Сложность

1.1. Сложность, присущая программному обеспечению

**Простые и сложные программные системы**

Мы знаем, что не все программные системы сложны. Существует множество программ, которые задумываются, разрабатываются, сопровождаются и используются одним и тем же человеком. Обычно это начинающий программист или профессионал, работающий изолированно. Нас интересует разработка того, что мы будем называть *промышленными программными продуктами.*

Существенная черта промышленной программы - уровень сложности: один разработчик практически не в состоянии охватить все аспекты такой системы. Грубо говоря, сложность промышленных программ превышает возможности человеческого интеллекта. Увы, но сложность, о которой мы говорим, по-видимому, присуща всем большим программных системам. Говоря *"присуща",* мы имеем в виду, что эта сложность здесь неизбежна: с ней можно справиться, но избавиться от нее нельзя.

**Почему программному обеспечению присуща сложность?**

Как говорит Брукс, "сложность программного обеспечения - отнюдь не случайное его свойство". Сложность вызывается четырьмя основными причинами:

сложностью реальной предметной области, из которой исходит заказ на разработку;

трудностью управления процессом разработки;

необходимостью обеспечить достаточную гибкость программы;

неудовлетворительными способами описания поведения больших дискретных систем.

**Пять признаков сложной системы**

Исходя из такого способа изучения, можно вывести пять общих признаков любой сложной системы. Основываясь на работе Саймона и Эндо, Куртуа предлагает следующее наблюдение [[7](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#1.7)]:

*1. "Сложные системы часто являются иерархическими и состоят из взаимозависимых подсистем, которые в свою очередь также могут быть разделены на подсистемы, и т.д., вплоть до самого низкого уровням."*

Саймон отмечает: "тот факт, что многие сложные системы имеют почти разложимую иерархическую структуру, является главным фактором, позволяющим нам понять, описать и даже "увидеть" такие системы и их части" [[8](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#1.8)]. В самом деле, скорее всего, мы можем понять лишь те системы, которые имеют иерархическую структуру.

Важно осознать, что архитектура сложных систем складывается и из компонентов, и из иерархических отношений этих компонентов. Речтин отмечает: "Все системы имеют подсистемы, и все системы являются частями более крупных систем... Особенности системы обусловлены отношениями между ее частями, а не частями как таковыми" [[9](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#1.9)].

Что же следует считать простейшими элементами системы? Опыт подсказывает нам следующий ответ:

*2. Выбор, 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.

Саймон называет иерархические системы *разложимыми,* если они могут быть разделены на четко идентифицируемые части, и *почти разложимыми,* если их составляющие не являются абсолютно независимыми. Это подводит нас к следующему общему свойству всех сложных систем:

*3. "Внутрикомпонентная связь обычно сильнее, чем связь между компонентами. Это обстоятельство позволяет отделять "высокочастотные" взаимодействия внутри компонентов от "низкочастотной" динамики взаимодействия между компонентами" [*[*10*](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#1.10)*].*

Это различие внутрикомпонентных и межкомпонентных взаимодействий обуславливает разделение функций между частями системы и дает возможность относительно изолированно изучать каждую часть.

Как мы уже говорили, многие сложные системы организованы достаточно экономными средствами. Поэтому Саймон приводит следующий признак сложных систем:

*4. "Иерархические системы обычно состоят из немногих типов подсистем, по-разному скомбинированных и организованных" [*[*11*](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#1.11)*].*

Иными словам и, разные сложные системы содержат одинаковые структурные части. Эти части могут использовать общие более мелкие компоненты, такие как клетки, или более крупные структуры, типа сосудистых систем, имеющиеся и у растений, и у животных.

Выше мы отмечали, что сложные системы имеют тенденцию к развитию во времени. Саймон считает, что сложные системы будут развиваться из простых гораздо быстрее, если для них существуют устойчивые промежуточные формы [[12](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#1.12)]. Гэлл [[13](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#1.13)] выражается более эффектно:

*5. "Любая работающая сложная система является результатом развития работавшей более простой системы... Сложная система, спроектированная "с нуля", никогда не заработает. Следует начинать с работающей простой системы".*

В процессе развития системы объекты, первоначально рассматривавшиеся как сложные, становятся элементарными, и из них строятся более сложные системы. Более того, невозможно сразу правильно создать элементарные объекты: с ними надо сначала повозиться, чтобы больше узнать о реальном поведении системы, и затем уже совершенствовать их.

Выводы

Программам присуща сложность, которая нередко превосходит возможности человеческого разума.

Задача разработчиков программных систем - создать у пользователя разрабатываемой системы иллюзию простоты.

Сложные структуры часто принимают форму иерархий; полезны обе иерархии: и классов, и объектов.

Сложные системы обычно создаются на основе устойчивых промежуточных форм.

Познавательные способности человека ограничены; мы можем раздвинуть их рамки, используя декомпозицию, выделение абстракций и создание иерархий.

Сложные системы можно исследовать, концентрируя основное внимание либо на объектах, либо на процессах; имеются веские основания использовать объектно-ориентированную декомпозицию, при которой мир рассматривается как упорядоченная совокупность объектов, которые в процессе взаимодействия друг с другом определяют поведение системы.

Объектно-ориентированный анализ и проектирование - метод, использующий объектную декомпозицию; объектно-ориентированный подход имеет свою систему условных обозначений и предлагает богатый набор логических и физических моделей, с помощью которых мы можем получить представление о различных аспектах рассматриваемой системы.

Объектная модель

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Объектно-ориентированная технология основывается на так называемой *объектной модели.*Основными ее принципами являются: абстрагирование, инкапсуляция, модульность, иерархичность, типизация, параллелизм и сохраняемость. Каждый из этих принципов сам по себе не нов, но в объектной модели они впервые применены в совокупности.  Объектно-ориентированный анализ и проектирование принципиально отличаются от традиционных подходов структурного проектирования: здесь нужно по-другому представлять себе процесс декомпозиции, а архитектура получающегося программного продукта в значительной степени выходит за рамки представлений, традиционных для структурного программирования. Отличия обусловлены тем, что структурное проектирование основано на структурном программировании, тогда как в основе объектно-ориентированного проектирования лежит методология объектно-ориентированного программирования, К сожалению, для разных людей термин "объектно-ориентированное программирование" означает разное. Ренч правильно предсказал: "В 1980-х годах объектно-ориентированное программирование будет занимать такое же место, какое занимало структурное программирование в 1970-х. но всем будет нравиться. Каждая фирма будет рекламировать свой продукт как зданный по этой технологии. Все программисты будут писать в этом стиле, причем все по-разному. Все менеджеры будут рассуждать о нем. И никто не будет знать, что же это такое" [Wegner, P. [J 1981]] [[1](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.1)]. Данные предсказания продолжают сбываться и в 1990-х годах.  В этой главе мы выясним, чем является и чем не является объектно-ориентированная разработка программ, и в чем отличия этого подхода к проектированию от других с учетом семи перечисленных выше элементов объектной модели.  2.1. Эволюция объектной модели  **Тенденции в проектировании**  **Поколения языков программирования.** Оглядываясь на короткую, но колоритную историю развития программирования, нельзя не заметить две сменяющих друг друга тенденции:  смещение акцентов от программирования отдельных деталей к программированию более крупных компонент;  развитие и совершенствование языков программирования высокого уровня.  Большинство современных коммерческих программных систем больше и существенно сложнее, чем были их предшественники даже несколько лет тому назад. Этот рост сложности вызвал большое число прикладных исследований по методологии проектирования, особенно, по декомпозиции, абстрагированию и иерархиям. Создание более выразительных языков программирования пополнило достижения в этой области. Возникла тенденция перехода от языков, указывающих компьютеру, что делать (императивные языки), к языкам, описывающим ключевые абстракции проблемной области (декларативные языки).  Вегнер сгруппировал некоторые из наиболее известных языков высокого уровня в четыре поколения в зависимости от того, какие языковые конструкции впервые в них появились:  Первое поколение (1954-1958)   |  |  | | --- | --- | | FORTRAN I | Математические формулы | | ALGOL-58 | Математические формулы | | Flowmatic | Математические формулы | | IPL V | Математические формулы |     Второе поколение (1959-1961)   |  |  | | --- | --- | | FORTRAN II | Подпрограммы, раздельная компиляция | | ALGOL-60 | Блочная структура, типы данных | | COBOL | Описание данных, работа с файлами | | Lisp | Обработка списков, указатели, сборка мусора |     Третье поколение(1962-1970)   |  |  | | --- | --- | | PL/I | FORTRAN+ALGOL+COBOL | | ALGOL-68 | Более строгий приемник ALGOL-60 | | Pascal | Более простой приемник ALGOL-60 | | Simula | Классы, абстрактные данные |     Потерянное поколение (1970-1980)  Много языков созданных, но мало выживших [Последняя фраза, очевидно, следует евангельскому "...много званных, но мало избранных" (Матф. 22:14). - Примеч. ред.] [[2](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.2)].  В каждом следующем поколении менялись поддерживаемые языками механизмы абстракции. Языки первого поколения ориентировались на научно-инженерные применения, и словарь этой предметной области был почти исключительно математическим. Такие языки, как FORTRAN I, были созданы для упрощения программирования математических формул, чтобы освободить программиста от трудностей ассемблера и машинного кода. Первое поколение языков высокого уровня было шагом, приближающим программирование к предметной области и удаляющим от конкретной машины. Во втором поколении языков основной тенденцией стало развитие алгоритмических абстракций. В это время мощность компьютеров быстро росла, а компьютерная индустрия позволила расширить области их применения, особенно в бизнесе. Главной задачей стало инструктировать машину, что делать: сначала прочти эти анкеты сотрудников, затем отсортируй их и выведи результаты на печать. Это было еще одним шагом к предметной области и от конкретной машины. В конце 60-х годов с появлением транзисторов, а затем интегральных схем, стоимость компьютеров резко снизилась, а их производительность росла почти экспоненциально. Появилась возможность решать все более сложные задачи, но это требовало умения обрабатывать самые разнообразные типы данных. Такие языки как ALGOL-68 и затем Pascal стали поддерживать абстракцию данных. Программисты смогли описывать свои собственные типы данных. Это стало еще одним шагом к предметной области и от привязки к конкретной машине.  70-е годы знаменовались безумным всплеском активности: было создано около двух тысяч различных языков и их диалектов. Неадекватность более ранних языков написанию крупных программных систем стала очевидной, поэтому новые языки имели механизмы, устраняющие это ограничение. Лишь немногие из этих языков смогли выжить (попробуйте найти свежий учебник по языкам Fred, Chaos, Tranquil), однако многие их принципы нашли отражение в новых версиях более ранних языков. Таким образом, мы получили языки Smalltalk (новаторски переработанное наследие Simula), Ada (наследник ALGOL-68 и Pascal с элементами Simula, Alphard и CLU), CLOS (объединивший Lisp, LOOPS и Flavors), C++ (возникший от брака С и Simula) и Eiffel (произошел от Simula и Ada). Наибольший интерес для дальнейшего изложения представляет класс языков, называемых *объектными* и *объектно-ориентированными,* которые в наибольшей степени отвечают задаче объектно-ориентированной декомпозиции программного обеспечения.  **Топология языков первого и начала второго поколения.** Для пояснения сказанного рассмотрим структуры, характерные для каждого поколения. На рис. 2-1 показана топология, типичная для большинства языков первого поколения и первой стадии второго поколения. Говоря "топология", мы имеем в виду основные элементы языка программирования и их взаимодействие. Можно отметить, что для таких языков, как FORTRAN и COBOL, основным строительным блоком является подпрограмма (параграф в терминах COBOL). Программы, реализованные на таких языках, имеют относительно простую структуру, состоящую только из глобальных данных и подпрограмм. Стрелками на рисунке обозначено влияние подпрограмм на данные. В процессе разработки можно логически разделить разнотипные Данные, но механизмы языков практически не поддерживают такого разделения. Ошибка в какой-либо части программы может иметь далеко идущие последствия, так как область данных открыта всем подпрограммам. В больших системах трудно гарантировать целостность данных при внесении изменений в какую-либо часть системы. В процессе эксплуатации уже через короткое время возникает путаница из-за большого количества перекрестных связей между подпрограммами, запутанных схем управления, неясного смысла данных, что угрожает надежности системы и определенно снижает ясность программы.   http://www.helloworld.ru/texts/comp/other/oop/pic02_01.gif  *Рис. 2-1. Топология языков первого и начала второго поколения.*  **Топология языков позднего второго и раннего третьего поколения.** Начиная с середины 60-х годов стали осознавать роль подпрограмм как важного промежуточного звена между решаемой задачей и компьютером [[3](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.3)]. Шоу отмечает: "Первая программная абстракция, названная процедурной абстракцией, прямо вытекает из этого прагматического взгляда на программные средства... Подпрограммы возникли до 1950 года, но тогда они не были оценены в качестве абстракции... Их рассматривали как средства, упрощающие работу... Но очень скоро стало ясно, что подпрограммы это абстрактные программные функции" [[4](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.4)].  Использование подпрограмм как механизма абстрагирования имело три существенных последствия. Во-первых, были разработаны языки, поддерживавшие разнообразные механизмы передачи параметров. Во-вторых, были заложены основания структурного программирования, что выразилось в языковой поддержке механизмов вложенности подпрограмм и в научном исследовании структур управления и областей видимости. В-третьих, возникли методы структурного проектирования, стимулирующие разработчиков создавать большие системы, используя подпрограммы как готовые строительные блоки. Архитектура языков программирования этого периода (рис. 2-2), как и следовало ожидать, представляет собой вариации на темы предыдущего поколения. В нее внесены кое-какие усовершенствования, в частности, усилено управление алгоритмическими абстракциями, но остается нерешенной проблема программирования "в большом" и проектирования данных.  **Топология языков конца третьего поколения.** Начиная с FORTRAN II и далее, для решения задач программирования "в большом" начал развиваться новый важный механизм структурирования. Разрастание программных проектов означало увеличение размеров и коллективов программистов, а, следовательно, необходимость независимой разработки отдельных частей проекта. Ответом на эту потребность стал отдельно компилируемый модуль, который сначала был просто более или менее случайным набором данных и подпрограмм (рис. 2-3). В такие модули собирали подпрограммы, которые, как казалось, скорее всего будут изменяться совместно, и мало кто рассматривал их как новую технику абстракции. В большинстве языков этого поколения, хотя и поддерживалось модульное программирование, но не вводилось никаких правил, обеспечивающих согласование интерфейсов модулей. Программист, сочиняющий подпрограмму в одном из модулей, мог, например, ожидать, что ее будут вызывать с тремя параметрами: действительным числом, массивом из десяти элементов и целым числом, обозначающим логическое значение. Но в каком-то другом модуле, вопреки предположениям автора, эта подпрограмма могла по ошибке вызываться с фактическими параметрами в виде: целого числа, массива из пяти элементов и отрицательного числа. Аналогично, один из модулей мог завести общую область данных и считать, что это его собственная область, а другой модуль мог нарушить это предположение, свободно манипулируя с этими данными. К сожалению, поскольку большинство языков предоставляло в лучшем случае рудиментарную поддержку абстрактных данных и типов, такие ошибки выявлялись только при выполнении программы.   http://www.helloworld.ru/texts/comp/other/oop/pic02_02.gif  *Рис. 2-2. Топология языков позднего второго и раннего третьего поколения.*  **Топология объектных и объектно-ориентированных языков.** Значение абстрактных типов данных в разрешении проблемы сложности систем хорошо выразил Шанкар: "Абстрагирование, достигаемое посредством использования процедур, хорошо подходит для описания абстрактных действий, но не годится для описания абстрактных объектов. Это серьезный недостаток, так как во многих практических ситуациях сложность объектов, с которыми нужно работать, составляет основную часть сложности всей задачи" [[5](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.5)]. Осознание этого влечет два важных вывода. Во-первых, возникают методы проектирования на основе потоков данных, которые вносят упорядоченность в абстракцию данных в языках, ориентированных на алгоритмы. Во-вторых, появляется теория типов, которая воплощается в таких языках, как Pascal.  Естественным завершением реализации этих идей, начавшейся с языка Simula и развитой в последующих языках в 1970-1980-е годы, стало сравнительно недавнее появление таких языков, как Smalltalk, Object Pascal, C++, CLOS, Ada и Eiffel. По причинам, которые мы вскоре объясним, эти языки получили название *объектных* или *объектно-ориентированных.* На рис. 2-4 приведена топология таких языков применительно к задачам малой и средней степени сложности. Основным элементом конструкции в указанных языках служит *модуль,* составленный из логически связанных классов и объектов, а не подпрограмма, как в языках первого поколения.   http://www.helloworld.ru/texts/comp/other/oop/pic02_03.gif  *Рис. 2-3. Топология языков конца третьего поколения.*  Другими словами: "Если процедуры и функции - глаголы, а данные - существительные, то процедурные программы строятся из глаголов, а объектно-ориентированные - из существительных" [[6](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.6)]. По этой же причине структура программ малой и средней сложности при объектно-ориентированном подходе представляется графом, а не деревом, как в случае алгоритмических языков. Кроме того, уменьшена или отсутствует область глобальных данных. Данные и действия организуются теперь таким образом, что основными логическими строительными блоками наших систем становятся классы и объекты, а не алгоритмы.  В настоящее время мы продвинулись много дальше программирования "в большом" и предстали перед программированием "в огромном". Для очень сложных систем классы, объекты и модули являются необходимыми, но не достаточными средствами абстракции. К счастью, объектный подход масштабируется и может быть применен на все более высоких уровнях. Кластеры абстракций в больших системах могут представляться в виде многослойной структуры. На каждом уровне можно выделить группы объектов, тесно взаимодействующих для решения задачи более высокого уровня абстракции. Внутри каждого кластера мы неизбежно найдем такое же множество взаимодействующих абстракций (рис. 2-5). Это соответствует подходу к сложным системам, изложенному в главе 1.  **Основные положения объектной модели**  Методы структурного проектирования помогают упростить процесс разработки сложных систем за счет использования алгоритмов как готовых строительных блоков. Аналогично, методы объектно-ориентированного проектирования созданы, чтобы помочь разработчикам применять мощные выразительные средства объектного и объектно-ориентированного программирования, использующего в качестве блоков классы и объекты.  Но в объектной модели отражается и множество других факторов. Как показано во врезке ниже, объектный подход зарекомендовал себя как унифицирующая идея всей компьютерной науки, применимая не только в программировании, но также в проектировании интерфейса пользователя, баз данных и даже архитектуры компьютеров. Причина такой широты в том, что ориентация на объекты позволяет нам справляться со сложностью систем самой разной природы.   http://www.helloworld.ru/texts/comp/other/oop/pic02_04.gif  *Рис. 2-4. Топология малых и средних приложений в объектных и объектно-ориентированных языках.*  Объектно-ориентированный анализ и проектирование отражают эволюционное, а не революционное развитие проектирования; новая методология не порывает с прежними методами, а строится с учетом предшествующего опыта. К сожалению, большинство программистов в настоящее время формально и неформально натренированы на применение только методов структурного проектирования. Разумеется, многие хорошие проектировщики создали и продолжают совершенствовать большое количество программных систем на основе этой методологии. Однако алгоритмическая декомпозиция помогает только до определенного предела, и обращение к объектно-ориентированной декомпозиции необходимо. Более того, при попытках использовать такие языки, как C++ или Ada, в качестве традиционных, алгоритмически ориентированных, мы не только теряем их внутренний потенциал - скорее всего результат будет даже хуже, чем при использовании обычных языков С и Pascal. Дать электродрель плотнику, который не слышал об электричестве, значит использовать ее в качестве молотка. Он согнет несколько гвоздей и разобьет себе пальцы, потому что электродрель мало пригодна для замены молотка.  **OOP, OOD и ООА**  Унаследовав от многих предшественников, объектный подход, к сожалению, перенял и запутанную терминологию. Программист в Smalltalk пользуется термином *метод,* в C++ - термином *виртуальная функция,* в CLOS - *обобщенная функция.*   http://www.helloworld.ru/texts/comp/other/oop/pic02_05.gif  *Рис. 2-5. Топология больших приложений в объектных и объектно-ориентированных языках.*  *В* Object Pascal используется термин *приведение типов,* а в языке Ada то же самое называется *преобразование типов.*Чтобы уменьшить путаницу, следует определить, что является объектно-ориентированным, а что - нет. Определение наиболее употребительных терминов и понятий вы найдете в глоссарии в конце книги.  Термин *объектно-ориентированный,* по мнению Бхаскара, "затаскан до потери смысла, как "материнство", "яблочный пирог" и "структурное программирование"" [[7](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.7)]. Можно согласиться, что понятие объекта является центральным во всем, что относится к объектно-ориентированной методологии. В главе 1 мы определили объект как осязаемую сущность, которая четко проявляет свое поведение. Стефик и Бобров определяют объекты как "сущности, объединяющие процедуры и данные, так как они производят вычисления и сохраняют свое локальное состояние" [[8](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.8)]. Определение объекта как сущности в какой-то мере отвечает на вопрос, но все же главным в понятии объекта является объединение идей абстракции данных и алгоритмов. Джонс уточняет это понятие следующим образом: "В объектном подходе акцент переносится на конкретные характеристики физической или абстрактной системы, являющейся предметом программного моделирования... Объекты обладают целостностью, которая не должна - а, в действительности, не может - быть нарушена. Объект может только менять состояние, вести себя, управляться или становиться в определенное отношение к другим объектам. Иначе говоря, свойства, которые характеризуют объект и его поведение, остаются неизменными. Например, лифт характеризуется теми неизменными свойствами, что он может двигаться вверх и вниз, оставаясь в пределах шахты... Любая модель должна учитывать эти свойства лифта, так как они входят в его определение" [[32](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.32)].    |  | | --- | | Основные положения объектной модели  Йонесава и Токоро свидетельствуют: "термин "объект" появился практически независимо в различных областях, связанных с компьютерами, и почти одновременно в начале 70-х годов для обозначения того, что может иметь различные проявления, оставаясь целостным. Для того, чтобы уменьшить сложность программных систем, объектами назывались компоненты системы или фрагменты представляемых знании" [[9](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.9)]. По мнению Леви, объектно-ориентированный подход был связан со следующими событиями:  "прогресс в области архитектуры ЭВМ;  развитие языков программирования, таких как Simula, Smalltalk, CLU, Ada;  развитие методологии программирования, включая принципы модульности и скрытия данных" [[10](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.10)].  К этому еще следует добавить три момента, оказавшие влияние на становление объектного подхода:  развитие теории баз данных;  исследования в области искусственного интеллекта;  достижения философии и теории познания.  Понятие "объект" впервые было использовано более 20 лет назад при конструировании компьютеров с descriptor-based и capability-based архитектурами [[11](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.11)]. В этих работах делались попытки отойти от традиционной архитектуры фон Неймана и преодолеть барьер между высоким уровнем программной абстракции и низким уровнем ЭВМ [[12](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.12)]. По мнению сторонников этих подходов, тогда были созданы более качественные средства, обеспечивающие: лучшее выявление ошибок, большую эффективность реализации программ, сокращение набора инструкций, упрощение компиляции, снижение объема требуемой памяти. Ряд компьютеров имеет объектно-ориентированную архитектуру: Burroughs 5000, Plessey 250, Cambridge CAP [[13](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.13)], SWARD [[14](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.14)], Intel 432 [[15](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.15)], Caltech's СОМ [[16](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.16)], IBM System/38 [[17](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.17)], Rational R1000, BiiN 40 и 60.  С объектно-ориентированной архитектурой тесно связаны объектно-ориентированные операционные системы (ОС). Дейкстра, работая над мультипрограммной системой THE, впервые ввел понятие машины с уровнями состояния в качестве средства построения системы [[18](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.18)]. Среди первых объектно-ориентированных ОС следует отметить: Plessey/System 250 (для мультипроцессора Plessey 250), Hydra (для CMU C.mmp), CALTSS (для CDC 6400), CAP (для компьютера Cambridge CAP), UCLA Secure UNIX (для PDP 11/45 и 11/70), StarOS (для CMU Cm\*), Medusa (также для CMU Cm\*) и iMAX (для Intel 432) [[19](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.19)]. Следующее поколение операционных систем, таких, как Microsoft Cairo и Taligent Pink, будет, по всей видимости, объектно-ориентированным.  Наиболее значительный вклад в объектный подход внесен объектными и объектно-ориентированными языками программирования. Впервые понятия классов и объектов введены в языке Simula 67. Система Flex и последовавшие за ней диалекты Smalltalk-72, -74, -76 и, наконец, -80, взяв за основу методы Simula, довели их до логического завершения, выполняя все действия на основе классов. В 1970-х годах создан ряд языков, реализующих идею абстракции данных: Alphard, CLU, Euclid, Gypsy, Mesa и Modula. Затем методы, используемые в языках Simula и Smalltalk, были использованы в традиционных языках высокого уровня. Внесение объектно-ориентированного подхода в С привело к возникновению языков C++ и Objective С. На основе языка Pascal возникли Object Pascal, Eiffel и Ada. Появились диалекты LISP, такие, как Flavors, LOOPS и CLOS (Common LISP Object System), с возможностями языков Simula и Smalltalk. Более подробно особенности этих языков изложены в приложении.  Первым, кто указал на необходимость построения систем в виде структурированных абстракций, был Дейкстра. Позднее Парнас ввел идею скрытия информации [[20](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.20)], а в 70-х годах ряд исследователей, главным образом Лисков и Жиль [[21](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.21)], Гуттаг [[22](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.22)], и Шоу [[23](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.23)], разработал механизмы абстрактных типов данных. Хоар дополнил эти подходы теорией типов и подклассов [[24](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.24)].  Развивавшиеся достаточно независимо технологии построения баз данных также оказали влияние на объектный подход [[25](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.25)], в первую очередь благодаря так называемой модели "сущность-отношение" (ER, entity-relationship) [[26](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.26)]. В моделях ER, впервые предложенных Ченом [[27](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.27)], моделирование происходит в терминах сущностей, их атрибутов и взаимоотношений.  Разработчики способов представления данных в области искусственного интеллекта также внесли свой вклад в понимание объектно-ориентированных абстракций. В 1975 г. Мински выдвинул теорию фреймов для представления реальных объектов в системах распознавания образов и естественных языков [[28](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.28)]. Фреймы стали использоваться в качестве архитектурной основы в различных интеллектуальных системах.  Объектный подход известен еще издавна. Грекам принадлежит идея о том, что мир можно рассматривать в терминах как объектов, так и событий. А в XVII веке Декарт отмечал, что люди обычно имеют объектно-ориентированный взгляд на мир [[29](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.29)]. В XX веке эту тему развивала Рэнд в своей философии объективистской эпистемологии [[30](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.30)]. Позднее Мински предложил модель человеческого мышления, в которой разум человека рассматривается как общность различно мыслящих агентов [[31](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.31)]. Он доказывает, что только совместное действие таких агентов приводит к осмысленному поведению человека. |   **Объектно-ориентированное программирование.** Что же такое объектно-ориентированное программирование (object-oriented programming, OOP)? Мы определяем его следующим образом:  *Объектно-ориентированное программирование - это методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования.*  В данном определении можно выделить три части: 1) OOP использует в качестве базовых элементов *объекты,* а не алгоритмы (иерархия "быть частью", которая была определена в главе 1); 2) каждый объект является *экземпляром*какого-либо определенного *класса;* 3) классы организованы *иерархически* (см. понятие об иерархии "is а" там же). Программа будет объектно-ориентированной только при соблюдении всех трех указанных требований. В частности, программирование, не основанное на иерархических отношениях, не относится к OOP, а называется*программированием на основе абстрактных типов данных.*  В соответствии с этим определением не все языки программирования являются объектно-ориентированными. Страуструп определил так: "если термин *объектно-ориентированный язык* вообще что-либо означает, то он должен означать язык, имеющий средства хорошей поддержки объектно-ориентированного стиля программирования... Обеспечение такого стиля в свою очередь означает, что в языке удобно пользоваться этим стилем. Если написание программ в стиле OOP требует специальных усилий или оно невозможно совсем, то этот язык не отвечает требованиям OOP" [[33](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.33)]. Теоретически возможна имитация объектно-ориентированного программирования на обычных языках, таких, как Pascal и даже COBOL или ассемблер, но это крайне затруднительно. Карделли и Вегнер говорят, что: "язык программирования является объектно-ориентированным тогда и только тогда, когда выполняются следующие условия:  Поддерживаются объекты, то есть абстракции данных, имеющие интерфейс в виде именованных операций и собственные данные, с ограничением доступа к ним.  Объекты относятся к соответствующим типам (классам).  Типы (классы) могут наследовать атрибуты супертипов (суперклассов)" [[34](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.34)].  Поддержка наследования в таких языках означает возможность установления отношения "is-a" ("есть", "это есть", " - это"), например, красная роза - это цветок, а цветок - это растение. Языки, не имеющие таких механизмов, нельзя отнести к объектно-ориентированным. Карделли и Вегнер назвали такие языки *объектными,* но не *объектно-ориентированными.* Согласно этому определению объектно-ориентированными языками являются Smalltalk, Object Pascal, C++ и CLOS, a Ada - объектный язык. Но, поскольку объекты и классы являются элементами обеих групп языков, желательно использовать и в тех, и в других методы объектно-ориентированного проектирования.  **Объектно-ориентированное проектирование.** Программирование прежде всего подразумевает правильное и эффективное использование механизмов конкретных языков программирования. Проектирование, напротив, основное внимание уделяет правильному и эффективному структурированию сложных систем. Мы определяем объектно-ориентированное проектирование следующим образом:  *Объектно-ориентированное проектирование - это методология проектирования, соединяющая в себе процесс объектной декомпозиции и приемы представления логической и физической, а также статической и динамической моделей проектируемой системы.*  В данном определении содержатся две важные части: объектно-ориентированное проектирование 1) основывается на объектно-ориентированной декомпозиции; 2) использует многообразие приемов представления моделей, отражающих логическую (классы и объекты) и физическую (модули и процессы) структуру системы, а также ее статические и динамические аспекты.  Именно объектно-ориентированная декомпозиция отличает объектно-ориентированное проектирование от структурного; в первом случае логическая структура системы отражается абстракциями в виде классов и объектов, во втором - алгоритмами. Иногда мы будем использовать аббревиатуру *OOD,* object-oriented design, для обозначения метода объектно-ориентированного проектирования, изложенного в этой книге.  **Объектно-ориентированный анализ.** На объектную модель повлияла более ранняя модель жизненного цикла программного обеспечения. Традиционная техника структурного анализа, описанная в работах Де Марко [[35](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.35)], Иордана [[36](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.36)], Гейна и Сарсона [[37](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.37)], а с уточнениями для режимов реального времени у Варда и Меллора [[38](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.38)] и Хотли и Пирбхая [[39](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.39)], основана на потоках данных в системе. Объектно-ориентированный анализ (или *OOA, object-oriented analysis*) направлен на создание моделей реальной действительности на основе объектно-ориентированного мировоззрения.  *Объектно-ориентированный анализ - это методология, при которой требования к системе воспринимаются с точки зрения классов и объектов, выявленных в предметной области.*  Как соотносятся ООА, OOD и OOP? На результатах ООА формируются модели, на которых основывается OOD; OOD в свою очередь создает фундамент для окончательной реализации системы с использованием методологии OOP.  2.2. Составные части объектного подхода  **Парадигмы программирования**  Дженкинс и Глазго считают, что "в большинстве своем программисты используют в работе один язык программирования и следуют одному стилю. Они программируют в парадигме, навязанной используемым ими языком. Часто они оставляют в стороне альтернативные подходы к цели, а следовательно, им трудно увидеть преимущества стиля, более соответствующего решаемой задаче" [[40](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.40)]. Бобров и Стефик так определили понятие стиля программирования: "Это способ построения программ, основанный на определенных принципах программирования, и выбор подходящего языка, который делает понятными программы, написанные в этом стиле" [[41](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.41)]. Эти же авторы выявили пять основных разновидностей стилей программирования, которые перечислены ниже вместе с присущими им видами абстракций:    |  |  | | --- | --- | | ® процедурно-ориентированный | алгоритмы | | ® объектно-ориентированный | классы и объекты | | ® логико-ориентированный | цели, часто выраженные в терминах исчисления предикатов | | ® ориентированный на правила | правила "если-то" | | ® ориентированный на ограничения | инвариантные соотношения |   Невозможно признать какой-либо стиль программирования наилучшим во всех областях практического применения. Например, для проектирования баз знаний более пригоден стиль, ориентированный на правила, а для вычислительных задач - процедурно-ориентированный. По нашему опыту объектно-ориентированный стиль является наиболее приемлемым для широчайшего круга приложений; действительно, эта парадигма часто служит архитектурным фундаментом, на котором мы основываем другие парадигмы.  Каждый стиль программирования имеет свою концептуальную базу. Каждый стиль требует своего умонастроения и способа восприятия решаемой задачи. Для объектно-ориентированного стиля концептуальная база - это *объектная модель.* Она имеет четыре главных элемента:  абстрагирование;  инкапсуляция;  модульность;  иерархия.  Эти элементы являются *главными* в том смысле, что без любого из них модель не будет объектно-ориентированной. Кроме главных, имеются еще три дополнительных элемента:  типизация;  параллелизм;  сохраняемость.  Называя их *дополнительными,* мы имеем в виду, что они полезны в объектной модели, но не обязательны.  Без такой концептуальной основы вы можете программировать на языке типа Smalltalk, Object Pascal, C++, CLOS, Eiffel или Ada, но из-под внешней красоты будет выглядывать стиль FORTRAN, Pascal или С. Выразительная способность объектно-ориентированного языка будет либо потеряна, либо искажена. Но еще более существенно, что при этом будет мало шансов справиться со сложностью решаемых задач.  **Абстрагирование**  **Смысл абстрагирования.** Абстрагирование является одним из основных методов, используемых для решения сложных задач. Хоар считает, что "абстрагирование проявляется в нахождении сходств между определенными объектами, ситуациями или процессами реального мира, и в принятии решений на основе этих сходств, отвлекаясь на время от имеющихся различий" [[42](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.42)]. Шоу определила это понятие так: "Упрощенное описание или изложение системы, при котором одни свойства и детали выделяются, а другие опускаются. Хорошей является такая абстракция, которая подчеркивает детали, существенные для рассмотрения и использования, и опускает те, которые на данный момент несущественны" [[43](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.43)]. Берзинс, Грей и Науман рекомендовали, чтобы "идея квалифицировалась как абстракция только, если она может быть изложена, понята и проанализирована независимо от механизма, который будет в дальнейшем принят для ее реализации" [[44](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.44)]. Суммируя эти разные точки зрения, получим следующее определение абстракции:  *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. Абельсон и Суссман назвали такое разделение смысла и реализации *барьером абстракции* [[45](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.45)], который основывается на принципе минимизации связей, когда интерфейс объекта содержит только существенные аспекты поведения и ничего больше [[46](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.46)]. Мы считаем полезным еще один дополнительный принцип, называемый *принципом наименьшего удивления,* согласно которому абстракция должна охватывать все поведение объекта, но не больше и не меньше, и не привносить сюрпризов или побочных эффектов, лежащих вне ее сферы применимости.  Выбор правильного набора абстракций для заданной предметной области представляет собой главную задачу объектно-ориентированного проектирования. Ввиду важности этой темы ей целиком посвящена глава 4.  По мнению Сейдвица и Старка "существует целый спектр абстракций, начиная с объектов, которые почти точно соответствуют реалиям предметной области, и кончая объектами, не имеющими право на существование" [[47](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.47)]. Вот эти абстракции, начиная от наиболее полезных к наименее полезным:    |  |  | | --- | --- | | ® Абстракция сущности | Объект представляет собой полезную модель некой сущности в предметной области | | ® Абстракция поведения | Объект состоит из обобщенного множества операций | | ® Абстракция виртуальной машины | Объект группирует операции, которые либо вместе используются более высоким уровнем управления, либо сами используют некоторый набор операций более низкого уровня | | ® Произвольная абстракция | Объект включает в себя набор операций, не имеющих друг с другом ничего общего |   Мы стараемся строить абстракции сущности, так как они прямо соответствуют сущностям предметной области.  *Клиентом* называется 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. Такой подход концентрирует внимание на внешних проявлениях объекта и приводит к идее, которую Мейер назвал *контрактной моделью* программирования [[48](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.48)]: 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 [[49](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.49)].  Каждая операция, предусмотренная этим контрактом, однозначно определяется ее формальными параметрами 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 способы, которыми объект может действовать или подвергаться воздействию, полностью определяя тем самым внешнее поведение абстракции со статической и динамической точек зрения.   http://www.helloworld.ru/texts/comp/other/oop/pg057.gif  *Абстракция фокусируется на существенных с точки зрения наблюдателя характеристиках объекта.*  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* (инварианты предполагаемые операцией) и *постусловия* (инварианты, которым удовлетворяет операция). Изменение инварианта нарушает контракт, связанный с абстракцией. В частности, если нарушено предусловие, то клиент не соблюдает свои обязательства и сервер не может выполнить свою задачу правильно. Если же нарушено постусловие, то свои обязательства нарушил сервер, и клиент не может более ему доверять. В случае нарушения какого-либо условия возбуждается исключительная ситуация. Как мы увидим далее, некоторые языки имеют средства для работы с исключительными ситуациями: объекты могут возбуждать исключения, чтобы запретить дальнейшую обработку и предупредить о проблеме другие объекты, которые в свою очередь могут принять на себя перехват исключения и справиться с проблемой.  Заметим, что понятия *операция, метод и функция-член* происходят от различных традиций программирования (Ada, Smalltalk и C++ соответственно). Фактически они обозначают одно и то же и в дальнейшем будут взаимозаменяемы.  Все абстракции обладают как статическими, так и динамическими свойствами. Например, файл как объект требует определенного объема памяти на конкретном устройстве, имеет имя и содержание. 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. В процедурном стиле программирования действия, изменяющие динамические характеристики объектов, составляют суть программы. Любые события связаны с вызовом подпрограмм и с выполнением операторов. Стиль программирования, ориентированный на правила, характеризуется тем, что под влиянием определенных условий активизируются определенные правила, которые в свою очередь вызывают другие правила, и т.д. Объектно-ориентированный стиль программирования связан с воздействием на объекты (в терминах Smalltalk с *передачей объектам сообщений*). Так, операция над объектом порождает некоторую реакцию этого объекта. Операции, которые можно выполнить по отношению к данному объекту, и реакция объекта на внешние воздействия определяют поведение этого объекта.  **Примеры абстракций.** Для иллюстрации сказанного выше приведем несколько примеров. В данном случае мы сконцентрируем внимание не столько на выделении абстракций для конкретной задачи (это подробно рассмотрено в главе 4), сколько на способе выражения абстракций.  В тепличном хозяйстве, использующем гидропонику, растения выращиваются на питательном растворе без песка, гравия или другой почвы. Управление режимом работы парниковой установки - очень ответственное дело, зависящее как от вида выращиваемых культур, так и от стадии выращивания. Нужно контролировать целый ряд факторов: температуру, влажность, освещение, кислотность (показатель рН) и концентрацию питательных веществ. В больших хозяйствах для решения этой задачи часто используют автоматические системы, которые контролируют и регулируют указанные факторы. Попросту говоря, цель автоматизации состоит здесь в том, чтобы при минимальном вмешательстве человека добиться соблюдения режима выращивания.  Одна из ключевых абстракций в такой задаче - датчик. Известно несколько разновидностей датчиков. Все, что влияет на урожай, должно быть измерено, так что мы должны иметь датчики температуры воды и воздуха, влажности, рН, освещения и концентрации питательных веществ. С внешней точки зрения датчик температуры - это объект, который способен измерять температуру там, где он расположен. Что такое температура? Это числовой параметр, имеющий ограниченный диапазон значений и определенную точность, означающий число градусов по Фаренгейту, Цельсию или Кельвину. Что такое местоположение датчика? Это некоторое идентифицируемое место в теплице, температуру в котором нам необходимо знать; таких мест, вероятно, немного. Для датчика температуры существенно не столько само местоположение, сколько тот факт, что данный датчик расположен именно в данном месте и это отличает его от других датчиков. Теперь можно задать вопрос о том, каковы обязанности датчика температуры? Мы решаем, что датчик должен знать температуру в своем местонахождении и сообщать ее по запросу. Какие же действия может выполнять по отношению к датчику клиент? Мы принимаем решение о том, что клиент может калибровать датчик и получать от него значение текущей температуры.  Для демонстрации проектных решений будет использован язык C++. Читатели, недостаточно знакомые с этим языком, а также желающие уточнить свои знания по другим объектным и объектно-ориентированным языкам, упоминаемым в этой книге, могут найти их краткие описания с примерами в приложении. Итак, вот описания, задающие абстрактный датчик температуры на C++.  **// Температура по Фаренгейту**  **typedef float Temperature;**  **// Число, однозначно определяющее положение датчика**  **typedef unsigned int Location;**  **class TemperatureSensor {**  **public:**  **TemperatureSensor (Location);**  **~TemperatureSensor();**  **void calibrate(Temperature actualTemperature);**  **Temperature currentTemperature() const;**  **private:**  **...**  **};**  Здесь два оператора определения типов **Temperature** и **Location** вводят удобные псевдонимы для простейших типов, и это позволяет нам выражать свои абстракции на языке предметной области [К сожалению, конструкция**typedef** не определяет нового типа данных и не обеспечивает его защиты. Например, следующее описание в C++: "**typedef int Count;**" просто вводит синоним для примитивного типа int. Как мы увидим в следующем разделе, другие языки, такие как Ada и Eiffel, имеют более изощренную семантику в отношении строгой типизации базовых типов]. **Temperature** - это числовой тип данных в формате с плавающей точкой для записи температур в шкале Фаренгейта. Значения типа Location обозначают места фермы, где могут располагаться температурные датчики.  Класс **TemperatureSensor** - это только спецификация датчика; настоящая его начинка скрыта в его закрытой (private) части. Класс **TemperatureSensor** это еще не объект. Собственно датчики - это его *экземпляры,* и их нужно создать, прежде чем с ними можно будет оперировать. Например, можно написать так:  **Temperature temperature;**  **TemperatureSensor greenhouse1Sensor(1);**  **TemperatureSensor greenhouse2Sensor(2);**  **temperature = greenhouse1Sensor.currentTemperature();**  Рассмотрим инварианты, связанные с операцией currentTemperature. Предусловие включает предположение, что датчик установлен в правильным месте в теплице, а постусловие - что датчик возвращает значение температуры в градусах Фаренгейта.  До сих пор мы считали датчик пассивным: кто-то должен запросить у него температуру, и тогда он ответит. Однако есть и другой, столь же правомочный подход. Датчик мог бы активно следить за температурой и извещать другие объекты, когда ее отклонение от заданного значения превышает заданный уровень. Абстракция от этого меняется мало: всего лишь несколько иначе формулируется ответственность объекта. Какие новые операции нужны ему в связи с этим? Обычной идиомой для таких случаев является обратный вызов. Клиент предоставляет серверу функцию (функцию обратного вызова), а сервер вызывает ее, когда считает нужным. Здесь нужно написать что-нибудь вроде:  **class ActiveTemperatureSensor {  public:**  **ActiveTemperatureSensor (Location,**  **void (\*f)(Location, Temperature));**  **~ActiveTemperatureSensor();  void calibrate(Temperature actualTemperature);  void establishSetpoint(Temperature setpoint,**  **Temperature delta);**  **Temperature currentTemperature() const;**  **private:  ...  };**  Новый класс **ActiveTemperatureSensor** стал лишь чуть сложнее, но вполне адекватно выражает новую абстракцию. Создавая экземпляр датчика, мы передаем ему при инициализации не только место, но и указатель на функцию обратного вызова, параметры которой определяют место установки и температуру. Новая функция установки**establishSetpoint** позволяет клиенту изменять порог срабатывания датчика температуры, а ответственность датчика состоит в том, чтобы вызывать функцию обратного вызова каждый раз, когда текущая температура**actualTemperature** отклоняется от **setpoint** больше чем на **delta**. При этом клиенту становится известно место срабатывания и температура в нем, а дальше уже он сам должен знать, что с этим делать.  Заметьте, что клиент по-прежнему может запрашивать температуру по собственной инициативе. Но что если клиент не произведет инициализацию, например, не задаст допустимую температуру? При проектировании мы обязательно должны решить этот вопрос, приняв какое-нибудь разумное допущение: пусть считается, что интервал допустимых изменений температуры бесконечно широк.  Как именно класс **ActiveTemperatureSensor** выполняет свои обязательства, зависит от его внутреннего представления и не должно интересовать внешних клиентов. Это определяется реализацией его закрытой части и функций-членов.  Рассмотрим теперь другой пример абстракции. Для каждой выращиваемой культуры должен быть задан план выращивания, описывающий изменение во времени температуры, освещения, подкормки и ряда других факторов, обеспечивающих высокий урожай. Поскольку такой план является частью предметной области, вполне оправдана его реализация в виде абстракции.  Для каждой выращиваемой культуры существует свой отдельный план, но общая форма планов у всех культур одинакова. Основу плана выращивания составляет таблица, сопоставляющая моментам времени перечень необходимых действий. Например, для некоторой культуры на 15-е сутки роста план предусматривает поддержание в течении 16 часов температуры 78ЂF, из них 14 часов с освещением, а затем понижение температуры до 65ЂF на остальное время суток. Кроме того, может потребоваться внесение удобрений в середине дня, чтобы поддержать заданное значение кислотности.  Таким образом, план выращивания отвечает за координацию во времени всех действий, необходимых при выращивании культуры. Наше решение заключается в том, чтобы не поручать абстракции плана само выполнение плана, - это будет обязанностью другой абстракции. Так мы ясно разделим понятия между различными частями системы и ограничим концептуальный размер каждой отдельной абстракции.  С точки зрения интерфейса объекта-плана, клиент должен иметь возможность устанавливать детали плана, изменять план и запрашивать его. Например, объект может быть реализован с интерфейсом "человек-компьютер" и ручным изменением плана. Объект, который содержит детали плана выращивания, должен уметь изменять сам себя. Кроме того, должен существовать объект-исполнитель плана, умеющий читать план. Как видно из дальнейшего описания, ни один объект не обособлен, а все они взаимодействуют для обеспечения общей цели. Исходя из такого подхода, определяются границы каждого объекта-абстракции и протоколы их связи.  На C++ план выращивания будет выглядеть следующим образом. Сначала введем новые типы данных, приближая наши абстракции к словарю предметной области (день, час, освещение, кислотность, концентрация):  **// Число, обозначающее день года**  **typedef unsigned int Day;**  **// Число, обозначающее час дня**  **typedef unsigned int Hour;**  **// Булевский тип**  **enum Lights {OFF, ON};**  **// Число, обозначающее показатель кислотности в диапазоне от 1 до 14**  **typedef float pH;**  **// Число, обозначающее концентрацию в процентах: от 0 до 100**  **typedef float Concentration;**  Далее, в тактических целях, опишем следующую структуру:  **// Структура, определяющая условия в теплице**  **struct Condition {**  **Temperature temperature;**  **Lights lighting;**  **pH acidity;**  **Concentration concentration;**  **};**  Мы использовали структуру, а не класс, поскольку Condition - это просто механическое объединение параметров, без какого-либо внутреннего поведения, и более богатая семантика класса здесь не нужна.  Наконец, вот и план выращивания:  **class GrowingPlan (  public:**  **GrowingPlan (char \*name);  virtual ~GrowingPlan();  void clear();  virtual void establish(Day, Hour, const Condition&);  const char\* name() const;  const Condition& desiredConditions(Day, Hour) const;**  **protected:  ...  };**  Заметьте, что мы предусмотрели одну новую обязанность: каждый план имеет имя, и его можно устанавливать и запрашивать. Кроме того заметьте, что операция **establish** описана как virtual для того, чтобы подклассы могли ее переопределять.  В открытую (public) часть описания вынесены конструктор и деструктор объекта (определяющие процедуры его порождения и уничтожения), две процедуры модификации (очистка всего плана clear и определение элементов плана establish) и два селектора-определителя состояния (функции name и **desiredCondition**). Мы опустили в описании закрытую часть класса, заменив ее многоточием, поскольку сейчас нам важны внешние ответственности, а не внутреннее представление класса.  **Инкапсуляция**  **Что это значит?** Хотя мы описывали нашу абстракцию **GrowingPlan** как сопоставление действий моментам времени, она не обязательно должна быть реализована буквально как таблица данных. Действительно, клиенту нет никакого дела до реализации класса, который его обслуживает, до тех пор, пока тот соблюдает свои обязательства. На самом деле, абстракция объекта всегда предшествует его реализации. А после того, как решение о реализации принято, оно должно трактоваться как секрет абстракции, скрытый от большинства клиентов. Как мудро замечает Ингалс: "Никакая часть сложной системы не должна зависеть от внутреннего устройства какой-либо другой части" [[50](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.50)]. В то время, как абстракция "помогает людям думать о том, что они делают", инкапсуляция "позволяет легко перестраивать программы" [[51](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.51)].  Абстракция и инкапсуляция дополняют друг друга: абстрагирование направлено на наблюдаемое поведение объекта, а инкапсуляция занимается внутренним устройством. Чаще всего инкапсуляция выполняется посредством скрытия информации, то есть маскировкой всех внутренних деталей, не влияющих на внешнее поведение. Обычно скрываются и внутренняя структура объекта и реализация его методов.  Инкапсуляция, таким образом, определяет четкие границы между различными абстракциями. Возьмем для примера структуру растения: чтобы понять на верхнем уровне действие фотосинтеза, вполне допустимо игнорировать такие подробности, как функции корней растения или химию клеточных стенок. Аналогичным образом при проектировании базы данных принято писать программы так, чтобы они не зависели от физического представления данных; вместо этого сосредотачиваются на схеме, отражающей логическое строение данных [[52](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.52)]. В обоих случаях объекты защищены от деталей реализации объектов более низкого уровня.  Дисков прямо утверждает, что "абстракция будет работать только вместе с инкапсуляцией" [[53](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.53)]. Практически это означает наличие двух частей в классе: интерфейса и реализации. *Интерфейс* отражает внешнее поведение объекта, описывая абстракцию поведения всех объектов данного класса. Внутренняя *реализация* описывает представление этой абстракции и механизмы достижения желаемого поведения объекта. Принцип разделения интерфейса и реализации соответствует сути вещей: в интерфейсной части собрано все, что касается взаимодействия данного объекта с любыми другими объектами; реализация скрывает от других объектов все детали, не имеющие отношения к процессу взаимодействия объектов. Бритон и Парнас назвали такие детали "тайнами абстракции" [[54](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.54)].   http://www.helloworld.ru/texts/comp/other/oop/pg063.gif  *Инкапсуляция скрывает детали реализации объекта.*  Итак, инкапсуляцию можно определить следующим образом:  *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.*  **Примеры инкапсуляции.** Вернемся к примеру гидропонного тепличного хозяйства. Еще одной из ключевых абстракций данной предметной области является нагреватель, поддерживающий заданную температуру в помещении. Нагреватель является абстракцией низкого уровня, поэтому можно ограничиться всего тремя действиями с этим объектом: включение, выключение и запрос состояния. Нагреватель не должен отвечать за поддержание температуры, это будет поведением более высокого уровня, совместно реализуемым нагревателем, датчиком температуры и еще одним объектом. Мы говорим о поведении *более высокого уровня,* потому что оно основывается на простом поведении нагревателя и датчика, добавляя к ним кое-что еще, а именно *гистерезис* (или запаздывание), благодаря которому можно обойтись без частых включений и выключении нагревателя в состояниях, близких к граничным. Приняв такое решение о разделении ответственности, мы делаем каждую абстракцию более цельной.  Как всегда, начнем с типов.  **// Булевский тип**  **enum Boolean {FALSE, TRUE};**  В дополнение к трем предложенным выше операциям, нужны обычные мета-операции создания и уничтожения объекта (конструктор и деструктор). Поскольку в системе может быть несколько нагревателей, мы будем при создании каждого из них сообщать ему место, где он установлен, как мы делали это с классом датчиков температуры**TemperatureSensor**. Итак, вот класс **Heater** для абстрактных нагревателей, написанный на C++:  **class Heater {  public:**  **Heater(Location);  ~Heater();  void turnOn();  void tum0ff();  Boolean is0n() const;**  **private:  };**  Вот и все, что посторонним надо знать о классе **Heater**. Внутренность класса это совсем другое дело. Предположим, проектировщики аппаратуры решили разместить управляющие компьютеры вне теплицы (где слишком жарко и влажно), и соединить их с датчиками и исполнительными устройствами с помощью последовательных интерфейсов. Разумно ожидать, что нагреватели будут коммутироваться с помощью блока реле, а оно будет управляться командами, поступающими через последовательный интерфейс. Скажем, для включения нагревателя передается текстовое имя команды, номер места нагревателя и еще одно число, используемое как сигнал включения нагревателя.  Вот класс, выражающий абстрактный последовательный порт.  **class SerialPort {  public:**  **SerialPort();  ~SerialPort();  void write(char\*);  void write(int);  static SerialPort ports[10];**  **private:  };**  Экземпляры этого класса будут настоящими последовательными портами, в которые можно выводить строки и числа.  Добавим еще три параметра в класс **Heater**.  **class Heater {  public:  ...  protected:**  **const Location repLocation;  Boolean repIsOn;  SerialPort\* repPort;**  **};**  Эти параметры **repLocation**, **repIsOn**, **repPort** образуют его инкапсулированное состояние. Правила C++ таковы, что при компиляции программы, если клиент попытается обратиться к этим параметрам напрямую, будет выдано сообщение об ошибке.  Определим теперь реализации всех операций этого класса.  **Heater::Heater(Location 1)**  **: repLocation(1),  repIsOn(FALSE),  repPort(&SerialPort::ports[l]) {}**  **Heater::Heater() {}**  **void Heater::turnOn()  {**  **if (!repls0n) {**  **repPort->write("\*");  repPort->write(repLocation);  repPort->write(1);  repIsOn = TRUE;**  **}**  **}**  **void Heater::turn0ff()  {**  **if (repIsOn) {**  **repPort->write("\*");  repPort->write(repLocation);  repPort->write(0);  repIsOn = FALSE;**  **}**  **}**  **Boolean Heater::is0n() const  {**  **return repIsOn;**  **}**  Такой стиль реализации типичен для хорошо структурированных объектно-ориентированных систем: классы записываются экономно, поскольку их специализация осуществляется через подклассы.  Предположим, что по какой-либо причине изменилась архитектура аппаратных средств системы и вместо последовательного порта управление должно осуществляться через фиксированную область памяти. Нет необходимости изменять интерфейсную часть класса - достаточно переписать реализацию. Согласно правилам C++, после этого придется перекомпилировать измененный класс, но не другие объекты, если только они не зависят от временных и пространственных характеристик прежнего кода (что крайне нежелательно и совершенно не нужно).  Обратимся теперь к реализации класса **GrowingPlan**. Как было сказано, это, в сущности, временной график действий. Вероятно, лучшей реализацией его был бы словарь пар время-действие с открытой хеш-таблицей. Нет смысла запоминать действия час за часом, они происходят не так часто, а в промежутках между ними система может интерполировать ход процесса.  Инкапсуляция скроет от посторонних взглядов два секрета: то, что в действительности график использует открытую хеш-таблицу, и то, что промежуточные значения интерполируются. Клиенты вольны думать, что они получают данные из почасового массива значений параметров.  Разумная 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, чтобы реализовать более эффективные алгоритмы или оптимизировать алгоритм по критерию памяти, заменяя хранение данных вычислением. Важным преимуществом ограничения доступа является возможность внесения изменений в объект без изменения других объектов.  В идеальном случае попытки обращения к данным, закрытым для доступа, должны выявляться во время компиляции программы. Вопрос реализации этих условий для конкретных языков программирования является предметом постоянных обсуждений. Так, Smalltalk обеспечивает защиту от прямого доступа к экземплярам другого класса, обнаруживая такие попытки во время компиляции. В тоже время Object Pascal не инкапсулирует представление класса, так что ничто в этом языке не предохраняет клиента от прямых ссылок на внутренние поля другого объекта. Язык CLOS занимает в этом вопросе промежуточную позицию, возлагая все обязанности по ограничению доступа на программиста. В этом языке все *слоты* могут сопровождаться атрибутами **:reader**, **:writer** и **:accessor**, разрешающими соответственно чтение, запись или полный доступ к данным (то есть и чтение, и запись). При отсутствии атрибутов слот полностью инкапсулирован. По соглашению, признание того, что некоторая величина хранится в слоте, рассматривается как нарушение абстракции, так что хороший стиль программирования на CLOS требует, чтобы при публикации интерфейса класса, документировались бы только имена его функций, а тот факт, что слот имеет функции полного доступа, должен скрываться [[55](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.55)]. В языке C++ управление доступом и видимостью более гибко. Члены класса могут быть отнесены к открытой, закрытой или защищенной частям. Открытая часть доступна для всех объектов; закрытая часть полностью закрыта для других объектов; защищенная часть видна только экземплярам данного класса и его подклассов. Кроме того, в C++ существует понятие "друзей" (friends), для которых открыта закрытая часть.  Скрытие информации - понятие относительное: то, что спрятано на одном уровне абстракции, обнаруживается на другом уровне. Забраться внутрь объектов можно; правда, обычно требуется, чтобы разработчик класса-сервера об этом специально позаботился, а разработчики классов-клиентов не поленились в этом разобраться. Инкапсуляция не спасает от глупости; она, как отметил Страуструп, "защищает от ошибок, но не от жульничества" [[56](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.56)]. Разумеется, язык программирования тут вообще ни при чем; разве что операционная система может ограничить доступ к файлам, в которых описаны реализации классов. На практике же иногда просто необходимо ознакомиться с реализацией класса, чтобы понять его назначение, особенно, если нет внешней документации.  **Модульность**  **Понятие модульности.** По мнению Майерса "Разделение программы на модули до некоторой степени позволяет уменьшить ее сложность... Однако гораздо важнее тот факт, что внутри модульной программы создаются множества хорошо определенных и документированных интерфейсов. Эти интерфейсы неоценимы для исчерпывающего понимания программы в целом" [[57](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.57)]. В некоторых языках программирования, например в Smalltalk, модулей нет, и классы составляют единственную физическую основу декомпозиции. В других языках, включая Object Pascal, C++, Ada, CLOS, модуль - это самостоятельная языковая конструкция. В этих языках классы и объекты составляют логическую структуру системы, они помещаются в *модули,* образующие физическую структуру системы. Это свойство становится особенно полезным, когда система состоит из многих сотен классов.  Согласно Барбаре Лисков "модульность - это разделение программы на фрагменты, которые компилируются по отдельности, но могут устанавливать связи с другими модулями". Мы будем пользоваться определением Парнаса: "Связи между модулями - это их представления друг о друге" [[58](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.58)]. В большинстве языков, поддерживающих принцип модульности как самостоятельную концепцию, интерфейс модуля отделен от его реализации. Таким образом, модульность и инкапсуляция ходят рука об руку. В разных языках программирования модульность поддерживается по-разному. Например, в C++ модулями являются раздельно компилируемые файлы. Для C/C++ традиционным является помещение интерфейсной части модулей в отдельные файлы с расширением .h (так называемые *файлы-заголовки).*Реализация, то есть текст модуля, хранится в файлах с расширением .с (в программах на C++ часто используются расширения .ее, .ср и .срр). Связь между файлами объявляется директивой макропроцессора #include. Такой подход строится исключительно на соглашении и не является строгим требованием самого языка. В языке Object Pascal принцип модульности формализован несколько строже. В этом языке определен особый синтаксис для интерфейсной части и реализации модуля (unit). Язык Ada идет еще на шаг дальше: модуль (называемый package) также имеет две части - спецификацию и тело. Но, в отличие от Object Pascal, допускается раздельное определение связей с модулями для спецификации и тела пакета. Таким образом, допускается, чтобы тело модуля имело связи с модулями, невидимыми для его спецификации.  Правильное разделение программы на модули является почти такой же сложной задачей, как выбор правильного набора абстракций. Абсолютно прав Зельковиц, утверждая: "поскольку в начале работы над проектом решения могут быть неясными, декомпозиция на модули может вызвать затруднения. Для хорошо известных приложений (например, создание компиляторов) этот процесс можно стандартизовать, но для новых задач (военные системы или управление космическими аппаратами) задача может быть очень трудной" [[59](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.59)].  Модули выполняют роль физических контейнеров, в которые помещаются определения классов и объектов при логическом проектировании системы. Такая же ситуация возникает у проектировщиков бортовых компьютеров. Логика электронного оборудования может быть построена на основе элементарных схем типа НЕ, И-НЕ, ИЛИ-НЕ, но можно объединить такие схемы в стандартные интегральные схемы (модули), например, серий 7400, 7402 или 7404.  Для небольших задач допустимо описание всех классов и объектов в одном модуле. Однако для большинства программ (кроме самых тривиальных) лучшим решением будет сгруппировать в отдельный модуль логически связанные классы и объекты, оставив открытыми те элементы, которые совершенно необходимо видеть другим модулям. Такой способ разбиения на модули хорош, но его можно довести до абсурда. Рассмотрим, например, задачу, которая выполняется на многопроцессорном оборудовании и требует для координации своей работы механизм передачи сообщений. В больших системах, подобных описываемым в главе 12, вполне обычным является наличие нескольких сотен и даже тысяч видов сообщений. Было бы наивным определять каждый класс сообщения в отдельном модуле. При этом не только возникает кошмар с документированием, но даже просто поиск нужных фрагментов описания становится чрезвычайно труден для пользователя. При внесении в проект изменений потребуется модифицировать и перекомпилировать сотни модулей. Этот пример показывает, что скрытие информации имеет и обратную сторону [[60](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.60)]. Деление программы на модули бессистемным образом иногда гораздо хуже, чем отсутствие модульности вообще.   http://www.helloworld.ru/texts/comp/other/oop/pg068.gif  *Модульность позволяет хранить абстракции раздельно.*  В традиционном структурном проектировании модульность - это искусство раскладывать подпрограммы по кучкам так, чтобы в одну кучку попадали подпрограммы, использующие друг друга или изменяемые вместе. В объектно-ориентированном программировании ситуация несколько иная: необходимо физически разделить классы и объекты, составляющие логическую структуру проекта.  На основе имеющегося опыта можно перечислить приемы и правила, которые позволяют составлять модули из классов и объектов наиболее эффективным образом. Бритон и Парнас считают, что "конечной целью декомпозиции 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" [[61](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.61)]. Прагматические соображения ставят предел этим руководящим указаниям. На практике перекомпиляция тела модуля не является трудоемкой операцией: заново компилируется только данный модуль, и программа перекомпонуется. Перекомпиляция *интерфейсной* части модуля, напротив, более трудоемка. В строго типизированных языках приходится перекомпилировать интерфейс и тело самого измененного модуля, затем все модули, связанные с данным, модули, связанные с ними, и так далее по цепочке. В итоге для очень больших программ могут потребоваться многие часы на перекомпиляцию (если только среда разработки не поддерживает фрагментарную компиляцию), что явно нежелательно. Поэтому следует стремиться к тому, чтобы интерфейсная часть модулей была возможно более узкой (в пределах обеспечения необходимых связей). Наш стиль программирования требует скрыть все, что только возможно, в реализации модуля. Постепенный перенос описаний из реализации в интерфейсную часть гораздо менее опасен, чем "вычищение" избыточного интерфейсного кода.  Таким образом, программист должен находить баланс между двумя противоположными тенденциями: стремлением скрыть информацию и необходимостью обеспечения видимости тех или иных абстракций в нескольких модулях. Парнас, Клеменс и Вейс предложили следующее правило: "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" [[62](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.62)]. Другими словами, следует стремиться построить модули так, чтобы объединить логически связанные абстракции и минимизировать взаимные связи между модулями. Исходя из этого, приведем определение модульности:  *Модульность - это свойство системы, которая была разложена на внутренне связные, но слабо связанные между собой модули.*  Таким образом, принципы абстрагирования, инкапсуляции и модульности являются взаимодополняющими. Объект логически определяет границы определенной абстракции, а инкапсуляция и модульность делают их физически незыблемыми.  В процессе разделения системы на модули могут быть полезными два правила. Во-первых, поскольку модули служат в качестве элементарных и неделимых блоков программы, которые могут использоваться в системе повторно, распределение классов и объектов по модулям должно учитывать это. Во-вторых, многие компиляторы создают отдельный сегмент кода для каждого модуля. Поэтому могут появиться ограничения на размер модуля. Динамика вызовов подпрограмм и расположение описаний внутри модулей может сильно повлиять на локальность ссылок и на управление страницами виртуальной памяти. При плохом разбиении процедур по модулям учащаются взаимные вызовы между сегментами, что приводит к потере эффективности кэш-памяти и частой смене страниц.  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. При этом более опытные программисты обычно отвечают за интерфейс модулей, а менее опытные - за реализацию. На более крупном уровне такие же соотношения справедливы для отношений между субподрядчиками. Абстракции можно распределить так, чтобы быстро установить интерфейсы модулей по соглашению между компаниями, участвующими в работе. Изменения в интерфейсе вызывают много крика и зубовного скрежета, не говоря уже об огромном расходе бумаги, - все эти факторы делают интерфейс крайне консервативным. Что касается документирования проекта, то оно строится, как правило, также по модульному принципу - модуль служит единицей описания и администрирования. Десять модулей вместо одного потребуют в десять раз больше описаний, и поэтому, к сожалению, иногда требования по документированию влияют на декомпозицию проекта (в большинстве случаев негативно). Могут сказываться и требования секретности: часть кода может быть несекретной, а другая - секретной; последняя тогда выполняется в виде отдельного модуля (модулей).  Свести воедино столь разноречивые требования довольно трудно, но главное уяснить: вычленение классов и объектов в проекте и организация модульной структуры - *независимые* действия. Процесс вычленения классов и объектов составляет часть процесса логического проектирования системы, а деление на модули - этап физического проектирования. Разумеется, иногда невозможно завершить логическое проектирование системы, не завершив физическое проектирование, и наоборот. Два этих процесса выполняются итеративно.  **Примеры модульности.** Посмотрим, как реализуется модульность в гидропонной огородной системе. Допустим, вместо закупки специализированного аппаратного обеспечения, решено использовать стандартную рабочую станцию с графическим интерфейсом пользователя GUI (Graphical User Interface). С помощью рабочей станции оператор может формировать новые планы выращивания, модифицировать имеющиеся планы и наблюдать за их исполнением. Так как абстракция плана выращивания - одна из ключевых, создадим модуль, содержащий все, относящееся к плану выращивания. На C++ нам понадобится примерно такой файл-заголовок (пусть он называется gplan.h).  **// gplan.h**  **#ifndef \_GPLAN\_H**  **#define \_GPLAN\_H 1  #include "gtypes.h"  #include "except.h"  #include "actions.h"  class GrowingPlan ...  class FruitGrowingPlan ...  class GrainGrowingPlan ...**  **#endif**  Здесь мы импортируем в файл три других заголовочных файла с определением интерфейсов, на которые будем ссылаться: gtypes.h, except .h и actions.h. Собственно код классов мы поместим в модуль реализации, в файл с именем gplan.cpp.  Мы могли бы также собрать в один модуль все программы, относящиеся к окнам диалога, специфичным для данного приложения. Этот модуль наверняка будет зависеть от классов, объявленных в gplan.h, и от других файлов-заголовков с описанием классов GUI.  Вероятно, будет много других модулей, импортирующих интерфейсы более низкого уровня. Наконец мы доберемся до главной функции - точки запуска нашей программы операционной системой. При объектно-ориентированном проектировании это скорее всего будет самая малозначительная и неинтересная часть системы, в то время, как в традиционном структурном подходе головная функция - это краеугольный камень, который держит все сооружение. Мы полагаем, что объектно-ориентированный подход более естественен, поскольку, как замечает Мейер, "на практике программные системы предлагают некоторый набор услуг. Сводить их к одной функции можно, но противоестественно... Настоящие системы не имеют верхнего уровня" [[63](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.63)].  **Иерархия**  **Что такое иерархия?** Абстракция - вещь полезная, но 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 абстракций. 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. Определим иерархию следующим образом:  *Иерархия - это упорядочение абстракций, расположение их по уровням.*  Основными видами иерархических структур применительно к сложным системам являются структура классов (иерархия "is-a") и структура объектов (иерархия "part of").  Примеры иерархии: одиночное наследование. Важным элементом объектно-ориентированных систем и основным видом иерархии "is-a" является упоминавшаяся выше концепция наследования. Наследование означает такое отношение между классами (отношение родитель/потомок), когда один класс заимствует структурную или функциональную часть одного или нескольких других классов (соответственно, *одиночное* и *множественное наследование*). Иными словами, наследование создает такую иерархию абстракций, в которой подклассы наследуют строение от одного или нескольких суперклассов. Часто подкласс достраивает или переписывает компоненты вышестоящего класса.  Семантически, наследование описывает отношение типа "is-a". Например, медведь есть млекопитающее, дом есть недвижимость и "быстрая сортировка" есть сортирующий алгоритм. Таким образом, наследование порождает иерархию "обобщение-специализация", в которой подкласс представляет собой специализированный частный случай своего суперкласса. "Лакмусовая бумажка" наследования - обратная проверка; так, если B не есть A, то B не стоит производить от A.  Рассмотрим теперь различные виды растений, выращиваемых в нашей огородной системе. Мы уже ввели обобщенное представление абстрактного плана выращивания растений. Однако разные культуры требуют разных планов. При этом планы для фруктов похожи друг на друга, но отличаются от планов для овощей или цветов. Имеет смысл ввести на новом уровне абстракции обобщенный "фруктовый" план, включающий указания по опылению и сборке урожая. Вот как будет выглядеть на C++ определение плана для фруктов, как наследника общего плана выращивания.  **// Тип Урожай  typedef unsigned int Yield;**  **class FruitGrowingPlan : public GrowingPlan {  public:**  **FruitGrowingPlan(char\* name);  virtual ~FruitGrowingPlan();  virtual void establish(Day, Hour, Condition&);  void scheduleHarvest(Day, Hour);  Boolean isHarvested() const;  unsigned daysUntilHarvest() const;  Yield estimatedYield() const;**  **protected:**  **Boolean repHarvested;  Yield repYield;**  http://www.helloworld.ru/texts/comp/other/oop/pg072.gif  *Абстракции образуют иерархию.*  Это означает, что план выращивания фруктов **FruitGrowingPlan** является разновидностью плана выращивания**GrowingPlan**. В него добавлены параметры **repHarvested** и **repYield**, определены четыре новые функции и переопределена функция establish. Теперь мы могли бы продолжить специализацию - например, определить на базе "фруктового" плана "яблочный" класс **AppleGrowingPlan**.  В наследственной иерархии общая часть структуры и поведения сосредоточена в наиболее общем суперклассе. По этой причине говорят о наследовании, как об иерархии *обобщение-специализация.* Суперклассы при этом отражают наиболее общие, а подклассы - более специализированные абстракции, в которых члены суперкласса могут быть дополнены, модифицированы и даже скрыты. Принцип наследования позволяет упростить выражение абстракций, делает проект менее громоздким и более выразительным. Кокс пишет: "В отсутствие наследования каждый класс становится самостоятельным блоком и должен разрабатываться "с нуля". Классы лишаются общности, поскольку каждый программист реализует их по-своему. Стройность системы достигается тогда только за счет дисциплинированности программистов. Наследование позволяет вводить в обращение новые программы, как мы обучаем новичков новым понятиям - сравнивая новое с чем-то уже известным" [[64](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.64)].  Принципы абстрагирования, инкапсуляции и иерархии находятся между собой в некоем здоровом конфликте. Данфорт и Томлинсон утверждают: "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" [[65](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.65)]. Для любого класса обычно существуют два вида клиентов: объекты, которые манипулируют с экземплярами данного класса, и подклассы-наследники. Лисков поэтому отмечает, что существуют три способа нарушения инкапсуляции через наследование: "подкласс может получить доступ к переменным экземпляра своего суперкласса, вызвать закрытую функцию и, наконец, обратиться напрямую к суперклассу своего суперкласса" [[66](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.66)]. Различные языки программирования по-разному находят компромисс между наследованием и инкапсуляцией; наиболее гибким в этом отношении является C++. В нем интерфейс класса может быть разделен на три части: закрытую (private), видимую только для самого класса; защищенную (protected), видимую также и для подклассов; и открытую (public), видимую для всех.  **Примеры иерархии: множественное наследование.** В предыдущем примере рассматривалось одиночное наследование, когда подкласс **FruitGrowingPlan** был создан только из одного суперкласса **GrowingPlan**. В ряде случаев полезно реализовать наследование от нескольких суперклассов. Предположим, что нужно определить класс, представляющий разновидности растений.  **class Plant {  public:**  **Plant(char\* name, char\* species);  virtual ~Plant();  void setDatePlanted(Day);  virtual establishGrowingConditions(const Condition&);  const char\* name() const;  const char\* species() const;  Day datePlantedt) const;**  **protected:**  **char\* repName;  char\* repSpecies;  Day repPlanted;**  **private:  ...  };**  Каждый экземпляр класса **plant** будет содержать имя, вид и дату посадки. Кроме того, для каждого вида растений можно задавать особые оптимальные условия выращивания. Мы хотим, чтобы эта функция переопределялась подклассами, поэтому она объявлена *виртуальной* при реализации в C++. Три параметра объявлены как защищенные, то есть они будут доступны и классу, и подклассам (закрытая часть спецификации доступна только самому классу).  Изучая предметную область, мы приходим к выводу, что различные группы культивируемых растений - цветы, фрукты и овощи, - имеют свои особые свойства, существенные для технологии их выращивания. Например, для цветов важно знать времена цветения и созревания семян. Аналогично, время сбора урожая важно для абстракций фруктов и овощей. Создадим два новых класса - цветы **(Flower)** и фрукты-овощи **(FruitVegetable)**; они оба наследуют от класса**Plant**. Однако некоторые цветочные растения имеют плоды! Для этой абстракции придется создать третий класс,**FlowerFruitVegetable**, который будет наследовать от классов **Flower** и **FruitVegetablePlant**.  Чтобы не было избыточности, в данном случае очень пригодится множественное наследование. Сначала давайте опишем отдельно цветы и фрукты-овощи.  **class FlowerMixin {  public:**  **FlowerMixin(Day timeToFlower, Day timeToSeed);  virtual ~FlowerMixin();  Day timeToFlower() const;  Day timeToSeed() const;**  **protected:  ...  };**  **class FruitVegetableMixin {  public:**  **FruitVegetableMixin(Day timeToHarvest);  virtual ~FruitVegetableMixin();  Day timeToHarvest() const;**  **protected:  ...  };**  Мы намеренно описали эти два класса без наследования. Они ни от кого не наследуют и специально предназначены для того, чтобы их *подмешивали* (откуда и имя **Mixin**) к другим классам. Например, опишем розу:  **class Rose : public Plant, public FlowerMixin...**  А вот морковь:  **class Carrot : public Plant, public FruiteVegetableMixin {};**  В обоих случаях классы наследуют от двух суперклассов: экземпляры подкласса **Rose** включают структуру и поведение как из класса **Plant**, так и из класса **FlowerMixin**. И вот теперь определим вишню, у которой товаром являются как цветы, так и плоды:  **class Cherry : public Plant,** **public FlowerMixin,** **FruitVegetableMixin...**  Множественное наследование - вещь нехитрая, но оно осложняет реализацию языков программирования. Есть две проблемы - конфликты имен между различными суперклассами и повторное наследование. Первый случай, это когда в двух или большем числе суперклассов определено поле или операция с одинаковым именем. В C++ этот вид конфликта должен быть явно разрешен вручную, а в Smalltalk берется то, которое встречается первым. Повторное наследование, это когда класс наследует двум классам, а они порознь наследуют одному и тому же четвертому. Получается ромбическая структура наследования и надо решить, должен ли самый нижний класс получить одну или две отдельные копии самого верхнего класса? В некоторых языках повторное наследование запрещено, в других конфликт решается "волевым порядком", а в C++ это оставляется на усмотрение программиста. Виртуальные базовые классы используются для запрещения дублирования повторяющихся структур, в противном случае в подклассе появятся копии полей и функций и потребуется явное указание происхождения каждой из копий.  Множественным наследованием часто злоупотребляют. Например, сладкая вата - это частный случай сладости, но никак не ваты. Применяйте ту же "лакмусовую бумажку": если B не есть A, то ему не стоит наследовать от A. Часто плохо сформированные структуры множественного наследования могут быть сведены к единственному суперклассу плюс агрегация других классов подклассом.  **Примеры иерархии: агрегация.** Если иерархия "is а" определяет отношение "обобщение/специализация", то отношение "part of" (часть) вводит иерархию агрегации. Вот пример.  **class Garden {  public:**  **Garden();  virtual ~Garden();**  **protected:**  **Plant\* repPlants[100];  GrowingPlan repPlan;**  **};**  Это - абстракция огорода, состоящая из массива растений и плана выращивания.  Имея дело с такими иерархиями, мы часто говорим об уровнях абстракции, которые впервые предложил Дейкстра [[67](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.67)]. В иерархии классов вышестоящая абстракция является обобщением, а нижестоящая - специализацией. Поэтому мы говорим, что класс **Flower** находится на более высоком уровне абстракции, чем класс **Plant**. В иерархии "part of" класс находится на более высоком уровне абстракции, чем любой из использовавшихся при его реализации. Так класс**Garden** стоит на более высоком уровне, чем класс **Plant**.  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.  В связи с агрегацией возникает проблема владения, или принадлежности объектов. В нашем абстрактном огороде одновременно растет много растений, и от удаления или замены одного из них огород не становится другим огородом. Если мы уничтожаем огород, растения остаются (их ведь можно пересадить). Другими словами, огород и растения имеют свои отдельные и независимые сроки жизни; мы достигли этого благодаря тому, что огород содержит не сами объекты **Plant**, а указатели на них. Напротив, мы решили, что объект **GrowingPlan** внутренне связан с объектом**Garden** и не существует независимо. План выращивания физически содержится в каждом экземпляре огорода и погибает вместе с ним. Подробнее про семантику владения мы будем говорить в следующей главе.  **Типизация**  **Что такое типизация?** Понятие *типа* взято из теории абстрактных типов данных. Дойч определяет тип, как "точную характеристику свойств, включая структуру и поведение, относящуюся к некоторой совокупности объектов" [[68](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.68)]. Для наших целей достаточно считать, что термины *тип* и *класс* взаимозаменяемы [Тип и класс не вполне одно и то же; в некоторых языках их различают. Например, ранние версии языка Trellis/Owl разрешали объекту иметь и класс, и тип. Даже в Smalltalk объекты классов SmallInteger, LargeNegativeInteger, LargePositiveInteger относятся к одному типу Integer, хотя и к разным классам [[69](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.69)]. Большинству смертных различать типы и классы просто противно и бесполезно. Достаточно сказать, что класс реализует понятие типа]. Тем не менее, типы стоит обсудить отдельно, поскольку они выставляют смысл абстрагирования в совершенно другом свете. В частности, мы утверждаем, что:  *Типизация - это способ защититься от использования объектов одного класса вместо другого, или по крайней мере управлять таким использованием.*  Типизация заставляет нас выражать наши абстракции так, чтобы язык программирования, используемый в реализации, поддерживал соблюдение принятых проектных решений. Вегнер замечает, что такой способ контроля существенен для программирования "в большом" [[70](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.70)].  Идея согласования типов занимает в понятии типизации центральное место. Например, возьмем физические единицы измерения [[71](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.71)]. Деля 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 накладывает правила и ограничения на использование и сочетание абстракций.  **Примеры сильной и слабой типизации.** Конкретный язык программирования может иметь сильный или слабый механизм типизации, и даже не иметь вообще никакого, оставаясь объектно-ориентированным. Например, в Eiffel соблюдение правил использования типов контролируется непреклонно, - операция не может быть применена к объекту, если она не зарегистрирована в его классе или суперклассе. В сильно типизированных языках нарушение согласования типов может быть обнаружено во время трансляции программы. С другой стороны, в Smalltalk типов нет: во время исполнения любое сообщение можно послать любому объекту, и если класс объекта (или его надкласс) не понимает сообщение, то генерируется сообщение об ошибке. Нарушение согласования типов может не обнаружиться во время трансляции и обычно проявляется как ошибка исполнения. C++ тяготеет к сильной типизации, но в этом языке правила типизации можно игнорировать или подавить полностью.  Рассмотрим абстракцию различных типов емкостей, которые могут использоваться в нашей теплице. Вероятно, в ней есть емкости для воды и для минеральных удобрений; хотя первые предназначены для жидкостей, а вторые для сыпучих веществ, они имеют достаточно много общего, чтобы устроить иерархию классов. Начнем с типов.  **// Число, обозначающее уровень от 0 до 100 процентов**  **typedef float Level;**  Операторы typedef в C++ не вводят новых типов. В частности, и Level и Concentration - на самом деле другие названия для float, и их можно свободно смешивать в вычислениях. В этом смысле C++ имеет слабую типизацию: значения примитивных типов, таких, как int или float неразличимы в пределах данного типа. Напротив, Ada и Object Pascal предоставляют сильную типизацию для примитивных типов. В Ada можно объявить самостоятельным типом интервал значений или подмножество с ограниченной точностью.   http://www.helloworld.ru/texts/comp/other/oop/pg078.gif  *Строгая типизация предотвращает смешивание абстракций.*  Построим теперь иерархию классов для емкостей:  **class StorageTank {  public:**  **StorageTank();  virtual ~StorageTank();  virtual void fill();  virtual void startDraining();  virtual void stopDraining();  Boolean isEmpty() const;  Level level() const;**  **protected:  ...  };**  **class WaterTank : public StorageTank{  public:**  **WaterTank();  virtual ~WaterTank();  virtual void fill();  virtual void startDraining();  virtual void stopDraining();  void startHeating();  void stopHeating();  Temperature currentTemperature() const;**  **protected:  ...  };**  **class NutrientTank : public StorageTank {  public:**  **NutrientTank();  virtual ~NutrientTank();  virtual void startDrainingt();  virtual void stopDraining();**  **protected:  ...  };**  Класс **StorageTank** - это базовый класс иерархии. Он обеспечивает структуру и поведение общие для всех емкостей: возможность их наполнять или опустошать. Классы **WaterTank** (емкость для воды) и **NutrientTank** (для удобрений) наследуют свойства **StorageTank**, частично переопределяют их и добавляют кое-что свое: например, класс**WaterTank** вводит новое поведение, связанное с температурой.  Предположим, что мы имеем следующие описания:  **StorageTank s1, s2;  WaterTank w;  NutrientTank n;**  Заметьте, переменные такие как s1, s2, w или n - это не экземпляры соответствующих классов. На самом деле, это просто имена, которыми мы обозначаем объекты соответствующих классов: когда мы говорим "объект s1" мы на самом деле имеем ввиду экземпляр **StorageTank**, обозначаемый переменной s1. Мы вернемся к этому тонкому вопросу в следующей главе.  При проверке типов у классов, C++ типизирован гораздо строже. Под этим понимается, что выражения, содержащие вызовы операций, проверяются на согласование типов во время компиляции. Например, следующее правильно:  **Level l = s1.level();  w.startDrainingt();  n.stopDraining();**  Действительно, такие селекторы есть в классах, к которым принадлежат соответствующие переменные. Напротив, следующее неправильно и вызовет ошибку компиляции:  **s1.startHeating(); // Неправильно  n.stopHeating(); // Неправильно**  Таких функций нет ни в самих классах, ни в их суперклассах. Но следующее  **n.fill();**  совершенно правильно: функции fill нет в определении **NutrientTank**, но она есть в вышестоящем классе.  Итак, сильная типизация заставляет нас соблюдать правила использования абстракций, поэтому она тем полезнее, чем больше проект. Однако у нее есть и теневая сторона. А именно, даже небольшие изменения в интерфейсе класса требуют перекомпиляции всех его подклассов. Кроме того, не имея параметризованных классов, о которых речь пойдет в главах 3 и 9, трудно представить себе, как можно было бы создать собрание разнородных объектов. Предположим, что мы хотим ввести абстракцию инвентарного списка, в котором собирается все имущество, связанное с теплицей. Обычная для С идиома применима и в C++: нужно использовать класс-контейнер, содержащий указатели на void, то есть на объекты произвольного типа.  **class Inventory {  public:**  **Inventory();  ~Inventory();  void add(void\*);  void remove(void\*);  void\* mostRecent() const;  void apply(Boolean (\*)(void\*));**  **private:  ...  };**  Операция apply - это так называемый итератор, который позволяет применить какую-либо операцию ко всем объектам в списке. Подробнее об итераторах см. в следующей главе.  Имея экземпляр класса Inventory, мы можем добавлять и уничтожать указатели на объекты любых классов. Но эти действия не безопасны с точки зрения типов - в списке могут оказаться как осязаемые объекты (емкости), так и неосязаемые (температура или план выращивания), что нарушает нашу абстракцию материального учета. Более того, мы могли бы внести в список объекты классов **WaterTank** и **TemperatureSensor**, и по неосторожности ожидая от функции mostRecent объекта класса **WaterTank** получить **StorageTank**.  Вообще говоря, у этой проблемы есть два общих решения. Во-первых, можно сделать контейнерный класс, безопасный с точки зрения типов. Чтобы не манипулировать с нетипизированными указателями void, мы могли бы определить инвентаризационный класс, который манипулирует только с объектами класса **TangibleAsset** (осязаемого имущества), а этот класс будет подмешиваться ко всем классам, такое имущество представляющим, например, к **WaterTank**, но не к**GrowingPlan**. Тем самым можно отсечь проблему первого рода, когда неправомочно смешиваются объекты разных типов. Во-вторых, можно ввести проверку типов в ходе выполнения, для того, чтобы знать, с объектом какого типа мы имеем дело в данный момент. Например, в Smalltalk можно запрашивать у объектов их класс. В C++ такая возможность не входила в стандарт до недавнего времени, хотя на практике, конечно, можно ввести в базовый класс операцию, возвращающую код класса (строку или значение перечислимого типа). Однако для этого надо иметь очень серьезные причины, поскольку проверка типа в ходе выполнения ослабляет инкапсуляцию. Как будет показано в следующем разделе, необходимость проверки типа можно смягчить, используя полиморфные операции.  В языках с сильной типизацией гарантируется, что все выражения будут согласованы по типу. Что это значит, лучше пояснить на примере. Следующие присваивания допустимы:  **s1 = s2;  s1 = w;**  Первое присваивание допустимо, поскольку переменные имеют один и тот же класс, а второе - поскольку присваивание идет снизу вверх по типам. Однако во втором случае происходит потеря информации (известная в C++ как "проблема срезки"), так как класс переменной w, **WaterTank**, семантически богаче, чем класс переменной s1, то есть**StorageTank**.  Следующие присваивания неправильны:  **w = s1; // Неправильно**  **w = n; // Неправильно**  В первом случае неправильность в том, что присваивание идет сверху вниз по иерархии, а во втором классы даже не находятся в состоянии подчиненности.  Иногда необходимо преобразовать типы. Например, посмотрите на следующую функцию:  **void checkLevel(const StorageTank& s);**  Мы можем привести значение вышестоящего класса к подклассу в том и только в том случае, если фактическим параметром при вызове оказался объект класса **WaterTank**. Или вот еще случай:  **if (((WaterTank&)s).currentTemperature() < 32.0) ...**  Это выражение согласовано по типам, но не безопасно. Если при выполнении программы вдруг окажется, что переменная s обозначала объект класса **NutrientTank**, приведение типа даст непредсказуемый результат во время исполнения. Вообще говоря, преобразований типа надо избегать, поскольку они часто представляют собой нарушение принятой системы абстракций.  Теслер отметил следующие важные преимущества строго типизированных языков:  "Отсутствие контроля типов может приводить к загадочным сбоям в программах во время их выполнения.  В большинстве систем процесс редактирование-компиляция-отладка утомителен, и раннее обнаружение ошибок просто незаменимо.  Объявление типов улучшает документирование программ.  Многие компиляторы генерируют более эффективный объектный код, если им явно известны типы" [[72](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.72)].  Языки, в которых типизация отсутствует, обладают большей гибкостью, но даже в таких языках, по мнению Борнинга и Ингалса: "Программисты обычно знают, какие объекты ожидаются в качестве аргументов и какие будут возвращаться" [[73](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.73)]. На практике, особенно при программировании "в большом", надежность языков со строгой типизацией с лихвой компенсирует некоторую потерю в гибкости по сравнению с нетипизированными языками.  **Примеры типизации: статическое и динамическое связывание.** Сильная и статическая типизация - разные вещи. Строгая типизация следит за соответствием типов, а статическая типизация (иначе называемая *статическим* или*ранним связыванием)* определяет время, когда имена связываются с типами. Статическая связь означает, что типы всех переменных и выражений известны во время компиляции; *динамическое связывание* (называемое также *поздним связыванием)* означает, что типы неизвестны до момента выполнения программы. Концепции типизации и связывания являются независимыми, поэтому в языке программирования может быть: типизация - сильная, связывание - статическое (Ada), типизация - сильная, связывание - динамическое (C++, Object Pascal), или и типов нет, и связывание динамическое (Smalltalk). Язык CLOS занимает промежуточное положение между C++ и Smalltalk: определения типов, сделанные программистом, могут быть либо приняты во внимание, либо не приняты.  Прокомментируем это понятие снова примером на C++. Вот "свободная", то есть не входящая в определение какого-либо класса, функция [Свободная функция - функция, не входящая ни в какой класс. В чисто объектно-ориентированных языках, типа Smalltalk, свободных процедур не бывает, каждая операция связана с каким-нибудь классом]:  **void balanceLevels(StorageTank& s1, StorageTank& s2);**  Вызов этой функции с экземплярами класса **StorageTank** или любых его подклассов в качестве параметров будет согласован по типам, поскольку тип каждого фактического параметра происходит в иерархии наследования от базового класса **StorageTank**.  При реализации этой функции мы можем иметь что-нибудь вроде:  **if (s1.level()> s2.level()) s2.fill();**  В чем особенность семантики при использовании селектора level? Он определен только в классе **StorageTank**, поэтому, независимо от классов объектов, обозначаемых переменными в момент выполнения, будет использована одна и та же унаследованная ими функция. Вызов этой функции статически связан при компиляции - мы точно знаем, какая операция будет запущена.  Иное дело fill. Этот селектор определен в **StorageTank** и переопределен в **WaterTank**, поэтому его придется связывать динамически. Если при выполнении переменная s2 будет класса **WaterTank**, то функция будет взята из этого класса, а если - **NutrientTank**, то из **StorageTank**. В C++ есть специальный синтаксис для явного указания источника; в нашем примере вызов fill будет разрешен, соответственно, как **WaterTank::fill** или **StorageTank::fill**[Так синтаксис C++ определяет явную квалификацию имени].  Это особенность называется *полиморфизмом:* одно и то же имя может означать объекты разных типов, но, имея общего предка, все они имеют и общее подмножество операций, которые можно над ними выполнять [[74](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.74)]. Противоположность полиморфизму называется *мономорфизмом;* он характерен для языков с сильной типизацией и статическим связыванием (Ada).  Полиморфизм возникает там, где взаимодействуют наследование и динамическое связывание. Это одно из самых привлекательных свойств объектно-ориентированных языков (после поддержки абстракции), отличающее их от традиционных языков с абстрактными типами данных. И, как мы увидим в следующих главах, полиморфизм играет очень важную роль в объектно-ориентированном проектировании.  **Параллелизм**  **Что такое 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, параллельная система имеет много таких потоков: век одних недолог, а другие живут в течении всего сеанса работы системы. Реальная параллельность достигается только на многопроцессорных системах, а системы с одним процессором имитируют параллельность за счет алгоритмов разделения времени.  Кроме этого "аппаратного" различия, мы будем различать "тяжелую" и "легкую" параллельность по потребности в ресурсах. "Тяжелые" процессы управляются операционной системой независимо от других, и под них выделяется отдельное защищенное адресное пространство. "Легкие" сосуществуют в одном адресном пространстве. "Тяжелые" процессы общаются друг с другом через операционную систему, что обычно медленно и накладно. Связь "легких" процессов осуществляется гораздо проще, часто они используют одни и те же данные.  Многие современные операционные системы предусматривают прямую поддержку параллелизма, и это обстоятельство очень благоприятно сказывается на возможности обеспечения параллелизма в объектно-ориентированных системах. Например, системы UNIX предусматривают системный вызов *fork,* который порождает новый процесс. Системы Windows NT и OS/2 - многопоточные; кроме того они обеспечивают программные интерфейсы для создания процессов и манипулирования с ними.  Лим и Джонсон отмечают, что "возможности проектирования параллельности в объектно-ориентированных языках не сильно отличаются от любых других, - на нижних уровнях абстракции параллелизм и OOP развиваются совершенно независимо. С OOP или без, все традиционные проблемы параллельного программирования сохраняются" [[75](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.75)]. Действительно, создавать большие программы и так непросто, а если они еще и параллельные, то надо думать о возможном простое одного из потоков, неполучении данных, взаимной блокировке и т.д.  К счастью, как отмечают те же авторы далее: "на верхних уровнях OOP упрощает параллельное программирование для рядовых разработчиков, пряча его в повторно-используемые абстракции" [[76](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.76)]. Блэк и др. сделали следующий вывод: "объектная модель хороша для распределенных систем, поскольку она неявно разбивает программу на (1) распределенные единицы и (2) сообщающиеся субъекты" [[77](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.77)].  В то время, как объектно-ориентированное программирование основано на абстракции, инкапсуляции и наследовании, параллелизм главное внимание уделяет абстрагированию и синхронизации процессов [[78](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.78)]. Объект есть понятие, на котором эти две точки зрения сходятся: каждый объект (полученный из абстракции реального мира) может представлять собой отдельный поток управления (абстракцию процесса). Такой объект называется *активным.* Для систем, построенных на основе OOD, 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:   http://www.helloworld.ru/texts/comp/other/oop/pg084.gif  *Параллелизм позволяет различным объектам действовать одновременно.*  *Параллелизм - это свойство, отличающее активные объекты от пассивных.*  **Примеры параллелизма.** Ранее мы обзавелись классом ActiveTemperatureSensor, поведение которого предписывает ему периодически измерять температуру и обращаться к известной ему функции вызова, когда температура отклоняется на некоторую величину от установленного значения. Как он будет это делать, мы в тот момент не объяснили. При всех секретах реализации понятно, что это - активный объект и, следовательно, без параллелизма тут не обойтись. В объектно-ориентированном проектировании есть три подхода к параллелизму.  Во-первых, параллелизм - это внутреннее свойство некоторых языков программирования. Так, для языка Ada механизм параллельных процессов реализуется как *задача.* В Smalltalk есть класс process, которому наследуют все активные объекты. Есть много других языков со встроенными механизмами для параллельного выполнения и синхронизации процессов - Actors, Orient 84/K, ABCL/1, которые предусматривают сходные механизмы параллелизма и синхронизации. Во всех этих языках можно создавать активные объекты, код которых постоянно выполняется параллельно с другими активными объектами.  Во-вторых, можно использовать библиотеку классов, реализующих какую-нибудь разновидность "легкого" параллелизма. Например, библиотека AT&T для C++ содержит классы Shed, Timer, Task и т.д. Ее реализация, естественно, зависит от платформы, хотя интерфейс достаточно хорошо переносим. При этом подходе механизмы параллельного выполнения не встраиваются в язык (и, значит, не влияют на системы без параллельности), но в то же время практически воспринимаются как встроенные.  Наконец, в-третьих, можно создать иллюзию многозадачности с помощью прерываний. Для этого надо кое-что знать об аппаратуре. Например, в нашей реализации класса **ActiveTemperatureSensor** мы могли бы иметь аппаратный таймер, периодически прерывающий приложение, после чего все датчики измеряли бы температуру и обращались бы, если нужно, к своим функциям вызова.  Как только в систему введен параллелизм, сразу возникает вопрос о том, как синхронизировать отношения активных объектов друг с другом, а также с остальными объектами, действующими последовательно. Например, если два объекта посылают сообщения третьему, должен быть какой-то механизм, гарантирующий, что объект, на который направлено действие, не разрушится при одновременной попытке двух активных объектов изменить его состояние. В этом вопросе соединяются абстракция, инкапсуляция и параллелизм. В параллельных системах недостаточно определить поведение объекта, надо еще принять меры, гарантирующие, что он не будет растерзан на части несколькими независимыми процессами.  **Сохраняемость**  Любой программный объект существует в памяти и живет во времени. Аткинсон и др. предположили, что есть непрерывное множество продолжительности существования объектов: существуют объекты, которые присутствуют лишь во время вычисления выражения, но есть и такие, как базы данных, которые существуют независимо от программы. Этот спектр сохраняемости объектов охватывает:  "Промежуточные результаты вычисления выражений.  Локальные переменные в вызове процедур.  Собственные переменные (как в ALGOL-60), 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" [[79](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.79)].  Традиционно, первыми тремя уровнями занимаются языки программирования, а последними - базы данных. Этот конфликт культур приводит к неожиданным решениям: программисты разрабатывают специальные схемы для сохранения объектов в период между запусками программы, а конструкторы баз данных переиначивают свою технологию под короткоживущие объекты [[80](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.80)].  Унификация принципов параллелизма для объектов позволила создать параллельные языки программирования. Аналогичным образом, введение сохраняемости, как нормальной составной части объектного подхода приводит нас к объектно-ориентированным базам данных (OODB, object-oriented databases). На практике подобные базы данных строятся на основе проверенных временем моделей - последовательных, индексированных, иерархических, сетевых или реляционных, но программист может ввести абстракцию объектно-ориентированного интерфейса, через который запросы к базе данных и другие операции выполняются в терминах объектов, время жизни которых превосходит время жизни отдельной программы. Как мы увидим в главе 10, эта унификация значительно упрощает разработку отдельных видов приложений, позволяя, в частности, применить единый подход к разным сегментам программы, одни из которых связаны с базами данных, а другие не имеют такой связи.  Языки программирования, как правило, не поддерживают понятия сохраняемости; примечательным исключением является Smalltalk, в котором есть протоколы для сохранения объектов на диске и загрузки с диска. Однако, записывать объекты в неструктурированные файлы - это все-таки наивный подход, пригодный только для небольших систем. Как правило, сохраняемость достигается применением (немногочисленных) коммерческих OODB [[81](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.81)]. Другой вариант - создать объектно-ориентированную оболочку для реляционных СУБД; это лучше, в частности, для тех, кто уже вложил средства в реляционную систему. Мы рассмотрим такую ситуацию в главе 10.  Сохраняемость - это не только проблема сохранения *данных.* В OODB имеет смысл сохранять и *классы,* так, чтобы программы могли правильно интерпретировать данные. Это создает большие трудности по мере увеличения объема данных, особенно, если класс объекта вдруг потребовалось изменить.  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 для распределенных систем желательно обеспечивать возможность перенесения объектов в пространстве, так, чтобы их можно было переносить с машины на машину и даже при необходимости изменять форму представления объекта в памяти. Этими вопросами мы займемся в главе 12.  В заключение определим сохраняемость следующим образом:  *Сохраняемость - способность объекта существовать во времени, переживая породивший его процесс, и (или) в пространстве, перемещаясь из своего первоначального адресного пространства.*  2.3. Применение объектной модели  **Преимущества объектной модели**  Как уже говорилось выше, объектная модель принципиально отличается от моделей, которые связаны с более традиционными методами структурного анализа, проектирования и программирования. Это не означает, что объектная модель требует отказа от всех ранее найденных и испытанных временем методов и приемов. Скорее, она вносит некоторые новые элементы, которые добавляются к предшествующему опыту. Объектный подход обеспечивает ряд существенных удобств, которые другими моделями не предусматривались. Наиболее важно, что объектный подход позволяет создавать системы, которые удовлетворяют пяти признакам хорошо структурированных сложных систем. Согласно нашему опыту, есть еще пять преимуществ, которые дает объектная модель.  Во-первых, объектная модель позволяет в полной мере использовать выразительные возможности объектных и объектно-ориентированных языков программирования. Страуструп отмечает: "Не всегда очевидно, как в полной мере использовать преимущества такого языка, как C++. Существенно повысить эффективность и качество кода можно просто за счет использования C++ в качестве "улучшенного C" с элементами абстракции данных. Однако гораздо более значительным достижением является введение иерархии классов в процессе проектирования. Именно это называется OOD и именно здесь преимущества C++ демонстрируются наилучшим образом" [[82](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.82)]. Опыт показал, что при использовании таких языков, как Smalltalk, Object Pascal, C++, CLOS и Ada вне объектной модели, их наиболее сильные стороны либо игнорируются, либо применяются неправильно.   http://www.helloworld.ru/texts/comp/other/oop/pg087.gif  *Сохраняемость поддерживает состояние и класс объекта в пространстве и во времени.*  Во-вторых, использование объектного подхода существенно повышает уровень унификации разработки и пригодность для повторного использования не только программ, но и проектов, что в конце концов ведет к созданию среды разработки [[83](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.83)]. Объектно-ориентированные системы часто получаются более компактными, чем их не объектно-ориентированные эквиваленты. А это означает не только уменьшение объема кода программ, но и удешевление проекта за счет использования предыдущих разработок, что дает выигрыш в стоимости и времени.  В-третьих, использование объектной модели приводит к построению систем на основе стабильных промежуточных описаний, что упрощает процесс внесения изменений. Это дает системе возможность развиваться постепенно и не приводит к полной ее переработке даже в случае существенных изменений исходных требований.  В-четвертых, в главе 7 показано, как объектная модель уменьшает риск разработки сложных систем, прежде всего потому, что процесс интеграции растягивается на все время разработки, а не превращается в единовременное событие. Объектный подход состоит из ряда хорошо продуманных этапов проектирования, что также уменьшает степень риска и повышает уверенность в правильности принимаемых решений.  Наконец, объектная модель ориентирована на человеческое восприятие мира, или, по словам Робсона, "многие люди, не имеющие понятия о том, как работает компьютер, находят вполне естественным объектно-ориентированный подход к системам" [[84](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#2.84)].  **Использование объектного подхода**  Возможность применения объектного подхода доказана для задач самого разного характера. На рис. 2-6 приведен перечень областей, для которых реализованы объектно-ориентированные системы. Более подробные сведения об этих и других проектах можно найти в приведенной библиографии.  В настоящее время объектно-ориентированное проектирование - единственная методология, позволяющая справиться со сложностью, присущей очень большим системам. Однако, следует заметить, что иногда применение OOD может оказаться нецелесообразным, например, из-за неподготовленности персонала или отсутствия подходящих средств разработки. К этой теме мы вернемся в главе 7.  **Открытые вопросы**  Чтобы успешно использовать объектный подход, нам предстоит ответить на следующие вопросы:  Что такое классы и объекты?  Как идентифицировать классы и объекты в конкретных приложениях?  Как описать схему объектно-ориентированной системы?  Как создавать хорошо структурированные объектно-ориентированные системы?  Как организовать управление процессом разработки согласно OOD? Этим вопросам посвящены следующие пять глав.  Выводы  Развитие программной индустрии привело к созданию методов объектно-ориентированного анализа, проектирования и программирования, которые служат для программирования "в большом".  В программировании существует несколько парадигм, ориентированных на процедуры, объекты, логику, правила и ограничения.  Абстракция определяет существенные характеристики некоторого объекта, которые отличают его от всех других видов объектов и, таким образом, абстракция четко очерчивает концептуальную границу объекта с точки зрения наблюдателя.  Инкапсуляция - это процесс разделения устройства и поведения объекта; инкапсуляция служит для того, чтобы изолировать контрактные обязательства абстракции от их реализации.   |  |  | | --- | --- | | Авиационное оборудование | Обработка коммерческой информации | | Автоматизация учреждений | Операционные системы | | Автоматизированное проектирование | Планирование инвестиций | | Автоматизированное обучение | Повторно используемые компоненты | | Автоматизированное производство программного обеспечения | Подготовка документов | | Анимация | Программные средства космических станций | | Базы данных | Проектирование интерфейса пользователя | | Банковское дело | Проектирование СБИС | | Гипермедиа | Распознавание образов | | Кинопроизводство | Робототехника | | Контроль программного обеспечения | Системы телеметрии | | Математический анализ | Системы управления и регулирования | | Медицинская электроника | Средства разработки программ | | Моделирование авиационной и космической техники | Телекоммуникации | | Музыкальная композиция | Управление воздушным движением | | Написание сценариев | Управление химическими процессами | | Нефтяная промышленность | Экспертные системы |   *Рис. 2-6. Применения объектной модели.*  *Модульность - это состояние системы, разложенной на внутренне связные и слабо связанные между собой модули.*  *Иерархия - это ранжирование или упорядочение абстракций.*  *Типизация - это способ защититься от использования объектов одного класса вместо другого, или по крайней мере способ управлять такой подменой.*  *Параллелизм - это свойство, отличающее активные объекты от пассивных.*  *Сохраняемость - способность объекта существовать во времени и (или) в пространстве.* |

Классы и объекты

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| И инженер, и художник должны хорошо чувствовать материал, с которым они работают. В объектно-ориентированной методологии анализа и создания сложных программных систем основными строительными блоками являются классы и объекты. Выше было дано всего лишь неформальное определение этих двух элементов. В этой главе мы рассмотрим природу классов и объектов, взаимоотношения между ними, а также сообщим несколько полезных правил проектирования хороших абстракций.  3.1. Природа объекта  **Что является и что не является объектом?**  Способностью к распознанию объектов физического мира человек обладает с самого раннего возраста. Ярко окрашенный мяч привлекает внимание младенца, но, если спрятать мяч, младенец, как правило, не пытается его искать: как только предмет покидает поле зрения, он перестает существовать для младенца. Только в возрасте около года у ребенка появляется представление о предмете: навык, который незаменим для распознавания. Покажите мяч годовалому ребенку и спрячьте его: скорее всего, ребенок начнет искать спрятанный предмет. Ребенок связывает понятие предмета с постоянством и индивидуальностью формы независимо от действий, выполняемых над этим предметом [[1](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#3.1)].  В предыдущей главе объект был неформально определен как осязаемая реальность, проявляющая четко выделяемое поведение. С точки зрения восприятия человеком объектом может быть:  осязаемый и (или) видимый предмет;  нечто, воспринимаемое мышлением;  нечто, на что направлена мысль или действие.  Таким образом, мы расширили неформальное определение объекта новой идеей: объект моделирует часть окружающей действительности и таким образом существует во времени и пространстве. Термин *объект* в программном обеспечении впервые был введен в языке Simula и применялся для моделирования реальности [[2](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#3.2)].  Объектами реального мира не исчерпываются типы объектов, интересные при проектировании программных систем. Другие важные типы объектов вводятся на этапе проектирования, и их взаимодействие друг с другом служит механизмом отображения поведения более высокого уровня [[3](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#3.3)]. Это приводит нас к более четкому определению, данному Смитом и Токи: "Объект представляет собой конкретный опознаваемый предмет, единицу или сущность (реальную или абстрактную), имеющую четко определенное функциональное назначение в данной предметной области" [[4](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#3.4)]. В еще более общем плане объект может быть определен как нечто, имеющее четко очерченные границы [[5](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#3.5)].  Представим себе завод, на котором создаются композитные материалы для таких различных изделий как, скажем, велосипедные рамы и крылья самолетов. Заводы часто разделяются на цеха: механический, химический, электрический и т.д. Цеха подразделяются на участки, на каждом из которых установлено несколько единиц оборудования: штампы, прессы, станки. На производственных линиях можно увидеть множество емкостей с исходными материалами, из которых с помощью химических процессов создаются блоки композитных материалов. Затем из них делается конечный продукт - рамы или крылья. Каждый осязаемый предмет может рассматриваться как объект. Токарный станок имеет четко очерченные границы, которые отделяют его от обрабатываемого на этом станке композитного блока; рама велосипеда в свою очередь имеет четкие границы по отношению к участку с оборудованием.  Существуют такие объекты, для которых определены явные концептуальные границы, но сами объекты представляют собой неосязаемые события или процессы. Например, химический процесс на заводе можно трактовать как объект, так как он имеет четкую концептуальную границу, взаимодействует с другими объектами посредством упорядоченного и распределенного во времени набора операций и проявляет хорошо определенное поведение. Рассмотрим систему пространственного проектирования CAD/CAM. Два тела, например, сфера и куб, имеют как правило нерегулярное пересечение. Хотя эта линия пересечения не существует отдельно от сферы и куба, она все же является самостоятельным объектом с четко определенными концептуальными границами.  Объекты могут быть осязаемыми, но иметь размытые физические границы: реки, туман или толпы людей [Это верно только на достаточно высоком уровне абстракции. Для человека, идущего через полосу тумана, бессмысленно отличать "мой туман" от "твоего тумана". Однако, рассмотрим карту погоды: полосы тумана в Сан-Франциско и в Лондоне представляют собой совершенно разные объекты]. Подобно тому, как взявший в руки молоток начинает видеть во всем окружающем только гвозди, проектировщик с объектно-ориентированным мышлением начинает воспринимать весь мир в виде объектов. Разумеется, такой взгляд несколько упрощен, так как существуют понятия, явно не являющиеся объектами. К их числу относятся атрибуты, такие, как время, красота, цвет, эмоции (например, любовь или гнев). Однако, потенциально все перечисленное - это свойства, присущие объектам. Можно, например, утверждать, что некоторый человек (объект) любит свою жену (другой объект), или что конкретный кот (еще один объект) - серый.   http://www.helloworld.ru/texts/comp/other/oop/pg093.gif  *Объект имеет состояние, обладает некоторым хорошо определенным поведением и уникальной идентичностью.*  Полезно понимать, что объект - это нечто, имеющее четко определенные границы, но этого недостаточно, чтобы отделить один объект от другого или дать оценку качества абстракции. На основе имеющегося опыта можно дать следующее определение:  *Объект обладает состоянием, поведением и идентичностью; структура и поведение схожих объектов определяет общий для них класс; термины "экземпляр класса" и "объект" взаимозаменяемы.*  **Состояние**  **Семантика.** Рассмотрим торговый автомат, продающий напитки. Поведение такого объекта состоит в том, что после опускания в него монеты и нажатия кнопки автомат выдает выбранный напиток. Что произойдет, если сначала будет нажата кнопка выбора напитка, а потом уже опущена монета? Большинство автоматов при этом просто ничего не сделают, так как пользователь нарушил их основные правила.  Другими словами, автомат играл роль (ожидание монеты), которую пользователь игнорировал, нажав сначала кнопку. Или предположим, 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 мы видим, что 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 низкоуровневое определение:  *Состояние объекта характеризуется перечнем (обычно статическим) всех свойств данного объекта и текущими (обычно динамическими) значениями каждого из этих свойств.*  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 концептуальная 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 (их можно пить, а автомат нет, и совершать с ними иные действия).  Таким образом, мы установили различие между объектами и простыми величинами: простые количественные характеристики (например, число 3) являются "постоянными, неизменными и непреходящими", тогда как объекты "существуют во времени, изменяются, имеют внутреннее состояние, преходящи и могут создаваться, уничтожаться и разделяться" [[6](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#3.6)].  Тот факт, что всякий объект имеет состояние, означает, что всякий объект занимает определенное пространство (физически или в памяти компьютера).  **Примеры.** Предположим, что на языке C++ нам нужно создать регистрационные записи о сотрудниках. Можно сделать это следующим образом:  **struct PersonnelRecord {**  **char name[100];  int socialSecurityNumber;  char department[10];  float salary;**  **};**  Каждый компонент в приведенной структуре обозначает конкретное свойство нашей абстракции регистрационной записи. Описание определяет не объект, а класс, поскольку оно не вводит какой-либо конкретный экземпляр [Точнее, это описание определяет структуру в C++, семантика которой соответствует классу, у которого все поля открыты. Таким образом, структуры - это неинкапсулированные абстракции]. Для того чтобы создать объекты данного класса, необходимо написать следующее:  **PersonnelRecord deb, dave, karen, jim, torn, denise, kaitlyn, krista, elyse;**  В данном случае объявлено девять различных объектов, каждый из которых занимает определенный участок в памяти. Хотя свойства этих объектов являются общими (их состояние представляется единообразно), в памяти объекты не пересекаются и занимают каждый свое место. На практике принято ограничивать доступ к состоянию объекта, а не делать его общедоступным, как в предыдущем определении класса. С учетом сказанного, изменим данное определение следующим образом:  **class PersonnelRecord {  public:**  **char\* employeeName() const;  int employeeSocialSecurityNumber() const;  char\* employeeDepartment() const;**  **protected:**  **char name[100];  int socialSecurityNumber;  char department[10];  float salary;**  **};**  Новое определение несколько сложнее предыдущего, но по ряду соображений предпочтительнее [К вопросу о стилях: по критериям, которые вводятся в этой главе далее, предложенное определение класса PersonnelRecord - это далеко не шедевр. Мы хотим здесь только показать семантику состояния класса. Иметь в классе функцию, которая возвращает значение char\*, часто опасно, так как это нарушает парадигму защиты памяти: если метод отводит себе память, за которую получивший к ней доступ клиент не отвечает, результатом будет замусоривание памяти. В наших системах мы предпочитаем использовать параметризованный класс строк переменной длины, который можно найти в базовой библиотеке классов, вроде той, что описана в главе 9. И еще: классы - это больше чем структуры из С с синтаксисом классов C++; как объясняется в главе 4, классификация требует определенного согласования структуры и поведения]. В частности, в новом определении реализация класса скрыта от других объектов. Если реализация класса будет в дальнейшем изменена, код придется перекомпилировать, но семантически клиенты не будут зависеть от этих изменении (то есть их код сохранится). Кроме того, решается также проблема занимаемой объектом памяти за счет явного определения операций, которые разрешены клиентам над объектами данного класса. В частности, мы даем всем клиентам право узнать имя, код социальной защиты и место работы сотрудника, но только особым клиентам (а именно, подклассам данного класса) разрешено устанавливать значения указанных параметров. Только этим специальным клиентам разрешен доступ к сведениям о заработной плате. Другое достоинство последнего определения связано с возможностью его повторного использования. В следующем разделе мы увидим, что механизм наследования позволяет повторно использовать абстракцию, а затем уточнить и многими способами специализировать ее.  В заключение скажем, что все объекты в системе инкапсулируют некоторое состояние, и все состояние системы инкапсулировано в объекты. Однако, инкапсуляция состояния объекта - это только начало, которого недостаточно, чтобы мы могли охватить полный смысл абстракций, которые мы вводим при разработке. По этой причине нам нужно разобраться, как объекты функционируют.  **Поведение**  **Что такое 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.*  Иными словами, поведение объекта - это его наблюдаемая и проверяемая извне деятельность.  Операцией называется определенное воздействие одного объекта на другой с целью вызвать соответствующую реакцию. Например, клиент может активизировать операции append и pop для того, чтобы управлять объектом-очередью (добавить или изъять элемент). Существует также операция length, которая позволяет определить размер очереди, но не может изменить это значение. В чисто объектно-ориентированном языке, таком как Smalltalk, принято говорить о передаче сообщений между объектами. В языках типа C++, в которых четче ощущается процедурное прошлое, мы говорим, что один объект вызывает функцию-член другого. В основном понятие *сообщение* совпадает с понятием *операции* над объектами, хотя механизм передачи различен. Для наших целей эти два термина могут использоваться как синонимы.  В объектно-ориентированных языках операции, выполняемые над данным объектом, называются *методами* и входят в определение класса объекта. В C++ они называются *функциями-членами*. Мы будем использовать эти термины как синонимы.  Передача сообщений - это одна часть уравнения, задающего 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). Итак, поведение объекта определяется выполняемыми над ним операциями и его состоянием, причем некоторые операции имеют побочное действие: они изменяют состояние. Концепция побочного действия позволяет уточнить наше определение состояния:  *Состояние объекта представляет суммарный результат его поведения.*  Наиболее интересны те объекты, состояние которых не статично: их состояние изменяется и запрашивается операциями.  Примеры. Опишем на языке C++ класс Queue (очередь):  **class Queue {  public:**  **Queue();  Queue(const Queue&);  virtual ~Queue();  virtual Queue& operator=(const Queue&);  virtual int operator==(const Queue&) const;  int operator!=(const Queue&) const;  virtual void clear();  virtual void append(const void\*);  virtual void pop();  virtual void remove(int at);  virtual int length() const;  virtual int isEmpty() const;  virtual const void\* front() const;  virtual int location(const void\*);**  **protected:  ...  };**  В определении класса используется обычная для С идиома ссылки на данные неопределенного типа с помощью void\*, благодаря чему в очередь можно вставлять объекты разных классов. Эта техника не безопасна - клиент должен ясно понимать, с каким (какого класса) объектом он имеет дело. Кроме того, при использовании void\* очередь не "владеет" объектами, которые в нее помещены. Деструктор ~Queue() уничтожает очередь, но не ее участников. В следующем разделе мы рассмотрим параметризованные типы, которые помогают справляться с такими проблемами.  Так как определение Queue задает класс, а не объект, мы должны объявить экземпляры класса, с которыми могут работать клиенты:  **Queue a, b, c, d;**  Мы можем выполнять операции над объектами:  **a.append(&deb);  a.append(&karen);  a.append (&denise);  b = a;  a.pop();**  Теперь очередь **а** содержит двух сотрудников (первой стоит karen), а очередь **b** - троих (первой стоит deb). Таким образом, очереди имеют определенное состояние, которое влияет на их будущее поведение - например, одну очередь можно безопасно продвинуть (pop) еще два раза, а вторую - три.  **Операции.** Операция - это услуга, которую класс может предоставить своим клиентам. На практике типичный клиент совершает над объектами операции пяти видов [Липпман предложил несколько иную классификацию: функции управления, функции реализации, вспомогательные функции (все виды модификаторов) и функции доступа (эквивалентные селекторам) [[7](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#3.7)]]. Ниже приведены три наиболее распространенные операции:    |  |  | | --- | --- | | ® Модификатор | Операция, которая изменяет состояние объекта | | ® Селектор | Операция, считывающая состояние объекта, но не меняющая состояния | | ® Итератор | Операция, позволяющая организовать доступ ко всем частям объекта в строго определенной последовательности |   Поскольку логика этих операций весьма различна, полезно выбрать такой стиль программирования, который учитывает эти различия в коде программы. В нашей спецификации класса **Queue** мы вначале перечислили все модификаторы (функции-члены без спецификаторов const - clear, append, pop, remove), а потом все селекторы (функции со спецификаторами const - length, isEmpty, front и location). Позднее в главе 9, следуя нашему стилю, мы определим отдельный класс, который действует как агент, отвечающий за итеративный просмотр очередей.  Две операции являются универсальными; они обеспечивают инфраструктуру, необходимую для создания и уничтожения экземпляров класса:    |  |  | | --- | --- | | ® Конструктор | Операция создания объекта и/или его инициализации | | ® Деструктор | Операция, освобождающая состояние объекта и/или разрушающая сам объект |   В языке C++ конструктор и деструктор составляют часть описания класса, тогда как в Smalltalk и CLOS эти операторы определены в протоколе метакласса (то есть класса класса).  В чисто объектно-ориентированных языках, таких как Smalltalk, операции могут быть только методами, так как процедуры и функции вне классов в этом языке определять не допускается. Напротив, в языках Object Pascal, C++, CLOS и Ada допускается описывать операции как независимые от объектов подпрограммы. В C++ они называются функциями-нечленами; мы же будем здесь называть их свободными подпрограммами. Свободные подпрограммы - это процедуры и функции, которые выполняют роль операций высокого уровня над объектом или объектами одного или разных классов. Свободные процедуры группируются в соответствии с классами, для которых они создаются. Это дает основание называть такие пакеты процедур утилитами класса. Например, для определенного выше класса **Queue**можно написать следующую свободную процедуру:  **void copyUntilFound(Queue& from, Queue& to, void\* item)  {**  **while ((!from.isEmpty()) && (from.front() != item))  {**  **to.append(from.front());  from.pop();**  **}**  **}**  Смысл в том, что содержимое одной очереди переходит в другую до тех пор, пока в голове первой очереди не окажется заданный объект. Это операция высокого уровня; она строится на операциях-примитивах класса Queue.  В C++ (и Smalltalk) принято собирать все логически связанные свободные подпрограммы и объявлять их частью некоторого класса, не имеющего состояния. Все такие функции будут статическими.  Таким образом, можно утверждать, что все методы - операции, но не все операции - методы: некоторые из них представляют собой свободные подпрограммы. Мы склонны использовать только методы, хотя, как будет показано в следующем разделе, иногда трудно удержаться от искушения, особенно если операция по своей природе выполняется над несколькими объектами разных классов и нет никаких причин объявить ее операцией именно одного класса, а не другого.  **Роли и ответственности.** Совокупность всех методов и свободных процедур, относящихся к конкретному объекту, образует протокол этого объекта. Протокол, таким образом, определяет поведение объекта, охватывающее все его статические и динамические аспекты. В самых нетривиальных абстракциях полезно подразделять протокол на частные аспекты поведения, которые мы будет называть ролями. Адамс говорит, что роль - это маска, которую носит объект [[8](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#3.8)]; она определяет контракт абстракции с ее клиентами.  Объединяя наши определения состояния и поведения объекта, Вирфс-Брок вводит понятие ответственности. "Ответственности объекта имеют две стороны - знания, которые объект поддерживает, и действия, которые объект может исполнить. Они выражают смысл его предназначения и место в системе. Ответственность понимается как совокупность всех услуг и всех контрактных обязательств объекта" [[9](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#3.9)]. Таким образом можно сказать, что состояние и поведение объекта определяют исполняемые им роли, а те, в свою очередь, необходимы для выполнения ответственности данной абстракции.  Действительно большинство интересных объектов исполняют в своей жизни разные роли, например [[10](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#3.10)]:  Банковский счет может быть в хорошем или плохом состоянии (две роли), и от этой роли зависит, что произойдет при попытке снятия с него денег.  Для фондового брокера пакет акций - это товар, который можно покупать или продавать, а для юриста это знак обладания определенными правами.  В течении дня одна и та же персона может играть роль матери, врача, садовника и кинокритика.  Роли банковского счета являются динамическими и взаимоисключающими. Роли пакета акций слегка перекрываются, но каждая из них зависит от того, что клиент с ними делает. В случае персоны роли динамически изменяются каждую минуту.  Как мы увидим в главах 4 и 6, мы часто начинаем наш анализ с перечисления разных ролей, которые может играть объект. Во время проектирования мы выделяем эти роли, вводя конкретные операции, выполняющие ответственности каждой роли.  **Объекты как автоматы.** Наличие внутреннего состояния объектов означает, что порядок выполнения операций имеет существенное значение. Это наводит на мысль представить объект в качестве маленькой независимой машины [[11](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#3.11)]. Действительно, для ряда объектов такой временной порядок настолько существен, что наилучшим способом их формального описания будет конечный автомат. В главе 5 мы введем обозначения для описания иерархических конечных автоматов, которые можно использовать для выражения соответствующей семантики.  Продолжая аналогию с машинами, можно сказать, что объекты могут быть активными и пассивными. Активный объект имеет свой поток управления, а пассивный - нет. Активный объект в общем случае автономен, то есть он может проявлять свое поведение без воздействия со стороны других объектов. Пассивный объект, напротив, может изменять свое состояние только под воздействием других объектов. Таким образом, активные объекты системы - источники управляющих воздействий. Если система имеет несколько потоков управления, то и активных объектов может быть несколько. В последовательных системах обычно в каждый момент времени существует только один активный объект, например, главное окно, диспетчер которого ловит и обрабатывает все сообщения. В таком случае остальные объекты пассивны: их поведение проявляется, когда к ним обращается активный объект. В других видах последовательных архитектур (системы обработки транзакций) нет явного центра активности, и управление распределено среди пассивных объектов системы.  **Идентичность**  **Семантика.** Хошафян и Коуплэнд предложили следующее определение:  *"Идентичность - это такое свойство объекта, которое отличает его от всех других объектов" [*[*12*](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#3.12)*].*  Они отмечают, что "в большинстве языков программирования и управления базами данных для различения временных объектов их именуют, тем самым путая адресуемость и идентичность. Большинство баз данных различают постоянные объекты по ключевому атрибуту, тем самым смешивая идентичность и значение данных". Источником множества ошибок в объектно-ориентированном программировании является неумение отличать имя объекта от самого объекта.  **Примеры.** Начнем с определения точки на плоскости.  **struct Point {**  **int x;  int y;  Point() : x(0), y(0) {}  Point(int xValue, int yValue) : x(xValue), y(yValue) {)**  **};**  Мы определили point как структуру, а не как полноценный класс. Правило, на основании которого мы так поступили, очень просто. Если абстракция представляет собой собрание других объектов без какого-либо собственного поведения, мы делаем ее структурой. Однако, когда наша абстракция подразумевает более сложное поведение, чем простой доступ к полям структуры, то нужно определять класс. В данном случае абстракция point - это просто пара координат (x, y). Для удобства предусмотрено два конструктора: один инициализирует точку нулевыми значениями координат, а другой - некоторыми заданными значениями.  Теперь определим экранный объект (DisplayItem). Это абстракция довольно обычна для систем с графическим интерфейсом (GUI) - она является базовым классом для всех объектов, которые можно отображать в окне. Мы хотим сделать его чем-то большим, чем просто совокупностью точек. Надо, чтобы клиенты могли рисовать, выбирать объекты и перемещать их по экрану, а также запрашивать их положение и состояние. Мы записываем нашу абстракцию в виде следующего объявления на C++:  **class DisplayItem { public:**  **DisplayItem();  DisplayItem(const Point& location);  virtual ~DisplayItem();  virtual void draw();  virtual void erase();  virtual void select();  virtual void unselect();  virtual void move(const Point& location);  int isSelected() const;  Point location() const;  int isUnder(const Point& location) const;**  **protected:  ...  };**  В этом объявлении мы намеренно опустили конструкторы, а также операторы для копирования, присваивания и проверки на равенство. Их мы оставим до следующего раздела.  Мы ожидаем, что у этого класса будет много наследников, поэтому деструктор и все модификаторы объявлены виртуальными. В особенности это относится к draw. Напротив, селекторы скорее всего не будут переопределяться в подклассах. Заметьте, что один из них, isUnder, должен вычислять, накрывает ли объект данную точку, а не просто возвращать значение какого-то свойства.  Объявим экземпляры указанных классов:  **DisplayItem item1;  DisplayItem\* item2 = new DisplayItem(Point(75, 75));  DisplayItem\* item3 = new DisplayItem(Point(100, 100));  DisplayItem\* item4 = 0;**  Рис. 3-1а показывает, что при выполнении этих операторов возникают четыре имени и три разных объекта. Конкретно, в памяти будут отведены четыре места под имена **item1**, **item2**, **item3**, **item4**. При этом **item1** будет именем объекта класса **DisplayItem**, а три других будут указателями. Кроме того, лишь **item2** и **item3** будут на самом деле указывать на объекты класса **DisplayItem**. У объектов, на которые указывают **item2** и **item3**, к тому же нет имен, хотя на них можно ссылаться "разыменовывая" соответствующие указатели: например, \***item2**. Поэтому мы можем сказать, что**item2** указывает на отдельный объект класса DisplayItem, на имя которого мы можем косвенно ссылаться через \***item2**. Уникальная идентичность (но не обязательно имя) каждого объекта сохраняется на все время его существования, даже если его внутреннее состояние изменилось. Эта ситуация напоминает парадокс Зенона о реке: может ли река быть той же самый, если в ней каждый день течет разная вода?   http://www.helloworld.ru/texts/comp/other/oop/pic03_01.gif  *Рис. 3-1. Идентичность объектов.*  Рассмотрим результат выполнения следующих операторов (рис. 3-1б):  **item1.move(item2->location());  item4 = item3;  item4->move(Point(38, 100));**  Объект **item1** и объект, на который указывает **item2**, теперь относятся к одной и той же точке экрана. Указатель **item4**стал указывать на тот же объект, что и **item3**. Кстати, заметьте разницу между выражениями "объект **item2**" и "объект, на который указывает **item2**". Второе выражение более точно, хотя для краткости мы часто будем использовать их как синонимы.  Хотя объект **item1** и объект, на который указывает **item2**, имеют одинаковое состояние, они остаются разными объектами. Кроме того, мы изменили состояние объекта **\*item3**, использовав его новое косвенное имя **item4**. Эта ситуация, которую мы называем структурной зависимостью, подразумевая под этим ситуацию, когда объект именуется более чем одним способом несколькими синонимичными именами. 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. Например, если мы уничтожим объект через указатель **item3**, то значение указателя **item4** окажется бессмысленным; эта ситуация называется повисшей ссылкой. На рис. 3-1в иллюстрируется результат выполнения следующих действий:  **item2 = &item1;  item4->move(item2->location());**  В первой строке создается синоним: **item2** указывает на тот же объект, что и **item1**. Во второй доступ к состоянию**item1** получен через этот новый синоним. К сожалению, при этом произошла утечка памяти, - объект, на который первоначально указывала ссылка **item2**, никак не именуется ни прямо, ни косвенно, и его идентичность потеряна. В Smalltalk и CLOS память, отведенная под объекты, будет вновь возвращена системе сборщиком мусора. В языках типа C++ такая память не освобождается, пока не завершится программа, создавшая объект. Такие утечки памяти могут вызвать и просто неудобство, и крупные сбои, особенно, если программа должна непрерывно работать длительное время [Представьте себе утечку памяти в программе управления спутником или сердечным стимулятором. Перезапуск компьютера на спутнике в нескольких миллионах километров от Земли очень неудобен. Аналогично, непредсказуемая сборка мусора в программе, управляющей стимулятором, может оказаться смертельным для пациента. В таких случаях разработчики систем реального времени предпочитают воздерживаться от динамического распределения памяти].  Копирование, присваивание и равенство. Структурная зависимость имеет место, когда объект имеет несколько имен. В наиболее интересных приложениях объектно-ориентированного подхода использование синонимов просто неизбежно. Например, рассмотрим следующие две функции:  **void highLight(DisplayItem& i);  void drag(DisplayItem i); // Опасно**  Если вызвать первую функцию с параметром **item1**, будет создан псевдоним: формальный параметр **i** означает указатель на фактический параметр, и следовательно **item1** и **i** именуют один и тот же объект во время выполнения функции. При вызове второй функции с аргументом **item1** ей передается новый объект, являющийся копией **item1**: **i**обозначает совершенно другой объект, хотя и с тем же состоянием, что и **item1**. В C++ различается передача параметров по ссылке и по значению. Надо следить за этим, иначе можно нечаянно изменить копию объекта, желая изменить сам объект [В Smalltalk семантика передачи объектов методам в качестве аргументов является по своему духу эквивалентом передачи параметра по ссылке в C++]. Как мы увидим в следующем разделе, передача объектов по ссылке в C++ необходима для программирования полиморфного поведения. В общем случае, передача объектов по ссылке крайне желательна для достаточно сложных объектов, поскольку при этом копируется ссылка, а не состояние, и следовательно, достигается большая эффективность (за исключением тех случаев, когда передаваемое значение очень простое).  В некоторых обстоятельствах, однако, подразумевается именно копирование. В языках типа C++ семантику копирования можно контролировать. В частности, мы можем ввести копирующий конструктор в определение класса, как в следующем фрагменте кода, который можно было бы включить в описание класса **DisplayItem**:  **DisplayItem(const DisplayItem&);**  В C++ копирующий конструктор может быть вызван явно (как часть описания объекта) или неявно (с передачей объекта по значению). Отсутствие этого специального конструктора вызывает копирующий конструктор, действующий по умолчанию, который копирует объект поэлементно. Однако, когда объект содержит ссылки или указатели на другие объекты, такая операция приводит к созданию синонимов указателей на объекты, что делает поэлементное копирование опасным. Мы предлагаем эмпирическое правило: разрешать неявное размножение путем копирования только для объектов, содержащих исключительно примитивные значения, и делать его явным для более сложных объектов.  Это правило поясняет то, что некоторые языки называют "поверхностным" и "глубоким" копированием. Чтобы копировать объект, в языке Smalltalk введены методы **shallowCopy** (метод копирует только объект, а состояние является разделяемым) и **deepCopy** (метод копирует объект и состояние, если нужно - рекурсивно). Переопределяя эти методы для классов с агрегацией, можно добиваться эффекта "глубокого" копирования для одних частей объекта, и "поверхностного" копирования остальных частей.  Присваивание - это, вообще говоря, копирование. В C++ его смысл тоже можно изменять. Например, мы могли бы добавить в определение класса **DisplayItem** следующую строку:  **virtual DisplayItem& operator=(const DisplayItem&);**  Этот оператор намеренно сделан виртуальным, так как ожидается, что подклассы будут его переопределять. Как и в случае копирующего конструктора, копирование можно сделать "глубоким" и "поверхностным". Если оператор присваивания не переопределен явно, то по умолчанию объект копируется поэлементно.  С вопросом присваивания тесно связан вопрос равенства. Хотя вопрос кажется простым, равенство можно понимать двумя способами. Во-первых, два имени могут обозначать один и тот же объект. Во-вторых, это может быть равенство состояний у двух разных объектов. В примере на рис. 3-1в оба варианта тождественности будут справедливы для **item1**и **item2**. Однако для **item2** и **item3** истинным будет только второй вариант.  В C++ нет предопределенного оператора равенства, поэтому мы должны определить равенство и неравенство, объявив эти операторы при описании:  **virtual int operator=(const DisplayItem&) const; int operator!=(const DisplayItem&) const;**  Мы предлагаем описывать оператор равенства как виртуальный (так как ожидаем, что подклассы могут переопределять его поведение), и описывать оператор неравенства как невиртуальный (так как хотим, чтобы он всегда был логическим отрицанием равенства: подклассам не следует переопределять это).  Аналогичным образом мы можем создавать операторы сравнения объектов типа >= и <=.  **Время жизни объектов.** Началом времени существования любого объекта является момент его создания (отведение участка памяти), а окончанием - возвращение отведенного участка памяти системе.  Объекты создаются явно или неявно. Есть два способа создать их явно. Во-первых, это можно сделать при объявлении (как это было с **item1**): тогда объект размещается в стеке. Во-вторых, как в случае **item3**, можно разместить объект, то есть выделить ему память из "кучи". В C++ в любом случае при этом вызывается конструктор, который выделяет известное ему количество правильно инициализированной памяти под объект. В Smalltalk этим занимаются метаклассы, о семантике которых мы поговорим позже.  Часто объекты создаются неявно. Так, передача параметра по значению в C++ создает в стеке временную копию объекта. Более того, создание объектов транзитивно: создание объекта тянет за собой создание других объектов, входящих в него. Переопределение семантики копирующего конструктора и оператора присваивания в C++ разрешает явное управление тем, когда части объекта создаются и уничтожаются. К тому же в C++ можно переопределять и оператор **new**, тем самым изменяя политику управления памятью в "куче" для отдельных классов.  В Smalltalk и некоторых других языках при потере последней ссылки на объект его забирает сборщик мусора. В языках без сборки мусора, типа C++, объекты, созданные в стеке, уничтожаются при выходе из блока, в котором они были определены, но объекты, созданные в "куче" оператором **new**, продолжают существовать и занимать место в памяти: их необходимо явно уничтожать оператором **delete**. Если объект "забыть", не уничтожить, это вызовет, как уже было сказано выше, утечку памяти. Если же объект попробуют уничтожить повторно (например, через другой указатель), последствием будет сообщение о нарушении памяти или полный крах системы.  При явном или неявном уничтожении объекта в C++ вызывается соответствующий деструктор. Его задача не только освободить память, но и решить, что делать с другими ресурсами, например, с открытыми файлами [Деструкторы не освобождают автоматически память, размещенную оператором **new**, программисты должны явно освободить ее].  Уничтожение долгоживущих объектов имеет несколько другую семантику. Как говорилось в предыдущей главе, некоторые объекты могут быть долгоживущими; под этим понимается, что их время жизни может выходить за время жизни породивших их программ. Обычно такие объекты являются частью некой долговременной объектной структуры, поэтому вопросы их жизни и смерти относятся скорее к политике соответствующей объектно-ориентированной базы данных. В таких системах для обеспечения долгой жизни наиболее принят подход на основе постоянных "подмешиваемых классов". Все объекты, которым мы хотим обеспечить долгую жизнь, должны наследовать от этих классов.  3.2. Отношения между объектами  **Типы отношений**  Сами по себе объекты не представляют никакого интереса: только в процессе взаимодействия объектов реализуется система. По выражению Ингалса: "Вместо процессора, беззастенчиво перемалывающего структуры данных, мы получаем сообщество хорошо воспитанных объектов, которые вежливо просят друг друга об услугах" [[13](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#3.13)]. Самолет, по определению, "совокупность элементов, каждый из которых по своей природе стремится упасть на землю, но за счет совместных непрерывных усилий преодолевающих эту тенденцию" [[14](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#3.14)]. 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 иерархических соотношений объектов:  связи;  агрегация.  Зайдевиц и Старк назвали эти два типа отношений отношениями старшинства и "родитель/потомок" соответственно [[15](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#3.15)].  **Связи**  **Семантика.** Мы заимствуем понятие связи у Румбаха, который определяет его как "физическое или концептуальное соединение между объектами" [[16](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#3.16)]. Объект сотрудничает с другими объектами через связи, соединяющие его с ними. Другими словами, связь - это специфическое сопоставление, через которое клиент запрашивает услугу у объекта-сервера или через которое один объект находит путь к другому.  На рис. 3-2 показано несколько разных связей. Они отмечены линиями и означают как бы пути прохождения сообщений. Сами сообщения показаны стрелками (соответственно их направлению) и помечены именем операции. На рисунке объект **aController** связан с двумя объектами класса **DisplayItem** (объекты **a** и **b**). В свою очередь, оба, вероятно, связаны с **aView**, но нам была интересна только одна из этих связей. Только вдоль связи один объект может послать сообщение другому.  Связь между объектами и передача сообщений обычно односторонняя (как на рисунке; хотя технически она может быть и взаимной). Как мы увидим в главе 5, подобное разделение прав и обязанностей типично для хорошо структурированных объектных систем [На самом деле организация объектов, показанная на рис. 3-2, встречается настолько часто, что ее можно считать типовым проектным решением. В Smalltalk аналогичный механизм известен как MVC, model/view/controller (модель/представление/контроллер). Как мы увидим в следующей главе, хорошо структурированные системы имеют много таких опознаваемых типовых решений]. Заметьте также, что хотя передаваемое сообщение инициализировано клиентом (в данном случае **aController**), данные передаются в обоих направлениях. Например, когда **aController**вызывает операцию move для пересылки данных объекту а, данные передаются от клиента серверу, но при выполнении операции **isUnder** над объектом **b**, результат передается от сервера клиенту.  Участвуя в связи, объект может выполнять одну из следующих трех ролей:  ® **Актер** [Actor - это деятель, исполнитель. А исполнитель ролей, это и есть актер. - Примеч. ред.]. Объект может воздействовать на другие объекты, но сам никогда не подвергается воздействию других объектов; в определенном смысле это соответствует понятию активный объект.   http://www.helloworld.ru/texts/comp/other/oop/pic03_02.gif  *Рис. 3-2. Связи.*  ® **Сервер**. Объект может только подвергаться воздействию со стороны других объектов, но он никогда не выступает в роли воздействующего объекта.  ® **Агент**. Такой объект может выступать как в активной, так и в пассивной роли; как правило, объект-агент создается для выполнения операций в интересах какого-либо объекта-актера или агента.  На рис. 3-2 объект **aController** выступает как актер, объект **a** - как агент и объект **aView** - как сервер.  **Пример.** Во многих промышленных процессах требуется непрерывное изменение температуры. Необходимо поднять температуру до заданного значения, выдержать заданное время и понизить до нормы. Профиль изменения температуры у разных процессов разный; зеркало телескопа надо охлаждать очень медленно, а закаляемую сталь очень быстро.  Абстракция нагрева имеет достаточно четкое поведение, что дает нам право на описание такого класса. Сначала определим тип, значение которого задает прошедшее время в минутах.  **// Число прошедших минут typedef unsigned int Minute;**  Теперь опишем сам класс **TemperatureRamp**, который по смыслу задает функцию времени от температуры:  **class TemperatureRamp {  public:**  **TemperatureRamp();  virtual ~TemperatureRamp();  virtual void clear();  virtual void bind (Temperature, Minute);  Temperature TemperatureAt (Minute);**  **protected:  ...  };**  Выдерживая наш стиль, мы описали некоторые из операций виртуальными, так как ожидаем, что этот класс будет иметь подклассы.  На самом деле в смысле поведения нам надо нечто большее, чем просто зависимость температуры от времени. Пусть, например, известно, что на 60-й минуте должна быть достигнута температура 250ЂF, а на 180-й - 150ЂF. Спрашивается, какой она должна быть на 120-й минуте? Это требует линейной интерполяции, так что требуемое от абстракции поведение усложняется.  Вместе с тем, управления нагревателем, поддерживающего требуемый профиль, мы от этой абстракции не требуем. Мы предпочитаем разделение понятий, при котором нужное поведение достигается взаимодействием трех объектов: экземпляра **TemperatureRamp**, нагревателя и контроллера. Класс **TemperatureController** можно определить так:  **class TemperatureController {  public:**  **TemperatureController(Location);  ~TemperatureController();  void process(const TemperatureRamp&);  Minute schedule(const TemperatureRamp&) const;**  **private:  ...  };**  Тип **Location** был определен в главе 2. Заметьте, что мы не ожидаем наследования от этого класса и поэтому не объявляем в нем никаких виртуальных функций.  Операция **process** обеспечивает основное поведение этой абстракции; ее назначение - передать график изменения температуры нагревателю, установленному в конкретном месте. Например, объявим:  **TemperatureRamp growingRamp;  TemperatureController rampController(7);**  Теперь зададим пару точек и загрузим план в контроллер::  **growingRamp.bind (250, 60);  growingRamp.bind(150, 180);  rampController.process(growingRamp);**  В этом примере **rampController** - агент, ответственный за выполнение температурного плана, он использует объект**growingRamp** как сервер. Эта связь проявляется хотя бы в том, что **rampController** явно получает **growingRamp** в качестве параметра одной из своих операций.  Одно замечание по поводу нашего стиля. На первый взгляд может показаться, что наши абстракции - лишь объектные оболочки для элементов, полученных в результате обычной функциональной декомпозиции. Пример операции **schedule**показывает, что это не так. Объекты класса **TemperatureController** имеют достаточно интеллекта, чтобы определять расписание для конкретных профилей, и мы включаем эту операцию как дополнительное поведение нашей абстракции. В некоторых энергоемких технологиях (например, плавка металлов) можно существенно выиграть, если учитывать остывание установки и тепло, остающееся после предыдущей плавки. Поскольку существует операция **schedule**, клиент может запросить объект **TemperatureController**, чтобы тот рекомендовал оптимальный момент запуска следующего нагрева.  Видимость. Пусть есть два объекта А и в и связь между ними. Чтобы А мог послать сообщение в, надо, чтобы в был в каком-то смысле видим для А. Мы можем не заботиться об этом на стадии анализа, но когда дело доходит до реализации системы, мы должны обеспечить видимость связанных объектов.  В предыдущем примере объект **rampController** видит объект **growingRamp**, поскольку оба они объявлены в одной области видимости и потому, что **growingRamp** передается объекту **rampController** в качестве параметра. В принципе есть 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.  Какой именно из этих способов выбрать - зависит от тактики проектирования.  **Синхронизация.** Когда один объект посылает по связи сообщение другому, связанному с ним, они, как говорят, синхронизируются. В строго последовательном приложении синхронизация объектов и состоит в запуске метода (см. врезку ниже). Однако в многопоточной системе объекты требуют более изощренной схемы передачи сообщений, чтобы разрешить проблемы взаимного исключения, типичные для параллельных систем. Активные объекты сами по себе выполняются как потоки, поэтому присутствие других активных объектов на них обычно не влияет. Если же активный объект имеет связь с пассивным, возможны следующие три подхода к синхронизации:  Последовательный - семантика пассивного объекта обеспечивается в присутствии только одного активного процесса.  Защищенный - семантика пассивного объекта обеспечивается в присутствии многих потоков управления, но активные клиенты должны договориться и обеспечить взаимное исключение.  Синхронный - семантика пассивного объекта обеспечивается в присутствии многих потоков управления; взаимное исключение обеспечивает сервер.  Все объекты, описанные в этой главе, были последовательными. В главе 9 мы рассмотрим остальные варианты более подробно.  **Агрегация**  **Семантика.** В то время, как связи обозначают равноправные или "клиент-серверные" отношения между объектами, агрегация описывает отношения целого и части, приводящие к соответствующей иерархии объектов, причем, идя от целого (агрегата), мы можем придти к его частям (атрибутам). В этом смысле агрегация - специализированный частный случай ассоциации. На рис. 3-3 объект rampController имеет связь с объектом **growingRamp** и атрибут **h** класса **Heater**(нагреватель). В данном случае **rampController** - целое, a **h** - его часть. Другими словами, **h** - часть состояния**rampController**. Исходя из **rampController**, можно найти соответствующий нагреватель. Однако по **h** нельзя найти содержащий его объект (называемый также его контейнером), если только сведения о нем не являются случайно частью состояния **h**.   http://www.helloworld.ru/texts/comp/other/oop/pic03_03.gif  *Рис. 3-3. Агрегация.*  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. Это, несомненно, отношение агрегации, но скорее концептуальное, чем физическое по своей природе.  Выбирая одно из двух - связь или агрегацию - надо иметь в виду следующее. Агрегация иногда предпочтительнее, поскольку позволяет скрыть части в целом. Иногда наоборот предпочтительнее связи, поскольку они слабее и менее ограничительны. Принимая решение, надо взвесить все.  Объект, являющийся атрибутом другого объекта (агрегата), имеет связь со своим агрегатом. Через эту связь агрегат может посылать ему сообщения.  **Пример.** Добавим в спецификацию класса **TemperatureController** описание нагревателя:  **Heater h;**  После этого каждый объект **TemperatureController** будет иметь свой нагреватель. В соответствии с нашим определением класса Heater в предыдущей главе мы должны инициализировать нагреватель при создании нового контроллера, так как сам этот класс не предусматривает конструктора по умолчанию. Мы могли бы определить конструктор класса **TemperatureController** следующим образом:  **TemperatureController::TemperatureController(Location 1) : h(1) {}**  3.3. Природа классов  **Что такое класс?**  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. Таким образом, можно говорить о классе "Млекопитающие", который включает характеристики, общие для всех млекопитающих. Для указания на конкретного представителя млекопитающих необходимо сказать "это - млекопитающее" или "то - млекопитающее".   http://www.helloworld.ru/texts/comp/other/oop/pg111.gif  *Класс представляет набор объектов, которые обладают общей структурой и одинаковым поведением.*  В общепонятных терминах можно дать следующее определение класса: "группа, множество или вид с общими свойствами или общим свойством, разновидностями, отличиями по качеству, возможностями или условиями" [[17](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#3.17)]. В контексте объектно-ориентированного анализа дадим следующее определение класса:  *Класс - это некое множество объектов, имеющих общую структуру и общее поведение.*  Любой конкретный объект является просто экземпляром класса. Что же не является классом? Объект не является классом, хотя в дальнейшем мы увидим, что класс может быть объектом. Объекты, не связанные общностью структуры и поведения, нельзя объединить в класс, так как по определению они не связаны между собой ничем, кроме того, что все они объекты.  Важно отметить, что классы, как их понимают в большинстве существующих языков программирования, необходимы, но не достаточны для декомпозиции сложных систем. Некоторые абстракции так сложны, что не могут быть выражены в терминах простого описания класса. Например, на достаточно высоком уровне абстракции графический интерфейс пользователя, база данных или система учета как целое, это явные объекты, но не классы [Можно попытаться выразить такие абстракции одним классом, но повторной используемости и возможности наследования не получится. Иметь громоздкий интерфейс - плохая практика, так как большинство клиентов использует только малую его часть. Более того, изменение одной части этого гигантского интерфейса требует обновления каждого из клиентов, независимо от того, затронуло ли его это изменение по сути. Вложенность классов не устраняет этих проблем, а только откладывает их]. Лучше считать их некими совокупностями (кластерами) сотрудничающих классов. Страуструп называет такие кластеры компонентами [[18](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#3.18)]. Мы же, по причинам, которые будут объяснены в главе 5, называем такие кластеры категориями классов.  **Интерфейс и реализация**  Мейер [[19](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#3.19)] и Снайдерс [[20](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#3.20)] высказали идею контрактного программирования:  большие задачи надо разделить на много маленьких и перепоручить их мелким субподрядчикам. Нигде эта идея не проявляет себя так ярко, как в проектировании классов.  По своей природе, класс - это генеральный контракт между абстракцией и всеми ее клиентами. Выразителем обязательств класса служит его интерфейс, причем в языках с сильной типизацией потенциальные нарушения контракта можно обнаружить уже на стадии компиляции.  Идея контрактного программирования приводит нас к разграничению внешнего облика, то есть интерфейса, и внутреннего устройства класса, реализации. Главное в интерфейсе - объявление операций, поддерживаемых экземплярами класса. К нему можно добавить объявления других классов, переменных, констант и исключительных ситуаций, уточняющих абстракцию, которую класс должен выражать. Напротив, реализация класса никому, кроме него самого, не интересна. По большей части реализация состоит в определении операций, объявленных в интерфейсе класса.  Мы можем разделить интерфейс класса на три части:  открытую (public) - видимую всем клиентам;  защищенную (protected) - видимую самому классу, его подклассам и друзьям (friends);  закрытую (private) - видимую только самому классу и его друзьям.  Разные языки программирования предусматривают различные комбинации этих частей. Разработчик может задать права доступа к той или иной части класса, определив тем самым зону видимости клиента.  В частности, в C++ все три перечисленных уровня доступа определяются явно. В дополнение к этому есть еще и механизм друзей, с помощью которого посторонним классам можно предоставить привилегию видеть закрытую и защищенную области класса. Тем самым нарушается инкапсуляция, поэтому, как и в жизни, друзей надо выбирать осторожно. В Ada объявления могут быть сделаны закрытыми или открытыми. В Smalltalk все переменные - закрыты, а методы - открыты. В Object Pascal все поля и операции открыты, то есть никакой инкапсуляции нет [Речь идет о старых версиях Object Pascal - Прим. редактора-2]. В CLOS обобщенные функции открыты, а слоты могут быть закрытыми, хотя узнать их значения все равно можно.  Состояние объекта задается в его классе через определения констант или переменных, помещаемые в его защищенной или закрытой части. Тем самым они инкапсулированы, и их изменения не влияют на клиентов.  Внимательный читатель может спросить, почему же представление объекта определяется в интерфейсной части класса, а не в его реализации. Причины чисто практические: в противном случае понадобились бы объектно-ориентированные процессоры или очень хитроумные компиляторы. Когда компилятор обрабатывает объявление объекта, например, такое:  **DisplayItem item1;**  он должен знать, сколько отвести под него памяти. Если бы эта информация содержалась в реализации класса, нам пришлось бы написать ее полностью, прежде, чем мы смогли бы задействовать его клиентов. То есть, весь смысл отделения интерфейса от реализации был бы потерян.  Константы и переменные, составляющие представление класса, известны под разными именами. В Smalltalk их называют переменные экземпляра, в Object Pascal - поля, в C++ - члены класса, а в CLOS - слоты. Мы часто будем использовать эти термины как синонимы.  **Жизненный цикл класса**  В поведении простых классов можно разобраться, изучая операции их открытой части. Однако поведение более интересных классов (такое как перемещение объекта класса **DisplayItem** или составление расписания для экземпляра класса **TemperatureController**) включает взаимодействие разных операций, входящих в класс. Как уже говорилось выше, объекты таких классов действуют как маленькие машины, части которых взаимодействуют друг с другом, и так как все такие объекты имеют одно и то же поведение, можно использовать класс для описания их общей семантики, упорядоченной по времени и событиям. Как будет показано в главе 5, мы можем описывать динамику поведения объектов, используя модель конечного автомата.  3.4. Отношения между классами  **Типы отношений**  Рассмотрим сходства и различия между следующими классами: цветы, маргаритки, красные розы, желтые розы, лепестки и божьи коровки. Мы можем заметить следующее:  Маргаритка - цветок.  Роза - (другой) цветок.  Красная и желтая розы - розы.  Лепесток является частью обоих видов цветов.  Божьи коровки питаются вредителями, поражающими некоторые цветы.  Из этого простого примера следует, что классы, как и объекты, не существуют изолированно. В каждой проблемной области ключевые абстракции взаимодействуют многими интересными способами, что мы и должны отразить в проекте [[21](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#3.21)].  Отношения между классами могут означать одно из двух. Во-первых, у них может быть что-то общее. Например, и маргаритки, и розы - это разновидности цветов: и те, и другие имеют ярко окрашенные лепестки, испускают аромат и так далее. Во-вторых, может быть какая-то семантическая связь. Например, красные розы больше похожи на желтые розы, чем на маргаритки. Но между розами и маргаритками больше общего, чем между цветами и лепестками. Также существует симбиотическая связь между цветами и божьими коровками: божьи коровки защищают цветы от вредителей, которые, в свою очередь, служат пищей божьим коровкам.  Известны три основных типа отношений между классами [[22](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#3.22)]. Во-первых, это отношение "обобщение/специализация" (общее и частное), известное как "is-a". Розы суть цветы, что значит: розы являются специализированным частным случаем, подклассом более общего класса "цветы". Во вторых, это отношение "целое/ часть", известное как "part of". Так, лепестки являются частью цветов. В-третьих, это семантические, смысловые отношения, ассоциации. Например, божьи коровки ассоциируются с цветами - хотя, казалось бы, что у них общего. Или вот: розы и свечи - и то, и другое можно использовать для украшения стола.  Языки программирования выработали несколько общих подходов к выражению отношений этих трех типов. В частности, большинство объектно-ориентированных языков непосредственно поддерживают разные комбинации следующих видов отношений:  ассоциация  наследование  агрегация  использование  инстанцирование  метакласс.  Альтернативой наследованию является делегирование, при этом объекты рассматриваются как прототипы, которые делегируют свое поведение родственным им объектам. Таким образом, классы становятся не нужны [[23](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#3.23)].  Из шести перечисленных видов отношений наиболее общим и неопределенным является ассоциация. Как мы увидим в главе 6, обычно аналитик констатирует наличие ассоциации и, постепенно уточняя проект, превращает ее в какую-то более специализированную связь.  Наследование, вероятно, следует считать самым интересным семантически. Оно выражает отношение общего и частного. Однако, по нашему опыту, одного наследования недостаточно, чтобы выразить все многообразие явлений и отношений жизни. Полезна также агрегация, отражающая отношения целого и части между экземплярами классов. Нелишне добавить отношение использования, означающее наличие связи между экземплярами классов. Имея дело с языками Ada, Eiffel и C++, нам не обойтись без инстанцирования, которое, подобно наследованию, является специфической разновидностью обобщения. "Метаклассовые" отношения - это нечто совершенно иное, в явном виде встречающееся только в языках Smalltalk и CLOS. Метакласс - это класс классов, что позволяет нам трактовать классы как объекты.   http://www.helloworld.ru/texts/comp/other/oop/pic03_04.gif  *Рис. 3-4. Ассоциация.*  **Ассоциация**  **Пример.** Желая автоматизировать розничную торговую точку, мы обнаруживаем две абстракции - товары и продажи. На рис. 3-4 показана ассоциация, которую мы при этом усматриваем. Класс **Product** - это то, что мы продали в некоторой сделке, а класс **Sale** - сама сделка, в которой продано несколько товаров. Надо полагать, ассоциация работает в обе стороны: задавшись товаром, можно выйти на сделку, в которой он был продан, а пойдя от сделки, найти, что было продано.  В C++ это можно выразить с помощью того, что Румбах называет погребенными указателями [[24](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#3.24)]. Вот две выдержки из объявления соответствующих классов:  **class Product;**  **class Sale;**  **class Product {  public:  ...  protected:**  **Sale\* lastSale;**  **};**  **class Sale {  public:  ...  protected:**  **Product\*\* productSold;**  **};**  Это ассоциация вида "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.  **Семантические зависимости.** Как показывает этот пример, ассоциация - смысловая связь. По умолчанию, она не имеет направления (если не оговорено противное, ассоциация, как в данном примере, подразумевает двухстороннюю связь) и не объясняет, как классы общаются друг с другом (мы можем только отметить семантическую зависимость, указав, какие роли классы играют друг для друга). Однако именно это нам требуется на ранней стадии анализа. Итак, мы фиксируем участников, их роли и (как будет сказано далее) мощность отношения.  **Мощность.** В предыдущем примере мы имели ассоциацию "один ко многим". Тем самым мы обозначили ее мощность (то есть, грубо говоря, количество участников). На практике важно различать три случая мощности ассоциации:  "один-к-одному"  "один-ко-многим"  "многие-ко-многим".  Отношение "один-к-одному" обозначает очень узкую ассоциацию. Например, в розничной системе продаж примером могла бы быть связь между классом **Sale** и классом **CreditCardTransaction**: каждая продажа соответствует ровно одному снятию денег с данной кредитной карточки. Отношение "многие-ко-многим" тоже нередки. Например, каждый объект класса **Customer** (покупатель) может инициировать транзакцию с несколькими объектами класса **Saleperson**(торговый агент), и каждый торговый агент может взаимодействовать с несколькими покупателями. Как мы увидим в главе 5, все три вида мощности имеют разного рода вариации.  **Наследование**  **Примеры.** Находящиеся в полете космические зонды посылают на наземные станции информацию о состоянии своих основных систем (например, источников энергоснабжения и двигателей) и измерения датчиков (таких как датчики радиации, масс-спектрометры, телекамеры, фиксаторы столкновений с микрометеоритами и т.д.). Вся совокупность передаваемой информации называется телеметрическими данными. Как правило, они передаются в виде потока данных, состоящего из заголовка (включающего временные метки и ключи для идентификации последующих данных) и нескольких пакетов данных от подсистем и датчиков. Все это выглядит как простой набор разнотипных данных, поэтому для описания каждого типа данных телеметрии сами собой напрашиваются структуры:  **class Time...**  **struct ElectricalData {**  **Time timeStamp;  int id;  float fuelCell1Voltage, fuelCell2Voltage;  float fuelCell1Amperes, fuelCell2Amperes;  float currentPower;**  **};**  Однако такое описание имеет ряд недостатков. Во-первых, структура класса **ElectricalData** не защищена, то есть клиент может вызвать изменение такой важной информации, как **timeStamp** или **currentPower** (мощность, развиваемая обеими электробатареями, которую можно вычислить из тока и напряжения). Во-вторых, эта структура является полностью открытой, то есть ее модификации (добавление новых элементов в структуру или изменение типа существующих элементов) влияют на клиентов. Как минимум, приходится заново компилировать все описания, связанные каким-либо образом с этой структурой. Еще важнее, что внесение в структуру изменений может нарушить логику отношений с клиентами, а следовательно, логику всей программы. Кроме того, приведенное описание структуры очень трудно для восприятия. По отношению к такой структуре можно выполнить множество различных действий (пересылка данных, вычисление контрольной суммы для определения ошибок и т.д.), но все они не будут связаны с приведенной структурой логически. Наконец, предположим, что анализ требований к системе обусловил наличие нескольких сотен разновидностей телеметрических данных, включающих показанную выше структуру и другие электрические параметры в разных контрольных точках системы. Очевидно, что описание такого количества дополнительных структур будет избыточным как из-за повторяемости структур, так и из-за наличия общих функций обработки.   http://www.helloworld.ru/texts/comp/other/oop/pg117.gif  *Дочерний класс может унаследовать структуру и поведение родительских классов.*  Лучше было бы создать для каждого вида телеметрических данных отдельный класс, что позволит защитить данные в каждом классе и увязать их с выполняемыми операциями. Но этот подход не решает проблему избыточности.  Значительно лучше построить иерархию классов, в которой от общих классов с помощью наследования образуются более специализированные; например, следующим образом:  **class TelemetryData {**  **public:**  **TelemetryData();**  **virtual ~TelemetryData();**  **virtual void transmit();**  **Time currentTime() const;**  **protected:**  **int id;**  **Time timeStamp;**  **};**  В этом примере введен класс, имеющий конструктор, деструктор (который иаследники могут переопределить) и функции**transmit** и **currentTime**, видимые для всех клиентов. Защищенные элементы **id** и **timeStamp** несколько лучше инкапсулированы - они доступны только классу и его подклассам. Заметьте, что функция currentTlrne сделана открытой, благодаря чему значение **timeStamp** можно читать (но не изменять).  Теперь разберемся с **ElectricalData**:  **class ElectricalData : public TelemetryData {**  **public:**  **ElectricalData(float v1, float v2, float a1, float a2);**  **virtual ~ElectricalData();**  **virtual void.transmit();**  **float currentPower() const;**  **protected:**  **float fuelCell1Voltage, fuelCell2Voltage;**  **float fuelCell1Amperes, fuelCell2Amperes;**  **};**  Этот класс - наследник класса **TelemetryData**, но исходная структура дополнена (четырьмя новыми элементами), а поведение - переопределено (изменена функция **transmit**). Кроме того, добавлена функция **currentPower**.  **Одиночное наследование.** Попросту говоря, наследование - это такое отношение между классами, когда один класс повторяет структуру и поведение другого класса (одиночное наследование) или других (множественное наследование) классов. Класс, структура и поведение которого наследуются, называется суперклассом. Так, **TelemetryData**. является суперклассом для **ElectricalData**. Производный от суперкласса класс называется подклассом. Это означает, что наследование устанавливает между классами иерархию общего и частного. В этом смысле **ElectricalData** является более специализированным классом более общего **TelemetryData**. Мы уже видели, что в подклассе структура и поведение исходного суперкласса дополняются и переопределяются. Наличие механизма наследования отличает объектно-ориентированные языки от объектных.  Подкласс обычно расширяет или ограничивает существующую структуру и поведение своего суперкласса. Например, подкласс **GuardedQueue** может добавлять к поведению суперкласса **Queue** операции, которые защищают состояние очереди от одновременного изменения несколькими независимыми потоками. Обратный пример: подкласс**UnselectableDisplayItem** может ограничить поведение своего суперкласса **DisplayItem**, запретив выделение объекта на экране. Часто подклассы делают и то, и другое.  Отношения одиночного наследования от суперкласса **TelemetryData** показаны на рис. 3-5. Стрелки обозначают отношения общего к частному. В частности, **Cameradata** - это разновидность класса **SensorData**, который в свою очередь является разновидностью класса **TelemetryData**. Такой же тип иерархии характерен для семантических сетей, которые часто используются специалистами по распознаванию образов и искусственному интеллекту для организации баз знаний [[25](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#3.25)]. В главе 4 мы покажем, что правильная организация иерархии абстракций - это вопрос логической классификации.   http://www.helloworld.ru/texts/comp/other/oop/pic03_05.gif  *Рис. 3-5. Одиночное наследование.*  Можно ожидать, что для некоторых классов на рис. 3-5 будут созданы экземпляры, а для других - нет. Наиболее вероятно образование объектов самых *специализированных классов***ElectricalData** и **SpectrometerData** (такие классы называют конкретными классами, или листьями иерархического дерева). Образование объектов из классов, занимающих промежуточное положение (**SensorData** или тем более **TelemetryData**), менее вероятно. Классы, экземпляры которых не создаются, называются абстрактными. Ожидается, что подклассы абстрактных классов доопределят их до жизнеспособной абстракции, наполняя класс содержанием. В языке Smalltalk разработчик может заставить подкласс переопределить метод, помещая в реализацию метода суперкласса вызов метода**SubclassResponsibility**. Если метод не переопределен, то при попытке выполнить его генерируется ошибка. Аналогично, в C++ существует возможность объявлять функции чисто виртуальными. Если они не переопределены, экземпляр такого класса невозможно создать.  Самый общий класс в иерархии классов называется базовым. В большинстве приложений базовых классов бывает несколько, и они отражают наиболее общие абстракции предметной области. На самом деле, особенно в C++, хорошо сделанная структура классов - это скорее лес из деревьев наследования, чем одна многоэтажная структура наследования с одним корнем. Но в некоторых языках программирования определен базовый класс самого верхнего уровня, который является единым суперклассом для всех остальных классов. В языке Smalltalk эту роль играет класс object.  У класса обычно бывает два вида клиентов [[26](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#3.26)]:  экземпляры;  подклассы.  Часто полезно иметь для них разные интерфейсы [[27](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#3.27)]. В частности, мы хотим показать только внешне видимое поведение для клиентов-экземпляров, но нам нужно открыть служебные функции и представления клиентам-подклассам. Этим объясняется наличие открытой, защищенной и закрытой частей описания класса в языке C++: разработчик может четко разделить, какие элементы класса доступны Для экземпляров, а какие для подклассов. В языке Smalltalk степень такого разделения меньше: данные видимы для подклассов, но не для экземпляров, а методы общедоступны (можно ввести закрытые методы, но язык не обеспечивает их защиту).  Есть серьезные противоречия между потребностями наследования и инкапсуляции. В значительной мере наследование открывает наследующему классу некоторые секреты. На практике, чтобы понять, как работает какой-то класс, часто надо изучить все его суперклассы в их внутренних деталях.  Наследование подразумевает, что подклассы повторяют структуры их суперклассов. В предыдущем примере экземпляры класса **ElectricalData** содержат элементы структуры суперкласса (**id** и **timeStamp**) и более специализированные элементы (**fuelCell1Voltage, fuelCell2Voltage, fuelCell1Amperes, fuelCell2Amperes**) [Некоторые языки объектно-ориентированного программирования, главным образом экспериментальные, позволяют подклассу сокращать структуру его суперкласса].  Поведение суперклассов также наследуется. Применительно к объектам класса **ElectricalData** можно использовать операции **currentTime** (унаследована от суперкласса), **currentPower** (определена в классе) и **transmit**(переопределена в подклассе). В большинстве языков допускается не только наследование методов суперкласса, но также добавление новых и переопределение существующих методов. В Smalltalk любой метод суперкласса можно переопределить в подклассе.  В C++ степень контроля за этим несколько выше. Функция, объявленная виртуальной (функция **transmit** в предыдущем примере), может быть в подклассе переопределена, а остальные (**currentTime**) - нет.  **Одиночный полиморфизм.** Пусть функция **transmit** класса **TelemetryData** реализована следующим образом:  **void TelemetryData::transmit()**  **{**  **// передать id**  **// передать timeStamp**  **};**  В классе **ElectricalData** та же функция переопределена:  **void ElectricalData::transmit()**  **{**  **TelemetryData::transmit();**  **// передать напряжение**  **// передать силу тока**  **};**  Эта функция сначала вызывает одноименную функцию суперкласса с помощью ее явно квалифицированного имени**TelemetryData::transmit**(). Та передаст заголовок пакета (**id** и **timeStamp**), после чего в подклассе передаются его собственные данные.  Определим теперь экземпляры двух описанных выше классов:  **TelemetryData telemetry;**  **ElectricalData electrical(5.0, -5.0, 3.0, 7.0);**  Теперь определим свободную процедуру:  **void transmitFreshData (TelemetryData& d, const Time& t)**  **{**  **if (d.currentTime() >= t)**  **d.transmit();**  **);**  Что произойдет, если выполнить следующие два оператора?  **transmitFreshData(telemetry, Time(60));**  **transmitFreshData(electrical, Time(120));**  В первом операторе будет передан уже известный нам заголовок. Во втором будет передан он же, плюс четыре числа в формате с плавающей точкой, содержащие результаты измерений электрических параметров. Почему это так? Ведь функция **transmitFreshData** ничего не знает о классе объекта, она просто выполняет **d.transmit**()! Это был пример полиморфизма. Переменная d может обозначать объекты разных классов. У этих классов есть общий суперкласс и они, хотя и по разному, могут реагировать на одно и то же сообщение, одинаково понимая его смысл.  Карделли и Вегнер заметили, что "традиционные типизированные языки типа Pascal основаны на той идее, что функции и процедуры, а следовательно, и операнды должны иметь определенный тип. Это свойство называется мономорфизмом, то есть каждая переменная и каждое значение относятся к одному определенному типу. В противоположность мономорфизму полиморфизм допускает отнесение значений и переменных к нескольким типам" [[28](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#3.28)]. Впервые идею полиморфизма ad hoc описал Страчи [[29](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#3.29)], имея в виду возможность переопределять смысл символов, таких, как "+", сообразно потребности. В современном программировании мы называем это перегрузкой. Например, в C++ можно определить несколько функций с одним и тем же именем, и они будут автоматически различаться по количеству и типам своих аргументов. Совокупность этих признаков называется сигнатурой функции; в языке Ada к этому списку добавляется тип возвращаемого значения. Страчи говорил также о параметрическом полиморфизме, который мы сейчас называем просто полиморфизмом.  При отсутствии полиморфизма код программы вынуждено содержит множество операторов выбора case или sw**i**tch. Например, на языке Pascal невозможно образовать иерархию классов телеметрических данных; вместо этого придется определить одну большую запись с вариантами, включающую все разновидности данных. Для выбора варианта нужно проверить метку, определяющую тип записи. На языке Pascal процедура **TransmitFreshData** может быть написана следующим образом:  **const**  **Electrical = 1;**  **Propulsion = 2;**  **Spectrometer = 3;**  **Procedure Transmit\_Presh\_Data(TheData: Data; The\_Time: Time);**  **begin**  **if (The\_Data.Current\_Time >= The\_Time)**  **then**  **case TheData.Kind of**  **Electrical: Transmit\_Electrical\_Data(The\_Data);**  **Propulsion: Transmit\_Propulsion\_Data(The\_Data);**  **end;**  **end;**  Чтобы ввести новый тип телеметрических данных, нужно модифицировать эту вариантную запись, добавив новый тип в каждый оператор case. В такой ситуации увеличивается вероятность ошибок, и проект становится нестабильным.  Наследование позволяет различать разновидности абстракций, и монолитные типы становятся не нужны. Каплан и Джонсон отметили, что "полиморфизм наиболее целесообразен в тех случаях, когда несколько классов имеют одинаковые протоколы" [[30](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#3.30)]. Полиморфизм позволяет обойтись без операторов выбора, поскольку объекты сами знают свой тип.  Наследование без полиморфизма возможно, но не очень полезно. Это видно на примере Ada, где можно объявлять производные типы, но из-за мономорфизма языка операции жестко задаются на стадии компиляции.  Полиморфизм тесно связан с механизмом позднего связывания. При полиморфизме связь метода и имени определяется только в процессе выполнения программ. В C++ программист имеет возможность выбирать между ранним и поздним связыванием имени с операцией. Если функция виртуальная, связывание будет поздним и, следовательно, функция полиморфна. Если нет, то связывание происходит при компиляции и ничего изменить потом нельзя. Этому вопросу посвящена следующая врезка.  **Наследование и типизация.** Рассмотрим еще раз переопределение функции **transmit**:  **void ElectricalData::transmit()**  **{**  **TelemetryData::transmit();**  **// передать напряжение**  **// передать силу тока**  **};**  В большинстве объектно-ориентированных языков программирования при реализации метода подкласса разрешается вызывать напрямую метод какого-либо суперкласса. Как видно из примера, это допускается и в том случае, если метод подкласса имеет такое же имя и фактически переопределяет метод суперкласса. В Smalltalk метод вышестоящего класса вызывают с помощью ключевого слова super, при этом вызывающий может указывать на самого себя с помощью ключевого слова self. В C++ метод любого достижимого вышестоящего класса можно вызывать, добавляя имя класса в качестве префикса, формируя квалифицированное имя метода (как **TelemetryData::transmit**() в нашем примере). Вызывающий объект может ссылаться на себя с помощью предопределенного указателя **this**.  На практике метод суперкласса вызывается до или после дополнительных действий. Метод подкласса уточняет или дополняет поведение суперкласса [В CLOS эти различные роли метода выражаются явно с помощью дополнительных квалификаторов **:before**, **:after** или **:around**. Метод без дополнительного квалификатора считается первичным и выполняет основную работу, обеспечивающую требуемое поведение. Before-метод вызывается до первичного, after-метод - после первичного, around-метод действует как оболочка вокруг первичного метода, которая вызывается изнутри этого метода функцией call-next-method].  Все подклассы на рис. 3-5 являются также подтипами вышестоящего класса. В частности, **ElectricalData** является подтипом **TelemetryData**. Система типов, развивающаяся параллельно наследованию, обычна для объектно-ориентированных языков с сильной типизацией, включая C++. Для Smalltalk, который едва ли вообще можно считать типизированным, типы не имеют значения.    |  | | --- | | Поиск метода  Рассмотрим иерархию (рис. 3-6), в которой имеется базовый класс и три подкласса с именами **circle**, **Triangle** и**Rectangle**. Для класса **Rectangle** определен в свою очередь подкласс **SolidRectangle**. Предположим, что в классе**DisplayItem** определена переменная экземпляра **theCenter** (задающая координаты центра изображения), а также следующие операции:  **draw** - нарисовать изображение;  **move** - передвинуть изображение;  **location** - вернуть координаты изображения.  Операция **location** является общей для всех подклассов и не требует обязательного переопределения. Однако, поскольку только подклассы могут знать, как их изображать и передвигать, операции **draw** и **move** должны быть переопределены.   http://www.helloworld.ru/texts/comp/other/oop/pic03_06.gif  *Рис. 3-6. Диаграмма класса****DisplayItem***.  Класс **Circle** имеет переменную **theRadius** и соответственно операции для установки (set) и чтения значения этой переменной. Для этого класса операция **draw** формирует изображение окружности заданного радиуса с центром в точке **theCenter**. В классе **Rectangle** есть переменные **theHeight** и **theWidth** и соответствующие операции установки и чтения их значений. Операция draw в данном случае формирует изображение прямоугольника заданной высоты и ширины с центром в заданной точке **theCenter**. Подкласс **SolidRectangle** наследует все особенности класса **Rectangle**, но операция draw в этом подклассе переопределена. Сначала вызывается draw вышестоящего класса, а затем полученный контур заполняется цветом.  Теперь рассмотрим следующий фрагмент программы:  **DisplayItem\* items[10];** **for (unsigned index = 0; index < 10; index++) items[index]->draw();**  Вызов draw требует полиморфного поведения. У нас есть разнородный массив объектов, содержащий указатели на любые разновидности **DisplayItem**. Пусть некоторый клиент хочет, чтобы они все изобразили себя на экране. Наш подход - перебрать элементы массива и послать каждому указываемому объекту сообщение draw. Компилятор не может определить, какую функцию и откуда надо при этом вызвать, так как невозможно предсказать, на что будут указывать элементы массива во время выполнения программы. Посмотрим, как эта задача будет решаться в разных объектно-ориентированных языках.  Поскольку в Smalltalk нет типов, методы вызываются строго динамически. Когда клиент посылает сообщение draw очередному получателю, происходит следующее:  получатель ищет селектор сообщения в словаре методов своего класса;  если метод найден, то запускается его код;  если нет, поиск производится в словаре методов суперкласса.  Таким образом, поиск распространяется вверх по иерархии и заканчивается на классе **object**, который является "предком" всех классов. Если метод не найден и там, посылается сообщение **doesNotUnderstand**, то есть, генерируется ошибка.  Главным действующим лицом в этом алгоритме является словарь методов. Он формируется при создании класса и, являясь частью его реализации, скрыт от клиентов. Вызов метода в Smalltalk требует примерно в 1.5 раза больше времени, чем вызов простой подпрограммы. В коммерческих версиях Smalltalk вызов методов ускорен на 20-30% за счет кеширования доступа к словарю [[31](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#3.31)].  Операция **draw** в подклассе **solidRectangle** представляет собой особый случай. Мы уже отмечали, что вначале вызывается одноименный метод суперкласса **Rectangle**. В Smalltalk для вызова метода суперкласса используется ключевое слово super. Поиск метода super draw начинается сразу с суперкласса.  Исследования Дейча дают основание полагать, что полиморфизм в 85% случаев не нужен, так что вызов метода часто можно свести к обычному вызову процедуры [[32](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#3.32)]. Дафф замечает, что в таких ситуациях программист часто делает неявные предположения, которые бы позволили раннее связывание [[33](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#3.33)]. К сожалению, в нетипизированных языках у него нет способа сообщить об этом компилятору.  В более строго типизированных языках, таких как C++, такой способ есть. В этих языках алгоритм вызова методов несколько отличается от описанного выше и позволяет сократить во многих случаях время поиска, сохранив при этом свойства полиморфизма.  В C++ операции для позднего связывания объявляются виртуальными (virtual), а все остальные обрабатываются компилятором как обычные вызовы подпрограмм. В нашем примере **draw** - виртуальная функция, a **location** - обычная. Есть еще одно средство, используя которое можно выиграть в скорости. Невиртуальные методы могут быть объявлены подставляемыми (inline), при этом соответствующая подпрограмма не вызывается, а явно включается в код на манер макроподстановки.  Для управления виртуальными функциями в C++ используется концепция vtable (виртуальных таблиц), которые формируются для каждого объекта при его создании (то есть когда класс объекта уже известен). Такая таблица содержит список указателей на виртуальные функции. Например, при создании объекта класса **Rectangle**виртуальная таблица будет содержать запись для виртуальной функции **draw**, содержащую указатель на ближайшую в иерархии реализацию функции **draw**. Если в классе **DisplayItem** есть виртуальная функция **rotate**, которая в классе **Rectangle** не переопределена, то соответствующий указатель для **rotate** останется связан с классом**DisplayItem**. Во время исполнения программы происходит косвенное обращение через соответствующий указатель и сразу выполняется нужный код без всякого поиска [[34](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#3.34)].  Операция **draw** в классе **SolidRectangle** представляет собой особый случай и в языке C++. Чтобы вызвать метод draw из суперкласса, применяется специальный префикс, указывающий на место определения функции. Это выглядит следующим образом:  **Rectangle::Draw();**  Исследование Страуструпа показало, что вызов виртуальной функции по эффективности мало уступает вызову обычной функции [[35](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#3.35)]. Для одиночного наследования вызов виртуальной функции требует дополнительно выполнения трех-четырех операций доступа к памяти по отношению к обычному вызову; при множественном наследовании число таких дополнительных операций составляет пять или шесть.  Существенно сложнее выполняется поиск нужных функций в языке CLOS, здесь используются дополнительные квалификаторы: : before, : after, : around. Наличие множественного полиморфизма еще более усложняет проблему. При вызове метода в языке CLOS, как правило, реализуется следующий алгоритм:  Определяется тип аргументов.  Вычисляется множество допустимых методов.  Методы сортируются в направлении от наиболее специализированных к более общим.  Выполняются вызовы всех методов с квалификатором : before.  Выполняется вызов наиболее специализированного первичного метода.  Выполняются вызовы всех методов с квалификаторами : after.  Возвращается значение первичного метода [[36](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#3.36)].  В CLOS есть протокол для метаобъектов, который позволяет переопределять в том числе и этот алгоритм. На практике, однако, мало кто заходит так далеко. Как справедливо отметили Винстон и Хорн: "Алгоритмы, используемые в языке CLOS, сложны, и даже кудесники программирования стараются не вникать в их детали, так же как физики предпочитают иметь дело с механикой Ньютона, а не с квантовой механикой" [[37](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#3.37)]. |   Параллелей между типизацией и наследованием следует ожидать там, где иерархия общего и частного предназначена для выражения смысловых связей между абстракциями. Рассмотрим снова объявления в C++:  **TelenetryData telemetry;**  **ElectrycalData electrical(5.0, -5.0, 3.0, 7.0);**  Следующий оператор присваивания правомочен:  **telemetry = electrical; //electrical - это подтип telemetry**  Хотя он формально правилен, он опасен: любые дополнения в состоянии подкласса по сравнению с состоянием суперкласса просто срезаются. Таким образом, дополнительные четыре параметра, определенные в подклассе**electrical**, будут потеряны при копировании, поскольку их просто некуда записать в объекте **telemetry** клacca**TelemetryData**.  Следующий оператор неправилен:  **electrical = telemetry; //неправильно: telemetry - это не подтип electrical**  Можно сделать заключение, что присвоение объекту y значения объекта x допустимо, если тип объекта x совпадает с типом объекта y или является его подтипом.  В большинстве строго типизированных языков программирования допускается преобразование значений из одного типа в другой, но только в тех случаях, когда между двумя типами существует отношение класс/подкласс. В языке C++ есть оператор явного преобразования, называемый приведением типов. Как правило, такие преобразования используются по отношению к объекту специализированного класса, чтобы присвоить его значение объекту более общего класса. Такое приведение типов считается безопасным, поскольку во время компиляции осуществляется семантический контроль. Иногда необходимы операции приведения объектов более общего класса к специализированным классам. Эти операции не являются надежными с точки зрения строгой типизации, так как во время выполнения программы может возникнуть несоответствие (несовместимость) приводимого объекта с новым типом [Новейшие усовершенствования C++, направленные на динамическое определение типа, смягчили эту проблему]. Однако такие преобразования достаточно часто используются в тех случаях, когда программист хорошо представляет себе все типы объектов. Например, если нет параметризованных типов, часто создаются классы **set** или **bag**, представляющие собой наборы произвольных объектов. Их определяют для некоторого базового класса (это гораздо безопаснее, чем использовать идиому void\*, как мы делали, определяя класс **Queue**). Итерационные операции, определенные для такого класса, умеют возвращать только объекты этого базового класса. Внутри конкретного приложения разработчик может использовать этот класс, создавая объекты только какого-то специализированного подкласса, и, зная, что именно он собирается помещать в этот класс, может написать соответствующий преобразователь. Но вся эта стройная конструкция рухнет во время выполнения, если в наборе встретится какой-либо объект неожиданного типа.  Большинство сильно типизированных языков позволяют приложениям оптимизировать технику вызова методов, зачастую сводя пересылку сообщения к простому вызову процедуры. Если, как в C++, иерархия типов совпадает с иерархией классов, такая оптимизация очевидна. Но у нее есть недостатки. Изменение структуры или поведения какого-нибудь суперкласса может поставить вне закона его подклассы. Вот что об этом пишет Микаллеф: "Если правила образования типов основаны на наследовании и мы переделываем какой-нибудь класс так, что меняется его положение в иерархии наследования, клиенты этого класса могут оказаться вне закона с точки зрения типов, несмотря на то, что внешний интерфейс класса остается прежним" [[38](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#3.38)].  Тем самым мы подходим к фундаментальным вопросам наследования. Как было сказано выше, наследование используется в связи с тем, что у объектов есть что-то общее или между ними есть смысловая ассоциация. Выражая ту же мысль иными словами, Снайдерс пишет: "наследование можно рассматривать, как способ управления повторным использованием программ, то есть, как простое решение разработчика о заимствовании полезного кода. В этом случае механика наследования должна быть гибкой и легко перестраиваемой. Другая точка зрения: наследование отражает принципиальную родственность абстракций, которую невозможно отменить" [[39](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#3.39)]. В Smalltalk и CLOS эти два аспекта неразделимы. C++ более гибок. В частности, при определении класса его суперкласс можно объявить **public** (как**ElectricalData** в нашем примере). В этом случае подкласс считается также и подтипом, то есть обязуется выполнять все обязательства суперкласса, в частности обеспечивая совместимое с суперклассом подмножество интерфейса и обладая неразличимым с точки зрения клиентов суперкласса поведением. Но если при определении класса объявить его суперкласс как **private**, это будет означать, что, наследуя структуру и поведение суперкласса, подкласс уже не будет его подтипом [Мы можем также объявить суперкласс защищенным, что даст ту же семантику, что и в случае закрытого суперкласса, но открытые и защищенные элементы такого суперкласса будут доступны подклассам]. Это означает, что открытые и защищенные члены суперкласса станут закрытыми членами подкласса, и следовательно они будут недоступны подклассам более низкого уровня. Кроме того, тот факт, что подкласс не будет подтипом, означает, что класс и суперкласс обладают несовместимыми (вообще говоря) интерфейсами с точки зрения клиента. Определим новый класс:  **class InternalElectricalData: private ElectricalData {**  **public:**  **InternalElectricalData(float v1, float v2, float a1, float a2);**  **virtual ~InternalElectricalData();**  **ElectricalData::currentPower;**  **};**  Здесь суперкласс **ElectricalData** объявлен закрытым. Следовательно, его методы, такие, например, как **transmit**, недоступны клиентам. Поскольку класс **InternalElectricalData** не является подтипом **ElectricalData**, мы уже не сможем присваивать экземпляры подкласса объектам суперкласса, как в случае объявления суперкласса в качестве открытого. Отметим, что функция **currentPower** сделана видимой путем ее явной квалификации. Иначе она осталась бы закрытой. Как можно было ожидать, правила C++ запрещают делать унаследованный элемент в подклассе "более открытым", чем в суперклассе. Так, член **timeStamp**, объявленный в классе **TelemetryData** защищенным, не может быть сделан в подклассе открытым путем явного упоминания (как это было сделано для функции **currentpower**).  В языке Ada для достижения аналогичного эффекта вместо подтипов используется механизм производных типов. Определение подтипа означает не появление нового типа, а лишь ограничение существующего. А вот определение производного типа создает самостоятельный новый тип, который имеет структуру, заимствованную у исходного типа.  В следующем разделе мы покажем, что наследование с целью повторного использования и агрегация до некоторой степени противоречат друг другу.  **Множественное наследование.** Мы рассмотрели вопросы, связанные с одиночным наследованием, то есть, когда подкласс имеет ровно один суперкласс. Однако, как указали Влиссидес и Линтон: "одиночное наследование при всей своей полезности часто заставляет программиста выбирать между двумя равно привлекательными классами. Это ограничивает возможность повторного использования предопределенных классов и заставляет дублировать уже имеющиеся коды. Например, нельзя унаследовать графический элемент, который был бы одновременно окружностью и картинкой; приходится наследовать что-то одно и добавлять необходимое от второго" [[40](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#3.40)].  Множественное наследование прямо поддерживается в языках C++ и CLOS, а также, до некоторой степени, в Smalltalk. Необходимость множественного наследования в OOP остается предметом горячих споров. По нашему опыту, множественное наследование - как парашют: как правило, он не нужен, но, когда вдруг он понадобится, будет жаль, если его не окажется под рукой.  Представьте себе, что нам надо организовать учет различных видов материального и нематериального имущества - банковских счетов, недвижимости, акций и облигаций. Банковские счета бывают текущие и сберегательные. Акции и облигации можно отнести к ценным бумагам, управление ими совершенно отлично от банковских счетов, но и счета и ценные бумаги - это разновидности имущества.  Однако есть много других полезных классификаций тех же видов имущества. В каком-то контексте может потребоваться отличать то, что можно застраховать (недвижимость и, до некоторой степени, сберегательные вклады). Другой аспект - способность имущества приносить дивиденды; это общее свойство банковских счетов и ценных бумаг.  Очевидно, одиночное наследование в данном случае не отражает реальности, так что придется прибегнуть к множественному [В действительности, это - "лакмусовая бумажка" для множественного наследования. Если мы составим структуру классов, в которой конечные классы (листья) могут быть сгруппированы в множества по разным ортогональным признакам (как в нашем примере, где такими признаками были способность приносить дивиденды и возможность страховки) и эти множества перекрываются, то это служит признаком невозможности обойтись одной структурой наследования, в которой бы существовали какие-то промежуточные классы с нужным поведением. Мы можем исправить ситуацию, используя множественное наследование, чтобы соединить два нужных поведения там, где это необходимо]. Получившаяся структура классов показана на рис. 3-7. На нем класс **Security** (ценные бумаги) наследует одновременно от классов **InterestBearingItem** (источник дивидендов) и **Asset** (имущество). Сходным образом, **BankAccount** (банковский счет) наследует сразу от трех классов: **InsurableItem** (страхуемое) и уже известным **Asset** и **InterestBearingItem**.  Вот как это выражается на C++. Сначала базовые классы:  **class Asset ...  class InsurableItem ...  class InterestBearingItem ...**  Теперь промежуточные классы; каждый наследует от нескольких суперклассов:  **class BankAccount: public Asset, public InsurableItem, public InterestBearingItem ...  class RealEstate: public Asset, public InsurableItem ...  class Security: public Asset, public InterestBearingItem ...**  Наконец, листья:  **class SavingsAccount: public BankAccount ...  class CheckingAccount: public BankAccount ...  class Stock: public Security ...  class Bond: public Security ...**  http://www.helloworld.ru/texts/comp/other/oop/pic03_07.gif  *Рис. 3-7. Множественное наследование.*  Проектирование структур классов со множественным наследованием - трудная задача, решаемая путем последовательных приближений. Есть две специфические для множественного наследования проблемы - как разрешить конфликты имен между суперклассами и что делать с повторным наследованием.  Конфликт имен происходит, когда в двух или более суперклассах случайно оказывается элемент (переменная или метод) с одинаковым именем. Представьте себе, что как **Asset**, так и **InsurableItem** содержат атрибут **presentValue**, обозначающий текущую стоимость. Так как класс **RealEstate** наследует обоим этим классам, как понимать наследование двух операций с одним и тем же именем? Это, на самом деле, главная беда множественного наследования: конфликт имен может ввести двусмысленность в поведение класса с несколькими предками.  Борются с этим конфликтом тремя способами. Во-первых, можно считать конфликт имен ошибкой и отвергать его при компиляции (так делают Smalltalk и Eiffel, хотя в Eiffel конфликт можно разрешить, исправив имя). Во-вторых, можно считать, что одинаковые имена означают одинаковый атрибут (так делает CLOS). В третьих, для устранения конфликта разрешается добавить к именам префиксы, указывающие имена классов, откуда они пришли. Такой подход принят в C++ [В C++ конфликт имен элементов подкласса может быть разрешен полной квалификацией имени члена класса. Функции-члены с одинаковыми именами и сигнатурами семантическими считаются идентичными].  О второй проблеме, повторном наследовании, Мейер пишет следующее: "Одно тонкое затруднение при использовании множественного наследования встречается, когда один класс является наследником другого по нескольким линиям. Если в языке разрешено множественное наследование, рано или поздно кто-нибудь напишет класс D, который наследует от B и C, которые, в свою очередь, наследуют от A. Эта ситуация называется повторным наследованием, и с ней нужно корректно обращаться" [[41](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#3.41)]. Рассмотрим следующий класс:  **class MutualFund: public Stock,** **public Bond ...**  который дважды наследует от класса **security**.  Проблема повторного наследования решается тремя способами. Во-первых, можно его запретить, отслеживая при компиляции. Так сделано в языках Smalltalk и Eiffel (но в Eiffel, опять-таки допускается переименование для устранения неопределенности). Во-вторых, можно явно развести две копии унаследованного элемента, добавляя к именам префиксы в виде имени класса-источника (это один из подходов, принятых в C++). В-третьих, можно рассматривать множественные ссылки на один и тот же класс, как обозначающие один и тот же класс. Так поступают в C++, где повторяющийся суперкласс определяется как виртуальный базовый класс. Виртуальный базовый класс появляется, когда какой-либо подкласс именует другой класс своим суперклассом и отмечает этот суперкласс как виртуальный, чтобы показать, что это - общий (shared) класс. Аналогично, в языке CLOS повторно наследуемые классы "обобществляются" с использованием механизма, называемого список следования классов. Этот список заводят для каждого нового класса, помещая в него сам этот класс и все его суперклассы без повторений на основе следующих правил:  класс всегда предшествует своему суперклассу;  каждый класс сам определяет порядок следования своих непосредственных родителей.  В результате граф наследования оказывается плоским, дублирование устраняется, и появляется возможность рассматривать результирующую иерархию как иерархию с одиночным наследованием [[43](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#3.43)]. Это весьма напоминает топологическую сортировку классов. Если она возможна, то повторное наследование допускается. При этом теоретически могут существовать несколько равноправных результатов сортировки, но алгоритм так или иначе выдает какой-то один из них. Если же сортировка невозможна (например, в структуре возникают циклы), то класс отвергается.  При множественном наследовании часто используется прием создания примесей (**mixin**). Идея примесей происходит из языка Flavors: можно комбинировать (смешивать) небольшие классы, чтобы строить классы с более сложным поведением. Хендлер пишет об этом так: "примесь синтаксически ничем не отличается от класса, но назначение их разное. Примесь не предназначена для порождения самостоятельно используемых экземпляров - она смешивается с другими классами" [[44](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#3.44)]. На рис. 3-7 классы **InsurableItem** и **interestBearingItem** - это примеси. Ни один из них не может существовать сам по себе, они используются для придания смысла другим классам [Для языка CLOS при обогащении поведения существующих первичных методов обычной практикой является строить примесь, используя только **:before-** и **:after**-методы]. Таким образом, примесь - это класс, выражающий не поведение, а одну какую-то хорошо определенную повадку, которую можно привить другим классам через наследование. При этом повадка эта обычно ортогональна собственному поведению наследующего ее класса. Классы, сконструированные целиком из примесей, называют агрегатными.  **Множественный полиморфизм.** Вернемся к одной из функций-членов класса **DisplayItem**:  **virtual void draw();**  Эта операция изображает объект на экране в некотором контексте. Она объявлена виртуальной, то есть полиморфной, переопределяемой подклассами. Когда эту операцию вызывают для какого-то объекта, программа определяет, что, собственно, выполнять (см. врезку выше). Это одиночный полиморфизм в том смысле, что смысл сообщения зависит только от одного параметра, а именно, объекта, для которого вызывается операция.  На самом деле операция **draw** должна бы зависеть от характеристик используемой системы отображения, в частности от графического режима. Например, в одном случае мы хотим получить изображение с высоким разрешением, а в другом - быстро получить черновое изображение. Можно ввести две различных операции, скажем, **drawGraphic** и**drawText**, но это не совсем то, что хотелось бы. Дело в том, что каждый раз, когда требуется учесть новый вид устройства, его надо проводить по всей иерархии надклассов для класса **DisplayItem**.  В CLOS есть так называемые мультиметоды. Они полиморфны, то есть их смысл зависит от множества параметров (например, от графического режима и от объекта). В C++ мультиметодов нет, поэтому там используется идиома так называемый двойной диспетчеризации.  Например, мы могли бы вести иерархию устройств отображения информации от базового класса **DisplayDevice**, а атем определить метод класса **DisplayItem** так:  **virtual void draw(DisplayDevice&);**  При реализации этого метода мы вызываем графические операции, которые полиморфны относительно переданного параметра типа **DisplayItem**, таким образом происходит двойная диспетчеризация: **draw** сначала демонстрирует полиморфное поведение в зависимости от того, к какому подклассу класса **DisplayItem** принадлежит объект, а затем полиморфизм проявляется в зависимости от того, к какому подклассу класса **DisplayDevice** принадлежит аргумент. Эту идиому можно продолжить до множественной диспетчеризации.  **Агрегация**  **Пример.** Отношение агрегации между классами имеет непосредственное отношение к агрегации между их экземплярами. Рассмотрим вновь класс **TemperatureController**:  **class TemperatureController {**  **public:**  **TemperatureController(Location);**  **~TemratureController();**  **void process(const TemperatureRamp&);**  **Minute schedule(const TemperatureRamp&) const;**  **private:**  **Heater h;**  **};**   http://www.helloworld.ru/texts/comp/other/oop/pic03_08.gif  *Рис. 3-8. Агрегация.*  Как явствует из рис. 3-8, класс **TemperatureController** это, несомненно, целое, а экземпляр класса **Heater** - одна из его частей. Совершенно такое же отношение агрегации между экземплярами этих классов показано на рис. 3-3.  **Физическое включение.** В случае класса **TemperatureController** мы имеем агрегацию по значению; эта разновидность физического включения означает, что объект класса **Heater** не существует отдельно от объемлющего экземпляра класса **TemperatureController**.  Менее обязывающим является включение по ссылке. Мы могли бы изменить закрытую часть **TemperatureController**так [В качестве альтернативы мы могли бы описать **h** как ссылку на нагреватель (**Heater&** в C++), в этом случае семантика инициализации и модификации этого объекта будет совершенно отличной от семантики указателей]:  **Heater\* h;**  В этом случае класс **TemperatureController** по-прежнему означает целое, но его часть, экземпляр класса **Heater**, содержится в целом косвенно. Теперь эти объекты живут отдельно друг от друга: мы можем создавать и уничтожать экземпляры классов независимо. Чтобы избежать структурной зависимости через ссылки важно придерживаться какой-то договоренности относительно создания и уничтожения объектов, ссылки на которые могут содержаться в разных местах. Нужно, чтобы это делал кто-то один.  Агрегация является направленной, как и всякое отношение "целое/часть". Объект **Heater** входит в объект**TemperatureController**, и не наоборот. Физическое вхождение одного в другое нельзя "зациклить", а вот указатели - можно (каждый из двух объектов может содержать указатель на другой).  Конечно, как уже говорилось, агрегация не требует обязательного физического включения, ни по значению, ни по ссылке. Например, акционер владеет акциями, но они не являются его физической частью. Более того, время жизни этих объектов может быть совершенно различным, хотя концептуально отношение целого и части сохраняется и каждая акция входит в имущество своего акционера. Поэтому агрегация может быть очень косвенной. Например, объект класса**Shareholder** (акционер) может содержать ключ записи об этом акционере в базе данных акций. Это тоже агрегация без физического включения. "Лакмусовая бумажка" для выявления агрегации такова: если (и только если) налицо отношение "целое/часть" между объектами, их классы должны находиться в отношении агрегации друг с другом.   http://www.helloworld.ru/texts/comp/other/oop/pic03_09.gif  *Рис. 3-9. Отношение использования.*  Часто агрегацию путают с множественным наследованием. Действительно, в C++ скрытое (защищенное или закрытое) наследование почти всегда можно заменить скрытой агрегацией экземпляра суперкласса. Решая, с чем вы имеете дело - с наследованием или агрегацией - будьте осторожны. Если вы не уверены, что налицо отношение общего и частного (is а), вместо наследования лучше применить агрегацию или что-нибудь еще.  **Использование**  **Пример.** В недавнем примере объекты **rampController** и **growingRamp** иллюстрировали связь между объектами, которую мы представляли в виде отношения использования между их классами **TemperatureController** и**TemperatureRamp**.  **class TemperatureController {**  **public:**  **TemperatureController(Location);**  **~TemperatureController();**  **void process(const TemperatureRamp&);**  **Minute schedule(const TemperatureRamp&) const;**  **private:**  **Heater h;**  **};**  Класс **TemperatureRamp** упомянут как часть сигнатуры функции-члена process; это дает нам основания сказать, что класс **TemperatureController** пользуется услугами класса **TemperatureRamp**.  Клиенты и серверы. Отношение использования между классами соответствует равноправной связи между их экземплярами. Это то, во что превращается ассоциация, если оказывается, что одна из ее сторон (клиент) пользуется услугами другой (сервера). Пример клиент-серверных отношений показан на рис. 3-9.  На самом деле, один класс может использовать другой по-разному. В нашем примере это происходит в сигнатуре интерфейсной функции. Можно представить, что **TemperatureController** внутри реализации функции **schedule**использует, например, экземпляр класса **Predictor** (предсказатель). Отношения целого и части тут ни при чем, поскольку этот объект не входит в объект **TemperatureController**, а только используется. В типичном случае такое отношение использования проявляет себя, если в реализации какой-либо операции происходит объявление локального объекта используемого класса.  Строгое отношение использования иногда несколько ограничительно, поскольку клиент имеет доступ только к открытой части интерфейса сервера. Иногда по тактическим соображениям мы должны нарушить инкапсуляцию, для чего, собственно, и служат "дружеские" отношения классов в C++.  **Инстанцирование**  **Примеры.** Наша первая попытка сконструировать класс **Queue** (очередь) была не особенно успешной, поскольку нам не удалось сделать его безопасным в отношении типов. Мы можем значительно усовершенствовать нашу абстракцию, если прибегнем к конструкции параметризованных классов, которая поддерживается языками C++ и Eiffel.  **Template<class Item>**  **class Queue {**  **public:**  **Queue();**  **Queue(const Queue<Item>&);**  **virtual ~Queue();**  **virtual Queue<Item>& operator=(const Queue<Item>&);**  **virtual int operator==(const Queue<Item>&) const;**  **int operator!=(const Queue<Item>&) const;**  **virtual void clear();**  **virtual void append(const Item&);**  **virtual void pop();**  **virtual void remove(int at);**  **virtual int length() const;**  **virtual int isEmpty() const;**  **virtual const Item& front() const;**  **virtual int location(const void\*);**  **protected:**  **...**  **};**  В этом новом варианте не используется идиома void\*, вместо этого объекты помещаются в очередь и достаются из нее через класс **item**, объявленный как аргумент шаблона.  Параметризованный класс не может иметь экземпляров, пока он не будет инстанцирован. Объявим две конкретных очереди - очередь целых чисел и очередь экранных объектов:  **Queue<int> intQueue;**  **Queue<DisplayItem\*> itemQueue;**  Объекты **intQueue** и **itemQueue** - это экземпляры совершенно различных классов, которые даже не имеют общего суперкласса. Тем не менее, они получены из одного параметризованного класса **Queue**. По причинам, которые мы объясним позже в главе 9, во втором случае мы поместили в очередь указатели. Благодаря этому, любые объекты подклассов **DisplayItem**, помещенные в очередь, не будут "срезаться", но сохранят свое полиморфное поведение.   http://www.helloworld.ru/texts/comp/other/oop/pic03_10.gif  *Рис. 3-10. Инстанцирование.*  Это инстанцирование безопасно с точки зрения типов. По правилам C++ будет отвергнута любая попытка поместить в очередь или извлечь из нее что-либо кроме, соответственно, целых чисел и разновидностей **DisplayItem**.  Отношения между параметризованным классом **Queue**, его инстанцированием для класса **DisplayItem** и экземпляром**itemQueue** показаны на рис. 3-10.  **Обобщенные классы.** Существует четыре основных способа создавать такие классы, как параметризованный класс**Queue**. Во-первых, мы можем использовать макроопределения. Именно так это было в раннем C++, но, как пишет Страуструп, "данный подход годился только для небольших проектов" [[45](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#3.45)], так как макросы неуклюжи и находятся вне семантики языка, более того, при каждом инстанцировании создается новая копия программного кода. Во-вторых, можно положиться на позднее связывание и наследование, как это делается в Smalltalk [[46](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#3.46)]. При таком подходе мы можем строить только неоднородные контейнерные классы, так как в языке нет средства ввести нужный класс элементов контейнера; каждый элемент в контейнере трактуется как экземпляр некоторого удаленного базового класса. Третий способ реализован в языках семейства Object Pascal, которые имеют и сильные типы, и наследование, но не поддерживают никакой разновидности параметризованных классов. В этом случае приходится создавать обобщенные контейнеры, как в Smalltalk, но использовать явную проверку типа объекта, прежде чем помещать его в контейнер. Наконец, есть собственно параметризованные классы, впервые появившиеся в CLU. Параметризованный класс представляет собой что-то вроде шаблона для построения других классов; шаблон может быть параметризован другими классами, объектами или операциями. Параметризованный класс должен быть инстанцирован перед созданием экземпляров. Механизм обобщенных классов есть в C++ и Eiffel.  Как можно заметить из рис. 3-10, чтобы инстанцировать параметризованный класс Queue мы должны использовать другой класс, например, **DisplayItem**. Благодаря этому отношение инстанцирования почти всегда подразумевает отношение использования.  Мейер указывает, что наследование - более мощный механизм, чем обобщенные классы и что через наследование можно получить большинство преимуществ обобщенных классов, но не наоборот [[47](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#3.47)]. Нам кажется, что лучше, когда языки поддерживают и то, и другое.  Параметризованные классы полезны далеко не только для создания контейнеров. Например, Страуструп отмечает их значение для обобщенной арифметики [[48](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#3.48)].  При проектировании обобщенные классы позволяют выразить некоторые свойства протоколов классов. Класс экспортирует операции, которые можно выполнять над его экземплярами. Наоборот, параметризующий аргумент класса служит для импорта классов и значений, предоставляющих некоторый протокол. C++ проверяет их взаимное соответствие при компиляции, когда фактически и происходит инстанцирование. Например, мы могли бы определить упорядоченную очередь объектов, отсортированных по некоторому критерию. Этот параметризованный класс должен иметь аргумент (класс **Item**), и требовать от этого аргумента определенное поведение (наличие операции вычисления порядка). При инстанцировании в качестве класса **Item** годится любой класс, который имеет соответствующий протокол. Таким образом, поведение классов в семействе, происходящем от одного параметризованного класса, может изменяться в весьма широких пределах.  **Метаклассы**  Как было сказано, любой объект является экземпляром какого-либо класса. Что будет, если мы попробуем и с самими классами обращаться как с объектами? Для этого нам надо ответить на вопрос, что же такое класс класса? Ответ - это метакласс. Иными словами, метакласс - это класс, экземпляры которого суть классы. Метаклассы венчают объектную модель в чисто объектно-ориентированных языках. Соответственно, они есть в Smalltalk и CLOS, но не в C++.  Вот как Робсон мотивирует потребность в метаклассах: "классы доставляют программисту интерфейс для определения объектов. Если так, то желательно, чтобы и сами классы были объектами, так, чтобы ими можно было манипулировать, как всеми остальными описаниями" [[49](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#3.49)].  В языках типа Smalltalk первичное назначение метакласса - поддержка переменных класса (которые являются общими для всех экземпляров этого класса), операции инициализации переменных класса и создания единичного экземпляра метакласса [[50](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#3.50)]. По соглашению, метакласс Smalltalk обычно содержит примеры использования его классов. Например, как показано на рис. 3-11, мы могли бы задать переменную класса **nextId** для метакласса **TelemetryData**, чтобы вырабатывать идентифицирующие метки при создании каждого экземпляра **TelemetryData**. Аналогично, мы могли бы определить оператор порождения новых экземпляров класса, который изготавливал бы их, скажем, в некотором предварительно выделенном пуле памяти.  Хотя в C++ метаклассов нет, семантика его конструкторов и деструкторов служит целям, аналогичным тем, что вызвали к жизни метаклассы. C++ имеет средства поддержки и переменных класса, и операций метакласса. Конкретно, в C++ можно описать члены данных или функции класса как статические (**static**), что будет означать: этот элемент является общим для всех экземпляров класса. Статические члены класса в C++ эквивалентны переменным класса в Smalltalk. Статическая функция-член класса играет роль операций метакласса в Smalltalk.  Как мы уже отмечали, в CLOS аппарат метаклассов еще сильнее чем в Smalltalk. Через него можно изменять саму семантику элементов: следование классов, обобщенные функции и методы. Главное преимущество - возможность экспериментировать с другими объектно-ориентированными парадигмами и создавать такие инструменты для разработчика, как броузеры классов и объектов.   http://www.helloworld.ru/texts/comp/other/oop/pic03_11.gif  *Рис. 3-11. Метаклассы.*  В CLOS есть предопределенный класс с именем standard-class, который является метаклассом для всех нетипизированных классов, определенных с помощью defclass. В этом метаклассе есть метод make-instance, который создает экземпляры. Кроме того, в нем определена вся техника работы со списком следования классов. Все это можно изменить.  Методы и обобщенные функции в CLOS тоже можно рассматривать как объекты. Так как они несколько отличаются от обычных объектов, то в совокупности объекты, соответствующие классам, методам и обобщенным функциям, называются *метаобьектами*. Каждый метод является экземпляром предопределенного класса **standard-method**, а каждая функция является экземпляром предопределенного класса **standard-generic-function.** Поскольку поведение этих предопределенных классов можно изменить, удается влиять на трактовку методов и обобщенных функций.  3.5. Взаимосвязь классов и объектов.  **Отношения между классами и объектами**  Классы и объекты - это отдельные, но тесно связанные понятия. В частности, каждый объект является экземпляром какого-либо класса; класс может порождать любое число объектов. В большинстве практических случаев классы статичны, то есть все их особенности и содержание определены в процессе компиляции программы. Из этого следует, что любой созданный объект относится к строго фиксированному классу. Сами объекты, напротив, в процессе выполнения программы создаются и уничтожаются.  В качестве примера рассмотрим классы и объекты для задачи управления воздушным движением. Наиболее важные абстракции в этой сфере - самолеты, графики полетов, маршрут и коридоры в воздушном пространстве. Трактовка этих классов объектов по самому их определению достаточно статична. Иначе невозможно было бы построить никакого приложения, использующего такие общепонятные факты, как то, что самолеты могут взлетать, летать и приземляться, а также что никакие два самолета не должны находиться одновременно в одной и той же точке.  Объекты же этих классов, напротив, динамичны. Набор маршрутов полетов сменяется не очень часто. Существенно быстрее изменяется множество самолетов, находящихся в полете. Частота, с которой самолеты занимают и покидают воздушные коридоры, еще выше.  **Роль классов и объектов в анализе и проектировании**  На этапе анализа и ранних стадиях проектирования решаются две основные задачи:  Выявление классов и объектов, составляющих словарь предметной области.  Построение структур, обеспечивающих взаимодействие объектов, при котором выполняются требования задачи.  В первом случае говорят о ключевых абстракциях задачи (совокупность классов и объектов), во втором - о механизмах реализации (совокупность структур).  На ранних стадиях внимание проектировщика сосредоточивается на внешних проявлениях ключевых абстракций и механизмов. Такой подход создает логический каркас системы: структуры классов и объектов. На последующих фазах проекта, включая реализацию, внимание переключается на внутреннее поведение ключевых абстракций и механизмов, а также их физическое представление. Принимаемые в процессе проектирования решения задают архитектуру системы: и архитектуру процессов, и архитектуру модулей.  3.6. Качество классов и объектов  **Измерение качества абстракции**  По мнению Ингалса "для построения системы должен использоваться минимальный набор неизменяемых компонент; сами компоненты должны быть по возможности стандартизованы и рассматриваться в рамках единой модели" [[51](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#3.51)]. Применительно к объектно-ориентированному проектированию такими компонентами являются классы и объекты, отражающие ключевые абстракции системы, а единство обеспечивается соответствующими механизмами реализации.  Опыт показывает, что процесс выделения классов и объектов является последовательным, итеративным. За исключением самых простых задач с первого раза не удается окончательно выделить и описать классы. В главах 4 и 7 показано, как в процессе работы сглаживаются противоречия, возникающие при начальном определении абстракций. Очевидно, такой процесс связан с дополнительными затратами на перекомпиляцию, согласование и внесение изменений в проект системы. Очень важно, следовательно, с самого начала по возможности приблизиться к правильным решениям, чтобы сократить число последующих шагов приближения к истине. Для оценки качества классов и объектов, выделяемых в системе, можно предложить следующие пять критериев:  зацепление;  связность;  достаточность;  полнота;  примитивность.  Термин *зацепление* взят из структурного проектирования, но в более вольном толковании он используется и в объектно-ориентированном проектировании. Стивенс, Майерс и Константайн определяют зацепление как "степень глубины связей между отдельными модулями. Систему с сильной зависимостью между модулями гораздо сложнее воспринимать и модифицировать. Сложность системы может быть уменьшена путем уменьшения зацепления между отдельными модулями" [[52](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#3.52)]. Пример неправильного подхода к проблеме зацепления привел Пейдж-Джонс, описав модульную стереосистему, в которой источник питания размещен в одной из звуковых колонок [[53](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#3.53)].  Кроме зацепления между модулями в объектно-ориентированном анализе, существенно зацепление между классами и объектами. Существует определенное противоречие между явлениями зацепления и наследования. С одной стороны, желательно избегать сильного зацепления классов; с другой стороны, механизм наследования, тесно связывающий подклассы с суперклассами, помогает выгодно использовать сходство абстракций.  Понятие связности также заимствовано из структурного проектирования. Связность - это степень взаимодействия между элементами отдельного модуля (а для OOD еще и отдельного класса или объекта), характеристика его насыщенности. Наименее желательной является связность по случайному принципу, когда в одном классе или модуле собираются совершенно независимые абстракции. Для примера можно вообразить класс, соединяющий абстракции собак и космических аппаратов. Наиболее желательной является функциональная связность, при которой все элементы класса или модуля тесно взаимодействуют в достижении определенной цели. Так, например, класс **Dog** будет функционально связным, если он описывает поведение собаки, всей собаки, и ничего, кроме собаки.  К идеям зацепления и связности тесно примыкают понятия достаточности, полноты и примитивности. Под достаточностью подразумевается наличие в классе или модуле всего необходимого для реализации логичного и эффективного поведения. Иначе говоря, компоненты должны быть полностью пригодны к использованию. Для примера рассмотрим класс **set** (множество). Операция удаления элемента из множества в этом классе, очевидно, необходима, но будет ошибкой не включить в этот класс и операцию добавления элемента. Нарушение требования достаточности обнаруживается очень быстро, как только создается клиент, использующий абстракцию. Под полнотой подразумевается наличие в интерфейсной части класса всех характеристик абстракции. Идея достаточности предъявляет к интерфейсу минимальные требования, а идея полноты охватывает все аспекты абстракции. Полнотой характеризуется такой класс или модуль, интерфейс которого гарантирует все для взаимодействия с пользователями. Полнота является субъективным фактором, и разработчики часто ею злоупотребляют, вынося на верх такие операции, которые можно реализовать на более низком уровне. Из этого вытекает требование примитивности. Примитивными являются только такие операции, которые требуют доступа к внутренней реализации абстракции. Так, в примере с классом **set** операция Add (добавление к множеству элемента) примитивна, а операция добавления четырех элементов не будет примитивной, так как вполне эффективно реализуется через операцию добавления одного элемента. Конечно, эффективность тоже вещь субъективная. Операция, которая требует прямого доступа к структуре данных, примитивна по определению. Операция, которая может быть описана в терминах существующих примитивных операций, но ценой значительно больших вычислительных затрат, также является кандидатом на включение в разряд примитивных [Примером может служить операция добавления к множеству произвольного числа элементов (а не обязательно четырех). - Примеч. ред.].  **Как выбрать операции?**  **Функциональность.** Описание интерфейса класса или модуля - трудная работа. Обычно первое приближение делается, исходя из структурного смысла класса, а затем, когда появляются клиенты класса, интерфейс уточняется, модифицируется и дополняется. В частности может возникнуть потребность в создании новых классов или в изменении взаимодействия существующих.  В пределах каждого класса принято иметь только примитивные операции, отражающие отдельные аспекты поведения. Такие методы называются точными. Принято также отделять методы, не связанные между собой. Это облегчает образование подклассов с переопределением поведения. Решение о количестве методов может быть обусловлено двумя причинами: описание поведения в одном методе упрощает интерфейс, но усложняет и увеличивает размеры самого метода; расщепление метода усложняет интерфейс, но делает каждый из методов проще. По наблюдению Мейера "хороший проектировщик умеет найти компромисс между большим числом связей (дробление системы на фрагменты) и большим размером модулей (что может привести к потере управляемости)" [[54](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#3.54)].  В объектно-ориентированном проектировании принято рассматривать методы класса как единое целое, поскольку все они взаимодействуют друг с другом для реализации протокола абстракции. Таким образом, определив поведение, нужно решить, в каком из классов это поведение реализуется. Халберт и O'Брайен предложили следующие критерии для принятия такого решения:    |  |  | | --- | --- | | ® Повторная используемость | Будет ли это поведение полезно более чем в одном контексте? | | ® Сложность | Насколько трудно реализовать такое поведение? | | ® Применимость | Насколько данное поведение характерно для класса, в который мы хотим включить поведение? | | ® Знание реализации | Надо ли для реализации данного поведения знать секреты класса? |   Обычно операции объявляются, как методы класса, к объектам которого относятся данные действия. Однако в языках Object Pascal, C++, CLOS и Ada допускается описание операций в виде свободных подпрограмм (утилит класса). Свободная подпрограмма, в терминологии C++, - это функция, не являющаяся элементом класса. Свободные подпрограммы не могут переопределяться подобно обычным методам, в них нет такой общности. Наличие утилит позволяет выполнить требование примитивности и уменьшить зацепление между классами, особенно если эти операции высокого уровня задействуют объекты многих различных классов.  **Аспекты расхода памяти и времени.** После того, как мы приняли решение о необходимости конкретной функции и определили ее семантику, следует принять решение об использовании ею времени и памяти. Для выражения таких решений принято использовать понятие лучшего, среднего и худшего вариантов, где худший - это верхний допустимый предел расходов.  Раньше мы уже отмечали, что поскольку один объект посылает другому сообщение, эти два объекта должны быть каким-то образом синхронизированы. В случае многих потоков управления это означает, что передача сообщений сложнее, чем управление вызовами подпрограмм. Для большинства языков программирования синхронизация просто не нужна, поскольку в них программы однопотоковые, и все объекты действуют последовательно. Мы говорим в таких случаях о простой передаче сообщений, так как ее семантика больше похожа на простой вызов подпрограмм. Однако в языках, поддерживающих параллелизм [Ada и Smalltalk имеют прямую поддержку параллельности. Языки типа C++ такой поддержкой не обладают, но в них часто можно обеспечить семантику параллельности за счет расширения классами (зависящими от платформы): примером служит библиотека AT&T для C++], нужно побеспокоиться о более изощренных системах передачи сообщений, чтобы избежать случаев, когда два потока работают одновременно и несогласованно с одним и тем же объектом. Объекты, семантика которых сохраняется при многопоточности, являются или синхронизированными, или защищенными.  В некоторых обстоятельствах полезно отмечать параллельность как для отдельных операций, так и для объекта в целом, так как разные операции могут потребовать разных форм синхронизации. Выделяют следующие формы передачи сообщений:    |  |  | | --- | --- | | ® Синхронная | Операция активизируется только при готовности передающего и принимающего сообщения объектов; ожидание взаимной готовности может быть неопределенно долгим. | | ® С учетом задержки | То же, что и синхронная, однако, в случае, если принимающий не готов, передающий не выполняет операцию. | | ® С ограничением времени | То же, что и синхронная, однако, посылающий будет ждать готовности принимающего не дольше некоторого времени. | | ® Асинхронная | Операция выполняется вне зависимости от готовности принимающего. |   Нужная форма выбирается для каждой операции отдельно, но только после того, как ее функциональная семантика определена.  **Как выбирать отношения**  **Сотрудничество.** Отношения между классами и объектами связаны с конкретными действиями. Если мы хотим, чтобы объект X послал объекту Y сообщение M, то прямо или косвенно класс X должен иметь доступ к классу Y, иначе невозможно вызвать в классе X операцию M. Под доступностью мы понимаем способность одной абстракции видеть другую и обращаться к ее открытым ресурсам. Абстракции доступны одна другой только тогда, когда перекрываются их области видимости и даны необходимые права доступа (так, закрытая часть класса доступна только ему самому и его друзьям). Таким образом, зацепление связано с видимостью.  Одним из полезных правил является закон Деметера, который утверждает, что "методы любого класса не должны зависеть от структуры других классов, а только от структуры (верхнего уровня) самого класса. В каждом методе посылаются сообщения только объектам из предельно ограниченного множества классов" [[56](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#3.56)]. Следование этому закону позволяет создавать слабо зацепленные классы, реализация которых скрыта. Такие классы достаточно автономны и для понимания их логики нет необходимости знать строение других классов.  При анализе структуры классов системы в целом можно обнаружить, что иерархия наследования либо широкая и мелкая, либо узкая и глубокая, либо сбалансированная. В первом случае структура классов выглядит как лес из свободно стоящих деревьев. Классы могут свободно смешиваться и вступать во взаимоотношения [[57](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#3.57)]. Во втором случае структура классов напоминает одно дерево с ветвями классов, имеющих общего предка [[58](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#3.58)]. Каждый из вариантов имеет свои достоинства и недостатки. Классы, составляющие лес, независимы друг от друга, но, вероятно, не лучшим образом используют возможности специализации и обобществления кода. В случае дерева классов эта "коммунальность" используется максимально, поэтому каждый из классов имеет меньший размер. Однако в последнем случае классы невозможно понять без контекста всех их предков.  Иногда требуется выбирать между отношениями наследования, агрегации и использования. Например, должен ли класс**Car** (автомобиль) наследовать, содержать или использовать классы **Engine** (двигатель) и **wheel** (колесо)? В данном случае более целесообразны отношения использования. По мнению Мейера, между классами A и B "отношения наследования более пригодны тогда, когда любой объект класса B может одновременно рассматриваться и как объект A" [[59](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#3.59)]. С другой стороны, если объект является чем-то большим, чем сумма его частей, то отношение агрегации не совсем уместно.  **Механизмы и видимость.** Отношения между объектами определяется в основном механизмами их взаимодействия. Вопрос состоит только в том, кто о чем должен знать. Например, на ткацкой фабрике материалы (партии) поступают на участки для обработки. Как только они попадают на участок, об этом надо известить управляющего. Является ли поступление материала на участок операцией над участком, над материалом, или тем и другим сразу? Если это операция над участком, то класс участка должен быть видим для материала. Если это операция над материалом, то класс материала должен быть видим для участка, так как партия материала должна различать участки. В случае операции над помещением и участком нужно обеспечить взаимную видимость. Аналогично следует определить отношение между управляющим и участком (но не материалом и управляющим): либо управляющий должен знать об участке, либо участок об управляющем.  Иногда в процессе проектирования полезно явно определить видимость объектов. Существуют четыре основных способа сделать так, чтобы объект X (клиент) видел объект Y (сервер):  сервер является глобальным;  сервер передается клиенту в качестве параметра операции;  сервер является частью клиента в смысле классов;  сервер локально объявляется в области видимости клиента.  Эти варианты можно комбинировать. Y может быть частью X и при этом быть видимым другим объектам. В языке Smalltalk такой способ обычно означает зависимость между двумя объектами. Общая зона видимости приводит к структурной зависимости, то есть один объект не имеет исключительных прав доступа к другому: состояние этого другого объекта может быть изменено несколькими способами.  **Выбор реализации**  Внутреннее строение (реализация) классов и объектов разрабатывается только после завершения проектирования их внешнего поведения. При этом необходимо принять два проектных решения: выбрать способ представления класса или объекта и способ размещения их в модуле.  **Представление.** Представление классов и объектов почти всегда должно быть инкапсулировано (скрыто). Это позволяет вносить изменения (например, перераспределение памяти и временных ресурсов) без нарушения функциональных связей с другими классами и объектами. Как мудро отметил Вирт: "выбор способа представления является нелегкой задачей и не определяется одними лишь техническими средствами. Он всегда должен рассматриваться с точки зрения операций над данными" [[60](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#3.60)]. Рассмотрим, например, класс, соответствующий расписаниям полетов самолетов. Как его нужно оптимизировать - по эффективности поиска или по скорости добавления/удаления рейса? Поскольку невозможно реализовать и то, и другое одновременно, нужно сделать выбор, исходя из целей системы. Иногда такой выбор сделать непросто, и тогда создается семейство классов с одинаковым интерфейсом, но с принципиально разной реализацией для обеспечения вариативности поведения.  Одним из наиболее трудных решений является выбор между вычислением элементов состояния объекта и хранением их в виде полей данных. Рассмотрим, например, класс **Cone** (конус) с соответствующим ему методом **volume** (объем). Этот метод возвращает значение объема объекта. В структуре конуса в виде отдельных полей хранятся данные о его высоте и радиусе основания. Следует ли еще создать поле данных для объема или следует вычислять его по мере необходимости внутри метода **volume** [[60](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#3.60)]? Если мы хотим получать значение объема максимально быстро, нужно создавать соответствующее поле данных. Если важнее экономия памяти, лучше вычислить это значение. Оптимальный способ представления объекта всегда определяется характером решаемой задачи. В любом случае этот выбор не должен влиять на интерфейс класса.  **Модульная структура.** Аналогичные вопросы возникают при распределении деклараций классов и объектов по модулям. В языке Smalltalk эта проблема отсутствует, здесь модульный механизм не реализован. В языках Object Pascal, C++, CLOS и Ada существует понятие модуля как отдельной языковой конструкции. Решение о месте декларирования классов и объектов в этих языках является компромиссом между требованиями видимости и скрытия информации. В общем случае модули должны быть функционально связными внутри и слабо связанными друг с другом. При этом следует учитывать ряд нетехнических факторов, таких, как повторное использование, безопасность, документирование. Проектирование модулей - не более простой процесс, чем проектирование классов и объектов. О скрытии информации Парнас, Клеменс и Вейс говорят следующее: "Применение этого принципа не всегда очевидно. Принцип нацелен на минимизацию стоимости программных средств (в целом за время эксплуатации), для чего от проектировщика требуется способность оценивать вероятность изменений. Такие оценки основываются на практическом опыте и знаниях предметной области, включая понимание технологии программирования и аппаратных особенностей" [[61](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#3.61)].  Выводы  Объект характеризуется состоянием, поведением и идентичностью.  Структура и поведение одинаковых объектов описывается в общем для них классе.  Состояние объекта определяет его статические и динамические свойства.  Поведение объекта характеризуется изменением его состояния в процессе взаимодействия (посредством передачи сообщений) с другими объектами.  Идентичность объекта - это его отличия от всех других объектов.  Иерархия объектов может строиться на принципах связи или агрегации.  Множество объектов с одинаковой структурой и поведением является классом.  Шесть типов иерархий классов включают: ассоциирование, наследование, агрегация, использование, инстанцирование и метаклассирование.  Классы и объекты, образующие словарь предметной области, называются ключевыми абстракциями.  Структура, объединяющая множество объектов и обеспечивающая их совместное целенаправленное функционирование, называется механизмом.  Качество абстракций измеряется их зацеплением, связностью, достаточностью, полнотой и примитивностью. |

Классификация

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Классификация - средство упорядочения знаний. В объектно-ориентированном анализе определение общих свойств объектов помогает найти общие ключевые абстракции и механизмы, что в свою очередь приводит нас к более простой архитектуре системы. К сожалению, пока не разработаны строгие методы классификации и нет правила, позволяющего выделять классы и объекты. Нет таких понятий, как "совершенная структура классов", "правильный выбор объектов". Как и во многих технических дисциплинах, выбор классов является компромиссным решением.  На одной из конференций программистам был задан вопрос: "Какими правилами вы руководствуетесь при определении классов и объектов?" Страуструп, разработчик языка C++, ответил: "Это как поиск святого Грааля. Не существует панацеи". Габриель, один из разработчиков CLOS, сказал: "Это вопрос, на который нет простого ответа. Я просто пробую" [[1](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#4.1)]. К счастью, имеется богатый опыт классификации в других науках, на основе которого разработаны методики объектно-ориентированного анализа. Каждая такая методика предлагает свои правила (эвристики) идентификации классов и объектов. Они и будут предметом этой главы.  **4.1. Важность правильной классификации**  **Классификация и объектно-ориентированное проектирование**  Определение классов и объектов - одна из самых сложных задач объектно-ориентированного проектирования. Наш опыт показывает, что эта работа обычно содержит в себе элементы открытия и изобретения. С помощью открытий мы распознаем ключевые понятия и механизмы, которые образуют словарь предметной области. С помощью изобретения мы конструируем обобщенные понятия, а также новые механизмы, которые определяют правила взаимодействия объектов. Поэтому открытие и изобретение - неотъемлемые части успешной классификации. Целью классификации является нахождение общих свойств объектов. Классифицируя, мы объединяем в одну группу объекты, имеющие одинаковое строение или одинаковое поведение.   http://www.helloworld.ru/texts/comp/other/oop/pg148.gif  *Классификация есть средство упорядочение знаний.*  Разумная классификация, несомненно, - часть любой науки. Михальски и Степп утверждают: "неотъемлемой задачей науки является построение содержательной классификации наблюдаемых объектов или ситуаций. Такая классификация существенно облегчает понимание основной проблемы и дальнейшее развитие научной теории" [[2](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#4.2)]. Та же философия относится и к инженерному делу. В области строительной архитектуры и городского планирования, как отмечает Александер, для архитектора "его проектная деятельность, и скромная, и гигантская по размеру, управляется целиком образами, которые он держит в своем сознании в данный момент, и его способностью комбинировать эти образы при создании нового проекта" [[3](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#4.3)].  Неудивительно, что классификация затрагивает многие аспекты объектно-ориентированного проектирования. Она помогает определить иерархии обобщения, специализации и агрегации. Найдя общие формы взаимодействия объектов, мы вводим механизмы, которые станут фундаментом реализации нашего проекта. Классификация помогает правильно определить модульную структуру. Мы можем расположить объекты в одном или разных модулях, в зависимости от степени схожести объектов; зацепление и связность - всего лишь меры этой схожести.  Классификация играет большую роль при распределении процессов между процессорами. Мы направляем процессы на один процессор или на разные в зависимости от того, как эти процессы связаны друг с другом.  **Трудности классификации**  **Примеры классификации.** В главе 3 мы определили объект как нечто, имеющее четкие границы. На самом деле это не вполне так. Границы предметов часто неопределенны. Например, посмотрите на вашу ногу. Попытайтесь определить, где начинается и кончается колено. В разговорной речи трудно понять, почему именно эти звуки определяют слово, а не являются частью какого-то более длинного слова. Представьте себе, что вы проектируете текстовый редактор. Что считать классом - буквы или слова? Как понимать отдельные фразы, предложения, параграфы, документы? Как обращаться с произвольными, не обязательно осмысленными, блоками текста? Что делать с предложениями, абзацами и целыми документами - соответствуют ли такие классы нашей задаче?  То, что разумная классификация - трудная проблема, новостью не назовешь. И поскольку есть параллели с аналогичными трудностями в объектно-ориентированном проектировании, рассмотрим примеры классификации в двух других научных дисциплинах: биологии и химии.  Вплоть до XVIII века идея о возможности классификации живых организмов по степени сложности была господствующей. Мера сложности была субъективной, поэтому неудивительно, что человек оказался в списке на первом месте. В середине XVIII века шведский ботаник Карл Линней предложил более подробную таксономию для классификации организмов: он ввел понятия рода и вида. Век спустя Дарвин выдвинул теорию, по которой механизмом эволюции является естественный отбор и ныне существующие виды животных - продукт эволюции древних организмов. Теория Дарвина основывалась на разумной классификации видов. Как утверждает Дарвин, "натуралисты пытаются расположить виды, роды, семейства в каждом классе в то, что называется натуральной системой. Что подразумевается под этой системой? Некоторые авторы понимают некоторую простую схему, позволяющую расположить наиболее похожие живые организмы в один класс и различные - в разные классы" [[4](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#4.4)]. В современной биологии термин "классификация" обозначает "установление иерархической системы категорий на основе предположительно существующих естественных связей между организмами" [[5](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#4.5)]. Наиболее общее понятие в биологической таксономии - царство, затем, в порядке убывания общности: тип (отдел), класс, отряд (порядок), семейство, род и, наконец, вид. Исторически сложилось так, что место каждого организма в иерархической системе определяется на основании внешнего и внутреннего строения тела и эволюционных связей. В современной классификации живых существ выделяются группы организмов, имеющих общую генетическую историю, то есть организмы, имеющие сходные ДНК, включаются в одну группу. Классификация по ДНК полезна, чтобы различить организмы, которые похожи внешне, но генетически сильно отличаются. По современным воззрениям дельфины ближе к коровам, чем к форели [[6](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#4.6)].  Возможно, для программиста биология представляется зрелой, вполне сформировавшейся наукой с определенными критериями классификации организмов. Но это не так. Биолог Мэй сказал: "На сегодняшний день мы даже не знаем порядок числа видов растений и животных, населяющих нашу планету: классифицировано менее, чем 2 млн. видов, в то время как возможное число видов оценивается от 5 до 50 млн." [[7](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#4.7)]. Более того, различные критерии классификации одних и тех же животных приводят к разным результатам. Мартин утверждает, что "все зависит от того, что вы хотите получить. Если вы хотите, чтобы классификация говорила о кровном родстве видов, вы получите один ответ, если вы желаете отразить уровень приспособления, ответ будет другой" [[8](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#4.8)]. Можно заключить, что даже в строгих научных дисциплинах методы и критерии классификации сильно зависят от цели классификации.  Аналогичная ситуация сложилась и в химии [[9](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#4.9)]. В древние времена считалось, что все вещества суть комбинации земли, воздуха, огня и воды. В настоящее время такая классификация не может считаться сколько-нибудь удовлетворительной. В середине XVII в. Роберт Бойль предложил элементы как примитивные химические абстракции, из которых составляются более сложные вещества. Век спустя, в 1789 г., Лавуазье опубликовал первый список, содержащий 23 элемента, хотя впоследствии было открыто, что некоторые из них таковыми не являются. Но открытие новых элементов продолжалось, список увеличивался. Наконец, в 1869 г. Менделеев предложил периодический закон, который давал точные критерии для классификации известных элементов и даже мог предсказывать свойства еще не открытых элементов. Но даже периодический закон не был концом истории о классификации элементов. В начале XX в. были открыты элементы с одинаковыми химическими свойствами, но с разными атомными весами - изотопы.  Вывод прост. Как утверждал Декарт: "Открытие порядка - нелегкая задача, но если он найден, понять его совсем не трудно" [[10](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#4.10)]. Лучшие программистские решения выглядят просто, но, как показывает опыт, добиться простой архитектуры очень трудно.  **Итеративная суть классификации.** Все эти сведения мы привели здесь не для того, чтобы оправдать "долгострой" в программном обеспечении, хотя на самом деле многим менеджерам и пользователям кажется, что необходимы века, чтобы закончить начатую работу. Мы просто хотели подчеркнуть, что разумная классификация - работа интеллектуальная и лучший способ ее ведения - последовательный, итеративный процесс. Это становится очевидным при анализе разработки таких программных продуктов, как графический интерфейс, стандарты баз данных и языки программирования четвертого поколения. Шоу утверждает, что в разработке программного обеспечения "развитие какой-либо абстракции часто следует общей схеме. В начале проблема решается *ad hoc*, то есть как-нибудь, для каждого частного случая. По мере накопления опыта некоторые решения оказываются более удачными, чем другие, и возникает род фольклора, переходящего от человека к человеку. Удачные решения изучаются более систематически, они программируются и анализируются. Это позволяет развить модели, осуществить их автоматическую реализацию, и разработать теорию, обобщающую найденное решение. Это в свою очередь поднимает практику на более высокий уровень и позволяет взяться за еще более сложную задачу, к которой, в свою очередь, мы подходим *ad hoc*, тем самым начиная новый виток спирали" [[11](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#4.11)].  Итеративный подход к классификации накладывает соответствующий отпечаток и на процедуру конструирования иерархии классов и объектов при разработке сложного программного обеспечения. На практике обычно за основу берется какая-то определенная структура классов, которую постепенно совершенствуют.   http://www.helloworld.ru/texts/comp/other/oop/pg151.gif  *Разные наблюдатели классифицируют один и тот же объект по-разному.*  И только на поздней стадии разработки, когда уже получен некоторый опыт использования такой структуры, мы можем критически оценить качество получившейся классификации. Основываясь на полученном опыте, мы можем создать новый подкласс из уже существующих (вывод), или разделить большой класс на много маленьких (факторизация), или, наконец, слить несколько существующих в один (композиция). Возможно, в процессе разработки будут найдены новые общие свойства, ранее не замеченные, и мы сможем определить новые классы (абстракция) [[12](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#4.12)].  Почему же классификация так сложна? Мы объясняем это двумя причинами. Во-первых, отсутствием "совершенной" классификации, хотя, естественно, одни классификации лучше других. Кумбс, Раффья и Трал утверждают, что "существует столько способов деления мира на объектные системы, сколько ученых принимается за эту задачу" [[13](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#4.13)]. Любая классификация зависит от точки зрения субъекта. Флуд и Кэрсон приводят пример: "Соединенное Королевство... экономисты могут рассматривать как экономический институт, социологи - как общество, защитники окружающей среды - как гибнущий уголок природы, американские туристы - как достопримечательность, советские руководители - как военную угрозу, наконец, наиболее романтичные из нас, британцев - как зеленые луга родины" [[14](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#4.14)]. Во-вторых, разумная классификация требует изрядной доли творческого озарения. Бертвистл, Даль, Мюрхауг и Нюгард заключают, что "иногда ответ очевиден, иногда он - дело вкуса, а бывает, что все зависит от умения заметить главное" [[15](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#4.15)]. Все это напоминает загадку: "Почему лазерный луч похож на золотую рыбку?.. Потому, что ни тот, ни другой не умеют свистеть" [[16](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#4.16)]. Надо быть очень творческим мыслителем, чтобы найти общее в настолько несвязанных предметах.  4.2. Идентификация классов и объектов  **Классический и современный подходы**  Со времен Платона проблема классификации занимала умы бесчисленных философов, лингвистов, когнитивистов, математиков. Поэтому было бы разумно изучить накопленный опыт и применить его в объектно-ориентированном проектировании. Исторически известны только три подхода:  классическая категоризация;  концептуальная кластеризация;  теория прототипов [[17](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#4.17)].  **Классическая категоризация.** В классическом подходе "все вещи, обладающие данным свойством или совокупностью свойств, формируют некоторую категорию. Причем наличие этих свойств является необходимым и достаточным условием, определяющим категорию" [[18](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#4.18)]. Например, холостые люди - это категория: каждый человек или холост, или женат, и этот признак достаточен 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.  Классическая категоризация пришла к нам от Платона и Аристотеля. Последний в своей классификации растений и животных пользовался техникой рассуждений, напоминающей современную детскую игру в 20 вопросов (Это минерал, животное или растение? Это покрыто мехом или перьями? Может ли оно летать? Пахнет ли оно?) [[20](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#4.20)]. Такой подход нашел последователей, наиболее выдающимися из которых были: Фома Аквинский, Декарт, Локк. По утверждению Фомы Аквинского: "Мы можем именовать вещи согласно нашим знаниям об их природе, получаемым через познание их свойств и действий" [[21](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#4.21)].  Принципы классической категоризации отражены в современной теории развития ребенка. Пьяже утверждает, что после первого года жизни ребенок осознает существование объектов и затем начинает приобретать навыки их классификации, вначале пользуясь базовыми категориями, такими, как собаки, кошки и игрушки [[22](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#4.22)]. Позднее ребенок осознает, с одной стороны более общие (животные), а с другой стороны, более частные категории (колли, доги, овчарки) [[23](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#4.23)].  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. Мински предположил, что "лучшими являются такие наборы свойств, элементы которых мало взаимодействуют между собой. Этим объясняется всеобщая любовь к таким критериям как размер, цвет, форма и материал. Так как эти критерии не пересекаются, про какой-нибудь предмет можно утверждать, что он большой, серый, круглый и деревянный" [[24](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#4.24)]. Вообще говоря, свойства не обязательно должны быть измеряемыми, в качестве их можно использовать наблюдаемое поведение. То обстоятельство, что птицы летают, а рыбы нет, позволяет отличить орла от форели.    |  | | --- | | Проблема классификации  На рис. 4-1 показаны 10 поездов, обозначенных буквами от А до J. Каждое изображение состоит из паровоза и нескольких вагонов. Прежде чем продолжать чтение, попытайтесь за 10 минут определить несколько групп изображений, составленных по какому-то логическому признаку. Например, изображения можно разбить на три группы: в одной группе поезда имеют черные колеса, в другой группе - белые, а в третьей - и белые, и черные.  Этот пример взят из работы Степпа и Михальски о концептуальном объединении [[19](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#4.19)]. Очевидно, "правильного" разбиения на группы не существует. Наши изображения были классифицированы 93 различными способами. Наиболее распространенный способ классификаций по длине состава: были выделены три группы: составы с двумя, тремя и четырьмя вагонами. Второй по популярности вид классификации - по цвету колес поезда. Сорок из девяносто трех видов классификации были уникальными (то есть вид содержал только один экземпляр).  Экспериментируя с этим рисунком, мы убедились в правоте Степпа и Михальски. Большинство опрошенных нами предлагали один из двух наиболее популярных видов классификации (по длине состава и цвету колес поезда). Один опрошенный предложил следующее: в одной группе составы помечены буквами, нарисованными с помощью только прямых линий (A, Е, F, H и I), в другой - буквами с кривыми линиями. Вот уж, действительно, пример нетривиального мышления.  Если вы уже справились с заданием, давайте изменим условия. Представим, что круги обозначают груз с токсичными веществами, прямоугольники - лесоматериалы, все остальные знаки обозначают пассажиров. Попытайтесь теперь классифицировать изображения и заметьте, как дополнительная информация влияет на вашу точку зрения.  В наших опытах большинство опрошенных классифицировало поезда по тому, содержит состав токсичный груз или нет. Мы заключили, что новые сведения о реальной ситуации облегчают и улучшают классификацию. |   Какие конкретно свойства надо принимать во внимание? Это зависит от обстановки. Например, цвет автомобиля надо зафиксировать в задаче учета продукции автомобилестроительного завода, но он не интересен программе, управляющей уличным светофором. Вот почему мы говорим, что нет абсолютного критерия классификации, одна и та же структура классов может подходить для одной задачи и не годиться для другой. Джеймс пишет: "Нельзя утверждать, что некоторая схема классификации лучше других отражает структуру и порядок вещей в природе. Природе безразличны наши попытки в ней разобраться. Некоторые классификации действительно важнее других, но только в связи с нашими интересами, а не потому, что они вернее или полнее отражают реальность" [[25](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#4.25)].  Современное западное мышление по большей части насквозь пропитано классической категоризацией, однако, как показывает пример с высокими и низкими людьми, этот подход не всегда работает. Косок отмечает, что "естественные категории не четко отграничены друг от друга. Большинство птиц летает, но не все. Стул может быть деревянным, металлическим или пластмассовым, а количество ног у него целиком зависит от прихоти конструктора. Практически невозможно перечислить определяющие свойства естественной категории, так, чтобы не было исключений" [[26](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#4.26)]. Это, действительно, коренные пороки классической категоризации, которые и попытались исправить в современных подходах. Ими мы сейчас займемся.   http://www.helloworld.ru/texts/comp/other/oop/pic04_01.gif  *Рис. 4-1. Проблема классификации.*  **Концептуальная кластеризация.** Это более современный вариант классического подхода. Он возник из попыток формального представления знаний. Степп и Михальски пишут: "При таком подходе сначала формируются концептуальные описания классов (кластеров объектов), а затем мы классифицируем сущности в соответствии с этими описаниями" [[27](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#4.27)]. Например, возьмем понятие "любовная песня". Это именно понятие, а не признак или свойство, поскольку степень любовности песни едва ли можно измерить. Но если можно утверждать, что песня скорее про любовь, чем про что-то другое, то мы помещаем ее в эту категорию.  Концептуальную кластеризацию можно связать с теорией нечетких (многозначных) множеств, в которой объект может принадлежать к нескольким категориям одновременно с разной степенью точности. Концептуальная кластеризация делает в классификации абсолютные суждения, основываясь на наилучшем согласии.  **Теория прототипов.** Классическая категоризация и концептуальная кластеризация - достаточно выразительные методы, вполне пригодные для проектирования сложных программных систем. Но все же есть ситуации, в которых эти методы не работают. Рассмотрим более современный метод классификации, теорию прототипов, предпосылки которой можно найти в книге по психологии восприятия Рош и ее коллег [[28](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#4.28)].  Существуют некоторые абстракции, которые не имеют ни четких свойств, ни четкого определения. Лакофф объясняет эту проблему так: "По утверждению Виттгенстейна (Wittgenstein), существуют категории (например, игры), которые не соответствуют классически образцам, так как нет признаков, свойственных всем играм... По этой причине их можно объединить так называемой семейной схожестью... Виттгенстейн утверждает, что у категории игр нет четкой границы. Категорию можно расширить и включить новые виды игр при условии, что они напоминают уже известные игры" [[29](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#4.29)]. Вот почему этот подход называется теорией прототипов: класс определяется одним объектом-прототипом, и новый объект можно отнести к классу при условии, что он наделен существенным сходством с прототипом.  Лаков и Джонсон применяют классификацию на основе прототипов к упомянутой выше проблеме стульев. Они замечают, что "мы считаем мягкий пуф, парикмахерское кресло и складной стул стульями не потому, что они удовлетворяют некоторому фиксированному набору признаков прототипа, но потому, что они имеют достаточное фамильное сходство с прототипом... Не требуется никакого общего набора свойств прототипа, которое годилось бы и для пуфика и для парикмахерского кресла, но они оба - стулья, так как каждый из них в отдельности похож на прототипный стул, пусть даже каждый по-своему. Свойства, определяемые при взаимодействии с объектом (свойства взаимодействия), являются главными при определении семейного сходства" [[30](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#4.30)].  Понятие свойств взаимодействия - центральное для теории прототипов. В концептуальной кластеризации мы группируем в соответствии с различными концепциями. В теории прототипов классификация объектов производится по степени их сходства с конкретным прототипом.  **Применение классических и новых теорий.** Разработчику, озабоченному постоянно меняющимися требованиями к системе и вечно сражающемуся с напряженным планом при ограниченных ресурсах, предмет нашего обсуждения может показаться далеким от реальности. В действительности, три рассмотренных подхода к классификации имеют непосредственное отношение к объектно-ориентированному проектированию.  На практике мы идентифицируем классы и объекты сначала по свойствам, важным в данной ситуации, то есть стараемся выделить и отобрать структуры и типы поведения с помощью словаря предметной области. "Потенциально возможных абстракций, как правило, очень много" [[31](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#4.31)]. Если таким путем не удалось построить удобоваримой структуры классов, мы пробуем концептуальный подход. В этом случае в центре внимания уделяется поведение объектов, когда они взаимодействуют друг с другом. Наконец, мы пробуем выделить прототипы и ассоциировать с ними объекты.  Эти три способа классификации составляют теоретическую основу объектно-ориентированного подхода к анализу, предлагающего много практических советов и правил, которые можно применить для идентификации классов и объектов при проектировании сложной программной системы.  **Объектно-ориентированный анализ**  Границы между стадиями анализа и проектирования размыты, но решаемые ими задачи определяются достаточно четко. В процессе анализа мы моделируем проблему, *обнаруживая*классы и объекты, которые составляют словарь проблемной области. При объектно-ориентированном проектировании мы *изобретаем* абстракции и механизмы, обеспечивающие поведение, требуемое моделью [Обозначения и процессы, описанные в этой книге, в равной степени относятся к фазам и анализа и проектирования (в традиционном понимании), как мы увидим в главе 6. Именно по этой причине мы сменили во втором издании название книги на "Объектно-ориентированный анализ и проектирование"].  Теперь мы рассмотрим несколько проверенных практикой подходов к анализу объектно-ориентированных систем.  **Классические подходы.** Разные ученые находят различные источники классов и объектов, согласующихся с требованиями предметной области. Мы называем эти подходы классическими, поскольку они опираются на классическую категоризацию.  Например, Шлаер и Меллор предлагают следующих кандидатов в классы и объекты [[32](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#4.32)]:    |  |  | | --- | --- | | ® Осязаемые предметы | Автомобили, телеметрические данные, датчики давления | | ® Роли | Мать, учитель, политик | | ® События | Посадка, прерывание, запрос | | ® Взаимодействие | Заем, встреча, пересечение |   Что-то в этом роде предлагает Росс, исходя из перспектив моделирования баз данных [[33](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#4.33)]:    |  |  | | --- | --- | | ® Люди | Человеческие существа, выполняющие некоторые функции | | ® Места | Области, связанные с людьми или предметами | | ® Предметы | Осязаемый материальный объект или группа объектов | | ® Организации | Формально организованная совокупность людей, ресурсов, оборудования, которая имеет определенную цель и существование которой в целом не зависит от индивидуумов | | ® Концепции | Принципы и идеи, сами по себе неосязаемые, но предназначенные для организации деятельности и/или общения, или же для наблюдения за ними | | ® События | Нечто случающееся с чем-то в заданное время или последовательно |   Коад и Иордан предложили свой список [[34](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#4.34)]:    |  |  | | --- | --- | | ® Структуры | Отношения "целое-часть" и "общее-частное" | | ® Другие системы | Внешние системы, с которыми взаимодействует приложение | | ® Устройства | Устройства, с которыми взаимодействует приложение | | ® События | Происшествия, которые должны быть запомнены | | ® Разыгрываемые роли | Роли, которые исполняют пользователи, работающие с приложением | | ® Места | Здания, офисы и другие места, существенные для работы приложения | | ® Организационные единицы | Группы, к которым принадлежат пользователи |   На более высоком уровне абстракции Коад вводит понятие предметной области, которая в сущности является логически связанной группой классов, относящейся к высокоуровневым функциям системы.  **Анализ поведения.** В то время как классические подходы концентрируют внимание на осязаемых элементах предметной области, другая школа мысли объектно-ориентированного анализа сосредотачивается на динамическом поведении как на первоисточнике объектов и классов [Шлаер и Меллор дополнили свою более раннюю работу, обратив внимание также и на поведение. В частности, они изучали жизненный цикл объекта как средство понимания границ]. Это напоминает концептуальную кластеризацию, рассмотренную выше: мы формируем классы, основываясь на группах объектов, демонстрирующих сходное поведение.  Вирфс-Брок предлагает понятие ответственности объекта, под которыми следует понимать "его знания и умения. Ответственность - это способ выразить цель объекта и его место в системе. Ответственность объекта есть совокупность всех услуг, которые он может предоставлять по всем его контрактам" [[36](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#4.36)]. То есть, мы объединяем вместе те объекты, которые имеют сходные ответственности и строим иерархию классов, в которой каждый подкласс, выполняя обязательства суперкласса, привносит свои дополнительные услуги.  Рубин и Гольдберг предлагают идентифицировать классы и объекты, анализируя функционирование системы: "Наш подход основан на изучении поведения системы. Мы сопоставляем формы поведения с частями системы и пытаемся понять, какая часть инициирует поведение и какие части в нем участвуют... Инициаторы и участники, играющие существенные роли, опознаются как объекты и делаются ответственными за эти роли" [[37](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#4.37)].  Идеи Рубина тесно связаны с предложенным в 1979 году Альбрехтом подходом с точки зрения функций. По его определению, функция "определяется как отдельное бизнес-действие конечного пользователя" [[38](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#4.38)], то есть: ввод/вывод, запрос, файл или интерфейс. Очевидно, что эта концепция происходит из области информационных систем. Однако, она может быть применена к любой автоматизированной системе. По существу, функция - это любое достоверно видимое извне и имеющее отношение к делу поведение системы.  **Анализ предметной области.** 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, как, например, ведение историй болезни пациентов, торговля ценными бумагами, разработка компиляторов или системы управления ракетами. Если вы находитесь в середине разработки и застряли, анализ какой-нибудь узкой предметной области может помочь, указав вам на ключевые абстракции, оказавшиеся полезными в сходных системах. Анализ предметной области работает очень хорошо, исключая разве что лишь очень специальные ситуации, так как уникальные программные системы встречаются крайне редко.  Идею анализа предметной области впервые предложил Нейборс. Мы определим такой анализ как "попытку выделить те объекты, операции и связи, которые эксперты данной области считают наиболее важными" [[39](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#4.39)]. Мур и Байлин определяют следующие этапы в анализе области:  "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" [[40](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#4.40)].  Анализ области можно вести относительно аналогичных приложений (вертикально) или относительно аналогичных частей одного и того же приложения (горизонтально). Например, начиная проектировать систему учета пациентов, имеет смысл рассмотреть уже имеющиеся подобные системы, чтобы понять, какие ключевые абстракции и механизмы, использованные в них, будут вам полезны, а какие нет. Аналогично система бухгалтерского учета должна представлять различные виды отчетов. Если считать отчеты некой предметной областью, ее анализ может привести разработчика к пониманию ключевых абстракций и механизмов, которые обслуживают все виды отчетов. Полученные таким образом классы и объекты представляют собой множество ключевых абстракций и механизмов, отобранных с учетом цели исходной задачи: создания системы отчетов. Поэтому окончательный проект будет проще.  Определим теперь, кто такой эксперт? В роли эксперта часто выступает просто пользователь системы, например, инженер или диспетчер. Он не обязательно должен быть программистом, но должен быть близко знаком с исследуемой проблемой и разговаривать на языке этой проблемы.  Менеджеры проектов заинтересованы в непосредственном сотрудничестве пользователей и разработчиков системы. Но для очень сложных систем прикладной анализ является формальным процессом, для которого требуется большое число экспертов и разработчиков на длительный период времени. На практике такой формальный анализ требуется редко. Обычно для начального уяснения проблемы достаточно короткой встречи экспертов и разработчиков. Удивительно, как мало информации требуется для продуктивной работы разработчика. Однако мы считаем чрезвычайно полезными такие встречи в течение всей разработки. Анализ прикладной области лучше всего вести шаг за шагом - немного поанализировать, потом немного попроектировать и т.д.  **Анализ вариантов.** По отдельности классический подход, поведенческий подход и изучение предметной области, рассмотренные выше, сильно зависят от индивидуальных способностей и опыта аналитика. Для большинства реальных проектов одновременное применение всех трех подходов неприемлемо, так как процесс анализа становится недетерминированным и непредсказуемым.  Анализ вариантов - это подход, который можно успешно сочетать с первыми тремя, делая их применение более упорядоченным. Впервые его формализовал Джекобсон, определивший вариант применения, как "частный пример или образец использования, сценарий, начинающийся с того, что пользователь системы инициирует операцию или последовательность взаимосвязанных событий" [[41](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#4.41)].  Коротко говоря, этот вид анализа можно начинать вместе с анализом требований. В этот момент 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, как делают телевизионщики и кинематографисты [[42](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#4.42)]. При этом они устанавливают, какие объекты участвуют в сценарии, каковы обязанности каждого объекта и как они взаимодействуют в терминах операций. Тем самым группа разработчиков вынуждена четко распределить области влияния абстракций. Далее набор сценариев расширяется, чтобы учесть исключительные ситуации и вторичное поведение (Гольдстейн и Алджер называют это периферийными аспектами [[43](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#4.43)]). В результате появляются новые или уточняются существующие абстракции. Позже, в главе 6, мы покажем, как сценарии используются для тестирования.     **CRC-карточки.** CRC обозначает Class-Responsibilities-Collaborators (Класс/Ответственности/Участники). Это простой и замечательно эффективный способ анализа сценариев. Карты CRC впервые предложили Бек и Каннингхэм для обучения объектно-ориентированному программированию, но такие карточки оказались отличным инструментом для мозговых атак и общения разработчиков между собой.  Собственно, это обычные библиографические карточки 3х5 дюйма (если позволяет бюджет вашего проекта, купите 5х7; очень хорошо, если карточки будут линованными, а разноцветные - просто мечта). На карточках вы пишите (обязательно карандашом) сверху - название класса, снизу в левой половине - за что он отвечает, а в правой половине - с кем он сотрудничает. Проходя по сценарию, заводите по карточке на каждый обнаруженный класс и дописывайте в нее новые пункты. При этом каждый раз обдумывайте, что из этого получается, и "выделяйте излишек ответственности" в новый класс или, что случается чаще всего, перенесите ответственности с одного большого класса на несколько более детальных классов, или, возможно, передайте часть обязанностей другому классу.  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.  **Неформальное описание.** Радикальная альтернатива классическому анализу была предложена в чрезвычайно простом методе Аббота. Согласно этому методу надо описать задачу или ее часть на простом английском языке, а потом подчеркнуть существительные и глаголы [[45](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#4.45)]. Существительные - кандидаты на роль классов, а глаголы могут стать именами операций. Метод можно автоматизировать, и такая система была построена в Токийском технологическом институте и в Fujitsu [[46](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#4.46)].  Подход Аббота полезен, так как он прост и заставляет разработчика заниматься словарем предметной области. Однако он весьма приблизителен и непригоден для сколько-нибудь сложных проблем. Человеческий язык - ужасно неточное средство выражения, потому список объектов и операций зависит от умения разработчика записывать свои мысли. Тем более, что для многих существительных можно найти соответствующую глагольную форму и наоборот.  **Структурный анализ.** Вторая альтернатива классической технике объектно-ориентированного анализа использует структурный анализ как основу для объектно-ориентированного проектирования. Такой подход привлекателен потому, что много аналитиков применяют этот подход и имеется большое число программных CASE-средств, поддерживающих автоматизацию этих методов. Нам лично не нравится использовать структурный анализ как основу для объектно-ориентированного проектирования, но для некоторых организаций такой прагматический подход не имеет альтернативы.  После проведения структурного анализа мы уже имеем модель системы, описанную диаграммами потоков данных и другими продуктами структурного анализа. Эти диаграммы дают нам формальную модель проблемы. Исходя из модели, мы можем приступить к определению осмысленных классов и объектов тремя различными способами.  МакМенамин и Палмер предлагают сначала приступить к формированию словаря данных и затем к анализу контекстных диаграмм модели. Они говорят:  "рассматривая список основных структур данных, следует подумать, о чем они говорят или что описывают. Например, если они прилагательные, то какие существительные они описывают? Ответы на такие вопросы могут пополнить ваш список объектов" [[47](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#4.47)]. Эти кандидаты в объекты происходят из окружающей среды, из существенных входных и выходных данных, а также продуктов, услуг и других ресурсов, которыми она управляет.  Следующие два способа основаны на анализе отдельных диаграмм потоков данных. Если взять какую-нибудь диаграмму потоков (в терминологии Барда и Меллора [[48](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#4.48)]), то кандидаты в объекты это:  внешние сущности;  хранилища данных;  хранилища управляющих сущностей;  управляющие преобразования.  Кандидаты в классы:  потоки данных;  потоки управления.  Остается преобразование данных, которое мы можем рассматривать как операции над существующими объектами или как поведение некоторого объекта, который мы создали специально для выполнения нужного преобразования.  Зайдевиц и Старк предлагают еще один метод, который они называют анализом абстракций. Метод базируется на идентификации основных сущностей, которые по своей природе аналогичны основным преобразованиям в структурном проектировании. Как они говорят, "в структурном анализе входные и выходные данные изучаются до тех пор, пока не достигнут высшего уровня абстракции. Процесс преобразования входных данных в выходные есть основное преобразование. В абстрактном анализе разработчик делает то же самое, а также изучает основное преобразование для того, чтобы определить, какие процессы и состояния представляют наилучшую абстрактную модель системы" [[49](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#4.49)]. После определения основной сущности в диаграмме потоков данных аналитик приступает к изучению всей инфраструктуры, прослеживая входящие и исходящие потоки данных из центра, группируя процессы и состояния, встречающиеся по пути. Для практического использования авторы нашли анализ абстракций слишком сложным и в качестве альтернативы предлагают объектно-ориентированный анализ [[50](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#4.50)].  Необходимо отметить, что принципы структурного проектирования, которое, естественно, следует за структурным анализом, полностью ортогональны принципам объектно-ориентированного проектирования. Наш опыт показывает, что использование структурного анализа в процессе объектно-ориентированного проектирования часто приводит к полному провалу, если разработчик не способен сопротивляться желанию свалиться в структурную пропасть. Другая очень серьезная опасность заключается в том, что многие аналитики любят рисовать диаграммы потоков данных, которые представляют собой скорее описание проекта, чем модель существа системы. Очень трудно построить объектно-ориентированную систему, если модель столь очевидно ориентирована на алгоритмическую декомпозицию. Поэтому мы предпочитаем объектно-ориентированный анализ и анализ проблемной области как подготовительный этап для объектно-ориентированного проектирования. При этом уменьшается риск замусорить проект элементами алгоритмического анализа.  Если же по каким-либо уважительным причинам [Политические и исторические причины в качестве уважительных не принимаются] приходится взять за основу структурный анализ, прекратите строить диаграммы, как только они начинают смахивать на проект программы, а не на модель предметной области. Помните, что материалы проектирования, такие, как диаграммы потоков данных, это не конечный продукт, а инструмент разработчиков. Обычно строятся диаграммы, а затем разрабатываются механизмы, обеспечивающие необходимое поведение системы, то есть сам акт проектирования видоизменяет начальную модель. Поддержание соответствия модели и развивающегося проекта - дело трудное, и, честно говоря, бесполезное. Имеет смысл сохранять только продукт структурного анализа высокого уровня абстракции. Он отражает существенные черты и достаточно независим от проекта системы.  4.3. Ключевые абстракции и механизмы  **Ключевые абстракции**  **Поиск и выбор ключевых абстракций.** Ключевая абстракция - это класс или объект, который входит в словарь проблемной области. Самая главная ценность ключевых абстракций заключена 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. Как утверждает Голдберг, "правильный выбор объектов зависит от назначения приложения и степени детальности обрабатываемой информации" [[51](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#4.51)].  Как мы уже отмечали, 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 [[52](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#4.52)]. 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, диспетчер экрана, список, очередь и так далее. Эти ключевые абстракции созданы уже не предметной областью, а проектированием.  Наиболее мощный способ выделения ключевых абстракций - сводить задачу к уже известным классам и объектам. Как будет показано ниже в главе 6, при отсутствии таких повторно используемых абстракций мы рекомендуем пользоваться сценариями, чтобы вести процесс идентификации классов и объектов.  **Уточнение ключевых абстракций.** Определив кандидатов на роли ключевых абстракций, мы должны оценить их по критериям, описанным в предыдущих главах. По словам Страуструпа "программист должен задаваться вопросами: Как создаются объекты класса? Как можно копировать и/или уничтожать объекты данного класса? Какие операции могут быть выполнены над этим объектом? Если ответы на эти вопросы туманны, то, возможно, общая концепция не ясна и лучше сесть и подумать еще раз, чем бросаться программировать" [[53](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#4.53)].  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. Халберт и О'Брайен утверждают, что "нет особой необходимости строить иерархию классов, начиная с самого верхнего класса, и потом дополнять ее подклассами. Чаще вы создаете несколько независимых иерархий, осознаете их общие черты и выделяете один или несколько суперклассов. Требуется несколько проходов вверх и вниз по иерархии, чтобы создать программный проект" [[54](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#4.54)]. Это не карт-бланш на хакерство, а всего лишь наблюдение, основанное на опыте и подтверждающее тот факт, что объектно-ориентированное проектирование - процесс последовательных приближений. Сходное наблюдение делает Страуструп: "Наиболее частые реорганизации в иерархии классов - это сведение совпадающих частей двух классов в один и разделение класса на два новых" [[55](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#4.55)].   http://www.helloworld.ru/texts/comp/other/oop/pg163.gif  *Классы и объекты должны быть на надлежащем уровне абстракции: не слишком высоко и не слишком низко.*  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. Это называется продвижением класса [[56](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#4.56)]. Аналогично, можем прийти к выводу, что класс слишком обобщен, и это затрудняет наследование: происходит семантический разрыв или конфликт зернистости [[57](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#4.57)]. В обоих случаях мы пытаемся выявить зацепление или недостаточную связность абстракций и смягчить конфликт.  Программисты часто легкомысленно относятся к правильному наименованию классов и объектов, но на самом деле очень важно отразить в обозначении классов и объектов сущность описываемых ими предметов. Программы необходимо писать тщательно, как художественную литературу, дума я и о читателях, и о компьютере [[58](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#4.58)]. При идентификации одного только объекта вам нужно придумать имена: для него, для его класса и для модуля, в котором класс объявлен. Умножьте на тысячу объектов и сотни классов, и вы поймете, как остра проблема.  Мы предлагаем следующие правила:  Объекты следует называть существительными: **theSensor** или **shape**.  Классы следует называть обобщенными существительными: **Sensors**, **Shapes**.  Операции-модификаторы следует называть активными глаголами: **Draw**, **moveLeft**.  У операций-селекторов в имя должен включаться запрос или форма глагола "to be": **extentOf**, **isOpen**.  Подчеркивание и использование заглавных букв - на ваше усмотрение, постарайтесь лишь не противоречить сами себе.  **Идентификация механизмов**  **Как найти механизмы?** В предыдущем обсуждении мы называли механизмами структуры, посредством которых объекты взаимодействуют друг с другом и ведут себя так, как требуется. Так же как при разработке класса фактически определяется поведение отдельных объектов, так же и механизмы служат для задания поведения совокупности объектов. Таким образом, механизмы представляют шаблоны поведения.  Рассмотрим требование, предъявляемое к автомобилю: нажатие на акселератор должно приводить к увеличению оборотов двигателя, а отпускание акселератора - к их уменьшению. Как это происходит, водителю совершенно безразлично. Может быть использован любой механизм, обеспечивающий нужное поведение, и его выбор - дело вкуса разработчика. Например, допустимо любое из предложенных ниже инженерных решений:  Механическая связь между акселератором и карбюратором (обычное решение).  Под педалью ставится датчик давления, который соединяется с компьютером, управляющим карбюратором (механизм управления по проводам).  Карбюратора нет. Бак с горючим находится на крыше автомобиля и топливо свободно течет в двигатель. Поток топлива регулируется зажимом на трубке. Нажатие на педаль акселератора ослабляет зажим (очень дешево).  Какую именно реализацию выберет разработчик, зависит от таких параметров, как стоимость, надежность, технологичность и т.д.  Подобно тому, как было бы недопустимой невежливостью со стороны клиента нарушать правила пользования сервером, также и выход за пределы правил и ограничений поведения, заданных механизмом, социально неприемлем. Водитель был бы удивлен, если бы, нажав на педаль акселератора, он увидел зажегшиеся фары.  Ключевые абстракции определяют словарь проблемной области, механизмы определяют суть проекта. В процессе проектирования разработчик должен придумать не только начинку классов, но и то, как объекты этих классов будут взаимодействовать друг с другом. Но конкретный механизм взаимодействия все равно придется разложить на методы классов. В итоге протокол класса будет отражать поведение его объектов и работу механизмов, в которых они участвуют.  Механизмы, таким образом, представляют собой стратегические решения в проектировании, подобно проектированию структуры классов. С другой стороны, проектирование интерфейса какого-то одного класса - это скорее тактическое решение. Стратегические решения должны быть выполнены явно, иначе у нас получится неорганизованная толпа объектов, кидающихся выполнять работу, расталкивая друг друга. В наиболее элегантных, стройных и быстрых программах воплощены тщательно разработанные механизмы.   http://www.helloworld.ru/texts/comp/other/oop/pg165.gif  *Механизмы суть средства, с помощью которых объекты взаимодействуют друг с другом для достижения необходимого поведения более высокого уровня.*  Механизмы представляют только один из шаблонов, которые мы находим в структурированных системах. Так, на нижнем конце своеобразной биологической пирамиды находятся идиомы. Это обороты, специфические для языков программирования или программистских культур, и отражающие общепринятые способы выражаться [Определяющей характеристикой идиомы является то, что ее игнорирование или нарушение влечет немедленные социальные последствия: вы превращаетесь в йеху или, еще хуже, в чужака, не заслуживающего уважения]. Например, в CLOS не принято использовать подчеркивание в именах функций или переменных, хотя в Ada это дело обычное [[59](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#4.59)]. Изучая язык, приходится учить его идиомы, которые обычно передаются в форме фольклора. Однако, как отметил Коплейн, идиомы играют важную роль в кодификации шаблонов низкого уровня. Он заметил, что "многие общепрограммистские действия идиоматичны" и поэтому распознание таких идиом позволяет "использовать конструкции C++ для выражения функциональности вне самого этого языка с сохранением иллюзии, что они являются частью языка" [[60](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#4.60)].  Место на верху пирамиды занимают среды разработки. Среда разработки - это собрание классов, предназначенных для определенной прикладной ситуации. Среда дает готовые классы, механизмы и услуги, которыми можно сразу пользоваться или приспосабливать для своих нужд.  Если идиомы составляют часть программистской культуры, то среды разработки обычно - коммерческий продукт. Например, Apple MacApp и его преемник Bedrock - среды, написанные на C++ и предназначенные для построения приложений со стандартным интерфейсом пользователя Macintosh. Аналогичную роль для Windows играют Microsoft Foundation Classes и ObjectWindows корпорации Borland.  **Примеры механизмов.** Рассмотрим механизм рисования, применяемый обычно в графических интерфейсах пользователя. Для того, чтобы представить какой-либо рисунок на экране, необходимы несколько объектов: окно, вид, модель, которую надо показать, и, наконец, клиент, который знает, когда надо нарисовать модель, но не знает, как это сделать. Сначала клиент дает окну команду нарисовать себя. Так как окно может включать в себя ряд видов, оно в свою очередь приказывает каждому из них нарисовать себя. Каждый вид посылает сообщение своей модели нарисовать себя, в результате чего и появляется изображение на экране. В этом механизме модель полностью отделена от окна и вида, в котором она представлена: виды могут посылать сообщения моделям, но модели не могут посылать сообщения видам. Smalltalk использует вариант этого механизма, названный парадигмой Model-View-Controller, модель-вид-контроллер (MVC) [[61](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#4.61)].  Механизмы, таким образом, представляют уровень повторного использования в проектировании, более высокий, чем повторное использование индивидуальных классов. MVC, например, является основой интерфейса пользователя в языке Smalltalk. Эта парадигма в свою очередь строится на базе механизма зависимостей, который вложен в поведение базового класса языка Smalltalk (класса **object**) и часто используется библиотекой классов языка Smalltalk.  Примеры механизмов можно найти во многих системах. Структуру операционной системы, например, можно описать на высоком уровне абстракции по тем механизмам, которые используются для диспетчеризации программ. Система может быть монолитной (как MS-DOS), иметь ядро (UNIX) или представлять собой иерархию процессов (операционная система THE) [[62](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#4.62)]. В системах искусственного интеллекта использованы разнообразные механизмы принятия решений. Одним из наиболее распространенных является механизм рабочей области, в которую каждый индивидуальный источник знаний независимо заносит свои сведения. В таком механизме не существует центрального контроля, но любое изменение в рабочей области может явиться толчком для выработки системой нового пути решения поставленной задачи [[63](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#4.63)]. Коад похожим образом выявил ряд общих механизмов в объектно-ориентированных системах, включая шаблоны временных ассоциаций, протоколирование событий и широковещательную рассылку сообщений [[64](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#4.64)]. Во всех случаях эти механизмы проявляются не как индивидуальные классы, а как структуры сотрудничающих классов.  На этом завершается наше изучение классификации и понятий, являющихся основой объектно-ориентированного проектирования. Следующие три главы посвящены самому методу, в частности системе обозначений, процессу проектирования и рассмотрению практических примеров.  Выводы  Идентификация классов и объектов - важнейшая задача объектно-ориентированного проектирования; процесс идентификации состоит из открытия и изобретения.  Классификация есть проблема группирования (кластеризации) объектов.  Классификация - процесс последовательных приближений; трудности классификации обусловлены в основном тем, что есть много равноправных решений.  Есть три подхода к классификации: классическое распределение по категориям (классификация по свойствам), концептуальная кластеризация (классификация по понятиям) и теория прототипов (классификация по схожести с прототипом).  Метод сценариев - это мощное средство объектно-ориентированного анализа, его можно использовать для других методов: классического анализа, анализа поведения и анализа предметной области.  Ключевые абстракции отражают словарь предметной области; их находят либо в ней самой, либо изобретают в процессе проектирования.  Механизмы обозначают стратегические проектные решения относительно совместной деятельности объектов многих различных типов. |

Обозначения

Составление диаграмм - это еще не анализ и не проектирование. Диаграммы позволяют описать поведение системы (для анализа) или показать детали архитектуры (для проектирования). Если вы понаблюдаете за работой инженера (программиста, технолога, химика, архитектора), вы быстро убедитесь, что будущая система формируется в сознании разработчика и только в нем. Когда, спустя некоторое время, система будет понятна в общих чертах, она скорее всего будет представлена на таких высокотехнологичных носителях, как тетрадные листы, салфетки или старые конверты [[1](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#5.1)].

Однако, хорошо продуманная и выразительная система обозначений очень важна для разработки. Во-первых, общепринятая система позволяет разработчику описать сценарий или сформулировать архитектуру и доходчиво изложить свои решения коллегам. Символ транзистора понятен всем электронщикам мира. Аналогично, над проектом жилого дома, разработанным архитекторами в Нью-Йорке, строителям из Сан-Франциско скорее всего не придется долго ломать голову, решая, как надо расположить двери, окна или электрическую проводку. Во-вторых, как подметил Уайтхед в своей основополагающей работе по математике, "Освобождая мозг от лишней работы, хорошая система обозначений позволяет ему сосредоточиться на задачах более высокого порядка" [[2](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#5.2)]. В-третьих, четкая система обозначений позволяет автоматизировать большую часть утомительной проверки на полноту и правильность. Как говорится в отчете управления оборонных исследований:

"Разработка программного обеспечения всегда будет кропотливой работой... Хотя наши машины могут выполнять рутинную работу и помогать нам держать нить рассуждений, разработка концепций останется прерогативой человека... Что всегда будет в цене - это искусство построения концептуальной структуры, а заботы об ее описании уйдут в прошлое" [[3](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#5.3)].

5.1. Элементы обозначений

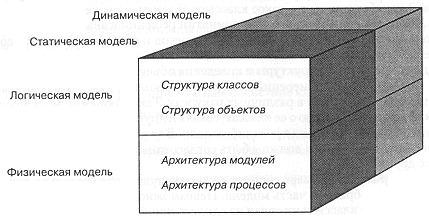
**Необходимость разных точек зрения**

Невозможно охватить все тонкие детали сложной программной системы одним взглядом. Как отмечают Клейн и Джингрич: "Необходимо понять как функциональные, так и структурные свойства объектов. Следует уяснить также таксономическую структуру классов объектов, используемые механизмы наследования, индивидуальное поведение объектов и динамическое поведение системы в целом. Эта задача в чем-то аналогична показу футбольного или теннисного матча, когда для вразумительной передачи происходящего действия требуется несколько камер, расположенных в разных углах спортивной площадки. Каждая камера передает свой аспект игры, недоступный другим камерам" [[4](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#5.4)].

На рис. 5-1 представлены различные типы моделей (описанные в главе 1), которые мы считаем главными в объектно-ориентированном подходе. Через них будут выражаться результаты анализа и проектирования, выполненные в рамках любого проекта. Эти модели в совокупности семантически достаточно богаты и универсальны, чтобы разработчик мог выразить все заслуживающие внимания стратегические и тактические решения, которые он должен принять при анализе системы и формировании ее архитектуры. Кроме того, эти модели достаточно полны, чтобы служить техническим проектом реализации практически на любом объектно-ориентированном языке программирования.

Наша система обозначений богата деталями, но это не означает, что в каждом проекте необходимо использовать все ее аспекты. На практике для описания большей части итогов анализа и проектирования достаточно некоторого малого подмножества этой системы; один наш коллега называет такие облегченные обозначения "нотацией *Booch Lite*" (сокращенная нотация Буча). В процессе знакомства с нашей системой обозначений мы четко выделим это подмножество. Зачем же тогда беспокоиться о деталях системы обозначений, не вошедших в минимальное подмножество? В основном они нужны, чтобы выражать некоторые важные тактические решения; а некоторые из них предназначены для создания программных инструментов CASE. Мы будем называть их *дополнениями*.

Как отмечает Вайнберг: "В некоторых областях (таких, как архитектура), в процессе проектирования графический план чаще всего представлен в виде общих набросков, а строго детализированные описания, пока не окончена творческая часть становления проекта, используются редко" [[5](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#5.5)]. Следует помнить, что обозначения - только средство выразить итоги размышлений над архитектурой и поведением системы, а никак не самоцель. Надо пользоваться исключительно теми элементами обозначений, которые необходимы, и не более того. Также, как опасно ставить слишком жесткие требования, нехорошо чересчур подробно описывать решение задачи. Например, на чертеже архитектор может показать расположение выключателя света в комнате, но его точное место не будет определяться, пока заказчик и прораб не произведут осмотр уже сооруженного здания для уточнения деталей отделки. Глупо было бы заранее указывать на чертеже трехмерные координаты этого выключателя (конечно, если это не является функционально важной деталью для заказчика: может быть, все члены его семьи имеют рост намного выше или ниже среднего). Если разработчики большой программной системы - высококвалифицированные специалисты и уже сработались друг с другом, им будет достаточно и грубых набросков, хотя и в этом случае они должны оставить свое творческое наследие эксплуатационщикам в удобоваримом виде. Если же разработчики неопытны, отделены друг от друга во времени или пространстве или если так установлено контрактом, то на протяжении всего процесса проектирования необходимы более детальные эскизы проекта. Система обозначений, которую мы представляем в этой главе, учитывает эти ситуации. 



*Рис. 5-1. Объектные модели.*

Различные языки программирования описывают одни и те же понятия различно. Наша система обозначений не зависит от конкретного языка. Конечно, некоторые ее элементы могут не иметь аналогов в языке, на котором будет написана проектируемая система. В этом случае просто не надо ими пользоваться. Например, в Smalltalk не бывает свободных подпрограмм, следовательно, в проект нет смысла закладывать утилиты классов. Аналогично, C++ не поддерживает метаклассы, следовательно, этот элемент должен быть исключен. Но нет ничего предосудительного в подгонке обозначений под конструкции выбранного языка. Например, в CLOS операции можно снабдить специальными пометками, чтобы определить методы **:before**, **:after** и **:around**. Подобным же образом, в системе автоматического проектирования для C++ вместо спецификаций класса можно пользоваться прямо заголовочными файлами.

Цель этой главы - определить синтаксис и семантику наших обозначений для объектно-ориентированного анализа и проектирования. В качестве примера мы будем использовать гидропонную теплицу, которая рассматривалась в главе 2. В настоящей главе не обсуждается, как, собственно, были получены представленные в примерах решения: это задача главы 6. В главе 7 обсуждаются практические аспекты этого процесса, а в главах 8-12 система обозначений демонстрируется в деле на серии примеров разработки приложений.

**Модели и ракурсы**

В главе 3 мы объяснили, что такое классы и объекты, а также какова связь между ними. Как показано на рис. 5-1, при принятии решений в анализе и проектировании полезно рассмотреть взаимодействие классов и объектов в двух измерениях: логическом/физическом и статическом/динамическом. Оба этих аспекта необходимы для определения структуры и поведения объектной системы.

В каждом из двух измерений мы строим несколько диаграмм, которые дают нам вид моделей системы в различных ракурсах. Таким образом, модели системы содержат всю информацию о ее классах, связях и других сущностях, а каждая диаграмма представляет одну из проекций модели. В установившемся состоянии проекта, все такие диаграммы должны быть согласованы с моделью, а, следовательно, и друг с другом.

Рассмотрим для примера систему, включающую в себя несколько сотен классов; эти классы образуют часть модели. Невозможно, а на самом деле и не нужно представлять все классы и их связи на единственной диаграмме. Вместо этого мы можем описать модель в нескольких диаграммах классов, каждая из которых представляет только один ее ракурс. Одна диаграмма может показывать структуру наследования некоторых ключевых классов; другая - транзитивное замыкание множества всех классов, используемых конкретным классом. Когда модель "устоится" (придет в *устойчивое состояние*), все такие диаграммы становятся семантически согласованными друг с другом и с моделью. Например, если по данному сценарию (который мы описываем на диаграмме объектов), объект **A** посылает сообщение **M** объекту **B**, то**M** должно быть прямо или косвенно определено в классе **B**. В соответствующей диаграмме классов должна быть надлежащая связь между классами **A** и **B**, так, чтобы экземпляры класса **A** действительно могли посылать сообщение**M**.

Для простоты на диаграммах все сущности с одинаковыми именами в одной области видимости рассматриваются как ссылки на одинаковые персонажи системы. Исключением из этого правила могут быть только операции, имена которых могут быть перегружены.

Чтобы различать диаграммы, мы должны дать им имена, которые отражали бы их предмет и назначение. Можно снабдить диаграммы и другими комментариями или метками, которые мы вскоре опишем; эти комментарии, как правило, не имеют дополнительной семантики.

**Логическая и физическая модели**

Логическое представление описывает перечень и смысл ключевых абстракций и механизмов, которые формируют предметную область или определяют архитектуру системы. Физическая модель определяет конкретную программно-аппаратную платформу, на которой реализована система.

При анализе мы должны задать следующие вопросы:

Каково требуемое поведение системы?

Каковы роли и обязанности объектов по поддержанию этого поведения?

Как было отмечено в предыдущей главе, чтобы выразить наши решения о поведении системы мы пользуемся сценариями. В логической модели важнейшим средством для описания сценариев служат диаграммы объектов. При анализе могут быть полезны диаграммы классов, позволяющие увидеть общие роли и обязанности объектов.

При проектировании мы должны задать следующие вопросы относительно архитектуры системы:

Какие существуют классы и какие есть между ними связи?

Какие механизмы регулируют взаимодействие классов?

Где должен быть объявлен каждый класс?

Как распределить процессы по процессорам и как организовать работу каждого процессора, если требуется обработка нескольких процессов?

Чтобы ответить на эти вопросы, мы используем следующие диаграммы:

диаграммы классов

диаграммы объектов

диаграммы модулей

диаграммы процессов.

**Статическая и динамическая модели**

Четыре введенные нами типа диаграмм являются по большей части статическими. Но практически во всех системах происходят события: объекты рождаются и уничтожаются, посылают друг другу сообщения, причем в определенном порядке, внешние события вызывают операции объектов. Не удивительно, что описание динамических событий на статическом носителе, например, на листе бумаги, будет трудной задачей, но эта же трудность встречается фактически во всех областях науки. В объектно-ориентированном проектировании мы отражаем динамическую семантику двумя дополнительными диаграммами:

диаграммами переходов из одного состояния в другое

диаграммами взаимодействия.

Каждый класс может иметь собственную диаграмму переходов, которая показывает, как объект класса переходит из состояния в состояние под воздействием событий. По диаграмме объектов, имея сценарий, можно построить диаграмму взаимодействий, чтобы показать порядок передачи сообщений.

**Инструменты проектирования**

Плохой разработчик, имея систему автоматического проектирования, сможет создать своего программного монстра за более короткий срок чем раньше. Великие проекты создаются великими проектировщиками, а не инструментами. Инструменты проектирования дают возможность проявиться индивидуальности, освобождают ее, чтобы она могла сосредоточиться исключительно на творческих задачах проектирования и анализа. Существуют вещи, которые автоматизированные системы проектирования могут делать хорошо, и есть вещи, которые они вообще не умеют. Например, если мы используем диаграмму объектов, чтобы показать сценарий с сообщением, посылаемым от одного объекта другому, автоматизированная система проектирования может гарантировать, что посылаемое сообщение будет в протоколе объекта; это пример проверки совместимости. Если мы введем инвариант, например, такой: "существует не более трех экземпляров данного класса", то мы ожидаем, что наш инструмент проследит за соблюдением данного требования; это пример проверки ограничения. Кроме того, система может оповестить вас, если какие-либо объявленные классы или методы не используются в проекте (проверка на полноту). Наконец, более сложная автоматическая система проектирования может определить длительность конкретной операции или достижимость состояния на диаграмме состояний; это пример автоматического анализа. Но, с другой стороны, никакая автоматическая система не сможет выявить новый класс, чтобы упростить вашу систему классов. Мы, конечно, можем попытаться создать такой инструмент, используя экспертные системы, но для этого понадобятся, во-первых, эксперты как в области объектно-ориентированного программирования, так и в предметной области, а во-вторых, четко сформулированные правила классификации и много здравого смысла. Мы не ожидаем появления таких средств в ближайшем будущем. В то же время, у нас есть вполне реальные проекты, которыми стоит заняться.

5.2. Диаграммы классов

**Существенное: классы и отношения между ними**

Диаграмма классов показывает классы и их отношения, тем самым представляя логический аспект проекта. Отдельная диаграмма классов представляет определенный ракурс структуры классов. На стадии анализа мы используем диаграммы классов, чтобы выделить общие роли и обязанности сущностей, обеспечивающих требуемое поведение системы. На стадии проектирования мы пользуемся диаграммой классов, чтобы передать структуру классов, формирующих архитектуру системы.

Два главных элемента диаграммы классов - это классы и их основные отношения.

**Классы.** На рис. 5-2 показано обозначение для представления класса на диаграмме. Класс обычно представляют аморфным объектом, вроде облака [Выбор графических обозначении - это трудная задача. Требуется осторожно балансировать между выразительностью и простотой, так что проектирование значков - это в большой степени искусство, а не наука. Мы взяли облачко из материалов корпорации Intel, документировавшей свою оригинальную объектно-ориентированную архитектуру iAPX432 [[6](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#5.6)]. Форма этого образа намекает на расплывчатость границ абстракции, от которых не ожидается гладкости и простоты. Пунктирный контур символизирует то, что клиенты оперируют обычно с экземплярами этого класса, а не с самим классом. Можно заменить эту форму прямоугольником, как сделал Румбах [[7](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#5.7)]: 

http://www.helloworld.ru/texts/comp/other/oop/pg177.gif

Однако, хотя прямоугольник проще рисовать, этот символ слишком часто используется в разных ситуациях и, следовательно, не вызывает ассоциаций. Кроме того, принятое Румбахом обозначение классов обычными прямоугольниками, а объектов - прямоугольниками с закругленными углами конфликтует с другими элементами его нотации (прямоугольники для актеров в диаграммах потоков данных и закругленные прямоугольники для состояний в диаграммах переходов). Облачко более удобно и для расположения пометок, которые, как мы увидим дальше, потребуются для абстрактных и параметризованных классов, и поэтому оно предпочтительнее в диаграммах классов и объектов. Аргумент простоты рисования прямоугольников спорен при использовании автоматизированной поддержки системы обозначений. Но чтобы сохранить возможность простого рисования и подчеркнуть связь с методом Румбаха, мы оставляем его обозначения классов и объектов в качестве допустимой альтернативы].

Каждый класс должен иметь имя; если имя слишком длинно, его можно сократить или увеличить сам значок на диаграмме. Имя каждого класса должно быть уникально в содержащей его категории. Для некоторых языков, в особенности - для C++ и Smalltalk, мы должны требовать, чтобы каждый класс имел имя, уникальное в системе. 



*Рис. 5-2. Значок класса.*

На некоторых значках классов полезно перечислять несколько атрибутов и операций класса. "На некоторых", потому что для большинства тривиальных классов это хлопотно и не нужно. Атрибуты и операции на диаграмме представляют прообраз полной спецификации класса, в которой объявляются все его элементы. Если мы хотим увидеть на диаграмме больше атрибутов класса, мы можем увеличить значок; если мы совсем не хотим их видеть - мы удаляем разделяющую черту и пишем только имя класса.

Как мы описывали в главе 3, атрибут обозначает часть составного объекта, или агрегата. Атрибуты используются в анализе и проектировании для выражения отдельных свойств класса [Точнее, атрибут эквивалентен отношению агрегации с физическим включением, метка которого совпадает с именем атрибута, а мощность равна в точности единице]. Мы используем следующий не зависящий от языка синтаксис, в котором атрибут может обозначаться именем или классом, или и тем и другим, и, возможно, иметь значение по умолчанию:

**A** - только имя атрибута;

**:C** - только класс;

**A:C** - имя и класс;

**A:C=E** - имя, класс и значение по умолчанию.

Имя атрибута должно быть недвусмысленно в контексте класса. В главе 3 говорилось, что операция - это услуга, предоставляемая классом. Операции обычно изображаются внутри значка класса только своим именем. Чтобы отличать их от атрибутов, к их именам добавляются скобки. Иногда полезно указать полную сигнатуру операции:

**N()** - только имя операции;

**RN(Аргументы)** - класс возвращаемого значения **(R)**, имя и формальные параметры (если есть).

Имена операций должны пониматься в контексте класса однозначно в соответствии с правилами перегрузки операций выбранного языка реализации.

Общий принцип системы обозначений: синтаксис элементов, таких, как атрибуты и операции, может быть приспособлен к синтаксису выбранного языка программирования. Например, на C++ мы можем объявить некоторые атрибуты как статические, или некоторые операции как виртуальные или чисто виртуальные [В C++ члены, общие для всех объектов класса, объявляются статическими; виртуальной называют полиморфную операцию; чисто виртуальной называют операцию, за реализацию которой отвечает подкласс]; в CLOS мы можем пометить операцию как метод **:around**. В любом случае мы пользуемся спецификой синтаксиса данного языка, чтобы обозначить детали. Как описывалось в главе 3, абстрактный класс - это класс, который не может иметь экземпляров. Так как абстрактные классы очень важны для проектирования хорошей структуры классов, мы вводим для них специальный значок треугольной формы с буквой А в середине, помещаемый внутрь значка класса (рис. 5-3). Общий принцип: украшения представляют вторичную информацию о некой сущности в системе. Все подобные типы украшений имеют такой же вид вложенного треугольника.

**Отношения между классами.** Классы редко бывают изолированы; напротив, как объяснялось в главе 3, они вступают в отношения друг с другом. Виды отношений показаны на рис. 5-4: ассоциация, наследование, агрегация (has) и использование. При изображении конкретной связи ей можно сопоставить текстовую пометку, документирующую имя этой связи или подсказывающую ее роль. Имя связи не обязано быть глобальным, но должно быть уникально в своем контексте.

Значок ассоциации соединяет два класса и означает наличие семантической связи между ними. Ассоциации часто отмечаются существительными, например **Employment** (место работы), описывающими природу связи. Класс может иметь ассоциацию с самим собой (так называемая рефлексивная ассоциация). Одна пара классов может иметь более одной ассоциативной связи. Возле значка ассоциации вы можете указать ее мощность (см. главу 3), используя синтаксис следующих примеров:

**1** - В точности одна связь

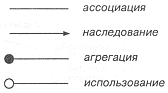
**N** - Неограниченное число (0 или больше)

**0..N** - Ноль или больше

**1..N** - Одна или больше



*Рис. 5-3. Значок абстрактного класса.*



*Рис. 5-4. Значки отношений между классами.*

**0..1** - Ноль или одна

**3..7** - Указанный интервал

**1..3, 7** - Указанный интервал или точное число

Обозначение мощности пишется у конца линии ассоциации и означает число связей между каждым экземпляром класса в начале линии с экземплярами класса в ее конце. Если мощность явно не указана, то подразумевается, что она не определена.

Обозначения оставшихся трех типов связи уточняют рисунок ассоциации дополнительными пометками. Это удобно, так как в процессе разработки проекта связи имеют тенденцию уточняться. Сначала мы заявляем о семантической связи между двумя классами, а потом, после принятия тактических решений об истинных их отношениях, уточняем эту связь как наследование, агрегацию или использование.

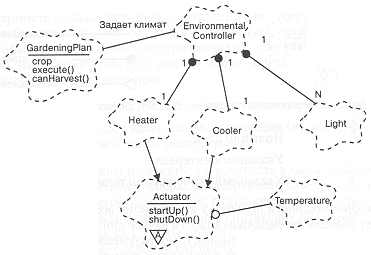
Значок наследования, представляющего отношение "общее/частное", выглядит как значок ассоциации со стрелкой, которая указывает от подкласса к суперклассу. В соответствии с правилами выбранного языка реализации, подкласс наследует структуру и поведение своего суперкласса. Класс может иметь один (одиночное наследование), или несколько (множественное наследование) суперклассов. Конфликты имен между суперклассами разрешаются в соответствии с правилами выбранного языка. Как правило, циклы в наследовании запрещаются. К наследованию значок мощности не приписывается.

Значок *агрегации* обозначает отношение "целое/часть" (связь "has") и получается из значка ассоциации добавлением закрашенного кружка на конце, обозначающем агрегат. Экземпляры класса на другом конце стрелки будут в каком-то смысле частями экземпляров класса-агрегата. Разрешается рефлексивная и циклическая агрегация. Агрегация не требует обязательного физического включения части в целое.

Знак использования обозначает отношение "клиент/сервер" и изображается как ассоциация с пустым кружком на конце, соответствующем клиенту. Эта связь означает, что клиент нуждается в услугах сервера, то есть операции класса-клиента вызывают операции класса-сервера или имеют сигнатуру, в которой возвращаемое значение или аргументы принадлежат классу сервера.

**Пример.** Описанные выше значки представляют важнейшие элементы всех диаграмм классов. В совокупности они дают разработчику набор обозначений, достаточный, чтобы описать фундамент структуры классов системы.

Рис. 5-5 показывает, как описывается в этих обозначениях задача обслуживания тепличной гидропонной системы. Эта диаграмма представляет только малую часть структуры классов системы. Мы видим здесь класс **GardeningPlan** (план выращивания), который имеет атрибут, названный **crop** (посев), одну операцию-модификатор **execute** (выполнить) и одну операцию-селектор **canHarvest** (можно собирать?). Имеется ассоциация между этим классом и классом**EnvironmentalController** (контроллер среды выращивания): экземпляры плана задают климат, который должны поддерживать экземпляры контроллера. 



*Рис. 5-5. Диаграмма классов гидропонной системы.*

Эта диаграмма также показывает, что класс **EnvironmentalController** 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 который использует класс **Temperature**.

**Существенное: категории классов**

Как объяснялось в главе 3, класс - необходимое, но недостаточное средство декомпозиции. Когда система разрастается до дюжины классов, можно заметить группы классов, связанные внутри, и слабо зацепляющиеся с другими. Мы называем такие группы *категориями классов*.

Многие объектно-ориентированные языки не поддерживают это понятие. Следовательно, выделение обозначений для категорий классов позволяет выразить важные архитектурные элементы, которые не могли быть непосредственно записаны на языке реализации [Среда программирования Smalltalk поддерживает концепцию категорий классов. Собственно это и подвигло нас на включение категорий в систему обозначений. Однако, в Smalltalk категории классов не имеют семантического содержания: они существуют только для более удобной организации библиотеки классов. В C++ категории классов связаны с концепцией компонент (Страуструп), они еще не являются чертой языка, хотя включение в него семантики пространства имен рассматривается [[8](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#5.8)]. (В настоящее время пространства имен включены в стандарт. - Примеч. ред.)]. 

http://www.helloworld.ru/texts/comp/other/oop/pic05_06.gif

*Рис. 5-6. Значок категории классов.*

Классы и категории классов могут сосуществовать на одной диаграмме. Верхние уровни логической архитектуры больших систем обычно описываются несколькими диаграммами, содержащими только категории классов.

**Категории классов.** Категории классов служат для разбиения логической модели системы. Категория классов - это агрегат, состоящий из классов и других категорий классов, в том же смысле, в котором класс - агрегат, состоящий из операций и других классов. Каждый класс системы должен "жить" в единственной категории или находиться на самом верхнем уровне системы. В отличие от класса, категория классов не имеет операций или состояний в явном виде, они содержатся в ней неявно в описаниях агрегированных классов.

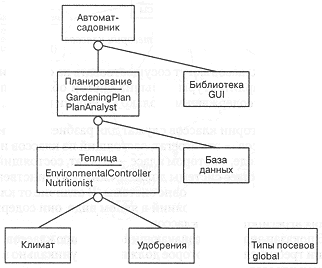
На рис. 5-6 показан значок, обозначающий категорию классов. Как и для класса, для категории требуется имя, которое должно быть уникально в данной модели и отлично от имен классов.

Иногда полезно на значке категории перечислить некоторые из содержащихся в ней классов. "Некоторые", потому, что зачастую категории содержат довольно много классов, и перечислять их все было бы хлопотно, да это и не нужно. Так же, как список атрибутов и операций на значке класса, список классов в значке категории представляет сокращенный вид ее спецификации. Если мы хотим видеть на значке категории больше классов, мы можем его увеличить. Можно удалить разделяющую черту и оставить в значке только имя категории.

Категория классов представляет собой инкапсулированное пространство имен. По аналогии с квалификацией имен в C++, имя категории можно использовать для однозначной квалификации имен содержащихся в ней классов и категорий. Например, если дан класс **A** из категории **B**, то его полным именем будет **A::B**. Таким образом, как будет обсуждаться далее, для вложенных категорий квалификация имен простирается на произвольную глубину.

Некоторые классы в категории могут быть открытыми, то есть экспортироваться для использования за пределы категории. Остальные классы могут быть частью реализации, то есть не использоваться никакими классами, внешними к этой категории. Для анализа и проектирования архитектуры это различие очень важно, так как позволяет разделить обязанности между экспортируемыми классами, которые берут на себя общение с клиентами, и внутренними классами в категории, которые, собственно, выполняют работу. На самом деле, во время анализа закрытые аспекты категории классов можно опустить. По умолчанию все классы в категории определяются как открытые, если явно не указано противное. Ограничение доступа будет обсуждаться ниже.

Категория может использовать невложенные категории и классы. С другой стороны, и классы могут использовать категории. Для единообразия мы обозначаем эти экспортно-импортные отношения так же, как отношение использования между классами (см. рис. 5-4). Например, если категория **A** использует категорию **B**, это означает, что классы из **A** могут быть наследниками, или содержать экземпляры, использовать или быть еще как-то ассоциированы с классами из **B**. 

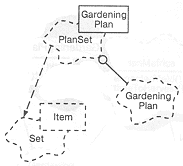


*Рис. 5-7. Диаграмма классов верхнего уровня для гидропонной системы.*

Когда в категории слишком много общих классов, вроде базовых классов-контейнеров или других базовых классов, подобных **Object** в Smalltalk, возникают практические затруднения. Такие классы будут использоваться чуть ли не всеми другими категориями, загромождая корневой уровень диаграммы. Чтобы выйти из положения, такие категории помечаются ключевым словом **global** в левом нижнем углу значка, показывающим, что категория по умолчанию может быть использована всеми остальными.

Диаграммы классов верхнего уровня, содержащие только категории классов, представляют архитектуру системы в самом общем виде. Такие диаграммы чрезвычайно полезны для визуализации слоев и разделов системы. Слой обозначает набор категорий классов одного уровня абстракции. Таким образом, слои представляют набор категорий классов, так же как категории классов - это кластеры классов. Слои обычно нужны, чтобы изолировать верхние уровни абстракции от нижних. Разделы обозначают связанные (каким-либо образом) категории классов на разных уровнях абстракции. В этом смысле слои представляют собой горизонтальные срезы системы, а разделы - вертикальные.

**Пример.** На рис. 5-7 приведен пример диаграммы классов верхнего уровня для тепличного хозяйства. Это типичная многослойная система. Здесь абстракции, которые ближе к реальности (а именно активаторы и датчики климата и удобрений), располагаются на самых нижних уровнях, а абстракции, отражающие понятия пользователя, - ближе к вершине. Категория классов **ТипыПосевов** - глобальна, то есть ее услуги доступны всем другим категориям. На значке категории классов **Планирование** показаны два ее важных класса: **GardeningPlan** (план выращивания) с рис. 5-5 и **PlanAnalyst** (анализатор планов). При увеличении любой из восьми категорий классов, показанных на рисунке, обнаружатся составляющие их классы. 



*Рис. 5-8. Значок параметризованного класса.*

**Дополнительные обозначения**

До сих пор мы занимались существенной частью нашей системы обозначений [Все существенные элементы в совокупности как раз и образуют нотацию Booch Lite]. Однако, чтобы передать некоторые часто встречающиеся стратегические и тактические решения, нам потребуется расширить ее. Общее правило: держаться существенных понятий и обозначений, а дополнительные применять только тогда, когда они действительно необходимы.

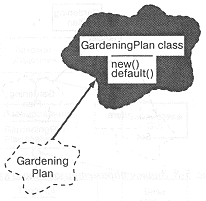
**Параметризованные классы.** В некоторых объектно-ориентированных языках программирования, в частности, C++, Eiffel и Ada можно создавать параметризованные классы. Как было сказано в главе 3, параметризованным классом называется семейство классов с общей структурой и поведением. Чтобы создать конкретный класс этого семейства, нужно подставить вместо формальных параметров фактические (процесс инстанцирования). *Конкретный класс* может порождать экземпляры.

Параметризованные классы достаточно сильно отличаются от обычных, что отмечается специальным украшением на их значках. Как показывает пример на рис. 5-8, параметризованный класс изображается значком обычного класса с пунктирным прямоугольником в правом верхнем углу, в котором указаны параметры. Инстанцированный класс изображается обычным значком класса с украшением в виде прямоугольника (со сплошной границей) с перечисленными в нем фактическими параметрами.

Связь между параметризованным классом и его инстанцированием изображается пунктирной линией, указывающей на параметризованный класс. Для получения инстанцированного класса необходим другой конкретный класс как фактический параметр **(GardeningPlan** в этом примере).

Параметризованный класс не может порождать экземпляры и не может использоваться сам в качестве параметра. Каждый инстанцированный класс является новым классом, отличающимся от других конкретных классов того же семейства.

**Метаклассы.** В некоторых языках, таких как Smalltalk и CLOS, есть метаклассы. Метакласс (см. главу 3) - это класс класса. В Smalltalk, например, метаклассы - это механизм поддержки переменных и операций класса (подобных статическим членам класса в C++), особенно фабрик класса (производящих операций), создающих экземпляры объектов данного класса. В CLOS метаклассы играют важную роль в возможности уточнения семантики языка [[9](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#5.9)]. 



*Рис. 5-9. Значок метакласса.*

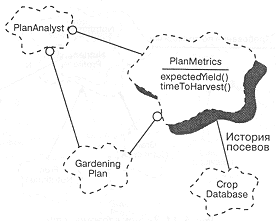
Метаклассы принципиально отличаются от обычных классов, и, чтобы подчеркнуть это, их значок закрашивается серым цветом, как это сделано на рис. 5-9. Связь между классом и его метаклассом (метасвязь) имеет вид жирной стрелки, направленной от класса к его метаклассу. Метакласс **GardeningPlan** обеспечивает методы-фабрики **new()** и**default()**, которые создают новые экземпляры класса **GardeningPlan**.

Метакласс не имеет экземпляров, но может любым образом быть ассоциирован с другими классами.

Метасвязь имеет еще одно применение. На некоторых диаграммах классов бывает полезно указать объект, который является статическим членом некоторого класса. Чтобы показать класс этого объекта, мы можем провести метасвязь "объект/ класс". Это согласуется с предыдущим употреблением: связь между некоторой сущностью (объектом или классом) и ее классом.

**Утилиты классов.** Благодаря своему происхождению, гибридные языки, такие как C++, Object Pascal и CLOS, позволяют разработчику применять как процедурный, так и объектно-ориентированный стиль программирования. Это контрастирует со Smalltalk, который целиком организован вокруг классов. В гибридном языке есть возможность описать функцию-не-член, называемую также *свободной подпрограммой*. Свободные подпрограммы часто возникают во время анализа и проектирования на границе объектно-ориентированной системы и ее процедурного интерфейса с внешним миром.

Утилиты классов употребляются одним из двух способов. Во-первых, утилиты класса могут содержать одну или несколько свободных подпрограмм, и тогда следует просто перечислить логические группы таких функций-не-членов. Во-вторых, утилиты класса могут обозначать класс, имеющий только переменные (и операции) класса (в C++ это означало бы класс только со статическими элементами [Программирующие на Smalltalk часто используют идиому утилит, чтобы достичь того же эффекта]). Таким классам нет смысла иметь экземпляры, потому что все экземпляры будут находиться в одном и том же состоянии. Такой класс сам выступает в роли своего единственного экземпляра. 



*Рис. 5-10. Значок утилиты классов.*

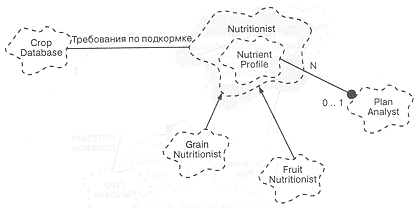
Как показано на рис. 5-10, утилита классов обозначается обычным значком класса с украшением в виде тени. В этом примере утилита классов **PlanMetrics** (параметры плана) предоставляет две важные операции: **expectedYield**(ожидаемый урожай) и **timeToHarvest** (время сбора урожая). Утилита обеспечивает эти две операции на основе услуг, предоставляемых классами нижнего уровня - **GardeningPlan** (план) и **CropDatabase** (база данных об урожае). Как показывает диаграмма, **PlanMetrics** зависит от **CropDatabase**: получает от нее информацию об истории посевов. В свою очередь, класс **PlanAnalyst** использует услуги **PlanHetrics**.

Рис. 5-10 иллюстрирует обычное использование утилит классов: здесь утилита предоставляет услуги, основанные на двух независимых абстракциях нижнего уровня. Вместо того, чтобы ассоциировать эти операции с классами высшего уровня, таких как **PlanAnalyst**, мы решили собрать их в утилиту классов и добились четкого разделения обязанностей между этими простыми процедурными средствами и более изощренной абстракцией класса-анализатора **PlanAnalyst**. Кроме того, включение свободных подпрограмм в одну логическую структуру повышает шансы на их повторное использование, обеспечивая более точное разбиение абстракции.

Связь классов с утилитой может быть отношением использования, но не наследования или агрегирования. В свою очередь, утилита класса может вступать в отношение использования с другими классами и содержать их статические экземпляры, но не может от них наследовать.

Подобно классам, утилиты могут быть параметризованы и инстанцированы. Для обозначения параметризованных утилит используются такие же украшения, как и для параметризованных классов (см. рис. 5-8). Аналогично, для обозначения связи между параметризованной утилитой класса и ее конкретизацией мы используем то же обозначение, что и для инстанцирования параметризованных классов.

**Вложенность.** Классы могут быть физически вложены в другие классы, а категории классов - в другие категории и т.д. Обычно это нужно для того, чтобы ограничить видимость имен. Вложение соответствует объявлению вложенной сущности в окружающем ее контексте. Мы изображаем вложенность физически вложенным значком; на рис. 5-11 полное имя вложенного класса - **Nutritionist::NutrientProfile**. 



*Рис. 5-11. Значок вложенности.*

В соответствии с правилами выбранного языка реализации, классы могут содержать экземпляры вложенного класса или использовать его. Языки обычно не допускают наследования от вложенного класса.

Обычно вложение классов является тактическим решением проектировщика, а вложение категорий классов - типично стратегическое архитектурное решение. В обоих случаях необходимость в использовании вложения на глубину более одного-двух уровней встречается крайне редко.

**Управление экспортом.** Все основные языки объектно-ориентированного программирования позволяют четко разделить интерфейс класса и его реализацию. Кроме того, как описано в главе 3, большинство из них позволяет разработчику определить более детально доступ к интерфейсу класса.

Например, в C++ элементы класса бывают открытыми (доступны всем клиентам), защищенными (доступны только подклассам, друзьям и самому классу) и закрытыми (доступны только самому классу и его друзьям). Кроме того, некоторые элементы могут быть частью реализации класса и тем самым быть недоступными даже друзьям этого класса [Например, объект или класс, описанный в .срр-файле, доступен только функциям-членам, реализованным в том же файле]. В Ada элементы класса могут быть открытыми или закрытыми. В Smalltalk все переменные экземпляров по умолчанию закрытые, а все операции - открытые. Доступ предоставляется самим классом и только явно: клиент ничего не может получить насильно.

Мы изображаем способ доступа следующими украшениями связи:

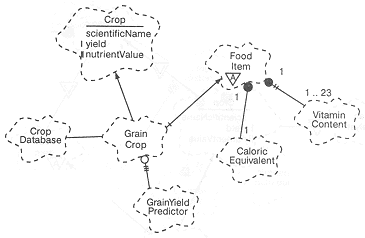
**<нет украшения>** - открытый (по умолчанию)

**|** - защищенный

**|| -** закрытый

**|||** - реализация

Мы ставим их как "засечки" на линии связи у источника. Например, на рис. 5-12 показано, что класс **GrainCrop**множественно наследует от классов **Crop** (посев) (открытый суперкласс) и **FoodItem** (пища) (защищенный суперкласс).



*Рис. 5-12. Значок управления доступом.*

**FoodItem** в свою очередь содержит от одного до двадцати трех закрытых экземпляров класса **VitaminContent**(содержание витаминов) и один открытый экземпляр класса **CaloricEquivalent** (калорийность). Заметим, что**CaloricEquivalent** мог бы быть записан как атрибут класса **FoodItem**, так как атрибуты эквивалентны агрегации, мощность которой равна 1:1. Кроме того, мы видим, что класс **GrainCrop** (посев зерновых) использует класс**GrainYieldPredictor** (предсказатель урожая зерновых) как часть своей реализации. Это обычно означает, что некоторый метод класса **GrainCrop** использует услуги, предоставляемые классом **GrainYieldPredictor**.

Кроме уже рассмотренных в этом примере случаев, обычная ассоциация так же может быть украшена символами доступа. Метасвязь (связь между инстанцированным классом и его метаклассом) не может получить таких украшений.

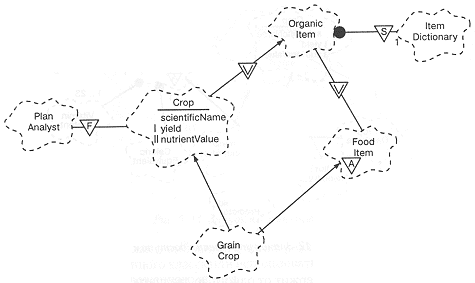
Символы ограничения доступа можно применять к вложенности во всех ее формах. На обозначении класса мы можем указать доступ к атрибутам, операциям или вложенным классам, добавив символ ограничения доступа в качестве префикса к имени. Например, на рис. 5-12 показано, что класс **Crop** имеет один открытый атрибут **scientificName**(ботаническое название), один защищенный - **yield** (урожай), и один закрытый - **nutrientValue** (количество удобрения). Такие же обозначения используются для вложенных классов или категорий классов. По умолчанию все вложенные классы и категории являются открытыми, но мы можем указать ограниченный доступ соответствующей меткой.

**Типы отношении.** В некоторых языках встречаются настолько всепроникающие типы отношений, с настолько фундаментальной семантикой, что было бы оправдано введение новых символов. В C++, например, имеется три таких конструкции:

**static -** переменная (или функция) класса;

**virtual -** совместно используемый базовый класс в ромбовидной структуре наследования;

**friend -** класс, которому даны права доступа к закрытым и защищенным элементам другого класса.



*Рис. 5-13. Значки отношений.*

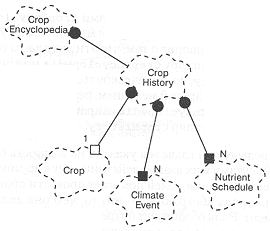
Логично использовать для них такое же украшение в виде треугольного значка, как и для абстрактного класса, но с символами **S**, **V** или **F** соответственно.

Рассмотрим пример на рис. 5-13, который представляет другой ракурс классов, показанных на предыдущем рисунке. Мы видим, что базовый класс **OrganicItem** (органический компонент) содержит один экземпляр класса **ItemDictionary**(словарь компонентов) и что этот экземпляр содержится самим классом, а не его экземплярами (то есть он является общим для всех экземпляров). В общем случае мы указываем обозначение **static** на одном из концов ассоциации или на конце связи агрегации.

Рассматривая класс **GrainCrop**, мы видим, что структура наследования приобретает ромбовидную форму (связи наследования, разветвившись, сходятся). По умолчанию, в C++ ромбовидная форма структуры наследования ведет к тому, что в классах-листьях дублируются структуры базового, дважды унаследованного класса. Чтобы класс **GrainCrop**получил единственную копию дважды унаследованных структур класса **OrganicItem**, мы должны применить виртуальное наследование, как показано на рисунке. Мы можем добавлять украшение виртуальной связи только к наследованию.

Значок дружественности можно присоединить к любому типу связи, расположив значок ближе к серверу, подразумевая, что сервер считает клиента своим другом. Например, на рис. 5-13 класс **PlanAnalyst** дружит с классом **Crop**, а, следовательно, имеет доступ к его закрытыми и защищенным элементам, включая оба атрибута **yield** и**scientificName**.

Физическое содержание. Как показано в главе 3, отношение агрегации является специальным случаем ассоциации. Агрегация обозначает иерархию "целое/часть" и предполагает, что по агрегату можно найти его части. Иерархия "целое/часть" не означает обязательного физического содержания: профсоюз имеет членов, но это не означает, что он владеет ими. С другой стороны, отдельная запись о посеве именно физически содержит в себе соответствующую информацию, такую, как имя посева, урожай и график подкормки. 



*Рис. 5-14. Физическое содержание.*

Агрегация обычно выявляется при анализе и проектировании; уточнение ее как физического содержания является детализирующим, тактическим решением. Однако, распознать этот случай важно, во-первых, для правильного определения конструкторов и деструкторов классов, входящих в агрегацию, и, во-вторых, для генерации и последовательного исправления кода.

Физическое содержание отмечается на диаграмме украшением на конце линии, обозначающей агрегацию; отсутствие этого украшения означает, что решение о физическом содержании не определено. В гибридных языках мы различаем два типа содержания:

по значению целое физически содержит часть

по ссылке целое физически содержит указатель или ссылку на часть.

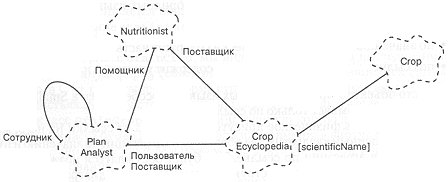
В чисто объектно-ориентированных языках, в особенности в Smalltalk, физическое содержание бывает только по ссылке.

Чтобы отличить физическое присутствие объекта от ссылки на него, мы используем закрашенный квадратик для обозначения агрегации по значению и пустой квадратик - для агрегации по ссылке. Как будет обсуждаться позже, этот стиль украшений согласуется с соответствующей семантикой на диаграммах объектов.

Рассмотрим пример, приведенный на рис. 5-14. Мы видим, что экземпляры класса **CropHistory** (история посева) физически содержат несколько экземпляров классов **NutrientSchedule** (график внесения удобрений) и **ClimateEvent**(климатическое событие). Физическое содержание частей агрегации по значению означает, что их создание или уничтожение происходит при создании или уничтожении самого агрегата. Таким образом, агрегация по значению гарантирует, что время жизни агрегата совпадает с временем жизни его частей. В противоположность этому, каждый экземпляр класса **CropHistory** обладает только ссылкой или указателем на один экземпляр класса **Crop**. Это означает, что времена жизни этих двух объектов независимы, хотя и здесь один является физической частью другого. Еще один случай - отношение агрегации между классами **CropEncyclopedia** (энциклопедия посевов) и **CropHistory**. В данном случае мы вообще не упоминаем физическое содержание. Диаграмма говорит о том, что эти два класса состоят в отношении "целое/часть", и что по экземпляру **CropEncyclopedia** можно найти соответствующий экземпляр**CropHistory**, но физическое содержание тут ни при чем. Вместо этого может быть разработан другой механизм, реализующий эту ассоциацию. Например, объект класса **CropEncyclopedia** запрашивает базу данных, и получает ссылку на подходящий экземпляр **CropHistory**.

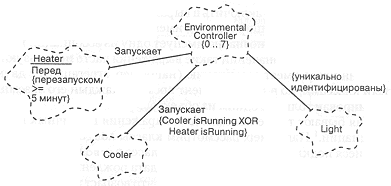
**Роли и ключи.** В предыдущей главе мы указали на важность описания различных ролей, играемых объектами в их взаимодействии друг с другом; в следующей главе мы изучим, как идентификация ролей помогает провести процесс анализа.

Коротко говоря, роль абстракции - это то, чем она является для внешнего мира в данный момент. Роль обозначает потребность или способность, в силу которых один класс ассоциируется с другим. Текстовое украшение, описывающее роль класса, ставится рядом с любой ассоциацией, ближе к выполняющему роль классу, как это видно на рис. 5-15. На этом рисунке классы **PlanAnalyst** (анализатор планов) и Nutritionist (агрохимик) оба являются поставщиками информации для объекта класса **CropEncyclopedia** (они оба добавляют информацию в энциклопедию), а объекты класса **PlanAnalyst** являются также и пользователями (они просматривают материал из энциклопедии). В любом случае, роль клиента определяет индивидуальное поведение и протокол, который он использует. Обратим внимание также на рефлексивную ассоциацию класса **PlanAnalyst**: мы видим, что несколько экземпляров этого класса могут сотрудничать друг с другом и при этом они используют особый протокол, отличающийся от их поведения в ассоциации, например, с классом **CropEncyclopedia**. 



*Рис. 5-15. Роли и ключи.*

На этом примере показана также ассоциация между классами **CropEncyclopedia** и **Crop**, но с другим типом украшения, которое представляет ключ (изображается как идентификатор в квадратных скобках). Ключ - это атрибут, значение которого уникально идентифицирует объект. В этом примере класс **CropEncyclopedia** использует атрибут**scientificName**, как ключ для поиска требуемой записи. Вообще говоря, ключ должен быть атрибутом объекта, который является частью агрегата, и ставится на дальнем конце связи-ассоциации. Возможно использование нескольких ключей, но значения ключей должны быть уникальны. 



*Рис. 5-16. Значок ограничения.*

**Ограничения.** Как говорилось в главе 3, ограничение - это выражение некоторого семантического условия, которое должно сохраняться. Иначе говоря, ограничение - инвариант класса или связи, который должен сохраняться, если система находится в стабильном состоянии. Подчеркнем - в стабильном состоянии, потому что возможны такие переходные явления, при которых меняется состояние системы в целом и система находится во внутренне рассогласованном состоянии, так что невозможно соблюсти все наложенные ограничения. Соблюдение ограничений гарантируется только в стабильном состоянии системы.

Мы используем для ограничений украшения, похожие на те, что использовались нами для обозначения ролей и ключей: помещаем заключенное в фигурные скобки выражение ограничения рядом с классом или связью, к которым оно прилагается. Ограничение присоединяется к отдельным классам, к ассоциации в целом или к ее участникам.

На рис. 5-16 мы видим, что для класса **EnviromentalController** наложено ограничение на мощность, постулирующее, что в системе имеется не более 7 экземпляров этого класса. При отсутствии ограничения на мощность класс может иметь сколько угодно экземпляров. Обозначение для абстрактного класса, введенное ранее, является специальным случаем ограничения (нуль экземпляров), но так как это явление очень часто встречается в иерархиях классов, оно получило собственный тип украшения (треугольник с буквой А).

Класс **Heater** (нагреватель) имеет ограничение другого типа. В рисунок включено требование гистерезиса в работе нагревателя: он не может быть включен, если с момента его последнего выключения прошло меньше пяти минут. Мы прилагаем это ограничение к классу **Heater**, считая, что контроль за его соблюдением возложен на экземпляры класса.

На этой диаграмме изображены еще два типа ограничений: ограничение на ассоциации. В ассоциации между классами**EnvironmentalController** и **Light** требуется, чтобы отдельные источники света были уникально индексированы относительно друг друга в контексте данной ассоциации. Имеется еще ограничение, наложенное на ассоциации**EnvironmentalController** с классами **Heater** и **Cooler**, состоящее в том, что диспетчер не может включить нагреватель и охладитель одновременно. Это ограничение прикладывается к ассоциации, а не к классам **Heater** и**Cooler**, потому что его соблюдение не может быть поручено самим нагревателям и охладителям.

При необходимости можно включить в выражение ограничения имена других ассоциаций с помощью квалифицированных имен, использованных в проекте. Например, **Cooler::** запускает однозначно именует одну из ассоциаций класса-диспетчера. В нашей системе обозначений такие выражения часто используются в ситуации, когда один класс имеет ассоциацию (например, агрегацию) с двумя (или более) другими классами, но в любой момент времени каждый его экземпляр может быть ассоциирован только с одним из объектов.

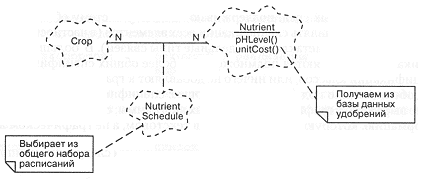
Ограничения бывают также полезны для выражения вторичных классов, атрибутов и ассоциаций [В терминологии Румбаха это называется производные сущности: для них он использует специальный значок. Нашего общего подхода к ограничениям достаточно, чтобы выразить семантику производных классов, атрибутов и ассоциации; этот подход облегчает повторное использование существующих значков и однозначное определение сущностей, от которых взяты производные]. Например, рассмотрим классы **Adult**(взрослые) и **Child** (дети), являющиеся подклассами абстрактного класса **Person** (Люди). Мы можем снабдить класс**Person** атрибутом **dateofbirth** (дата рождения) и добавить атрибут, называемый **age** (возраст), например, потому что возраст играет особую роль в нашей модели реального мира. Однако, **age** - атрибут вторичный: он может быть определен через **dateofbirth**. Таким образом, в нашей модели мы можем иметь оба атрибута, но должны указать ограничение, определяющее вывод одного из другого. Вопрос о том, какие атрибуты из каких выводятся, относится к тактике, но ограничение пригодится независимо от принятого нами решения.

Аналогично, мы могли бы иметь ассоциацию между классами **Adult** и **Child**, которая называлась бы **Parent** (родитель), а могли бы включить и ассоциацию, именуемую **Caretaker** (попечитель), если это нужно в модели (например, если моделируются официальные отношения родительства в системе социального обеспечения). Ассоциация **Caretaker**вторична: ее можно получить как следствие ассоциации **Parent**; мы можем указать этот инвариант как ограничение, наложенное на ассоциацию **Caretaker**.

**Ассоциации с атрибутами и примечания.** Последнее дополнительное понятие связано с задачей моделирования свойств ассоциаций; в системе обозначений задача решается введением элемента, который может быть приложен к любой диаграмме.

Рассмотрим пример на рис. 5-17. На нем показана ассоциация многие-ко-многим между классами **Crop** и **Nutrient**. Эта ассоциация означает, что к каждому посеву применяется N (любое число) удобрений, а каждое удобрение применяется к**N** (любому числу) посевов. Класс **NutrientSchedule** является как бы свойством этого отношения многие-ко-многим: каждый его экземпляр соответствует паре из посева и удобрения. Чтобы выразить этот семантический факт, мы рисуем на диаграмме пунктирную линию от ассоциации **Crop**/**Nutrient** (ассоциация с атрибутом) к ее свойству - классу**NutrientSchedule** (атрибут ассоциации). Каждая уникальная ассоциация может иметь не больше одного такого атрибута и ее имя должно соответствовать имени класса-атрибута.

Идея атрибутирования ассоциаций имеет обобщение: при анализе и проектировании появляется множество временных предположений и решений; их смысл и назначение часто теряются, потому что нет подходящего места для их хранения, а хранить все в голове - дело немыслимое. Поэтому полезно ввести обозначение, позволяющее добавлять произвольные текстовые примечания к любому элементу диаграммы. На рис. 5-17 имеется два таких примечания. Одно из них, приложенное к классу **NutrientSchedule**, сообщает нечто об ожидаемой уникальности его экземпляров (Выбирает из общего набора расписаний); другое (Получаем из базы данных удобрений) приложено к конкретной операции класса **Nutrient** и выражает наши пожелания к ее реализации. 



*Рис. 5-17. Ассоциация с атрибутом и примечание.*

Для таких примечаний мы используем значки, похожие на бумажки, и соединяем их с элементом, к которому они относятся, пунктирной линией. Примечания могут содержать любую информацию: обычный текст, фрагменты программ или ссылки на другую документацию (все это может пригодиться при разработке инструментов проектирования). Примечания могут быть не связаны ни с каким элементом, это значит, что они относятся к самой диаграмме [Значок, который мы используем, похож на обозначение примечаний во многих windows-системах, особенно следующих традициям Macintosh. Непосредственными вдохновителями нашего обозначения были предложения Гамма, Хелпа, Джонсона и Влиссидеса [[10](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#5.10)]].

**Спецификации**

Спецификация - это неграфическая форма, используемая для полного описания элемента системы обозначений: класса, ассоциации, отдельной операции или целой диаграммы. Просматривая диаграммы, можно относительно легко разобраться в большой системе; однако одного графического представления недостаточно: мы должны иметь некоторые пояснения к рисункам, и эту роль будут играть спецификации.

Как было сказано ранее, диаграмма - срез разрабатываемой модели системы. Спецификации же служат неграфическими обоснованиями каждого элемента обозначений. Таким образом, множество всех синтаксических и семантических фактов, нашедших свое отражение на диаграмме, должно быть подмножеством фактов, описанных в спецификации модели и согласовываться с ними. Очевидно, что важную роль в сохранении согласованности диаграмм и спецификаций может играть инструмент проектирования, поддерживающий такую систему обозначений.

В этом разделе мы рассмотрим сначала основные элементы двух важнейших спецификаций, а затем изучим их дополнительные свойства. Мы не ставим себе задачу подробного описания каждой спецификации, - оно зависит от пользовательского интерфейса конкретных сред, поддерживающих нашу систему обозначений. Мы также не будем представлять спецификации всех элементов (в частности, вне нашего внимания окажутся метакласс и отдельные типы связей). В большинстве такие спецификации или являются подмножеством более общих спецификаций, таких как спецификации классов, или ничего не добавляют к графическому представлению. Особенно важно подчеркнуть следующее: спецификация должна отражать то, что не выражено в графических элементах диаграммы; спецификации содержат ту информацию, которую лучше записать в текстовом, а не графическом виде.

**Общие элементы.** Все спецификации имеют как минимум следующие компоненты:

**Имя: идентификатор   
Определение: текст**

Уникальность имени зависит от именуемого элемента. Например, имена классов должны быть уникальны по крайней мере в содержащей их категории, тогда как имена операций имеют область видимости, локальную для содержащего их класса.

Определение - это текст, идентифицирующий представленное элементом понятие или функцию и пригодный для включения в словарь проекта (который обсуждается в следующей главе).

В каждой спецификации содержатся минимальные сведения. Конечно, используемый инструмент автоматического проектирования может вводить свои собственные графы для нужд конкретной программной среды. Однако, важно указать, что независимо от того, сколько граф включает в себя спецификация, не следует навязывать разработчику дурацкие правила, по которым он обязан заполнить все части спецификации, прежде чем приступит к следующему этапу разработки. Обозначения должны облегчать разработку, а не создавать дополнительные трудности.

**Спецификации класса.** Каждый класс в модели имеет ровно одну спецификацию, в которой содержатся как минимум следующие пункты:

**Обязанности: текст**   
**Атрибуты: список атрибутов**   
**Операции: список операций**   
**Ограничения: список ограничений**

Как говорилось в предыдущей главе, обязанности класса - это список предоставляемых им гарантий поведения. В следующей главе будет показано, как мы используем эту графу для регистрации обязанностей классов, которые мы открываем или изобретаем в процессе разработки.

Остальные пункты - атрибуты, операции, ограничения - соответствуют их графическим аналогам. Некоторые операции могут быть настолько важными, что следует снабдить их собственными спецификациями, которые мы обсудим ниже.

Перечисленные основные понятия могут быть представлены в терминах выбранного языка реализации. В частности, все эти сведения, как правило, однозначно фиксируются объявлением класса на C++ или спецификацией пакета в Ada.

Как говорилось в главе 3, часто поведение некоторых важных классов наилучшим образом выражается на языке конечного автомата, поэтому мы включим в спецификацию класса дополнительную графу:

**Автомат: ссылка на автомат**

Использование дополнительных элементов системы обозначений требует ввести в спецификацию класса следующие пункты:

**Управление экспортом: открытый | реализация**   
**Мощность: выражение**

Смысл этих пунктов вполне тождественен их графическим аналогам. Параметризованные и инстанцированные классы должны включать следующий пункт:

**Параметры: список формальных или фактических параметров**

Следующие необязательные пункты не имеют графических аналогов; они служат, чтобы указать некоторые функциональные аспекты класса:

**Устойчивость: мгновенный | постоянный**   
**Параллельность: последовательный | охраняемый | синхронный | активный**   
**Место в памяти: выражение**

Первое из этих свойств отражает продолжительность жизни объектов класса: постоянная сущность - это та, чье состояние может пережить сам объект, в отличие от мгновенных, состояние которых пропадает с истечением времени жизни объекта.

Второе свойство показывает в какой степени класс может работать в многопоточной системе (см. главу 2). По умолчанию объекты - последовательные, то есть рассчитаны на один поток. Охраняемый и синхронный классы "выдерживают" несколько потоков. При этом охраняемый класс ожидает, что клиентские потоки как-то договариваются о взаимном исключении, с тем чтобы в каждый момент времени с ним работал только один из них. Синхронный класс сам обеспечивает взаимное исключение клиентов. Наконец, активный класс имеет свой поток.

Последний пункт содержит сведения об абсолютном или относительном потреблении памяти объектами этого класса. Мы можем использовать эту графу для подсчета размера класса или его экземпляров.

**Спецификации операций.** Для всех операций-членов классов и свободных подпрограмм наши спецификации включают следующие основные пункты:

**Класс возвращаемого значения: ссылка на класс**   
**Аргументы: список формальных аргументов**

Эти графы можно заполнить на выбранном языке реализации. В соответствии с правилами языка можно включить еще один пункт:

**Квалификация: текст**

В C++, например, этот пункт может содержать утверждение о том, является ли операция статической, виртуальной, чисто виртуальной или константой.

Использование дополнительных элементов обозначений требует введения дополнительной графы:

**Доступ: открытый | защищенный | закрытый | реализация**

Содержание этой графы зависит от языка реализации. Например в Object Pascal все атрибуты и операции всегда открытые, в Ada операции могут быть открытыми или закрытыми, а в C++ возможны любые из четырех указанных случаев.

Использование дополнительных элементов обозначений требует также введения графы

**Протокол: текст**

Эта графа происходит из практики языка Smalltalk: протокол операции не имеет семантического значения, а служит просто для именования логической совокупности операций, вроде таких, как initialize-release (инициализация-освобождение) или model access (доступ к модели).

Следующие необязательные графы не имеют графических аналогов и служат для формального описания семантики операции:

**Предусловия: текст | ссылка на текст программы | ссылка на диаграмму объектов**   
**Семантика: текст | ссылка на текст программы | ссылка на диаграмму объектов**   
**Постусловия: текст | ссылка на текст программы | ссылка на диаграмму объектов**   
**Исключения: список исключительных ситуаций**

Первые три пункта могут быть заполнены в любой из перечисленных форм. Последний содержит список исключительных ситуаций, содержащий имена соответствующих классов.

Последняя серия необязательных граф служит для описания некоторых функциональных аспектов операции:

**Параллельность: последовательный | охраняемый | синхронный**   
**Память: выражение**   
**Время: выражение**

Первые две аналогичны одноименным графам в спецификации класса. Третья - относительные или абсолютные оценки времени выполнения операции.

5.3. Диаграммы состояний и переходов

**Существенное: состояния и переходы**

Диаграмма состоянии и переходов показывает: пространство состояний данного класса; события, которые влекут переход из одного состояния в другое; действия, которые происходят при изменении состояния. Мы приспособили обозначения, использованные Харелом [[11](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#5.11)]: его работа предоставляет простой, но очень выразительный подход, который гораздо эффективнее традиционных автоматов с конечным числом состояний [Мы дополнили его работу применительно к объектно-ориентированному программированию, следуя предложениям Румбаха [[12](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#5.12)] и Беара и др. [[13](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#5.13)]]. Отдельная диаграмма состояний и переходов представляет определенный ракурс динамической модели отдельного класса или целой системы. Мы строим диаграммы состояний и переходов только для классов, поведение которых (управляемое событиями) для нас существенно. Мы можем также представить диаграмму состояний и переходов для управляемого событиями поведения системы в целом. Эти диаграммы используются в ходе анализа, чтобы показать динамику поведения системы, а в ходе проектирования - для выражения поведения отдельных классов или их взаимодействия. 



*Рис. 5-18. Значок состояния.*

Два основных элемента диаграммы состояний и переходов - это, естественно, состояния и переходы между ними.

**Состояния.** Состояние представляет собой итоговый результат поведения системы. Например, только что включенный в сеть телефон находится в начальном состоянии: его предыдущее поведение несущественно, при этом он готов к тому, чтобы позвонить или принять звонок. Если кто-нибудь поднимет трубку, телефон перейдет в состояние готовности к набору номера; в этом состоянии мы не ожидаем, что телефон зазвонит, но приготовились к беседе с одним или несколькими абонентами. Если кто-либо наберет ваш номер, а телефон находится в начальном состоянии (трубка положена), то когда вы поднимете трубку, телефон перейдет в состояние с установленным соединением, и вы сможете поговорить со звонившим.

В любой момент времени состояние объекта определяет набор свойств (обычно статический) объекта и текущие (обычно динамические) значения этих свойств. Под "свойствами" подразумевается совокупность всех связей и атрибутов объекта. Мы можем обобщить понятие состояния так, чтобы оно было применимо и к объекту, и к классу, так как все объекты одного класса "живут" в одном пространстве состояний. Это пространство может представлять собой неопределенное, хотя конечное множество возможных (но не всегда ожидаемых или желаемых) состояний. На рис. 5-18 показано обозначение, которое мы используем для отдельного состояния.

Каждое состояние должно иметь имя; если оно оказывается слишком длинным, то его можно сократить или увеличить значок состояния. Каждое имя состояния должно быть уникально в своем классе. Состояния, ассоциированные со всей системой, глобальны, то есть видимы отовсюду, а область видимости вложенных состояний (дополнительное понятие) - ограничена соответствующей подсистемой. Все одноименные значки состояний на одной диаграмме обозначают одно и то же состояние.

На значках некоторых состояний полезно указать ассоциированные с ними действия. Как показано на рис. 5-18, действия обозначаются так же, как атрибуты и операции в значке класса. Мы можем увеличить значок, чтобы увидеть весь список действий, или, если нет необходимости указывать действия, можно удалить разделяющую линию и оставить только имя [Для совместимости с обозначениями Харела разделяющую линию можно вообще убрать]. Ассоциацию действий с состояниями мы обсудим позднее. 

http://www.helloworld.ru/texts/comp/other/oop/pic05_19.gif

*Рис. 5-19. Значок перехода из состояния в состояние.*

**Переходы.** Событием мы называем любое происшествие, которое может быть причиной изменения состояния системы. Изменение состояний называется переходом. На диаграмме переходов и состояний он изображается значком, показанным на рис. 5-19. Каждый переход соединяет два состояния. Состояние может иметь переход само в себя; обычно есть несколько различных переходов в одно и тоже состояние, но все переходы должны быть уникальны в том смысле, что ни при каких обстоятельствах не может произойти одновременно два перехода из одного состояния.

Например, в поведении гидропонной теплицы играют роль следующие события:

Посажена новая партия семян

.

Урожай созрел и готов к сбору

.

Из-за плохой погоды упала температура в теплице

.

Отказало охлаждающее устройство

.

Наступил заданный момент времени.

Как будет рассказано в следующей главе, идентификация событий, подобных этим, позволяет определить границы поведения системы и распределить обязанности по осуществлению этого поведения между отдельными классами.

Каждое из первых четырех перечисленных выше событий, вероятно, вызывает некоторое действие - например, начало или остановку выполнения некоторого плана сельскохозяйственных работ по посеву, включение нагревателя или посылку сигнала тревоги технику, обслуживающему систему. Отсчет времени - это другое дело: хотя секунды и минуты не имеют значения (посевы растут, очевидно, не так быстро), наступление нового часа или суток может вызвать некоторый сигнал, например, включить/выключить лампочки и изменить температуру в теплице, чтобы имитировать смену дня и ночи, необходимую для роста растений.

Действием мы называем операцию, которая, с практической точки зрения, требует нулевого времени на выполнение. Например, включение сигнала тревоги - действие. Обычно действие означает вызов метода, порождение другого события, запуск или остановку процесса. Деятельностью мы называем операцию, требующую некоторого времени на свое выполнение. Например, нагрев воздуха в теплице - деятельность, запускаемая включением нагревателя, который может оставаться включенным неопределенное время, до тех пор, пока не будет выключен явной командой.

Модель событий, передающих сообщения, которую предложил Харел, концептуально безупречна, но ее нужно приспособить к объектному подходу. При анализе мы можем давать предварительные названия событиям и действиям, в общих чертах отражая наше понимание предметной области. Однако, отображая эти понятия на классы, мы должны предложить конкретную стратегию реализации.

Событие может быть представлено символическим именем (или именованным объектом), классом или именем некоторой операции. Например, событие **CoolerFailure** (неисправность охлаждающего устройства) может обозначать либо литерал, либо имя объекта. Мы можем придерживаться той стратегии, что все события являются символическими именами и каждый класс с поведением, управляемым событиями, имеет операцию, которая распознает эти имена и выполняет соответствующие действия. Такая стратегия часто используется в архитектурах типа *модель-представление-котроллер* (model-view-controller), которая пришла из языка Smalltalk. Для большей общности можно считать события объектами и определить иерархию классов, которые представляют собой абстракции этих событий. Например, можно определить общий класс событий **DeviceFailure** (неисправность устройства) и его специализированные подклассы, такие как **CoolerFailure** (неисправность охлаждающего устройства) и **HeaterFailure** (неисправность нагревателя). Теперь извещение о событии можно связать с экземпляром класса-листа (например, **CoolerFailure**) или более общего суперкласса (**DeviceFailure**). И если выполнение некоторого действия назначено только при возникновении события класса **CoolerFailure**, то это означает, что все другие случаи отказа устройств должны намеренно игнорироваться. С другой стороны, если выполнение действия связано с событием **DeviceFailure**, то действие должно выполняться независимо от того, на каком устройстве произошел сбой. Продолжая в том же духе, мы можем сделать так, чтобы переходы из состояния в состояние были полиморфны относительно классов событий. Наконец, можно определить событие просто как операцию, такую как **GardeningPlan::execute()**. Это похоже на подход, который трактует события как имена, но в отличие от него здесь не требуется явного диспетчера событий.

Для нашего метода несущественно, какая из этих стратегий выбрана для разработки, если она последовательно проводится во всей системе. Обычно в замечаниях указывается, какая стратегия использована для данного конкретного автомата.

Действие можно записывать, используя синтаксис, показанный в следующих примерах:

**heater.startUp() -** действие

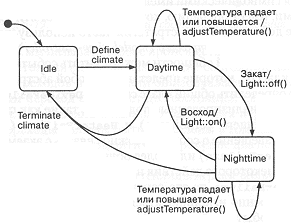
**DeviceFailure** **-** произошло событие

**start Heating** **-** начать некоторую деятельность

**stop Heating** **-** прекратить деятельность.

Имена операций или событий должны быть уникальны в области видимости диаграммы; там, где необходимо, они могут быть квалифицированы соответствующими именами классов или объектов. В случае начала или прекращения некоторой деятельности, она может быть представлена операцией (такой, как **Actuator::shutDown()**) или символическим именем (для событий). Когда деятельность соответствует некоторой функции системы, такой, как **harvest crop** (сбор урожая), мы обычно пользуемся символическими именами.

На каждой диаграмме состояний и переходов должно присутствовать ровно одно стартовое состояние; оно обозначается немаркированным переходом в него из специального значка, изображаемого в виде закрашенного кружка. Иногда бывает нужно указать также конечное состояние (обычно автомат, ассоциированный с классом или системой в целом, никогда не достигает конечного состояния; этот автомат просто перестает существовать после того, как содержащий его объект уничтожается). Мы обозначаем конечное состояние, рисуя немаркированный переход от него к специальному значку, изображаемому как кружок с закрашенной серединой. 



*Рис. 5-20. Диаграмма состояний и переходов для контроллера тепличной среды (****EnvironmentalController****).*

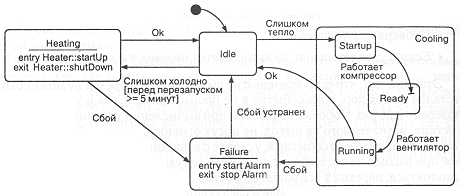
**Пример.** До сих пор вводились значки, описывающие существенные элементы диаграмм состояний и переходов. В совокупности они предоставляют разработчику систему обозначений, достаточную для моделирования простого конечного плоского автомата, пригодного для описания приложений с ограниченным числом состояний. Системы, имеющие много состояний или обладающие сильно запутанным событийным поведением, которое описывается переходами по условию или в результате предыдущих состояний, требуют для построения диаграмм переходов более сложных понятий.

На рис. 5-20 показан пример использования существенных обозначений. Пример опять описывает гидропонную систему. Мы видим диаграмму состояний и переходов для класса **EnvironmentalController**, впервые введенного на рис. 5-5.

На этой диаграмме все события представляются символическими именами. Мы видим, что все объекты этого класса начинают свою жизнь в начальном состоянии **Idle** (ожидание); затем они изменяют свое состояние по событию **Define climate**, для которого не предполагается явных действий (считается, что это событие, то есть ввод климатического задания, происходит только в дневное время). Дальше динамическое поведение этого класса состоит в переключении между со-стояниями **Daytime** и **Nighttime** (день и ночь); оно определяется событиями **Sunrise** и **Sunset** (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 (операция **adjustTemperature()**, которая является локальной в этом классе). Мы возвращаемся в состояние **Idle**, когда поступит событие **Terminate climate**, то есть будет отменено климатическое задание.

**Дополнительные понятия**

Элементы диаграмм состояний и переходов, которые мы только что описали, недостаточны для многих случаев сложных систем. По этой причине мы расширим наши обозначения, включив семантику карт состояний, предложенную Харелом. 



*Рис. 5-21. Действия, условные переходы и вложенные состояния.*

**Действия, ассоциированные с состояниями и условные переходы.** Как показано на рис. 5-18, с состояниями могут быть ассоциированы действия. В частности, можно назначить выполнение некоторого действия на входе или выходе из состояния, при этом используется синтаксис следующих примеров:

**entry start Alarm -** запуск процедуры при входе в состояние

**exit shutDown()** - вызов операции при выходе из состояния.

Как и для переходов, можно назначить любое действие после ключевых слов **entry** и **exit**(вход и выход).

Деятельность можно ассоциировать с состоянием, используя синтаксис следующего примера:

**do Cooling -** в данном состоянии заниматься этой деятельностью.

Этот синтаксис служит сокращенной записью явных указаний: "Начать деятельность при входе в состояние и окончить при выходе из него".

На рис. 5-21 мы видим пример использования этих обозначений. При входе в состояние **Heating** (нагревание) вызывается операция **Heater::startUp()**, а при выходе - операция **Heater::shutDown()**, то есть происходит запуск и остановка нагревания. При входе и выходе из состояния **Failure** (сбой), соответственно вызывается и прекращается сигнал тревоги (**Alarm**).

Рассмотрим также переход из состояния **Idle** в состояние **Heating**. Он совершается, если температура понизилась, но только в случае, если прошло больше пяти минут после того, как последний раз был выключен нагреватель. Это пример условного (или защищенного) перехода; условие представляется логическим выражением в скобках.

Вообще, каждый переход может быть ассоциирован либо с событием, либо с событием и условием. Допускаются и "переходы без события". В этом случае переход совершается сразу после завершения действия, связанного с состоянием, причем выполняется и действие, связанное с выходом из этого состояния. Если переход условный, он состоится только в случае, если условие выполнено.

Имеет значение порядок выполнения условного перехода. Пусть имеется состояние **S**, из которого при событии **E**совершается переход **T** с условием **C** и действием **A**. Переход **T** осуществляется в такой последовательности:

Происходит событие **E**.

Проверяется условие **C**.

Если **C** удовлетворено, то выполняется переход **T** и действие **A**.

Это означает, что если условие **C** не выполнено, то переход не может быть осуществлен до тех пор, пока событие **E** не произойдет еще раз и условие **C** не будет проверено еще раз. Побочные эффекты при вычислении условия или выполнении действия, назначенного на выход, не могут отменить переход. Например, предположим, что произошло событие **E**, условие **C** выполнилось, но действие **A**, выполняемое при выходе из состояния **S**, изменило ситуацию так, что условие **C** перестало выполняться: переход **T** все равно состоялся.

Мы можем использовать еще и следующий синтаксис:

**in Cooling** - выражение для текущего состояния.

Здесь используется имя состояния (которое может быть квалифицированным). Выражение истинно тогда и только тогда, когда система находится в указанном состоянии. Такие условия особенно полезны, когда некоторому внешнему состоянию нужно запустить переход по условию, связанному с некоторым вложенным состоянием.

Можно использовать в условии и выражение, налагающее ограничения по времени:

**timeout (Heating, 30)** - выражение ограничения по времени.

Это условие выполняется, если система более 30 секунд находилась в состоянии **Heating** и остается в нем в момент проверки. Этот тип условия употребляется в системах реального времени для "переходов без события", так как защищает систему от зависания на долгое время в одном состоянии. Это выражение можно использовать для указания нижней границы времени нахождения в данном состоянии. Если приложить временное ограничение к каждому переходу с событием, выводящим из данного состояния, это будет равнозначно требованию, что система находится в каждом состоянии как минимум время, указанное в ограничении [Харел предложил "обобщенную завитушку" для обозначения двухсторонних границ по времени, но мы не будем обсуждать здесь его обобщения, так как условия исчерпания времени достаточно выразительны].

Что случится, если некое событие произойдет, а перейти в другое состояние нельзя либо потому, что не существует перехода для данного события, либо не выполняется условие перехода? По умолчанию это надо считать ошибкой: игнорирование событий обычно является признаком неполного анализа задачи. Вообще, для каждого состояния нужно документировать события, которые оно намеренно игнорирует.

**Вложенные состояния.** Возможность вложения состояний друг в друга придает глубину диаграммам переходов; эта ключевая особенность карт состояний Харела предотвращает комбинаторный взрыв в структуре состояний и переходов, который часто случается в сложных системах.

На рис. 5-21 показаны внутренние детали состояния **Cooling**, то есть вложенные в него состояния; для простоты мы опустили все его действия, включая действия при входе и выходе.

Объемлющие состояния, такие, как **Cooling**, называются суперсостояниями, а вложенные, такие, как **Running**, - подсостояниями. Вложенность может достигать любой глубины, то есть подсостояние может быть суперсостоянием для вложенных состояний более низкого уровня. Данное суперсостояние **Cooling** содержит три подсостояния. Семантика вложенности подразумевает отношение **xor** (исключающее или) для вложенных состояний: если система находится в состоянии **Cooling** (охлаждение), то она находится ровно в одном из подсостояний **Startup** (начальное), **Ready**(готовность) или **Running** (выполнение).

Чтобы проще ориентироваться в диаграмме переходов с вложенными состоя-ниями мы можем увеличить или уменьшить ее масштаб относительно выбранного состояния. При уменьшении вложенные состояния исчезают, а при увеличении проявляются. Переходы в скрытые на диаграмме подсостояния и выходы из них показываются стрелкой с черточкой, как переход в состояние **Ready** на рисунке [Если быть точными, то переходы **Too hot** и **Ok**относительно состояния **Cooling** также должны быть показаны на рис. 5-21 с черточкой, так как это переходы между подсостояниями].

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

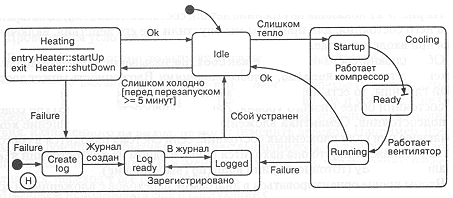
Переход между одноуровневыми состояниями (такой, как из **Failure** в **Idle** или из **Ready** в **Running**) - простейшая форма перехода; его семантика описана в предыдущем разделе.

Можно совершить переход непосредственно в подсостояние (как из **Idle** в **Startup**), или непосредственно из подсостояния (как из **Running** в **Idle**), или одновременно и то, и другое.

Указание перехода из суперсостояния (как из **Cooling** в **Failure** через событие **Failure**) означает, что он осуществляется из каждого подсостояния этого суперсостояния. Такой переход пронизывает все уровни до переопределения. Это упрощает диаграмму за счет удаления банальных переходов, общих для всех подсостояний.

Указание перехода в состояние с вложенными подсостояниями (например, предыдущий переход в состояние**Failure**) подразумевает переход к его начальному подсостоянию (по умолчанию).

**История.** Иногда, возвращаясь к суперсостоянию, мы хотели бы попасть в то его подсостояние, где мы были последний раз. Эту семантику мы будем изображать значком истории (буква **H** (History) внутри кружка, размещенного где-нибудь внутри значка суперсостояния). Например, на рис. 5-22 мы видим развернутое изображение состояния**Failure**. В самый первый раз, когда наша система переходит в него, она принимает начальное состояние по умолчанию**Create log** (создать журнал); что обозначено непомеченным переходом из закрашенного кружка внутри объемлющего состояния; когда журнал (**log**) создан, система переходит в состояние **Log** **ready**. После того, как сообщение о сбое занесено в журнал, мы возвращаемся обратно. Когда мы попадем в состояние **Failure** в следующий раз, нам не нужно будет опять создавать журнал, и мы перейдем прямо к **Log** **ready**, так как когда мы в последний раз выходили из состояния **Failure**, система находилась именно в этом подсостоянии. 



*Рис. 5-22. История событий.*

Действие "истории" распространяется только на тот уровень, на котором она указана. Если мы хотим распространить ее действие на все нижние подуровни, то мы обозначим это, пририсовав к ее значку звездочку. Можно получить промежуточный результат, пририсовав значок истории только к отдельным подсостояниям.

**Спецификации**

Каждый элемент диаграммы переходов может иметь спецификацию, которая дает его полное определение. В отличие от спецификаций классов, спецификации переходов и состояний ничего не добавляют к уже описанному в этом разделе, поэтому нет необходимости обсуждать их специально.

5.4. Диаграммы объектов

**Существенное: объекты и их отношения**

*Диаграмма объектов* показывает существующие объекты и их связи в логическом проекте системы. Иначе говоря, диаграмма объектов представляет собой мгновенный снимок потока событий в некоторой конфигурации объектов. Таким образом, диаграммы объектов являются своего рода прототипами: каждая представляет взаимодействие или структурные связи, которые могут возникнуть у данного множества экземпляров классов, безотносительно к тому, какие конкретно экземпляры участвуют в этом взаимодействии. В таком смысле, отдельная диаграмма объектов есть ракурс структуры объектов системы. При анализе мы используем диаграммы объектов для показа семантики основных и второстепенных сценариев, которые отслеживают поведение системы. При проектировании мы используем диаграммы объектов для иллюстрации семантики механизмов в логическом проектировании системы. Существенные элементы диаграммы объектов - объекты и их отношения. 



*Рис. 5-23. Значок объекта.*

*Объекты.* На рис. 5-23 показан значок, который изображает объект на диаграмме объектов. Как и в диаграммах классов, можно провести горизонтальную линию, разделяющую текст внутри значка объекта на две части: имя объекта и его атрибуты.

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

**A** - только имя объекта

**:C** - только класс объектов

**A:C** - имя объекта и класса

либо использовать синтаксис выбранного языка реализации. Если текст не умещается внутри значка, то следует или увеличить значок, или сократить текст. Если несколько значков объектов на одной диаграмме используют одно и то же неквалифицированное имя (то есть имя без указания класса), то они означают один и тот же объект. В противном случае каждый значок означает отдельный объект [На одной диаграмме могут присутствовать значки объектов с одинаковыми неквалифицированными именами, но относящиеся к разным классам, в том случае, если эти классы имеют общего предшественника. Это позволяет представить распространение операций от подкласса к суперклассу и наоборот]. Если на разных диаграммах есть объекты с одинаковыми неквалифицированными именами, то это разные объекты, если только их имена не квалифицированы явно.

Смысл неквалифицированных имен зависит от контекста диаграммы объектов. Более точно: диаграммы объектов, определенные на самом верхнем уровне системы, имеют глобальную область видимости; другие диаграммы объектов могут быть определены для категорий классов, отдельных классов или отдельных методов, а, значит, иметь соответствующие области видимости. Квалифицированное имя может быть использовано при необходимости явной ссылки на глобальные объекты, переменные классов (статические элементы в C++), параметры методов, атрибуты или локально определенные объекты в той же области видимости.

Если не указать класс объекта - ни явно, использовав ранее упомянутый синтаксис, ни косвенно, через спецификацию объекта, - то класс рассматривается как анонимный, при этом нельзя провести семантическую проверку ни операций, совершаемых над объектом, ни его связей с другими объектами на диаграмме. Если же указать только класс, то анонимным считается объект. Каждый значок без имени объекта обозначает отдельный анонимный объект.

В любом случае, имя класса объекта должно быть именем настоящего класса (или любого из его суперклассов) в области видимости диаграммы, использованного для инстанцирования объекта, даже если этот класс - абстрактный. Эти правила позволяют написать сценарий, не зная точно, к каким подклассам принадлежат объекты. 

http://www.helloworld.ru/texts/comp/other/oop/pic05_24.gif

*Рис. 5-24. Значок связи между объектами.*

На значках объектов бывает полезно указать несколько их атрибутов. "Несколько" - так как значок объекта представляет только один какой-то ракурс его структуры. Синтаксис атрибутов совпадает с описанным ранее синтаксисом атрибутов класса и позволяет указать выражение, используемое по умолчанию. Имена атрибутов объектов должны соответствовать атрибутам, определенным в классе объекта, или в любом из его суперклассов. Синтаксис имен атрибутов может быть приспособлен к синтаксису языка реализации.

Диаграмма объектов может также включать значки, обозначающие утилиты классов и метаклассы: эти понятия подобны объектам, так как они могут действовать как объекты, и с ними можно оперировать как с объектами.

**Отношения между объектами.** Как говорилось в главе 3, объекты взаимодействуют с другими объектами через связи, обозначение которых показано на рис. 5-24. Подобно тому, как объект является экземпляром класса, связь является экземпляром ассоциации.

Связь между двумя объектами (включая утилиты классов и метаклассы) может существовать тогда и только тогда, когда существует ассоциация между соответствующими классами. Ассоциация между классами может проявляться различными способами, например, как простая ассоциация, отношение наследования или отношение включения. Следовательно, существование ассоциаций между двумя классами означает существование коммуникации (то есть канала связи) между их экземплярами, по которой объекты могут посылать друг другу сообщения. Все классы неявно имеют ассоциации сами с собой и, следовательно, объект может послать сообщение сам себе.

Пусть имеются объекты **A** и **B** и связь **L** между ними. Тогда **A** может вызвать любую операцию, имеющуюся в классе **B** и доступную **A**. То же верно для операций над **A**, вызываемых **B**. Объект, вызывающий операцию, называется объект-клиент, а объект, который предоставляет операцию, - объект-сервер. Обычно отправитель сообщения знает получателя, но обратное необязательно.

В установившемся состоянии структуры классов и объектов системы должны быть согласованы. Если мы показываем на диаграмме операцию **M** на классе **B**, вызванную по связи **L**, то м должна быть объявлена в спецификации **B**, или в спецификациях его суперклассов.

Как показано на рис. 5-24, рядом с соответствующей связью на диаграмме можно записать набор сообщений. Каждое сообщение состоит из следующих трех элементов:

**D** - символ синхронизации, обозначающий направление вызова

**M** - вызов операции или извещение о событии

**S** - необязательный порядковый номер.

Мы показываем направление сообщения стрелкой, указывающей на объект-сервер. Этот символ означает простейшую форму передачи сообщений, семантика которой гарантирована только в присутствии единственного потока контроля. Существуют более развитые формы синхронизации, которые применимы в случае нескольких потоков. О них мы расскажем в следующем разделе.

Вызов операции - наиболее общая форма сообщения. Она подчиняется ранее описанному синтаксису операций, но, в отличие от него, здесь могут быть приведены фактические параметры, подходящие к сигнатуре операции:

**N() -** только имя операции

**RN(arguments) -** возвращаемое значение, имя и фактические параметры операции.

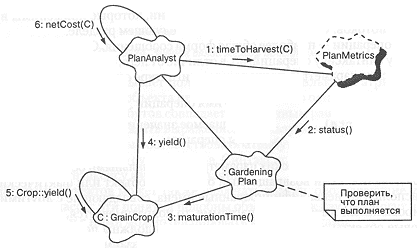
Сопоставление фактических параметров с формальными осуществляется в порядке следования. Если возвращаемый операцией объект или фактические параметры используют неквалифицированные имена, совпадающие с другими неквалифицированными именами на диаграмме, то подразумевается, что они именуют одинаковые объекты, а следовательно, их классы должны подходить к сигнатуре операции. Таким образом, мы можем представлять взаимодействия, в ходе которых объекты передаются в качестве параметров или возвращаются, как результат операции.

Сообщение может извещать о событии. Оно подчиняется определенному ранее синтаксису событий, и, следовательно, может представлять символьное имя, объект или имя некоторой операции. Во всяком случае, имя события должно быть определено на соответствующей классу объекта-сервера диаграмме переходов и состояний. Если извещение о событии является операцией, то оно может включать фактические параметры.

Если порядковый номер явно не указан, то сообщение может быть послано независимо от других сообщений, указанных на данной диаграмме объектов. Чтобы указать явный порядок событий, мы можем их пронумеровать. Нумерация начинается с единицы и добавляется как необязательный префикс к вызову операции или извещению о событии. Порядковый номер показывает относительный порядок посылки сообщений. Сообщения с одинаковыми номерами не упорядочены друг относительно друга; сообщение с меньшим порядковым номером посылается до сообщения с большим номером. Повторение порядковых номеров или их отсутствие говорит о частичной упорядоченности сообщений.

**Пример.** На рис. 5-25 показана диаграмма объектов для нашего тепличного хозяйства в контексте категории классов**Planning** (планирование; описана на рис. 5-7). Цель этой диаграммы - проиллюстрировать сценарий выполнения обычной функции системы, а именно, прогнозирование затрат на сбор урожая некоторого посева.

Выполнение этой функции требует сотрудничества нескольких различных объектов. Сценарий начинается с вызова объектом **PlanAnalyst** (анализатор планов) операции **timeToHarvest()** (время собирать урожай) над утилитой класса**PlanMetrics** (параметры планов). При этом объект с передается как фактический параметр операции. Затем утилита**PlanMetrics** вызывает операцию **status()** (состояние) на некотором неименованном объекте класса **GardeningPlan**(план выращивания). В пояснении говорится: "Надо проверить, что этот план действительно выполняется". В свою очередь, объект **GardeningPlan** вызывает операцию **maturationTime()** (время созревания) на выбранном объекте класса **GrainCrop** (посев зерновых), запрашивающую ожидаемое время созревания посева. Когда эта операция-селектор будет выполнена, управление возвращается объекту класса **PlanAnalyst**, который затем непосредственно вызывает операцию **C.yield()**, унаследованную от суперкласса (операция **Crop::yield()**). Управление снова возвращается объекту класса **PlanAnalyst**, который продолжает сценарий, выполняя над собой операцию **netCost()**. 



*Рис. 5-25. Диаграмма объектов гидропонной системы.*

На диаграмме показана связь между объектами классов **PlanAnalyst** и **GardeningPlan**. Хотя сообщения между ними не посылаются, связь отражает существование семантической зависимости между этими объектами.

**Дополнительные понятия**

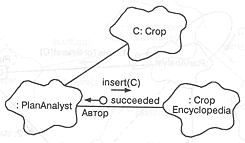
То, что мы описали, составляет существенные элементы диаграммы объектов. Однако многие запутанные вопросы разработки требуют некоторого расширения используемых обозначений. Мы предупреждали при описании диаграмм классов и хотим подчеркнуть опять: дополнительные понятия должны использоваться только при необходимости.

**Роли, ключи и ограничения.** Выше мы говорили, что на диаграмме классов при изображении ассоциации рядом с нею может быть написана ее роль, обозначающая намерение или мощность связи одного класса с другим. Для некоторых диаграмм объектов полезно заново написать эту роль при указании связи между объектами. Такая метка помогает объяснить, почему один объект оперирует над другим.

Рис. 5-26 дает пример использования этого дополнительного обозначения. Здесь мы видим, что некоторый объект класса **PlanAnalyst** заносит информацию об определенном посеве (Crop) в анонимный объект CropEncyclopedia (энциклопедия посевов) и делает это, пока находится в роли **Автор**.

Используя те же обозначения, что и для диаграммы классов, мы можем указать ключи или ограничения, ассоциированные с объектом или связью.

**Поток данных.** Как было описано в главе 3, данные могут передаваться по или против направления посылки сообщения. Иногда явное указание направления передачи данных помогает объяснить семантику конкретного сценария. Мы используем для этого значок, заимствованный из обозначений структурного проектирования. На рис. 5-26 дан пример его использования: здесь показано, что после завершения сообщения **insert** (вставить) возвращается значение **succeeded** (успех). Передаваться и возвращаться может либо объект, либо значение. 



*Рис. 5-26. Роли.*

**Видимость.** В некоторых запутанных сценариях полезно указать точно, насколько один объект видит другие. Ассоциации на диаграммах классов обозначают семантическую зависимость между классами, но не указывают точно, насколько их экземпляры видят друг друга. С этой целью мы можем украсить связи на наших диаграммах значками, иллюстрирующими видимость одного объекта другим. Эта информация важна и для инструментальных программ, генерирующих код, или наоборот, восстанавливающих по коду логическую модель.

Рис. 5-27 уточняет рис. 5-25 и содержит несколько украшений, дающих информацию о видимости. Они похожи на украшения для физического вхождения на диаграмме классов. Внутри этих украшений помещены буквенные обозначения типа видимости.

Например, канал связи от объекта **PlanAnalyst** к утилите классов **PlanMetrics** помечен буквой **G**; это значит, что утилита класса глобальна. Объект **C** по-разному виден объекту **PlanAnalyst** и объекту **GardeningPlan**: с точки зрения первого объект с класса **GrainCrop** виден как параметр некоторой операции (обозначается буквой **P**); с точки зрения второго **C** виден как атрибут или поле, то есть как часть агрегированного объекта (обозначен буквой **F** (field)).

Вообще, для указания видимости могут быть использованы следующие обозначения:

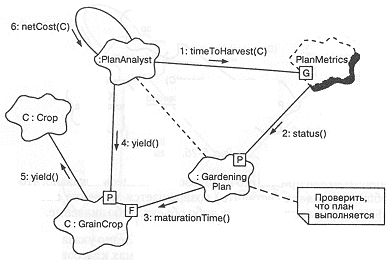
**G -** сервер глобален для клиента

**P** **-** сервер является параметром некоторой операции клиента

**F** **-** сервер является частью клиента

**L** **-** сервер локально определен в области видимости клиента.

В соответствии с украшением для физического вхождения, украшение для видимости представляет собой незакрашенный квадратик с буквой (если объект используется совместно) или закрашенный квадратик с буквой (если он не используется совместно). Если украшение видимости не указано, это означает, что решение о точном типе видимости осталось не уточненным. На практике эти украшения прилагаются только к нескольким ключевым каналам связи на диаграмме объектов. Наиболее часто эти украшения указываются для отношения "часть/целое" (агрегация) между двумя объектами; второе наиболее общее их использование - для представления объектов, которые по сценарию диаграммы посылаются как параметры. 



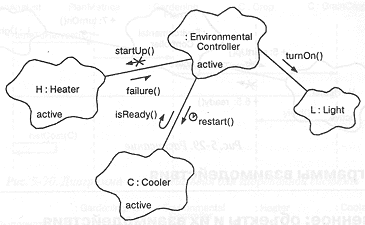
*Рис. 5-27. Значки видимости.*

**Активные объекты и синхронизация.** Как отмечалось в главе 3, некоторые объекты могут быть активными, то есть им отводится отдельный поток управления. Другие объекты могут существовать только в однопоточной среде. Третьи, будучи по природе однозадачными, гарантированно переносятся в многопоточную среду.

В каждом из этих случаев мы должны ответить на два вопроса: как выделить активные объекты, управляющие сценарием, и как представить различные формы синхронизации объектов.

Ранее, говоря о дополнительных элементах спецификаций класса, мы заметили, что есть четыре типа семантики: последовательная, защищенная, синхронизированная и активная. По существу, все объекты класса наследуют соответствующую семантику класса; все объекты считаются последовательными, если явно не указано обратное. Мы можем явно показать многозадачную семантику объекта на диаграмме объектов, указав в левом нижнем углу значка объекта одно из слов **sequential**, **guarded**, **synchronous** или **active**. Например, на рис. 5-28 мы видим, что объекты **H**,**C** и некий экземпляр класса **EnvironmentalController** - активные. Немаркированные объекты, такие как **L**, считаются последовательными.

Символ синхронизации сообщений, введенный ранее (простая стрелка), представляет обычную последовательную передачу сообщения. Однако, при наличии нескольких потоков управления мы должны указывать и другие формы синхронизации. Пример на рис. 5-28, возможно, несколько надуманный, иллюстрирует различные типы синхронизации сообщений, которые могут появиться на диаграмме объектов. Сообщение **turnOn()** (включить) - пример простой посылки сообщения; оно изображается простой стрелкой. Семантика простой посылки сообщения гарантирована только в однопоточной среде. Остальные сообщения из этого примера используют некоторые формы синхронизации процессов. Все такие дополнительные виды синхронизации применяются только к серверам, которые не являются последовательными. 



*Рис. 5-28. Активные объекты и синхронизация.*

Например, сообщение **startUp()** - синхронизированное, то есть клиент будет ждать до тех пор, пока сервер не примет сообщение. Посылка синхронизированного сообщения эквивалентна механизму свиданий задач в языке Ada (rendezvous). В случае сообщения **isReady()** клиент отложит сообщение, если сервер не сможет его немедленно обработать. Сообщение **restart()** будет отложено клиентом, если сервер не может его обработать за указанный промежуток времени.

В каждом из трех последних случаев клиент должен ждать, пока сервер обработает сообщение, или отложить пересылку, после чего может быть возобновлено управление. Сообщение **failure** имеет другую семантику. Это пример несинхронизированного сообщения, которое подразумевает, что клиент посылает событие серверу для обработки, сервер ставит сообщение в очередь, а клиент продолжает работать. Такие асинхронные сообщения сродни прерываниям.

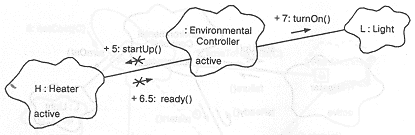
**Расписание.** В программах, имеющих ограничения по времени, важно отслеживать чистое время с момента начала каждого сценария. Для обозначения относи тельного времени (в секундах) мы ставим знак плюс. Например, на рис. 5-29 сообщение **startUp()** вызывается в первый раз спустя 5 секунд после начала сценария далее, через 6.5 секунд после начала сценария следует сообщение **ready()** и затем, спустя 7 секунд после начала сценария, - сообщение **turnOn()**.

**Спецификации**

Как и для диаграмм классов, за каждым элементом диаграммы объектов могут стоять спецификации. Спецификации объектов и их связей не несут никакой иной информации, кроме уже описанной. С другой стороны, спецификации диаграмм объектов как целого могут сообщить кое-что важное. Как упоминалось ранее, каждая диаграмма объектов существует в контексте. В спецификации контекст указывается следующим образом:

**Context: глобальный | категория | класс | операция**

В частности, область видимости диаграммы объектов может быть глобальной, или в контексте указанной категории классов, класса или операции (включая, как методы, так и свободные подпрограммы). 



*Рис. 5-29. Расписание.*

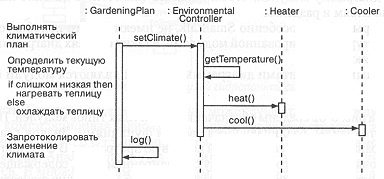
5.5. Диаграммы взаимодействия

**Существенное: объекты и их взаимодействия**

*Диаграмма взаимодействии* используется, чтобы проследить выполнение сценария в том же контексте, что и диаграмма объектов [Эти диаграммы обобщают диаграммы трассировки событий Румбаха и диаграммы взаимодействий Джекобсона [[15](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#5.15)]]. В известной степени диаграмма взаимодействия есть просто другой способ представления диаграммы объектов. Например, на рис. 5-30 мы видим диаграмму взаимодействия, которая дублирует большую часть семантики диаграммы объектов, показанной на рис. 5-25. Преимущество диаграммы взаимодействий в том, что на ней легче читается порядок посылки сообщений, а преимущество диаграммы объектов в том, что она лучше подходит для многих объектов со сложными вызовами и допускает включение другой информации: связи, значения атрибутов, роли, блок-схемы и видимость. Так как оба типа диаграмм имеют неоспоримые достоинства, мы пользуемся в нашем методе обоими [Диаграммы объектов и диаграммы взаимодействий настолько близки по семантике, что инструментальные средства могут генерировать одну диаграмму из другой с минимальной потерей информации].

Диаграммы взаимодействия не вводят новых понятий или обозначений. Скорее, они берут существенные элементы диаграммы объектов и перестраивают их. Как показывает рис. 5-30, диаграмма взаимодействий внешне напоминает таблицу. Имена объектов диаграммы взаимодействий (те же, что и на диаграмме объектов) записываются горизонтально в верхней ее строке. Под каждым из них рисуется вертикальная пунктирная линия. Отправления сообщений (которые могут обозначать события или вызовы операций) показываются горизонтальными стрелками, с тем же синтаксисом и обозначениями синхронизации, что и на диаграмме объектов. Линия, обозначающая посылку сообщения, проводится от вертикали клиента к вертикали сервера. Первое сообщение показывается на самом высоком уровне, второе ниже и т.д., таким образом отпадает надобность в их порядковых номерах.

Диаграммы взаимодействий часто лучше диаграмм объектов передают семантику сценариев на ранних этапах жизненного цикла разработки, когда еще не идентифицированы протоколы отдельных классов. Как мы расскажем в следующей главе, в начале разработки диаграммы взаимодействий обычно сконцентрированы скорее на событиях, чем на операциях, потому что события помогают определить границы системы. Когда же уточнились структуры классов, акцент смещается к диаграммам объектов, семантика которых более выразительна. 



*Рис. 5-31. Пояснения и переход управления.*

**Дополнительные понятия**

Диаграммы взаимодействия концептуально очень просты, но есть два поясняющих элемента, которые позволяют сделать их более выразительными при наличии сложных шаблонов взаимодействия.

**Пояснения.** Для сложных сценариев, использующих условия или итерации, диаграмма объектов может быть дополнена пояснениями. Как показано на примере (рис. 5-31), пояснения могут быть подписаны к любому сообщению слева от диаграммы на соответствующем уровне простым текстом, с элементами структуризации, или с использованием синтаксиса языка реализации.

**Передача управления.** Ни простейшие диаграммы объектов, ни диаграммы взаимодействий не показывают передач управления. Например, если мы показали, что объект **A** посылает сообщения **X** и **Y** другим объектам, то остается неясным, являются ли сообщения **X** и **Y** независимыми сообщениями из **A** или они были вызваны как части некоторого объемлющего сообщения **Z**. Как показано на рис. 5-31, мы можем нарисовать на вертикальной линии каждого объекта полоски, показывающие периоды, когда управление находится в этом объекте. На этом примере мы видим, что всем руководит анонимный экземпляр класса **GardeningPlan**, который, выполняя климатический план, вызывает другие методы, которые, в свою очередь,вызывают следующие методы, и, в конце концов, управление возвращается обратно к нему же.

5.6. Диаграммы модулей

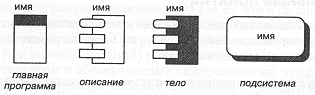
**Существенное: модули и их зависимость**

Диаграмма модулей показывает распределение классов и объектов по модулям в физическом проектировании системы. Каждая отдельная диаграмма модулей представляет некоторый ракурс структуры модулей системы. При разработке мы используем диаграмму модулей, чтобы показать физическое деление нашей архитектуры по слоям и разделам.

Некоторые языки, особенно Smalltalk, не имеют ничего подобного физической архитектуре, сформированной модулями; в таких случаях диаграммы модулей не употребляют.

Основными элементами диаграммы модулей являются модули и их зависимости.

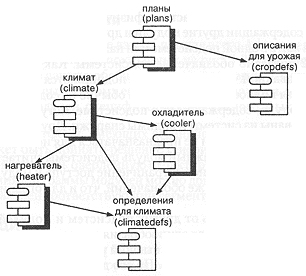
**Модули.** На рис. 5-32 сведены обозначения различных типов модулей. Первые три значка - это файлы, различающиеся своими функциями. Значок главной программы обозначает файл, содержащий корневую программу. В C++, например, это соответствовало бы некоторому файлу с расширением **.cpp** содержащему привилегированную функцию-неэлемент, называемую **main**. Обычно существует ровно один такой модуль на программу. Значок описания и значок тела обозначают файлы, которые содержат, соответственно, описания и реализации. В C++, например, модуль описаний соответствует заголовочному файлу с расширением **.h**, а модуль тела - файлу с текстом программы с расширением **.cpp**.



*Рис. 5-32. Значки модулей и подсистем.*

Смысл значка подсистемы мы раскроем в следующем разделе. Каждый модуль должен иметь имя; обычно это имя соответствующего физического файла в каталоге проекта. Как правило, такие имена пишутся без суффиксов, которые опознаются по типу значка. Если имя чересчур длинно, мы, как обычно, либо сокращаем его, либо расширяем значок. Каждое полное имя файла должно быть уникально в содержащей его подсистеме. В соответствии с правилами конкретной среды разработки, мы можем наложить ограничения на имена, такие, как условие на префиксы или требование уникальности в системе.

Каждый модуль содержит либо описание, либо определение классов и объектов, а также другие конструкции языка. По идее, "раскрыв" значок модуля на диаграмме, мы должны попасть внутрь соответствующего файла. 



*Рис. 5-33. Диаграмма модулей гидропонной системы.*

**Зависимости.** Единственная связь, которая может существовать между двумя модулями, - компиляционная зависимость - представляется стрелкой, выходящей из зависимого модуля. В C++, например, мы указываем такую зависимость директивой **#include**. Аналогично, в Ada компиляционная зависимость указывается фразой **with**. В множестве компиляционных зависимостей не могут встречаться циклы. Чтобы определить частичную упорядоченность компиляций, достаточно выполнить частичное упорядочение структуры модулей системы.

**Пример.** На рис. 5-33 показан пример обозначений модулей, в архитектуре системы тепличного гидропонного хозяйства. Мы видим здесь шесть модулей. Два из них, **climatedefs** и **cropdefs**, являются только описаниями и служат для предоставления общих типов и констант. Остальные четыре включают в себя и тела, и описания: это типичный стиль построения диаграмм модулей, так как описания и тела очень тесно связаны. На рисунке эти две части совмещены, и зависимость тела от описания получилась скрытой, хотя реально она существует. Также оказалось скрытым имя тела, но, по нашему соглашению, имена тела и описания различаются лишь суффиксами (**.cpp** и **.h**).

Зависимости на этой диаграмме предполагают частичное упорядочение компиляции. Например, тело модуля **climate**зависит от описания **heater**, которое, в свою очередь, зависит от описания **climatedefs**.Существенное: подсистемы. Как объяснялось в главе 2, 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. На практике разработчики стремятся следовать неформальному соглашению собирать связанные между собой модули в структуры типа каталогов. По этим соображениям мы введем понятие подсистемы на диаграмме модулей. Подсистемы представляют собой совокупности логически связанных модулей, примерно как категория классов представляет совокупность классов.

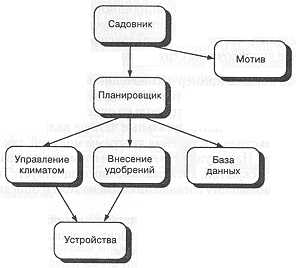
**Подсистемы.** Подсистемы служат частями физической модели системы. Подсистема - это агрегат, содержащий другие модули и другие подсистемы. Каждый модуль в системе должен жить в одной подсистеме или находиться на самом верхнем уровне.

На рис. 5-32 показано обозначение подсистемы. Как и модуль, подсистема должна быть именованной. Имена подсистем подчиняются тем же правилам, что и имена модулей, хотя полное имя подсистемы обычно не содержит суффиксов.

Некоторые модули, содержащиеся в подсистеме, могут быть общедоступны, то есть экспортированы из системы и видимы снаружи. Другие модули могут быть частью реализации подсистемы и не предназначаться для использования внешними модулями. По соглашению, каждый модуль подсистемы считается общедоступным, если явно не указано обратное. Ограничение доступа к модулям реализации достигается использованием тех же обозначений, что и для ограничения доступа в категории классов.

Подсистема может зависеть от других подсистем и модулей; модуль может также зависеть от подсистемы. Для единообразия мы используем прежнее обозначение зависимости. Система имеет один высший уровень, состоящий из подсистем и модулей высшего уровня абстракции. По его диаграмме разработчик получает представление об общей физической архитектуре системы.

**Пример.** На рис. 5-34 показан высший уровень диаграммы модулей для нашей системы тепличного хозяйства. Раскрыв любую из показанных семи подсистем, мы обнаружим все ее модули. 



*Рис. 5-34. Диаграмма модулей верхнего уровня для гидропонной системы.*

Рассмотрим, как связаны физическая и логическая (рис. 5-7) архитектуры этой системы. Они практически изоморфны, хотя имеются небольшие различия. В частности, мы приняли решение отделить классы устройств нижнего уровня от категорий классов **Климат** и **Удобрения**, и поместить соответствующие им модули в одну подсистему, названную**Устройства**. Кроме того, мы разделили категорию классов **Теплица** на две подсистемы, названные**УправлениеКлиматом** и **ВнесениеУдобрений**.

Дополнительные понятия

**Другие типы модулей.** Некоторые языки, прежде всего Ada, определяют типы модулей, отличные от простейших, показанных на рис. 5-32. Например, Ada предусматривает обобщенные пакеты, обобщенные подпрограммы и задачи как раздельно компилируемые единицы. Поэтому есть основания дополнить основные обозначения значками типов модулей, специфических для данного языка.

**Сегментация.** Для платформ, имеющих ограничения по адресации или физической памяти, может быть принято решение генерировать код в различных сегментах, или даже организовать оверлейную структуру. Чтобы отразить такую сегментацию обозначения диаграммы модулей можно дополнить, снабдив каждый модуль меткой, обозначающей соответствующий сегмент кода или данных.

**Спецификации.** Так же, как диаграммы классов и объектов, каждый элемент диаграммы модулей может иметь спецификацию, которая определяет его полностью. Спецификации модулей и их зависимостей содержат только ту информацию, которая уже описана в этом разделе, поэтому мы не будем их рассматривать.

В интегрированной инструментальной среде, поддерживающей наши обозначения, разумно использовать диаграммы модулей для визуализации программных модулей системы. "Раскрытие" модуля или подсистемы на диаграмме модулей открывает соответствующий физический файл или каталог и наоборот.

5.7. Диаграммы процессов.

Существенное: процессоры, устройства и соединения

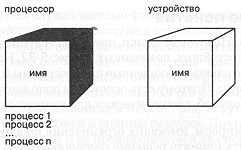
Диаграммы процессов используются, чтобы показать распределение процессов по процессорам в физическом проекте системы. Отдельная диаграмма процессов показывает один ракурс структуры процессов системы. При разработке проекта мы используем диаграмму процессов, чтобы показать физическую совокупность процессоров и устройств, обеспечивающих работу системы.

Основные элементы диаграммы процессов - процессоры, устройства и их соединения.

**Процессоры.** На рис. 5-35 показано обозначение процессора. Процессор - часть аппаратуры, способная выполнять программы. Каждый процессор должен иметь имя; никаких особых ограничений на имена процессоров нет, так как они обозначают "железо", а не программы.

Мы можем дополнить значок процессора списком процессов. Каждый процесс в таком списке обозначает или главную программу (функцию **main** из диаграммы модулей) или имя активного объекта (из диаграммы объектов).

**Устройства.** На рис. 5-35 показано обозначение устройства. Устройство - это часть аппаратной платформы, не способная выполнять программы (по крайней мере, в нашей логической модели). Как и процессорам, устройствам требуются имена, на которые не накладывается никаких существенных ограничений. 



*Рис. 5-35. Значки процессора и устройства нашей логической модели.*

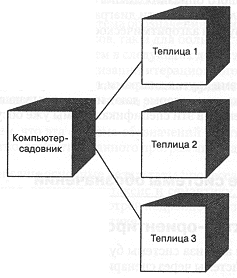
**Соединения.** Процессоры и устройства должны сообщаться друг с другом. На диаграмме процессов мы изображаем соединения между ними ненаправлеными линиями. Соединение обычно представляет непосредственную связь между аппаратурой, например, RS232, Ethernet, или даже доступ к разделяемой памяти, но эта связь может быть и не прямой, например, "Земля-спутник". Соединения обычно считаются двунаправленными, но при необходимости к их обозначению можно добавить стрелку, чтобы указать направление. Любое соединение может иметь необязательную именующую его метку.

**Пример.** На рис. 5-36 представлен пример использования этих обозначений, описывающий физическую архитектуру тепличной системы. Мы видим, что разработчики решили использовать четыре компьютера, один в качестве рабочего места оператора и по одному на каждую теплицу. Процессы, запущенные на выделенных теплицам компьютерах, не могут сообщаться друг с другом непосредственно, а только через рабочую станцию. Для простоты мы решили не показывать на этой диаграмме никаких устройств, хотя предполагаем, что система содержит большое число исполнительных устройств и датчиков.

Дополнительные понятия

**Обозначения.** На рис. 5-35 показаны стандартные обозначения, которые мы используем для процессора и устройства, но разумно и даже желательно учесть возможность их изменения. Например, можно было бы ввести специальные значки для встроенного микрокомпьютера (процессор), диска, терминала и выпрямителя тока (устройства), и использовать их на диаграммах процессов вместо стандартных. Поступая таким образом, мы предлагаем визуализацию физической платформы нашей реализации, которая предназначена непосредственно техникам и системщикам, а также конечным пользователям системы, которые, вероятно, не являются специалистами в разработке программного обеспечения.

**Вложенность.** Физическая конфигурация системы бывает очень сложной и может представлять собой иерархию процессоров и устройств. В таких случаях полезно иметь возможность выделить группы процессоров, устройств и соединений, так же, как категории классов представляют логическое группирование классов и объектов. Мы изображаем такие группы именованными пунктирными прямоугольниками с закругленными углами. Мы можем раскрыть такой значок на диаграмме процессов и обнаружить вложенные процессоры и устройства. Не представляет затруднений определить соединения между этими группами. 



*Рис. 5-36. Диаграмма процессов гидропонной системыми с закругленными углами.*

**Планирование процессов.** Мы должны некоторым образом определить порядок выполнения процессов на каждом процессоре. Имеется пять основных способов планирования, и мы можем указать на диаграмме для каждого процессора, какой из них использован, добавив к его значку одно из следующих имен: 

|  |  |
| --- | --- |
| ® Вытесняющее | Процесс с более высоким приоритетом, может отнимать процессор у исполняемого процесса с более низким приоритетом; обычно процессы с одинаковым приоритетом получают равные промежутки времени для выполнения, так что вычислительные ресурсы распределены справедливо. |
| ® Невытесняющее | Текущий процесс продолжает выполняться на процессоре до тех пор, пока сам не уступит контроль над ним. |
| ® Циклическое | Процессам по очереди выделяется равное количество процессорного времени, обычно называемое квантом времени, по истечении которого управление передается другому процессу. Процесс может получать время в квантах и подквантах. |
| ® Алгоритмическое | Переключением процессов управляет некоторый алгоритм. |
| ® Ручное | Пользователь извне системы управляет переключением процессов. |

Для более подробного описания диспетчеризации процессов на конкретном процессоре бывает полезно привести диаграмму объектов или взаимодействий, особенно если используется алгоритмическое переключение.

**Спецификации.** По аналогии с элементами других диаграмм, процессоры, устройства и соединения могут иметь спецификации, которые дают их исчерпывающее определение. Всю информацию, включаемую в эти спецификации, мы уже обсудили в текущем разделе.

5.8. Применение системы обозначений

Результат объектно-ориентированного проектирования

Обычно результатами анализа системы будут наборы диаграмм объектов (чтобы выразить поведение системы через сценарии), диаграмм классов (чтобы выразить роли и обязанности агентов по поддержанию заданного поведения системы) и диаграммы состояний и переходов (чтобы показать упорядоченное событиями поведение этих агентов). Проектирование системы, в которое входит разработка ее архитектуры и реализации, порождает диаграммы классов, объектов, модулей, процессов, а также динамические ракурсы этих диаграмм.

Существует сквозная связь между этими диаграммами, позволяющая нам проследить требования от реализации обратно к спецификации. Начав с диаграмм процессов, можно найти главную программу, которая определена на некоторой диаграмме модулей. Эта диаграмма модулей содержит наборы классов и объектов, определения которых мы найдем на подходящих диаграммах классов или объектов. Наконец, определения отдельных классов указывают на наши исходные требования, потому что эти классы, в общем, непосредственно отражают словарь предметной области.

Описанной в этой главе системой обозначений можно пользоваться вручную, хотя, конечно, она просто напрашивается на автоматизацию. Автоматизированным инструментам проектирования можно поручить проверку целостности, ограничений и полноты документации. Они также помогают разработчику легко и быстро просматривать результаты анализа и разработки. Например, глядя на диаграмму модулей, разработчик может пожелать выяснить устройство конкретного механизма, и автоматизированный инструмент поможет ему отыскать все классы, объявленные в каком-то модуле. А от диаграммы объектов, описывающей сценарий, в котором использован один из классов, разработчик может перейти к структуре наследования этого класса. Наконец, если в сценарии есть активный объект, разработчик может использовать автоматизированный инструмент проектирования, чтобы отыскать процессор, которому выделен соответствующий поток управления, и увидеть анимированное поведение конечного автомата класса на этом процессоре. Использование автоматизированных инструментов позволяет освободить разработчика от бремени согласования деталей, позволяя ему сосредоточиться на творческих аспектах процесса проектирования.

Увеличение и уменьшение масштаба

Мы считаем, что описанная здесь система обозначений годится как для маленьких систем, содержащих несколько классов, так и для больших проектов с несколькими тысячами классов. Как мы покажем в следующих двух главах, эта система обозначений особенно удобна для организации итерационного процесса разработки. К диаграммам не следует относиться как к застывшему догмату, а скорее наоборот, нужно постоянно отражать на них все новые решения, принятые в процессе проектирования.

Мы также считаем, что эта система обозначений годится для реализации на разных языках объектно-ориентированного программирования.

В этой главе были описаны основные результаты процесса объектно-ориентированного проектирования, включая синтаксис и семантику. В двух последующих главах процесс разработки будет рассмотрен подробнее. Оставшиеся пять глав повествуют о применении метода на практике.

Выводы

Проектирование - это не рисование диаграмм; диаграммы просто отражают результаты проектирования.

При проектировании сложной системы важно рассмотреть ее в различных ракурсах - как с точки зрения логической/физической структуры, так и статической/динамической семантики.

Система обозначений объектно-ориентированного проектирования включает четыре основных диаграммы (классов, объектов, модулей, процессов) и две дополнительные (состояний и переходов, взаимодействий).

Диаграмма классов показывает, какие существуют классы и связи между ними в логической структуре системы. Конкретная диаграмма классов - один из ракурсов полной структуры классов системы.

Диаграмма объектов показывает, какие существуют объекты и связи между ними в логической структуре системы. Диаграмма объектов используется для представления сценария.

Диаграмма модулей показывает распределение классов и объектов по модулям в физической структуре системы. Диаграмма модулей - один из ракурсов модульной архитектуры системы.

Диаграмма процессов показывает распределение процессов по процессорам в физической структуре системы. Каждая диаграмма процессов - один из ракурсов архитектуры процессов системы.

Диаграмма состояний и переходов показывает: (1) пространство состояний экземпляров данного класса; (2) события, которые влекут переход из одного состояния в другое; (3) действия, которые происходят при изменении состояния.

Диаграмма взаимодействий позволяет следить за выполнением сценария в контексте диаграммы объектов.

Процесс

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Программисты-любители все время ищут какой-то волшебный инструмент, который мог бы сделать процесс разработки программ тривиальным. Признак профессионализма - понимание того, что такой панацеи не существует. Любители стремятся действовать по "поваренной книге"; профессионалы же знают, что безупречно грамотный подход ведет к нелепым проектным решениям. За словом "система проектирования" разработчики пытаются спрятаться от ответственности за ошибки в проектных решениях. Любители либо игнорируют документацию вообще, либо выстраивают весь проект вокруг нее, заботясь больше о том, как продукт выглядит на бумаге, чем о его сути. Профессионал признает, что без документации не обойтись, но никогда не поступится ради нее полезными архитектурными новациями.  Процесс объектно-ориентированного анализа и проектирования не сводится к сумме рецептов, однако он определен достаточно хорошо, чтобы быть предсказуемым и воспроизводимым в умелых руках. В этой главе мы подробно рассмотрим его как итеративно развивающийся процесс, описав цели, виды деятельности, результаты и меры прогресса, характерные для его различных фаз.  6.1. Основные принципы  **Характерные черты удачных проектов**  Удачным проектом мы назовем тот, который удовлетворил (по возможности, превзошел) ожидания заказчика, уложился во временные и финансовые рамки, легко поддается изменению и адаптации. Пользуясь этим критерием, рассмотрим следующие две черты, которые оказались общими для всех встречавшихся нам удачных проектов, и, что замечательно, отсутствовали у тех, которые кажутся нам неудачными:  Ясное представление об архитектуре создаваемой системы;  Хорошо организованный итеративно развивающийся процесс работы над проектом.  **Архитектура.** Признак добротности архитектуры - ее концептуальное единство и целостность. По утверждению Брукса, "концептуальная целостность в проектировании важнее всего" [[1](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#6.1)]. Как показано в главах 1 и 5, архитектура объектно-ориентированной программной системы содержит структуры классов и объектов, имеющие горизонтальное и вертикальное слоение. Обычно конечному пользователю нет дела до архитектуры системы. Однако, как указывает Страуструп, "ясная внутренняя структура" играет важную роль в построении системы, которая будет понятна, тестируема, устойчива и сможет развиваться и перестраиваться [[2](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#6.2)]. Более того, именно ясность архитектуры дает возможность выявить общие абстракции и механизмы, которые можно свести воедино, тем самым делая систему проще, меньше и надежнее.  Не существует единственно верного способа классифицировать абстракции и разрабатывать архитектуру. В любой предметной области всегда достаточно глупейших путей проектирования, но, если поискать, можно найти и весьма элегантные. Как же отличить хорошую архитектуру от плохой?  Как правило, хорошая архитектура тяготеет к объектной ориентированности. Это не означает, что любая объектно-ориентированная архитектура оказывается хорошей, или что хороша только объектно-ориентированная архитектура. Однако, как было показано в главах 1 и 2, применение принципов объектно-ориентированной декомпозиции приводит к архитектуре, обладающей требуемыми свойствами организованной сложности.  Хорошей архитектуре присущи следующие свойства:  Она представляет собой многоуровневую систему абстракций. На каждом уровне абстракции сотрудничают друг с другом, имеют четкий интерфейс с внешним миром и основываются на столь же хорошо продуманных средствах нижнего уровня.  На каждом уровне интерфейс абстракции строго отграничен от реализации. Реализацию можно изменять, не затрагивая при этом интерфейс. Изменяясь внутренне, абстракции продолжают соответствовать ожиданиям внешних клиентов.  Архитектура проста, то есть не содержит ничего лишнего: общее поведение достигается общими абстракциями и механизмами.  Мы различаем стратегические и тактические решения. *Стратегическое решение* имеет важное архитектурное значение и связано с высоким уровнем системы. Механизмы обнаружения и обработки ошибок, парадигмы интерфейса пользователя, политика управления памятью, устойчивость объектов, синхронизация процессов, работающих в реальном масштабе времени, - все это стратегические архитектурные решения. В противоположность этому,*тактическое решение* имеет только локальное архитектурное значение и поэтому обычно связано с деталями интерфейса и реализации абстракций. Протокол класса, сигнатура метода, выбор алгоритма - все это тактические архитектурные решения.  Хорошая архитектура всегда демонстрирует баланс между стратегическими и тактическими решениями. При слабой стратегии даже очень изящно задуманный класс не сможет вполне соответствовать своей роли. Самые прозорливые стратегические решения будут разрушены, если не уделить должного внимания разработке отдельных классов. В обоих случаях пренебрежение архитектурой рождает программные эквиваленты анархии и неразберихи.  **Цикл итеративного развития.** Рассмотрим две крайности - полное отсутствие формализованного жизненного цикла разработки и очень жесткие, строго соблюдаемые правила разработки. В первом случае мы имеем анархию; тяжким трудом (преимущественно нескольких своих членов) команда разработчиков в конце концов может родить что-то стоящее, но состояние проекта всегда будет неизмеримо и непредсказуемо. Следует ожидать, что команда отработает весьма неэффективно, а, может быть, и вообще не создаст ничего пригодного для передачи заказчику. Это - пример проекта в свободном падении [Есть шанс, что проект в свободном падении приземлится благополучно, но вам не нужно ставить в связи с этим на кон будущее своей компании]. Во втором случае мы имеем диктатуру, в которой инициативы наказуемы, экспериментирование, которое могло бы привнести больше элегантности в архитектуру, не поощряется, и действительные требования заказчика никогда корректно не доходят до разработчиков нижнего уровня, скрытых за настоящей бумажной стеной, воздвигнутой бюрократией.  Встречавшиеся нам удачные объектно-ориентированные проекты не следовали ни анархическому, ни драконовскому жизненному циклу. Зато мы заметили, что удачная объектно-ориентированная архитектура создается в итеративно развивающемся процессе. Проектирование является *итеративным*, повторяющимся, в том смысле, что уже созданная архитектура вновь и вновь подвергается анализу и проектированию. При этом в каждом цикле анализ-проектирование-эволюция стратегические и тактические решения *развиваются*, приближаясь к требованиям конечного пользователя (часто даже не высказанным), оставаясь при этом простыми, надежными и открытыми для дальнейшего изменения.  Итеративно развивающийся процесс является антитезой традиционного "водопада" и не сводится к одностороннему движению сверху-вниз или снизу-вверх. Обнадеживающие прецеденты этого стиля есть в опыте создания как аппаратуры, так и программ [[3](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#6.3), [4](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#6.4)]. Например, пусть надо сформировать штат фирмы, занимающейся проектированием и изготовлением сложной уникальной аппаратуры. Можно использовать горизонтальный подход, когда проект катится водопадом, так, что архитекторы передают его конструкторам, а те электронщикам. Это - пример проектирования сверху-вниз, когда мы приглашаем узких (хотя и глубоких) специалистов в своей области [[5](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#6.5)]. Можно пойти по другому пути, наняв мастеров на все руки, каждому из которых можно поручить вертикальный сегмент проекта от начала до конца. Это уже гораздо больше похоже на итеративно развивающийся процесс.  По нашему мнению, процесс объектно-ориентированного проектирования не сводится к одностороннему движению сверху-вниз или снизу-вверх. Друк считает, что хорошо структурированные сложные системы можно создать методом "возвратного проектирования" (round-trip gestalt design). В этом методе основное внимание уделяется процессу поступательного итеративного развития путем совершенствования различных, но, тем не менее, совместимых между собой логических и физических моделей системы. Мы считаем, что возвратное проектирование составляет необходимую основу процесса объектно-ориентированного проектирования.  В отдельных случаях решаемая задача может быть уже хорошо изучена и много раз запрограммирована. Процесс разработки можно привести в идеальный порядок: проектировщики новой системы уже понимают, какие абстракции являются главными; они уже знают, какие механизмы нужно использовать и каким, в общих чертах, будет поведение системы. Творчество все еще важно в таком процессе, но здесь проблема достаточно сужена и большинство стратегических решений предопределены. Тогда, поскольку риск исключен, можно достичь очень высоких показателей производительности [[6](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#6.6)]. Чем больше мы знаем о задаче, тем легче ее решить.  Большинство промышленных задач не таковы: они связаны с балансированием уникальных требований к функциональности и эффективности и требуют полной творческой отдачи всего коллектива разработчиков. Более того, любая человеческая деятельность, которая требует творчества и инноваций, идет путем проб и ошибок, итеративно развивающегося процесса, который опирается на опыт, компетентность и талант каждого члена коллектива [Эксперименты Кертиса и его коллег подкрепляют эти наблюдения. Они изучали работу профессиональных разработчиков программного обеспечения, записывая видеокамерой их действия и затем анализируя их содержание (анализ, проектирование, реализация и т.п.) и время на выполнение. В результате исследований был сделан вывод, что "создание программ представляется набором итеративных, плохо упорядоченных и взаимно перекрывающихся процессов под приспосабливающимся управлением... Развитие по сбалансированной схеме сверху-вниз проявляется как особый случай, когда схема проектирования оказалась вполне подходящей или задача мала по размеру... Хорошие проектировщики работают одновременно на нескольких уровнях абстракции и детализации" [[8](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#6.8)]]. Так что нет и не будет стандартных рецептов для проектирования программных систем.  **Рациональный процесс проектирования**  Однако мы не можем обойтись без рецептов, описывая обещанную выше зрелую, воспроизводимую в любой организации технологию разработки. Поэтому мы и характеризовали ее, как управляемый итеративно развивающийся процесс - управляемый в том смысле, что он поддается проверке и измерению, но оставляет достаточную свободу для творчества.  Упорядоченный процесс проектирования чрезвычайно важен для организаций, разрабатывающих программное обеспечение. Хэмфри перечисляет следующие пять уровней зрелости таких процессов [[7](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#6.7)]:    |  |  | | --- | --- | | ® Начальный | Процесс разработки организован как придется и нередко хаотичен. На этой стадии налаживание элементарного управления проектом - уже прогресс. | | ® Воспроизводимый | Организация в разумной степени управляет своими планами и обязательствами. | | ® Определенный | Процесс разработки в разумной степени определен, понятен и применяется на практике; он позволяет выбирать команду и предсказывать ход разработки. Следующая цель - оформить выработанную практику разработки как инструментальную среду. | | ® Управляемый | Организация выработала количественные показатели процесса. Цель состоит в снижении затрат на сбор данных и налаживание механизмов обратной связи, позволяющих данным влиять на процесс. | | ® Оптимальный | Организация имеет отлаженный процесс, устойчиво выдающий результаты высокого качества, своевременно, предсказуемо и эффективно. |   К сожалению, как отмечают Парнас и Клеменс: "Мы никогда не отыщем процесс, который дал бы нам возможность проектировать программы строго рациональным образом", поскольку дело это творческое и новаторское по определению. Однако, продолжают они, "хорошей новостью является, то, что мы можем его имитировать... (Поскольку) разработчики нуждаются в руководстве, мы приблизимся к рациональной разработке, если будем следовать процессу, а не действовать, как попало. Когда организация занята многими программными продуктами, есть смысл иметь стандартную процедуру... Если мы держим в голове идеальный процесс, становится легче измерять успехи проекта" [[9](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#6.9)].  С приобретением опыта у организации встает вопрос: "Как примирить творчество и новации с возрастающей управляемостью?". Ответ состоит в разграничении макро- и микроэлементов процесса проектирования. Микропроцесс родственен спиральной модели развития, предложенной Боемом, и служит каркасом для итеративного подхода к развитию [[10](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#6.10)]. Макропроцесс близок к традиционному "водопаду" и задает направляющие рамки для микропроцесса. Примиряя эти два в корне различных процесса, мы имитируем полностью рациональный процесс разработки и обретаем основу для определенного уровня зрелости в деле создания программного обеспечения.  Мы должны подчеркнуть, что каждый проект уникален, и, следовательно, разработчик сам должен поддерживать баланс между неформальностью микропроцесса и формальностью макропроцесса. Для исследовательских приложений, разрабатываемых тесно сплоченной командой высококвалифицированных разработчиков, чрезмерная формальность негативно отразится на новациях; для очень сложных проектов, разрабатываемых большим коллективом разработчиков, отделенных друг от друга пространством и временем, недостаток формальности приводит к хаосу.  Оставшаяся часть этой главы дает обзор и детальное описание целей, результатов, видов деятельности и измеримых характеристик, составляющих микро- и макропроцессы разработки. В следующей главе мы рассмотрим практические проявления этих процессов, в первую очередь с точки зрения менеджеров, которые должны надзирать за ходом объектно-ориентированного проекта.  6.2. Микропроцесс проектирования  **Обзор**  Микропроцесс объектно-ориентированной разработки приводится в движение потоком сценариев и архитектурных продуктов, которые порождаются и последовательно уточняются в макропроцессе. Микропроцесс, по большей части, - повседневный труд отдельного разработчика или небольшого коллектива разработчиков.  Микропроцесс относится в равной степени к программисту и архитектору программной системы. С точки зрения программиста, микропроцесс предлагает руководство в принятии бесчисленного числа ежедневных тактических решений, которые являются частью процесса создания и подгонки архитектуры системы. С точки зрения архитектора, микропроцесс является основой для развития архитектуры и опробования альтернатив.  В микропроцессе традиционные фазы анализа и проектирования умышленно перемешаны, а управление осуществляется "по возможности". Как отмечает Страуструп, "не существует рецептов, которые могли бы заменить ум, опыт и хороший вкус в проектировании и программировании... Различные фазы программного проекта, такие, как проектирование, программирование и тестирование, неотделимы друг от друга" [[11](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#6.11)].  Как показано на рис. 6-1, микропроцесс обычно состоит из следующих видов деятельности:  выявление классов и объектов на данном уровне абстракции;  выяснение семантики этих классов и объектов;  выявление связей между этими классами и объектами;  спецификация интерфейса и реализация этих классов и объектов.  Теперь рассмотрим каждый из этих видов деятельности подробно.  **Выявление классов и объектов**  **Цель.** Цель выявления классов и объектов состоит в том, чтобы найти границы предметной области. Кроме того, эта деятельность является первым шагом в продумывании объектно-ориентированной декомпозиции разрабатываемой системы.  Мы применяем этот шаг в анализе, когда обнаруживаем абстракции, составляющие словарь предметной области и ограничиваем нашу задачу, решая, что важно, а что - нет. Такие действия необходимы при проектировании, когда мы изобретаем новые абстракции, которые являются составными частями решения. Переходя к программной реализации, мы применяем процедуру выявления, чтобы изобрести простые абстракции, из которых строятся более сложные, и обнаружить общие черты существующих абстракций, дабы упростить архитектуру системы.  **Результаты.** Главным результатом этого шага является обновляющийся по мере развития проекта словарь данных. Вначале достаточно составить список действующих лиц, состоящий из всех заметных классов и объектов, названых именами, отражающими их смысл [[12](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#6.12)]. Когда словарь разрастется, можно сделать простейшую базу данных, или более специальный инструмент проектирования, непосредственно поддерживающий выбранный метод разработки [Формально, словарь данных объектно-ориентированной разработки должен содержать спецификации каждого элемента архитектуры]. В своих более формальных разновидностях словарь данных служит предметным указателем для всех остальных компонентов проекта, включая диаграммы и спецификации обозначений объектно-ориентированного проектирования.   http://www.helloworld.ru/texts/comp/other/oop/pic06_01.gif  *Рис. 6-1. Микропроцесс.*  Таким образом, словарь данных - центральное хранилище относящихся к системе абстракций. Вначале допустимо держать словарь данных открытым для изменений: некоторые персонажи могут оказаться классами, некоторые - объектами, другие - атрибутами, а иные - просто синонимами других абстракций. Постепенно содержимое словаря уточняется путем введения новых, исключения лишних и объединения схожих абстракций.  Создание словаря данных на этом шаге дает три существенных выигрыша. Во-первых, сама работа с ним помогает выработать общепринятую и исчерпывающую терминологию, которой можно пользоваться на протяжении всего проекта. Во-вторых, словарь - естественное оглавление ко всем материалам проекта и система точек входа для доступа к проекту в произвольном порядке. Это особенно полезно, когда в команду принимается новый разработчик, который должен быстро войти в курс дел. В-третьих, словарь данных позволяет архитектору окинуть весь проект единым взглядом, что может привести к открытию новых общностей, которые иначе могли бы быть упущены.  **Виды деятельности.** Как мы описывали в главе 4, выявление классов и объектов связано с двумя видами творческих актов: открытием и изобретением.  Не каждый член команды должен быть равно искусен во всем. Аналитики, особенно работающие с экспертами в предметной области, должны уметь хорошо обнаруживать абстракции, то есть находить осмысленные классы и объекты в предметной области. Тем временем архитекторы и старшие разработчики придумывают классы и объекты, решающие чисто программистские проблемы. Мы обсудим природу этих творческих актов в следующей главе.  В любом случае основой для выявления классов и объектов служат методы классификации, описанные в главе 4. Обычный порядок действий таков:  Применить классический подход к классификации (см. раздел 4.2, "Объектно-ориентированный анализ"), чтобы получить множество кандидатов в классы и объекты. В начале жизненного цикла хорошими стартовыми точками являются материальные элементы и их роли. Затем исследовать последовательности событий, что даст другие абстракции первого и второго порядка: в конце концов, для каждого события мы должны иметь объект, который отвечает за его обнаружение и/или обработку.  Применить технику анализа поведения (см. там же) и выявить абстракции, которые непосредственно связаны с функциональными точками системы. Функциональные точки системы, как будет сказано подробнее в этой главе, берутся из макропроцесса и представляют отдельные проверяемые и внешне наблюдаемые поведения системы. Как и в случае событий, для каждого поведения можно найти классы и объекты, которые инициируют его и участвуют в нем.  Для соответствующих сценариев, созданных в макропроцессе, применить технику анализа вариантов (см. там же). В начале жизненного цикла мы исследуем самые общие сценарии поведения системы. В процессе разработки мы постепенно переходим ко все более детализированным сценариям, добираясь до самых темных уголков поведения системы.  В каждом из этих подходов CRC-карточки являются эффективным катализатором "мозгового штурма" и помогают теснее сплотить коллектив, подталкивая его членов к общению [Это ужасно банально, но некоторые проектировщики программ и в самом деле не очень общительны].  Некоторые классы и объекты будут определены в начале жизненного цикла проекта неправильно, но это не всегда плохо. Многие осязаемые вещи и роли, которые мы перечислим в жизненном цикле, пройдут через весь путь вплоть до реализации - настолько они фундаментальны для нашей концептуальной модели. Разбираясь в задаче, мы, вероятно, будем изменять границы некоторых абстракций, перераспределяя ответственности, объединяя подобные или (чаще всего), разбивая большие абстракции на группы взаимодействующих, формируя таким образом некоторые механизмы решения.  **Путевые вехи и характеристики.** Мы благополучно завершим эту фазу, когда будем иметь достаточно стабильный словарь данных. Поскольку микропроцесс развивается итеративно, следует ожидать, что словарь будет закончен и закрыт лишь на очень поздней стадии проекта. Пока нас удовлетворяет обильный, даже избыточный набор абстракций с содержательными именами и разумным распределением обязанностей.  Признаком качества, следовательно, будет то, что словарь не подвергается серьезным изменениям каждый раз, когда мы проходим новую итерацию микропроцесса. Неустойчивость словаря показывает, что разработчики еще не достигли желаемого, или в архитектуре что-то не так. По ходу разработки мы можем контролировать устойчивость нижних уровней архитектуры, отслеживая результаты локальных изменений взаимодействующих абстракций.  **Выяснение семантики классов и объектов**  **Цель.** Цель выяснения семантики классов и объектов - определить поведение и атрибуты каждой абстракции, выявленной на предыдущем шаге. При этом мы уточняем намеченные абстракции, продуманно и измеримо распределяя между ними обязанности.  На стадии анализа мы применяем этот шаг, чтобы распределить обязанности между различными видами поведения системы. На стадии проектирования мы применяем процедуру выяснения семантики, чтобы четко распределить обязанности между частями реализации. При реализации мы продвигаемся от описаний ролей и обязанностей в свободной форме к спецификациям конкретных протоколов для каждой абстракции и, в конечном счете, - к точным сигнатурам каждой операции.  **Результаты.** На этом шаге получаются несколько результатов. Первым является уточнение словаря данных, с помощью которого мы изначально присвоили обязанности абстракциям. В ходе проектирования мы можем выработать спецификации к каждой абстракции (как описано в главе 5), перечисляя имена операций в протоколе каждого класса. Затем, как можно скорее, мы выразим интерфейсы этих классов на языке реализации. Для C++ это означает создание .h-файлов, в Ada - спецификаций пакетов, в CLOS - обобщенных функций для каждого класса, в Smalltalk - это объявление, но не реализация методов каждого класса. Если проект связан с базой данных, особенно с объектно-ориентированной, на этом шаге мы получаем общий каркас нашей схемы данных.  В добавление к этим, по сути тактическим решениям, мы составляем диаграммы объектов и диаграммы взаимодействий, передающие семантику сценариев, создаваемых в ходе макропроцесса. Эти диаграммы формально отражают рас-кадровку каждого сценария и, таким образом, описывают явное распределение обязанностей среди взаимодействующих объектов. На этом шаге впервые появляются конечные автоматы для представления некоторых абстракций.  Чтобы команда разработчиков могла развивать согласованный язык обозначений и для учета обязанностей каждой абстракции, мы можем, как и на предыдущем шаге, использовать специализированную базу данных или другие, более специфические инструменты проектирования. Когда мы напишем на выбранном языке формальные интерфейсы классов, мы можем использовать наши инструменты проектирования для проверки и гарантии выполнения принятых решений.  Главная выгода большей формальности результатов на этом шаге состоит в том, что она помогает разработчику увидеть назначение всех протоколов абстракции. Невозможность четко определить смысл - признак зыбкости самих абстракций.  **Виды деятельности.** С этим шагом связано три вида деятельности: раскадровка, проектирование изолированных классов и поиск шаблонов.  Главными объектами раскадровки являются основные и второстепенные сценарии, полученные в макропроцессе. В ходе этой деятельности происходит нисходящее выяснение семантики. Там, где это касается функциональных точек системы, принимаются стратегические решения. Типичный ход выполнения действий может быть таким:  Выбрать сценарий (или группу сценариев), связанный с отдельной функциональной точкой; на основании результатов предыдущего шага определить относящиеся к этому сценарию абстракции.  Проследить действия в этом сценарии, наделяя каждую абстракцию обязанностями, достаточными, чтобы получить требуемое общее поведение. Если необходимо - выбрать атрибуты, которые будут представлять структурные элементы, требуемые для выполнения отдельных обязанностей.  По ходу раскадровки перераспределить обязанности так, чтобы сбалансировать поведение. Где возможно, использовать или адаптировать уже существующие обязанности. Очень распространенным приемом является деление больших обязанностей на малые; иногда тривиальные обязанности объединяются в более сложные.  Неформально мы можем использовать для раскадровки CRC-карточки. Для большей формальности команде разработчиков следует составить диаграммы объектов и взаимодействий. На стадии анализа раскадровка обычно выполняется командой, включающей, как минимум, аналитика, эксперта в предметной области, архитектора и контролера качества. На стадии проектирования и позже, при реализации, раскадровка выполняется архитектором и старшими разработчиками для доводки стратегических решений, и отдельными разработчиками - для доводки тактических решений. Привлечение дополнительных членов команды к участию в раскадровке - в высшей степени эффективный путь обучения начинающих разработчиков и передачи им сложившегося видения архитектуры.  В начале разработки проекта мы можем задавать семантику классов и объектов в свободной форме, просто описывая обязанности каждой абстракции. Обычно достаточно фразы или предложения; если этого мало - мы встречаем верный признак того, что данная обязанность является чрезмерно сложной и должна разделиться на меньшие. На более поздних стадиях разработки, когда мы будем заниматься доводкой протоколов отдельных абстракций, можно указать имена специфических операций, не определяя их полные сигнатуры, которые мы выясним потом. Таким образом, мы получим соответствие: каждая обязанность выполняется набором операций, а каждая операция как-либо участвует в выполнении обязанностей соответствующей абстракции. После этого, чтобы отразить динамическую семантику протоколов классов [Как мы описывали в главе 3, протокол определяет, что некоторые операции должны вызываться в определенном порядке. Для всех случаев кроме самых тривиальных операции редко встречаются в одиночестве; выполнение каждой из них имеет свои предусловия, проверка которых часто требует вызова других операции], имеющих управляемое событиями или зависящее от состояния поведение, мы можем построить конечные автоматы для них.  На этом шаге важно сосредоточить внимание больше на поведении, чем на структуре. Атрибуты представляют структурные элементы, а, значит, есть опасность, особенно на ранних стадиях анализа, преждевременным указанием некоторых атрибутов стеснить реализационные решения. Атрибуты должны идентифицироваться на этом этапе лишь настолько, насколько они необходимы в построении концептуальной модели сценария.  Проектирование изолированных классов - это восходящее выяснение семантики. Здесь мы концентрируем наше внимание на отдельных абстракциях и, применяя описанные в главе 3 эвристики для проектирования классов, рассматриваем их операции. Это действие по своей природе более тактическое, потому что здесь мы затрагиваем проектирование классов, а не архитектуры. Порядок его выполнения может быть следующим:  Выбрать одну абстракцию и перечислить ее роли и обязанности.  Определить необходимое множество операций, удовлетворяющих этим обязанностям. Попытаться, где возможно, использовать операции для концептуально схожих ролей и обязанностей повторно.  Рассмотреть каждую операцию абстракции: если она не примитивна - выделить и определить примитивы. Составные операции могут быть оставлены в самом классе (либо из-за их общности, либо по соображениям эффективности) или могут быть отправлены в утилиту классов (если они будут часто изменяться). Где это возможно следует рассмотреть минимальный набор примитивных операций.  Учесть конструирование, копирование и уничтожение объектов [[13](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#6.13)]. Если не имеется причин поступить иначе, лучше иметь общие стратегические принципы для таких операций, чем позволить отдельным классам вводить свои собственные решения.  Придать завершенность: добавить другие примитивные действия, которые не нужны существующим клиентам, но "округляют" абстракцию, что повышает вероятность использования ее новыми клиентами. Помня, что невозможно иметь полную завершенность, стремиться к простоте.  Важно избегать преждевременного определения отношения наследования - это часто ведет к потере целостности типа.  На ранних этапах разработки проектировать отдельные классы можно изолировано. Однако, как только мы определим структуры наследования, этот шаг будет включать в себя размещение операций в иерархии классов. Рассматривая операции, связанные с некоторым уровнем абстракции, мы должны решить, на каком уровне абстракции их разместить. Операции, которые могут быть использованы несколькими классами одного уровня, должны быть помещены в их общий суперкласс, который, возможно, придется создать. Действия, которые совместно используются никак не связанными классами, должны быть инкапсулированы в класс-примесь.  Третий вид деятельности - поиск шаблонов - связан с обобществлением абстракций. Выявляя семантику классов и объектов, мы должны отмечать шаблоны поведения, которые могут пригодиться где-нибудь еще. Этот процесс может проистекать в следующем порядке:  Имея полный набор сценариев на этом уровне абстракции, найти шаблоны взаимодействия абстракций. Такие взаимодействия могут представлять неявные идиомы или механизмы. Они должны быть исследованы, чтобы гарантировать, что не имеется никаких необоснованных различий в вызовах операций. Нетривиальные шаблоны взаимодействия нужно явно документировать как стратегические решения, чтобы они по возможности могли быть повторно использованы, а не изобретались заново. Это повышает архитектурную целостность.  Имея набор обязанностей для данного уровня абстракции, отыскать шаблоны поведения. Общие роли и обязанности должны быть унифицированы в форме общих классов - базовых, абстрактных или примесей.  Если уже специфицированы конкретные операции, найти шаблоны среди сигнатур операций. Если среди них встречаются часто повторяющиеся, устранить все непринципиальные различия и ввести классы-примеси или утилиты классов.  Выяснение и описание семантики применяется к категориям классов так же, как к отдельным классам. Семантика классов и их категорий определяет роли, обязанности и операции. Для отдельного класса операции могут быть со временем выражены как его функции-члены; в случае категории классов эти операции представляют экспортируемые из категории услуги, и в конечном счете реализуются набором сотрудничающих классов или отдельным классом. Таким образом, действия, описанные выше, применимы и к проектированию классов, и к проектированию архитектуры.  **Путевые вехи и характеристики.** Мы благополучно завершим этот шаг, когда будем иметь более или менее достаточный, примитивный и полный набор обязанностей и/или операций для каждой абстракции. В начале разработки достаточно иметь неформальный список обязанностей, а в дальнейшем мы постепенно уточняем семантику.  Качественные показатели включают все эвристики классов, описанные в главе 3. Сложные и туманные обязанности и операции говорят о том, что абстракции еще недостаточно определены. Невозможность написать конкретный файл заголовков или как-либо по другому формализовать интерфейс классов также говорит о том, что абстракции плохо сформулированы, или что основные понятия определяли не те люди [Остерегайтесь аналитиков и архитекторов, если они не хотят или не могут выразить конкретно семантику своих абстракции; это признак надменности или беспомощности].  **При просмотре сценариев ожидайте бурных дебатов.** Это помогает разработчикам делиться архитектурными представлениями и развивать искусство определения абстракций. Не проверенные абстракции не стоит пытаться кодировать.  **Выявление связей между классами и объектами**  **Цель.** Цель выявления связей между классами и объектами - уточнить границы каждой обнаруженной ранее в микропроцессе абстракции и опознать все сущности, с которыми она взаимодействует. Это действие формализует концептуальное и физическое размежевание между абстракциями, начатое на предыдущем шаге.  Мы применяем этот шаг в анализе для спецификации связей между классами и объектами (включая некоторые важные отношения наследования и агрегации).  Существование ассоциации подразумевает некоторую семантическую зависимость между двумя абстракциями и возможность перехода от одной сущности к другой. Этот этап проектирования нужен, чтобы специфицировать взаимодействия, которые формируют механизмы нашей архитектуры и группирование классов в категории и модулей в подсистемы. В ходе реализации мы приводим ассоциации к более конкретному виду: инстанцирование, использование и т.д.  **Результаты.** Основными результатами этого шага являются диаграммы классов, объектов и модулей. Хотя в конце концов мы должны выразить наши решения, принятые при анализе и проектировании, на языке программирования, диаграммы дают более широкий обзор архитектуры и, кроме того, позволяют раскрыть отношения, которые с трудом формулируются на используемом языке реализации.  При анализе мы составляем диаграммы классов, на которых указываются ассоциации между абстракциями, и добавляем к ним детали, полученные на предыдущем шаге (операции и атрибуты некоторых абстракций), необходимые, чтобы передать суть наших решений. При проектировании мы уточняем эти диаграммы, чтобы отразить принятые тактические решения о наследовании, агрегации, инстанцировании и использовании.  Нет ни возможности, ни необходимости создавать исчерпывающий набор диаграмм, которые определили бы все возможные виды связей между нашими абстракциями. Нужно сосредоточиться на "интересных отношениях", причем подразумевается, что в число "интересных" входят те связи между абстракциями, которые отражают фундаментальные архитектурные решения или выражают детали, необходимые для реализации.  Результатом анализа на данном этапе являются диаграммы классов, которые содержат категории классов, идентифицирующие кластеры абстракций, сгруппированные по слоям и разделам. Эти результаты пригодятся и для документирования.  При анализе мы также строим диаграммы объектов, завершая тем самым просмотр сценариев, начатый на предыдущем шаге. Отличие в том, что мы можем теперь рассмотреть взаимодействия между классами и объектами и обнаружить скрытые ранее общие механизмы взаимодействия, которыми следует воспользоваться. Обычно это приводит к локальным перестройкам структуры наследования. При проектировании мы пользуемся диаграммами объектов вместе с более детализированным описанием состояний, чтобы показать действие наших механизмов в динамике. Явный результат этого шага - набор диаграмм, которые идентифицируют механизмы взаимодействия.  При реализации мы должны принять решения о физическом разбиении нашей системы на модули и о распределении процессов по процессорам. Эти решения мы можем выразить на диаграммах модулей и процессов.  На этом же шаге также обновляется словарь данных. В нем отражаются распределения классов и объектов по категориям и модулей по подсистемам.  Основная польза полученных результатов в том, что они помогают наглядно показать и понять отношения, которыми связаны концептуально и физически далекие сущности.  **Виды деятельности.** С этим шагом связано три вида деятельности: спецификация ассоциаций, идентификация различных взаимодействий и уточнение ассоциаций. Спецификация ассоциаций является одним из основных действий в анализе и на ранней стадии проектирования. Как объяснялось в главе 3, ассоциации семантически слабы: они обозначают только некоторую семантическую зависимость, роль каждого участника связи и кардинальность связи и, возможно, направление допустимого перехода. Однако для анализа и ранней стадии проектирования этого часто достаточно, ибо передаются все важные детали связей между двумя абстракциями, при этом предохраняя нас от поспешных решений о реализации. Типичный порядок выполнения данного этапа таков:  Выбрать множество классов данного уровня абстракции или ассоциированных с некоторым набором сценариев; нанести на диаграммы все важнейшие операции и атрибуты, необходимые для иллюстрации существенных свойств моделируемой задачи.  Выяснить наличие зависимости между каждыми двумя классами и установить ассоциацию, если она присутствует. Необходимость перехода от одного объекта к другому и неизбежность использования некоторого поведения другого объекта являются причиной введения ассоциации. Чтобы устранить косвенные зависимости, следует ввести новые абстракции, которые служили бы агентами или посредниками. Некоторые ассоциации могут быть сразу идентифицированы как отношение "частное/общее" или агрегации.  Для каждой ассоциации определить роль каждого участника, если необходимо уточнить кардинальность и выявить другие ограничения.  Проверить годность этих решений, для чего просмотреть сценарий и убедиться, что имеющиеся ассоциации необходимы и достаточны для получения требуемых переходов и поведения абстракций этого сценария.  Диаграммы классов - основные модели, получаемые на данном этапе. Идентификация взаимодействий происходит главным образом при проектировании и, как описано в главе 4, является задачей классификации. А, значит, она также требует творчества и интуиции. В зависимости от текущего состояния макропроцесса, мы должны рассмотреть несколько различных типов взаимодействия:  Как часть формулировки наших стратегических решений, мы должны составить для каждого определенного на предыдущем шаге механизма диаграмму объектов, иллюстрирующую его динамическую семантику. Проверить каждый механизм в центральных и периферийных сценариях. Где возможен параллелизм, назначить объекты - актеры, агенты и серверы и способы синхронизации между ними. При этом может понадобиться ввести новые связи между объектами и устранить неиспользованные или избыточные.  Если между классами наблюдается общность, необходимо поместить эти классы в иерархию "общее/частное". Как говорилось в главе 3, обычно лучше создать "лес" классов, чем единое дерево. На предыдущем шаге мы уже определили кандидатов на базовые, абстрактные классы и классы-примеси; теперь нужно разместить их в структуре наследования. Для существенных классов следует рассмотреть диаграммы классов и оценить их качество, согласно эвристикам главы 3. В частности, требует особого внимания иерархическая структура: она не должна быть слишком высокой или слишком короткой, чересчур широкой или узкой. Там, где встречаются шаблоны в структуре или поведении, нужно реорганизовать иерархию так, чтобы максимизировать общность (но не в ущерб простоте).  Как часть архитектурного проектирования, мы должны рассмотреть группирование классов в категории и организацию модулей в подсистемы. Это - стратегические решения. Архитекторы могут использовать диаграммы классов, чтобы определить иерархию категорий классов, которая формирует слои и разделы разрабатываемой системы. Обычно это делается сверху вниз. Имея глобальное представление о системе, выделяют основные абстракции, выполняющие главные обязанности системы, которые являются логически связными и могут изменяться независимо. Архитектуру также можно модернизировать снизу вверх, когда при каждом прохождении через микропроцесс идентифицируются семантически замкнутые группы классов. Нужно также принять решения о распределении классов по категориям. Если существующие категории слишком раздуваются или обнаруживаются новые группы классов, можно ввести новые категории или реорганизовать старые. Выявление модулей (для физической модели системы) выполняется аналогично и принятые решения отражаются на диаграммах модулей.  Распределение классов и объектов по модулям является до некоторой степени локальным решением и чаще всего отражает отношения видимости абстракций. Как мы указывали в главе 5, отображение логической модели в физическую дает возможность разработчику открыть или ограничить доступ к каждой абстракции или упаковать вместе логически связанные абстракции, которые предполагается изменять по отдельности. Как мы обсудим в следующей главе, на отображение логической модели в физическую влияет также распределение обязанностей в команде проектировщиков. В любом случае все принятые решения можно выразить в виде диаграммы модулей.  Третий вид деятельности в этой фазе микропроцесса - уточнение ассоциаций - относится и к анализу, и к проектированию. При анализе мы можем провести вместо некоторых ассоциаций другие, семантически более точные связи, чтобы отразить наши достижения в понимании прикладной области. Таким образом, преобразовывая ассоциации и добавляя новые конкретные связи, мы готовим набросок реализации.  Отношения наследования, агрегации, инстанцирования и использования - важнейшие типы ассоциаций, представляющие для нас интерес вместе с такими свойствами, как метки, роли, кардинальность и т.д. Типичный порядок уточнения ассоциаций таков:  Имея набор классов, уже разбитый на группы, следует найти шаблоны поведения, указывающие на возможную связь "общее/частное". Далее необходимо разместить эти классы в существующей структуре наследования или построить новую подходящую структуру.  Если имеются шаблоны структуры, то, используя наследование с классами-примесями или агрегацию, попробовать ввести новые классы, отражающие общность структуры.  Найти классы с похожим поведением, которые либо находятся на одном уровне, либо еще не входят в структуру наследования и рассмотреть возможность введения общих параметризованных классов.  Рассмотреть существующие ассоциации с точки зрения переходов между ними и ограничить их насколько возможно. Если не требуется двустороннего перехода, считать связь простым отношением использования.  Определить тактические детали: указать роли, ключи, кардинальность, дружественность и т.д. Не требуется излишне детализировать: достаточно включить лишь важные результаты анализа и проектирования или то, что необходимо для реализации.  **Путевые вехи и характеристики.** Мы благополучно завершим эту фазу, когда достаточно полно определим семантику и связи интересующих абстракций, чтобы приступить к началу реализации.  Меры качества - связность, зацепление и полнота. Пересматривая связи, которые мы обнаружили или изобрели в течение этой фазы, мы хотим получить связные и слабо зацепленные между собой абстракции. При этом мы должны идентифицировать все важные связи на данном уровне абстракции, чтобы реализация не требовала введения новых существенных связей или неестественного использования тех, которые мы уже определили. Если на следующем шаге обнаружится, что наши абстракции неудобны для реализации, то это будет признаком того, что мы еще не определили подходящего набора связей между ними.  **Реализация классов и объектов**  **Цель.** На этапе анализа реализация классов и объектов нужна, чтобы довести существующие абстракции до уровня, достаточного для обнаружения новых классов и объектов на следующем уровне абстракции; они сами будут в дальнейшем поданы на новую итерацию микропроцесса. При проектировании целью реализации становится создание осязаемого представления наших абстракций путем выпуска последовательных исполнимых версий системы (макропроцесс).  Этот шаг намеренно выполняется позже всех, так как микропроцесс концентрирует внимание на поведении и откладывает насколько возможно решения о представлении. Такая стратегия оберегает разработчика от недозрелых решений, которые могут не оставить шансов на облегчение и упрощение архитектуры, и оставляет свободу выбора реализации (например, из соображений эффективности), гарантируя сохранение существующей архитектуры.  **Результаты.** На этом шаге мы принимаем решения о представлении каждой абстракции и об отображении этих абстракций в физическую модель. В начале процесса разработки мы формулируем эти тактические решения о представлении в форме уточненных спецификаций классов. Решения, имеющие общий интерес, или подходящие для повторного использования, мы документируем также на диаграммах классов (показывающих их статическую семантику), состояний и взаимодействия (показывающих их динамическую семантику). Когда становится ясно, на каком языке реализовывать проект, можно начинать программировать в псевдокоде или в исполнимом коде.  Чтобы раскрыть связи между логическим и физическим в нашей реализации системы, мы вводим диаграммы модулей, которые можно затем использовать, чтобы наглядно показать отображение нашей архитектуры в ее программную реализацию. Далее можно применить специфические инструментальные средства, которые позволяют либо генерировать код из диаграмм, либо восстанавливать диаграммы по реализации.  В этот шаг входит и обновление словаря данных, включая новые классы и объекты, которые были выявлены или изобретены при реализации существующих абстракций. Эти новые абстракции являются частью исходной информации для следующего цикла микропроцесса.  **Виды деятельности.** С реализацией связано одно главное действие: выбор структур и алгоритмов, которые представляют семантику определенных ранее микропроцессом абстракций. В отличие от первых трех стадий микропроцесса, сосредоточенных на внешних представлениях абстракций, этот этап акцентирует внимание на их внутреннем представлении.  На стадии анализа результаты этого действия относительно абстрактны: мы не так обеспокоены собственно реализацией, как заинтересованы в отыскании новых абстракций, которым можно делегировать обязанности. На стадии проектирования, особенно на поздних стадиях проектирования классов, мы действительно переходим к практическим решениям.  Типичный порядок действий таков:  Пересмотреть протокол каждого класса. Идентифицировать стереотипы его использования объектами-клиентами, чтобы определить, какие операции являются центральными и, следовательно, должны быть оптимизированы. Для облегчения реализации разработать точные сигнатуры всех важнейших операций.Рассмотреть возможность использования параметризованных классов, закрытого или защищенного наследования в реализации. Выбрать подходящие классы-примеси или параметризованные классы (или создать новые, если задача достаточно общая) и соответствующим образом изменить структуру наследования.  Рассмотреть объекты, которым можно делегировать обязанности. Для достижения эффективности может потребоваться незначительная реорганизация обязанностей и/или протокола абстракции нижнего уровня.  Если семантика абстракции не может быть выражена через наследование, инстанцирование или делегирование, рассмотреть подходящее представление из примитивов языка. Выбрать то представление, которое оптимизирует стереотипы использования, учитывая важность операций с точки зрения объектов-клиентов абстракции. Однако помните, что невозможно оптимизировать каждый случай использования. Получив эмпирическую информацию из последовательных версий-прототипов, мы можем выделить абстракции, которые неэффективно используют время или память и улучшить их реализацию, не опасаясь нарушить предположения клиентов относительно нашей абстракции.  Выбрать подходящий алгоритм для каждой операции. Ввести вспомогательные операции для расчленения сложных алгоритмов на более простые или более пригодные для повторного использования части. Рассмотреть возможные компромиссы, в частности, сделать выбор между хранением и вычислением отдельных членов-данных.  **Путевые вехи и характеристики.** На стадии анализа мы считаем, что благополучно завершили фазу реализации, когда идентифицировали все важные абстракции из тех, что необходимы для выполнения обязанностей абстракций, выявленных на этом цикле микропроцесса. На стадии проектирования реализация считается благополучно завершенной, когда мы получили исполнимую или почти исполнимую программную модель наших абстракций.  Главным показателем благополучия на этой фазе является простота. Сложные, неуклюжие или неэффективные реализации свидетельствуют о недостатках самой абстракции или о плохом ее представлении.  6.3. Макропроцесс проектирования  **Обзор**  Макропроцесс является контролирующим по отношению к микропроцессу. Макропроцесс предписывает ряд измеримых результатов и действий, которые позволяют команде разработчиков оценить риск, внести заблаговременные изменения в микропроцесс и сосредоточиться на коллективном анализе и проектировании. Макропроцесс - это деятельность всего коллектива в масштабе от недель до месяцев.  Многие элементы макропроцесса относятся к самой практике менеджмента программных проектов и поэтому выполняются одинаково, как для объектно-ориентированных, так и для других систем. Среди них - управление конфигурацией, гарантии качества, разбор программы и составление документации. В следующей главе мы рассмотрим эти практические вопросы в контексте объектно-ориентированного проектирования. Данная глава сосредоточена на описании специфики объектно-ориентированного подхода или (по определению Парнаса) на том, как мы уродуем рациональный процесс проектирования чтобы получить объектно-ориентированную систему.  Макропроцесс заботит в первую очередь технического руководителя команды разработчиков, цели которого несколько отличаются от задач отдельного разработчика. Они оба заинтересованы в качестве конечного программного продукта, удовлетворяющем требованиям заказчика [Ну, конечно, не все, а большинство. К сожалению, некоторые менеджеры больше заинтересованы в развитии своей империи, чем в развитии программного продукта. Прибавьте к этому предыдущее примечание относительно аналитиков и проектировщиков. Я думаю, Данте мог бы найти для них подходящее место]. Однако, конечного пользователя мало волнует, правильно ли использованы в проекте параметризованные классы или полиморфизм; заказчик гораздо более обеспокоен сроками, качеством, полнотой и правильностью работы программы. Поэтому макропроцесс сконцентрирован на управлении риском и выявлении общей архитектуры - двух управляемых компонентах, имеющих решающее значение для сроков, полноты и качества проекта.  В макропроцессе в большой степени сохранены традиционные фазы анализа и проектирования и процесс в меру упорядочен. Как показано на рис. 6-2, макропроцесс обычно включает следующие действия:  Выявление сущности требований к программному продукту (концептуализация).  Разработка модели требуемого поведения системы (анализ).  Создание архитектуры для реализации (проектирование).  Итеративное выполнение реализации (эволюция).  Управление эволюцией продукта в ходе эксплуатации (сопровождение).    http://www.helloworld.ru/texts/comp/other/oop/pic06_02.gif  *Рис. 6-2. Макропроцесс.*  У всех нетривиальных программных разработок макропроцесс продолжается и после создания и внедрения системы. Это особенно видно на примере организаций, специализирующихся на создании семейств программ, на которые часто выделяются значительные капиталовложения.  Основная философия макропроцесса состоит в постепенном развитии. Как его определяет Вонк, "при разработке методом последовательного развития, система выстраивается шаг за шагом, причем каждая новая версия содержит функциональность предыдущей, плюс новые функции" [[14](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#6.14)]. Этот подход чрезвычайно хорошо сочетается с объектно-ориентированной парадигмой и дает много возможностей для управления риском. Как утверждает Гилб: "Постепенная передача программ заказчику изобретена для того, чтобы заранее предупредить нас о надвигающихся неприятностях" [[15](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#6.15)].  Теперь детально рассмотрим каждое действие в макропроцессе. Естественно, одним из показателей зрелости организации, ведущей разработку, является знание случаев, когда надо обойти эти правила, что мы будем отдельно отмечать в нашем обзоре.  **Концептуализация**  **Цель.** Концептуализация должна установить основные требования к системе. Для каждой принципиально новой части программы или даже для нового применения существующей системы найдется такой момент, когда в голову разработчика, архитектора, аналитика или конечного пользователя западет идея о новом приложении.  Это может быть новое деловое предприятие, дополнительное изделие на поточной линии или, например, новая функция в существующей программной системе. Цель концептуализации не в том, чтобы полностью определить идею, а в том, чтобы выработать взгляд на нее и мысленно проверить ее.  **Результаты.** Первичными продуктами концептуализации являются прототипы системы. Определенно, каждой существенно новой программной системе необходим некоторый черновой прототип, пусть и выполненный "на скорую руку". Такие прототипы не полны по самой своей природе и разработаны лишь схематически. Однако, нужно сохранять интересные (пусть, возможно, и отвергнутые) прототипы, так как этим организация поддерживает корпоративную память о первоначальном замысле и сохраняет связь с исходными предположениями. При проектировании этот архив дает незаменимый материал для экспериментирования, к которому аналитики и архитекторы могут возвращаться, когда хотят опробовать новые идеи.  Очевидно, для грандиозных приложений (национального или международного значения), само построение прототипов может оказаться большим свершением. Ведь гораздо лучше столкнуться с трудностями при реализации, обнаружив, что неверны какие-то предположения о функциональности, эффективности, размере или сложности системы, чем пренебречь прогрессивным решением. Такое пренебрежение может грозить финансовой или социальной катастрофой.  Подчеркнем: прототипы хороши, но их следует выбросить. Нельзя позволять им непосредственно эволюционировать в готовую систему, если к этому не имеется достаточно серьезных оснований. Сжатые сроки не являются уважительной причиной: оптимизация краткосрочной разработки, игнорирующая последующие затраты владельца программного продукта, - типичный пример ложной экономии.  **Виды деятельности.** Концептуализация по самой своей природе - творческая деятельность, и, следовательно, она не должна быть скована жесткими правилами разработки. Возможно, самое важнее для организации - создать структуру, которая обеспечивала бы достаточные ресурсы для возникновения и исследования новых идей [Если организация не сделает этого сама, то отдельные разработчики все равно сделают это, не спрашиваясь у компании, в которой они работают. Так и возникают новые программистские фирмы. Их появление хорошо для индустрии в целом, но не для самой осиротевшей компании]. Новые идеи могут исходить из самых различных источников: конечных пользователей, групп пользователей, разработчиков, аналитиков, проектировщиков, распространителей и т.д. Для руководства важно вести регистрацию этих идей, располагая их по приоритетам и распределяя ограниченные ресурсы так, чтобы исследовать самые многообещающие из них. Когда для исследования выбрано конкретное направление, типичен следующий порядок дальнейших действий:  Решить, какие цели преследуются при опробовании концепции и каковы критерии того, что считать благополучным исходом.  Собрать подходящую команду для разработки прототипа. Часто она состоит из единственного члена (который и есть тот самый мечтатель). Самое лучшее, что организатор может сделать, чтобы облегчить усилия команды - не стоять на ее пути.  Оценить готовый прототип и принять ясное решение о проектировании конечного продукта или о дальнейшем исследовании. Решение приступить к разработке конечного продукта нужно принимать с разумным учетом потенциального риска, выявленного при опробовании концепции.  Концептуализация не содержит ничего специфически объектно-ориентированного. Каждая программная парадигма должна предусматривать опробование концепций. Однако, как часто бывает, разработка прототипов обычно происходит быстрее в тех случаях, когда на лицо зрелая объектно-ориентированная среда.  Довольно часто концепции опробуются на одном языке (например, на Smalltalk), а разработка конечного продукта ведется на другом (скажем, C++).  **Путевые вехи и характеристики.** Важно, чтобы для оценки прототипа были установлены четкие критерии. Работу над прототипом чаще планируют по срокам (имея в виду, что прототип должен быть завершен к определенной дате), чем по требованиям. Это не всегда плохо, так как искусственно ограничивает усилия по созданию прототипа и пресекает попытки выпустить концептуально недоношенный продукт.  Менеджеры верхнего звена могут оценить здоровье организации по ее отношению к новым идеям. Любая организация, которая сама не генерирует новые идеи, либо уже мертва, либо близка к этому. Наиболее благоразумное действие в такой ситуации - выделить независимые подразделения либо вообще уйти из бизнеса. С другой стороны, любая организация, заваленная новыми идеями, но неспособная определить их разумный приоритет, неуправляема. Такие компании часто тратят впустую существенные ресурсы, перескакивая к разработке изделия слишком рано, без исследования риска. Наиболее благоразумно здесь было бы формализовать процесс производства и наладить переход от концепции к продукту.  **Анализ**  **Цель.** Как утверждает Меллор, "цель анализа - дать описание задачи. Описание должно быть полным, непротиворечивым, пригодным для чтения и обозрения всеми заинтересованными сторонами, реально проверяемым" [[16](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#6.16)]. Говоря нашим языком, цель анализа - представить модель поведения системы.  Надо подчеркнуть, что анализ сосредоточен не на форме, а на поведении. На этой фазе неуместно заниматься проектированием классов, представлением или другими тактическими решениями. Анализ должен объяснить, что делает система, а не то, как она это делает. Любое, сделанное на стадии анализа (вопреки этому правилу) утверждение о том "как", может считаться полезным только для демонстрации поведения системы, а не как проверяемое требование к ее проектированию.  В этом отношении цели анализа и проектирования весьма различны. В анализе мы ищем модель мира, выявляя классы и объекты (их роли, обязанности и взаимодействия), которые формируют словарь предметной области. В проектировании мы изобретаем искусственные персонажи, которые реализуют поведение, требуемое анализом. В этом смысле, анализ - это деятельность, которая сводит вместе пользователей и разработчиков системы, объединяя их написанием общего словаря предметной области.  Сосредоточившись на поведении, мы приступаем к выяснению функциональных точек системы. Функциональные точки, впервые описанные Аланом Альбрехтом, обозначают видимые извне и поддающиеся проверке элементы поведения системы [[17](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#6.17)]. С точки зрения конечного пользователя, функциональная точка представляет некоторое простейшее действие системы в ответ на некоторое событие [Как отмечает Дрегер, в теории управления информационными системами функциональная точка представляет отдельную бизнес-функцию конечного пользователя [[18](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#6.18)]]. Функциональные точки часто (но не всегда) обозначают отображение входов на выходы и таким образом представляют преобразования, совершаемые системой. С точки зрения аналитика, функциональные точки представляют кванты поведения. Действительно, функциональные точки - мера сложности системы: чем их больше, тем она сложнее. На стадии анализа мы передаем семантику функциональных точек сценариями.  Анализ никогда не происходит независимо. Мы не стремимся к исчерпывающему пониманию поведения системы и даже утверждаем, что сделать полный анализ до начала проектирования не только невозможно, но и нежелательно. Процесс построения системы поднимает вопросы о ее поведении, на которые реально нельзя дать гарантированный ответ, занимаясь только анализом. Достаточно выполнить анализ всех первичных элементов поведения системы и некоторого количества вторичных, добавляемых для гарантии того, что никакие существенные шаблоны поведения не пропущены.  Достаточно полный и формальный анализ необходим в первую очередь для того, чтобы ход проекта можно было проследить. Возможность проследить проект нужна для обеспечения возможности его просчитать, дабы гарантировать, что не пропущено ни одной функциональной точки. Возможность проследить проект является также основой управления риском. При разработке любой нетривиальной системы, менеджеры столкнутся с необходимостью сделать нелегкий выбор либо в распределении ресурсов, либо в решении некоторой тактической проблемы. Имея возможность проследить процесс от функциональных точек до реализации, гораздо легче оценить влияние подобных проблем на архитектуру.  **Результаты.** ДеШампо считает, что результатом анализа должно быть описание назначения системы, сопровождаемое характеристиками производительности и перечислением требуемых ресурсов [[19](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#6.19)]. В объектно-ориентированном проектировании мы получаем такие описания с помощью сценариев. Каждый сценарий представляет одну функциональную точку. Мы используем первичные сценарии для иллюстрации ключевого поведения и вторичные для описания поведения в исключительных ситуациях.  Как говорилось в предыдущих главах, мы используем технику CRC-карточек для раскадровки сценариев, а потом применяем диаграммы объектов для более точной иллюстрации семантики каждого сценария. Такие диаграммы должны демонстрировать взаимодействие объектов, обеспечивающее выполнение функций системы, и упорядоченный процесс этого взаимодействия, состоящий в посылке объектами сообщений друг другу. Кроме диаграмм объектов, в рассмотрение можно включить диаграммы классов (чтобы показать существующие ассоциации между классами объектов) и состояний (чтобы показать жизненный цикл важнейших объектов).  Часто эти результаты анализа объединяют в один формальный документ, который формулирует требования анализа к поведению системы, иллюстрируя их диаграммами, и показывает такие неповеденческие аспекты системы, как эффективность, надежность, защищенность и переносимость [[20](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#6.20)].  Побочным результатом анализа будет оценка риска: выявление опасных мест, которые могут повлиять на процесс проектирования. Обнаружение имеющегося риска в начале процесса проектирования облегчит возможные архитектурные компромиссы на поздних этапах разработки.  **Виды деятельности.** С анализом связаны два основных вида деятельности: анализ предметной области и планирование сценариев.  Как мы описали в главе 4, анализ области должен идентифицировать обитающие в данной проблемной области классы и объекты. Прежде, чем взяться за разработку новой системы, обычно изучают уже существующие. В этом случае мы можем извлечь выгоду из опыта других проектов, в которых принимались сходные решения. Лучшим результатом анализа предметной области может явиться вывод, что нам не надо проектировать новый продукт, а следует повторно использовать или адаптировать существующую программу.  Планирование сценариев является центральным действием анализа. Интересно, что по этому вопросу, кажется, имеется совпадение мнений среди других методологов, особенно у Рубина и Голдберга (Rubin adn Goldberg), Адамса (Adams), Вирфс-Брока (Wirfs-Brock), Коада (Coad) и Джекобсона (Jacobson). Типичный порядок его выполнения следующий:  Идентифицировать основные функциональные точки системы и, если возможно, сгруппировать функционально связанные виды поведения. Рассмотреть возможность создания иерархии функций, в которой высшие функции вытекают из низших.  Для каждого представляющего интерес набора функциональных точек сделать раскадровку сценария, используя технику анализа поведения и примеров использования, описанную в главе 4 [Всесторонний анализ этого предмета можно найти в работах Джекобсона [[22](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#6.22)] и Рубина и Голдберга [[23](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#6.23)]]. В мозговом штурме каждого сценария эффективна техника CRC-карточек. Когда прояснится семантика сценариев, следует документировать их, используя диаграммы объектов, которые иллюстрируют объекты, инициирующие и обеспечивающие поведение, и их взаимодействие при выполнении действий сценария. Приложить описание событий, происходящих при выполнении сценария, и порядок выполняемых в результате действий. Кроме того, необходимо перечислить все предположения, ограничения и показатели эффективности для каждого сценария [[21](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#6.21)].  Если необходимо, сделать вторичные сценарии, иллюстрирующие поведение системы в исключительных ситуациях.  Для объектов с особо важным жизненным циклом описать диаграммы состояний (построить конечный автомат).  Найти в сценариях повторяющиеся шаблоны и выразить их в терминах более абстрактных обобщенных сценариев или в терминах диаграмм классов, показывающих связи между ключевыми абстракциями.  Внести изменения в словарь данных; включить в него новые классы и объекты, выявленные для каждого сценария, вместе с описанием их ролей и обязанностей.  Как описано в следующей главе, планирование сценариев выполняется аналитиками в сотрудничестве с экспертами в предметной области и архитекторами. В планировании сценария дополнительно должен участвовать контролер качества, так как сценарии представляют тестируемое поведение. Привлечение контролеров в самом начале процесса помогает сразу установить высокие стандарты качества. Эффективно также привлекать и других членов коллектива, чтобы дать им возможность включиться в процесс проектирования и ускорить понимание строения системы.  **Путевые вехи и характеристики.** Мы благополучно завершим эту фазу, когда мы будем иметь уточненные и подписанные сценарии для всех фундаментальных типов поведения системы. Говоря подписанные, мы предполагаем, что конечные результаты анализа проверялись экспертами, конечными пользователями, аналитиками и архитекторами; говоря фундаментальные, мы имеем в виду типы поведения, основные для данного приложения. Повторим, мы не ожидаем полного анализа, - достаточно рассмотреть только основные и несколько второстепенных видов поведения.  Степень совершенства анализа будет измеряться, в частности, его полнотой и простотой. Хороший анализ выявляет все первичные сценарии и, как правило, важнейшие вторичные. Разумный анализ включает также просмотр всех стратегически важных сценариев, так как это помогает привить единое видение системы всему коллективу разработчиков. Наконец, следует найти шаблоны поведения, которые давали бы возможно более простую структуру классов и учитывали бы все, что есть общего в различных сценариях.  Другой важной составной частью анализа является оценка риска, которая облегчит будущие стратегические и тактические компромиссы.  **Проектирование**  **Цель.** Цель проектирования - создать архитектуру развивающейся реализации и выработать единые тактические приемы, которыми должны пользоваться различные элементы системы. Мы начинаем процесс проектирования сразу после появления некоторой приемлемой модели поведения системы. Важно не начинать проектирование до завершения анализа. Равным образом важно избегать затягивания проектирования, пытаясь получить идеальную, а следовательно, недостижимую аналитическую модель [Такая ситуация обычно классифицируется как паралич анализа].  **Результаты.** Имеется два основных результата проектирования: описание архитектуры и выработка общих тактических приемов.  Мы можем описывать архитектуру путем построения диаграмм или создавая последовательные архитектурные релизы системы. Как описано в предыдущих главах, архитектура объектно-ориентированной системы выражает структуру классов и объектов в ней, поэтому можно использовать диаграммы классов и объектов, чтобы показать ее стратегическую организацию. Для описания архитектуры важно наглядно продемонстрировать группирование классов в категории классов (для логической архитектуры) и группирование модулей в подсистемы (для физической архитектуры). Можно распространять такие диаграммы, как часть формального документа, описывающего архитектуру, который должен быть доступен всем членам коллектива для ознакомления и внесения поправок при развитии архитектуры.  Мы используем архитектурные релизы системы как осязаемую демонстрацию строения архитектуры. Архитектурный релиз представляет собой как бы вертикальный разрез архитектуры, передающий важнейшую (но не полную) семантику существенных категорий и подсистем. Архитектурный релиз системы должен быть работающей программой, что позволяет измерять, изучать и оценивать архитектуру. Как мы увидим в следующем разделе, архитектурные релизы являются основой эволюции системы.  Общие тактические приемы - это локализованные механизмы, которые проявляются всюду в системе. К ним относятся такие аспекты проектирования, как принципы обнаружения и обработки ошибок, управление памятью, хранение и представление данных, подходы к управлению. Важно в явном виде описать эти приемы, чтобы не заставлять разработчиков отыскивать частные решения к общим задачам и не развалить нашу стратегическую архитектуру.  Мы описываем единые приемы в сценариях и действующих релизах каждого механизма.  **Виды деятельности.** С проектированием связано три действия: архитектурное планирование, тактическое проектирование и планирование релизов.  При архитектурном планировании мы занимаемся вертикальным и горизонтальным расчленением системы. Оно охватывает логическую декомпозицию, состоящую в группировании классов, и физическую декомпозицию, состоящую в разбиении на модули и назначении заданий процессорам. Типичный порядок действий таков:  Рассмотреть группирование функциональных точек (найденных в анализе) и распределить их по слоям и разделам архитектуры. Функции базирующиеся одна на другой должны попасть в разные слои; функции, сотрудничающие между собой для обеспечения требуемого поведения системы на данном уровне абстракции должны попасть в разделы системы, представляющие услуги на этом уровне.  Проверить архитектуру созданием действующих релизов, которые частично удовлетворяют семантике нескольких важнейших сценариев, предоставленных анализом.  Оценить достоинства и недостатки архитектуры. Определить риск изменения каждого ключевого архитектурного интерфейса, чтобы можно было заранее распределить ресурсы при эволюции системы.  Архитектурное планирование сконцентрировано на том, чтобы создать в самом начале жизненного цикла каркас системы, а потом постепенно развивать его.  Тактическое проектирование состоит в принятии решений о множестве общих приемов. Как описано ранее в этой главе, плохое тактическое проектирование может разрушить даже очень продуманную архитектуру. Мы можем уменьшить этот риск, явно выделив тактические приемы и решив твердо их придерживаться. Типичный порядок действий таков:  Перечислить все случаи, когда нужно следовать единым общим приемам. Некоторые из них окажутся фундаментальными, независимыми от предметной области, например, управление памятью, обработка ошибок и т.д. Другие будут специфичны для данной области и будут содержать свойственные этой области идиомы и механизмы, такие, как принципы управления системами реального времени или транзакциями и базами данных в информационных системах.  Для каждого приема составить сценарий, описывающий его семантику. Затем выразить ее в виде исполнимого прототипа, который может быть уточнен и представлен инструментально.  Документировать каждый принцип и распространить полученные документы, чтобы обеспечить единое архитектурное видение.  Программные релизы закладывают основы архитектурной эволюции системы. По полученным на стадии анализа функциональным точкам и оценкам риска, релизы выпускаются со все более широкими функциональными возможностями и, в конечном счете, достигают требований, предъявляемых к конечной системе. Типичный порядок действий таков:  Полученные в результате анализа сценарии упорядочить от основных к второстепенным. Приоритетность сценариев лучше выяснить вместе с экспертом в предметной области, аналитиком, архитектором и контролером качества.  Распределить функциональные точки по релизам так, чтобы последний релиз в серии представлял результирующую систему.  Определить цели и расписание релизов так, чтобы дать время на разработку и синхронизировать релизы с другими действиями, например, с разработкой документации и полевыми испытаниями.  Начать планирование задач, учитывая критические места проекта и ресурсы, отведенные на выпуск каждого релиза.  Естественным побочным результатом планирования релизов является план, в котором определены расписание работ, задачи коллектива и оценка риска.  **Путевые вехи и характеристики.** Мы благополучно закончим эту фазу, когда получим проверенную и утвержденную архитектуру, прошедшую прототипирование и формализованные обзоры. Кроме этого, должны быть утверждены все важные тактические приемы и план последовательных релизов.  Основным признаком совершенства является простота. Хорошая архитектура имеет характеристики организованной сложной системы (см. главу 1).  Главные выгоды от этой деятельности - раннее выявление архитектурных просчетов и утверждение единых приемов, которые позволяют получить более простую архитектуру.  **Эволюция**  **Цель.** Цель эволюции - наращивать и изменять реализацию, последовательно совершенствуя ее, чтобы в конечном счете создать готовую систему.  Эволюция архитектуры в значительной степени состоит в попытке удовлетворить нескольким взаимоисключающим требованиям ко времени, памяти и т.д. - одно всегда ограничивает другое. Например, если критичен вес компьютера (как при проектировании космических систем), то должен быть учтен вес отдельного чипа памяти. В свою очередь количество памяти, допустимое по весу, ограничивает размер программы, которая может быть загружена. Ослабьте любое ограничение, и станет возможным альтернативное решение; усильте ограничение, и некоторые решения отпадут. Эволюция при реализации программного проекта лучше чем монолитный набор приемов помогает определить, какие ограничения существенны, а какими можно пренебречь. По этой причине эволюционная разработка сосредоточена прежде всего на функциональности и только затем - на локальной эффективности. Обычно в начале проектирования мы слишком мало знаем, чтобы предвидеть слабое место в эффективности системы. Анализируя поведение каждого нового релиза, используя гистограммы и тому подобную технику, команда разработчиков через какое-то время сможет лучше понять, как настроить систему.  Таким образом, эволюция - это и есть процесс разработки программы. Как пишет Андерт, проектирование "есть время новшеств, усовершенствований, и неограниченной свободы изменять программный код, чтобы достигнуть целей. Производство - управляемый методичный процесс подъема качества изделия к надлежащему уровню" [[24](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#6.24)].  Пейдж-Джонс называет ряд преимуществ такой поступательной разработки:  "Обеспечивается обратная связь с пользователями, когда это больше всего необходимо, полезно и значимо.  Пользователи получают несколько черновых версий системы для сглаживания перехода от старой системы к новой.  Менее вероятно, что проект будет снят с финансирования, если он вдруг выбился из графика.  Главные интерфейсы системы тестируются в первую очередь и наиболее часто.  Более равномерно распределяются ресурсы на тестирование.  Реализаторы могут быстрее увидеть первые результаты работы системы, что их морально поддерживает.  Если сроки исполнения сжатые, то можно приступить к написанию и отладке программ до завершения проектирования".  **Результаты.** Основным результатом эволюции является серия исполнимых релизов, представляющих итеративные усовершенствования изначальной архитектурной модели. Вторичным продуктом следует признать выявление поведения, которое используется для исследования альтернативных подходов и дальнейшего анализа темных углов системы.  Действующие релизы выпускаются по графику, намеченному в начале планирования. Для скромного по размерам проекта, требующего 12-18 месяцев на разработку от начала до конца, это могло бы означать: по релизу каждые два или три месяца. Для более сложных проектов, требующих больше усилий разработчиков, можно выпускать релиз каждые шесть месяцев и реже. Более редкий график подозрителен, так как он не вынуждает разработчиков должным образом завершать микропроцессы и может скрыть опасные области.  Для кого делается действующий релиз программы? В начале процесса разработки основные действующие релизы передаются разработчиками контролерам качества, которые тестируют их по сценариям, составленным при анализе, и накапливают информацию о полноте, корректности и устойчивости работы релиза. Это раннее накопление данных помогает при выявлении проблем качества, которые будут учтены в следующих релизах. Позднее действующие релизы передаются конечным (альфа и бета) пользователям некоторым управляемым способом. "Управляемым" означает, что разработчики тщательно выверяют требования к каждому релизу и определяют аспекты, которые желательно проверить и оценить.  Специфика микропроцесса предполагает, что при многочисленных внутренних релизах разработчики выпускают наружу лишь некоторые исполнимые версии. Внутренние релизы представляют своего рода процесс непрерывной интеграции системы и завершают каждый цикл микропроцесса.  Косвенно подразумевается, что документация системы эволюционирует вместе с архитектурными релизами. Чтобы не относиться к ведению документации как к основному занятию, лучше всего получать ее, как естественный, полуавтоматически генерируемый побочный продукт эволюционного процесса.  **Виды деятельности.** Эволюция связана с двумя видами деятельности: микропроцесс и управление изменениями.  Работа, выполняемая между релизами, представляет процесс разработки в сжатом виде: это как раз и есть один цикл микропроцесса. Мы начинаем с анализа требований к следующему релизу, переходим к проектированию архитектуры и исследуем классы и объекты, необходимые для реализации этого проекта. Типичный порядок действий таков:  Определить функциональные точки, которые попадут в новый релиз, и области наивысшего риска, особенно те, которые были выявлены еще при эволюции предыдущего релиза.  Распределить задачи по релизам среди членов команды и начать новый микропроцесс. Контролировать микропроцесс, просматривая проект, и проверять состояние дел в важных промежуточных этапах с интервалами от нескольких дней до двух недель.  Когда потребуется понять семантику требуемого поведения системы, поручить разработчикам сделать прототип поведения. Четко установить назначение каждого прототипа и определить критерии готовности. После завершения решить, как включить результаты прототипирования в этот или последующие релизы.  Завершить микропроцесс интеграцией и очередным действующим релизом.  После каждого релиза следует перепроверить сроки и требования в основном плане релизов. Как правило, это незначительные корректировки дат или перенос функциональности из одного релиза в другой.  Управление изменениями необходимо именно в связи со стратегией итеративного развития. Всегда соблазнительно вносить неупорядоченные изменения в иерархию классов, их протоколы или механизмы, но это подтачивает стратегическую архитектуру и приводит к тому, что разработчики сами начинают путаться в собственном коде.  При эволюции системы на практике ожидаются следующие типы изменений:  Добавление нового класса или нового взаимодействия между классами.  Изменение реализации класса.  Изменение представления класса.  Реорганизация структуры классов.  Изменение интерфейса класса.  Каждый тип изменений имеет свою причину и стоимость.  Проектировщик вводит новые классы, если обнаружились новые абстракции или понадобились новые механизмы. Цена выполнения таких изменений обычно несущественна для управления разработкой. Если добавляется новый класс, нужно рассмотреть, куда он попадет в существующей структуре классов. Когда вводится новое взаимодействие классов, должен быть произведен минимальный анализ предметной области, чтобы убедиться, что оно действительно удовлетворяет одному из шаблонов взаимодействия.  Изменение реализации также обходится недорого. Обычно при объектно-ориентированной разработке сначала создается интерфейс класса, а потом пишется его реализация (то есть код функций-членов). Если только интерфейс в приемлемой степени стабилен, можно выбрать любое внутреннее представление этого класса и выполнить реализацию его методов. Реализация отдельного метода может быть изменена (обычно для исправления ошибки или повышения эффективности) позже. Можно скорректировать реализацию метода, чтобы воспользоваться преимуществами новых методов, определенных в существующем или во вновь введенном суперклассе. В любом случае изменение реализации метода обходится сравнительно недорого, особенно, если она была своевременно инкапсулирована.  Подобным образом можно было бы изменить представление класса (в C++ - защищенные и закрытые члены класса). Обычно это делается, чтобы получить более эффективные (с точки зрения памяти или скорости) экземпляры класса. Если представление класса инкапсулировано, что возможно в Smalltalk, C++, CLOS и Ada, то изменение в представлении не будет разрушать логику взаимодействия объектов-пользователей с экземплярами класса (если, конечно, новое представление обеспечивает ожидаемое поведение класса). С другой стороны, если представление класса не инкапсулировано, что также возможно в любом языке, то изменение в представлении класса чрезвычайно опасно, так как клиенты могут от него зависеть. Это особенно верно в случае подклассов: изменение представления суперкласса вызовет изменения представления всех его подклассов. Во всяком случае, изменение представления класса имеет цену: нужно произвести перекомпиляцию интерфейса и реализации класса, сделать то же для всех его клиентов, для клиентов тех клиентов и т.д.  Реорганизация структуры классов системы встречается довольно часто, хотя и реже, чем другие упомянутые виды изменений. Как отмечают Стефик и Бобров, "Программисты часто создают новые классы и реорганизуют имеющиеся, когда они видят удобную возможность разбить свои программы на части" [26]. Изменение структуры классов обычно происходит в форме изменения наследственных связей, добавления новых абстрактных классов и перемещения обязанностей и реализации общих методов в классы более верхнего уровня в иерархии классов. На практике структура классов системы особенно часто реорганизуется вначале, а потом, когда разработчики лучше поймут взаимодействие ключевых абстракций, стабилизируется. Реорганизация структуры классов поощряется на ранних стадиях проектирования, потому что в результате может получиться более лаконичная программа. Однако реорганизация структуры классов не обходится даром. Обычно изменение положения верхнего класса в иерархии делает устаревшими определения всех классов под ним и требует их перекомпиляции (а, значит, и перекомпиляции всех зависимых от них классов и т.д.).  Еще один важный вид изменений, к которому приходится прибегать при эволюции системы, - изменение интерфейса класса. Разработчик обычно изменяет интерфейс класса, чтобы добавить некоторый новый аспект, удовлетворить семантике некоторой новой роли объектов класса или добавить новую операцию, которая всегда была частью абстракции, но раньше не была экспортирована, а теперь понадобилась некоторому объекту-пользователю. На практике использование эвристик для построения классов, которые мы обсуждали в главе 3 (особенно требование примитивного, достаточного и полного интерфейса), сокращает вероятность таких изменений. Однако наш опыт никогда не бывает окончательным. Мы никогда не определим нетривиальный класс так, чтобы интерфейс его сразу оказался правильным.  Редко, но встречается удаление существующего метода; это обычно делается только для того, чтобы улучшить инкапсуляцию абстракции. Чаще мы добавляем новый метод или переопределяем метод, уже объявленный в некотором суперклассе. Во всех трех случаях это изменение дорого стоит, потому что оно логически затрагивает всех клиентов, требуя как минимум их перекомпиляции. К счастью, эти последние виды изменений, добавление и переопределение методов, совместимы снизу вверх. На самом деле вы обнаружите, что большинство изменений интерфейса, произведенного над определенными классами при эволюции системы, совместимы снизу вверх. Это позволяет для уменьшения воздействия этих изменений применить такие изощренные технологии, как инкрементная компиляция. Инкрементная компиляция позволяет нам вместо целых модулей перекомпилировать только отдельные описания и операторы, то есть перекомпиляции большинства клиентов можно избежать.  Почему перекомпиляция так неприятна? Для маленьких систем здесь нет проблем: перекомпиляция всей системы занимает несколько минут. Однако для больших систем это совсем другое дело. Перекомпиляция программы в сотни тысяч строк может занимать до половины суток машинного времени. Представьте себе, что вам понадобилось внести изменение в программное обеспечение компьютерной системы корабля. Как вы сообщите капитану, что он не может выйти в море, потому что вы все еще компилируете? В некоторых случаях цена перекомпиляции бывает так высока, что разработчикам приходится отказаться от внесения некоторых, представляющих разумные усовершенствования, изменений. Перекомпиляция представляет особую проблему для объектно-ориентированных языков, так как наследование вводит дополнительные компиляционные зависимости [[27](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#6.27)]. Для строго типизированных объектно-ориентированных языков программирования цена перекомпиляции может быть даже выше; в этих языках время компиляции принесено в жертву безопасности.  Все изменения, обсуждавшиеся до настоящего времени, сравнительно легкие: самый большой риск несут существенные изменения в архитектуре, которые могут погубить весь проект. Часто такие изменения производят чересчур блестящие инженеры, у которых слишком много хороших идей [[28](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#6.28)].  **Путевые вехи и характеристики.** Мы благополучно завершим фазу реализации, когда релизы перерастут в готовый продукт. Первой мерой качества, следовательно, будет то, в какой степени мы справились с реализацией функциональных точек, распределенных по промежуточным релизам, и насколько точно соблюдается график, составленный при их планировании.  Две других основных меры качества - скорость обнаружения ошибок и показатель изменчивости ключевых архитектурных интерфейсов и тактических принципов.  Грубо говоря, скорость обнаружения ошибок - это мера того, как быстро отыскиваются новые ошибки [[29](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#6.29)]. Вкладывая средства в контроль качества в начале разработки, мы можем получить количественные оценки качества для каждого релиза, которые менеджеры команды смогут использовать для определения областей риска и обновления команды разработчиков. После каждого релиза должен наблюдаться всплеск обнаружения ошибок. Стабильность этого показателя обычно свидетельствует о том, что ошибки не обнаруживаются, а его чрезмерная величина говорит о том, что архитектура еще не стабилизировалась или что новые элементы неверно спроектированы или реализованы. Эти характеристики используются при уточнении цели очередного релиза.  Показатель изменчивости архитектурного интерфейса или тактических принципов является основной характеристикой стабильности архитектуры [[30](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#6.30)]. Локальные изменения вероятны в течение всего процесса эволюции, но если структуры наследования или границы между категориями классов или подсистем постоянно перестраиваются, то это признак нерешенных проблем в архитектуре, что должно быть учтено как область риска при планировании следующего релиза.  **Сопровождение**  **Цель.** Сопровождение - это деятельность по управлению эволюцией продукта в ходе его эксплуатации. Она в значительной степени продолжает предыдущие фазы, за исключением того, что вносит меньше архитектурных новшеств. Вместо этого делаются более локализованные изменения, возникающие по мере учета новых требований и исправления старых ошибок.  Леман и Белади сделали несколько неоспоримых наблюдений, рассматривая процесс "созревания" уже внедренной программной системы:  "Эксплуатируемая программа должна непрерывно изменяться; в противном случае она будет становиться все менее и менее полезной (закон непрерывного изменения).  Когда эволюционирующая программа изменяется, ее структура становится более сложной, если не прилагаются активные усилия, чтобы этого избежать (закон возрастающей сложности)" [[31](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#6.31)].  Мы отличаем понятие сохранения системы программного обеспечения от ее сопровождения. При сопровождении разработчики вносят непрерывные усовершенствования в существующую систему; сопровождением обычно занимается другая группа людей, отличная от группы разработчиков. Сохранение же основано на привлечении дополнительных ресурсов для поддержания устаревшей системы (которая часто имеет плохо разработанную архитектуру и, следовательно, трудна для понимания и модификации). Итак, нужно принять деловое решение: если цена владения программным продуктом выше, чем цена разработки новой системы, то наиболее гуманный образ действий - оставить старую систему в покое или покончить с ней.  **Результаты.** Поскольку сопровождение является в определенном смысле продолжением эволюции системы, ее результаты похожи на то, чего мы добивались на предыдущих этапах. В дополнение к ним, сопровождение связано также с управлением списком новых заданий. Кроме тех требований, которые по каким-либо причинам не были учтены, вероятно, уже вскоре после выпуска работающей системы, разработчики и конечные пользователи обменяются множеством пожеланий и предложений, которые они хотели бы увидеть воплощенными в следующих версиях системы. Заметим, что когда с системой поработает больше пользователей, выявятся новые ошибки и неожиданные методы использования, которых не смогли предвидеть контролеры качества [Пользователи проявляют чудеса изобретательности в использовании системы самым необычным образом]. В список заносятся обнаруженные дефекты и новые требования, которые будут учтены при планировании новых релизов в соответствии с их приоритетом.  **Виды деятельности.** Сопровождение несколько отличается от эволюции системы. Если первоначальная архитектура удалась, добавление новых функций и изменение существующего поведения происходят естественным образом.  Кроме обычных действий по эволюции, при сопровождении нужно определить приоритеты задач, собранных в список замечаний и предложений. Типичный порядок действий таков:  Упорядочить по приоритетам предложения о крупных изменениях и сообщения об ошибках, связанных с системными проблемами, и оценить стоимость переработки.  Составить список этих изменений и принять их за функциональные точки в дальнейшей эволюции.  Если позволяют ресурсы, запланировать в следующем релизе менее интенсивные, более локализованные улучшения.  Приступить к разработке следующего эволюционного релиза программы.  **Путевые вехи и характеристики.** Путевыми вехами сопровождения являются продолжающееся производство эволюционирующих релизов и устранение ошибок.  Мы считаем, что занимаемся именно сопровождением системы, если архитектура выдерживает изменения; мы определим, что вошли в стадию сохранения, когда количество ресурсов, требуемых для достижения нужного улучшения, начнет резко нарастать.  Выводы  Удачные проекты обычно характеризуются ясным представлением об архитектуре и хорошо управляемым итеративным жизненным циклом.  Идеально рациональный процесс проектирования невозможен, но его можно имитировать, сочетая микро- и макропроцесс разработки.  Микропроцесс объектно-ориентированной разработки приводится в движение потоком сценариев и продуктов архитектурного анализа (макропроцесс); микропроцесс представляет ежедневную деятельность команды разработчиков.  Первый шаг в микропроцессе связан с идентификацией классов и объектов на данном уровне абстракции; основными видами деятельности являются открытие и изобретение.  Второй шаг микропроцесса состоит в выявлении семантики классов и объектов; основными видами деятельности здесь являются раскадровка сценариев, проектирование изолированных классов и поиск шаблонов.  Третий шаг микропроцесса - выявление связей между классами и объектами; основными действиями являются спецификация ассоциаций, выявление взаимодействий и уточнение ассоциаций.  Четвертый шаг микропроцесса связан с реализацией классов и объектов; основное действие - выбор структур данных и алгоритмов.  Макропроцесс объектно-ориентированной разработки управляет микропроцессом, определяет измеримые характеристики проекта и помогает контролировать риск.  Первый шаг макропроцесса - концептуализация, которая устанавливает основные требования к системе; она служит для опробования концепций и, по большей части, не должна контролироваться, чтобы предоставить неограниченную свободу фантазии.  Второй шаг макропроцесса - анализ. Его цель - получить модель поведения системы. Основными действиями на этом этапе являются анализ предметной области и планирование сценариев.  Третий шаг макропроцесса - проектирование. На этом шаге создается архитектура реализации и вырабатываются единые тактические приемы; основными действиями являются архитектурное планирование, тактическое проектирование и планирование релизов.  Четвертый шаг макропроцесса - эволюция, последовательно приближающая систему к желаемому результату. Основные действия - применение микропроцесса и управление изменениями.  Пятый шаг макропроцесса - сопровождение, то есть управление эволюцией системы в ходе ее эксплуатации; основные действия похожи на действия предыдущего шага, но к ним добавляется работа со списком улучшений и исправлений. |

Практические вопросы

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Разработка программ пока остается чрезвычайно трудоемким делом, в значительной степени она по-прежнему больше напоминает строительство коттеджей, чем промышленное возведение зданий [[1](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#7.1)]. Доклад Кишиды и др. свидетельствует, что даже в Японии на начальной стадии проектов "все еще по большей части полагаются на неформальный подход - карандаш и бумагу" [[2](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#7.2)].  Ситуация усугубляется тем обстоятельством, что проектирование - никак не точная наука. Возьмем проектирование баз данных, одну из технологий, предшествовавших объектно-ориентированному проектированию. Как замечает Хаврис-кевич: "Хотя все выглядит просто и ясно, неизбежно примешивается изрядная доля личного представления о важности различных объектов на предприятии. В результате процесс проектирования не воспроизводим: разные проектировщики могут создать разные модели одного и того же предприятия" [[3](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#7.3)].  Из этого можно сделать вывод, что при любом самом изощренном и теоретически обоснованном методе проектирования нельзя игнорировать практические соображения. Значит, мы должны принять во внимание управленческий опыт в таких областях, как подбор кадров, управление релизами и контроль качества. Для технолога это в высшей степени скучная материя, но для разработчика это реалии жизни, с которыми надо справляться, чтобы создавать сложные программные системы. Итак, в этой главе мы займемся практическими вопросами объектно-ориентированной разработки и влиянием объектной модели на управление.  7.1. Управление и планирование  Если мы при проектировании опираемся на метод итеративного развития, то важнее всего иметь сильное руководство, способное управлять ходом проекта и направлять его. Слишком много проектов сбились с пути из-за неспособности сосредоточиться на главном, и только сильная команда менеджеров может что-то с этим поделать.  **Управление риском**  В конечном счете, главная обязанность менеджера программного продукта - управление как техническим, так и нетехническим риском. Технический риск для объектно-ориентированной системы содержится в решении таких проблем, как выбор структуры наследования классов, обеспечивающий наилучший компромисс между удобством и гибкостью программного продукта. Серьезное решение приходится также принимать при выборе механизмов упрощения архитектуры и улучшения эффективности. Нетехнический риск содержит в себе такие вопросы, как контроль своевременности поставки программных продуктов от третьих фирм или регулирование отношений заказчика и разработчиков, что необходимо для выяснения реальных требований к системе на стадии анализа.  Как было описано в предыдущей главе, микропроцесс объектно-ориентированной разработки нестабилен по своей природе и требует активного управления, концентрации усилий. К счастью, существует макропроцесс разработки, который выдвигает ряд конкретных требований и характеристик. Менеджер проекта, изучая соответствие требований и фактических результатов, может оценить состояние разработки и, при необходимости, перенаправить ресурсы команды. Эволюционная суть макропроцесса разработки означает, что можно распознать проблемы в начале жизненного цикла и продуманно учесть связанный с ними риск прежде, чем проект окажется в опасности.  Многие виды деятельности по управлению разработкой программного обеспечения, например, планирование задач и просмотры, предусмотрены не только в объектно-ориентированной технологии. Однако при управлении объектно-ориентированным проектом намечаемые задачи и рассматриваемые результаты не совсем такие, как в других системах.  **Планирование задач**  Независимо от размера проекта, которым вы заняты, полезно раз в неделю проводить встречу всех разработчиков для обсуждения выполненной работы и действий на следующую неделю. Некоторая минимальная частота встреч необходима, чтобы способствовать общению между членами коллектива. С другой стороны, слишком частые встречи снижают продуктивность и обычно являются признаком потери курса. Объектно-ориентированная разработка требует, чтобы разработчики имели достаточное время для размышлений, введения новшеств и неформального общения с коллегами. Менеджеры команды должны учитывать в плане и это не структурированное время.  Проводимые встречи дают простую, но эффективную возможность гладкой подстройки планов в микропроцессе и распознания показавшихся на горизонте опасных ситуаций. Результатом такой встречи может быть небольшая корректировка в распределении работ, обеспечивающая устойчивость процесса: никакой проект не может позволить хотя бы одному из разработчиков сидеть сложа руки, ожидая, пока другие члены команды приведут в порядок свою часть архитектуры. Это особенно верно для объектно-ориентированных систем, в которых архитектура представляется набором классов и механизмов. Проект может заглохнуть, если разработчикам никак не удается разобраться с одним из ключевых классов.  Планирование задач связано с построением графика представления результатов макропроцесса. В промежутках между очередными релизами менеджеры команды должны оценить трудности, угрожающие проекту [Гилб замечает: "если вы не идете в атаку на трудности, трудности идут в атаку на вас" [[5](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#7.5)]], сконцентрировать ресурсы, чтобы разрешить возникшие проблемы, и далее заниматься новой итерацией микропроцесса, в результате которой нужно получить стабильную систему, удовлетворяющую сценариям, запланированным для нового релиза. Планирование задач на этом уровне очень часто оказывается неудачным из-за чрезмерно оптимистических графиков [[4](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#7.4)]. Разработка, которая рассматривалась как "просто вопрос программирования", растягивается на многие дни работы; графики выбрасываются в корзину, когда разработчик, занимаясь частью системы, предполагает определенные протоколы для других частей системы, а потом получает неполно или неправильно изготовленные классы. Смертельную опасность могут представлять внезапно обнаружившиеся ошибки в компиляторе или то, что программа не укладывается в заданное время исполнения. И то и другое часто приходится преодолевать, жертвуя принятыми ранее тактическими решениями.  Ключ к тому, чтобы не поддаваться чрезмерно оптимистическому планированию, - "калибровка" команды и ее инструментов разработки. Типичное планирование задач протекает следующим образом. Вначале менеджер направляет энергию разработчика на специфические части системы, например на проектирование классов для интерфейса с реляционной базой данных. Разработчик анализирует необходимые усилия и оценивает время исполнения, которое менеджер учитывает при планировании других его действий. Проблема в том, что эти оценки не всегда реальны: они обычно делаются в расчете на самый благоприятный случай. Один разработчик может согласиться на решение задачи за неделю, а другой на эту же задачу попросит месяц. Когда работа будет реально выполнена, может оказаться, что она отняла три недели рабочего времени у обоих разработчиков: первый разработчик недооценил усилия (общая проблема многих программистов), а второй разработчик оценил реальные усилия более точно (например потому, что он понимал разницу между действительным рабочим временем и календарным, которое часто заполнено множеством нефункциональных действий). Таким образом, чтобы разработать графики, к которым коллектив может иметь доверие, менеджерам необходимо ввести своего рода "калибровочные коэффициенты" для пересчета оценок времени, заявленных разработчиками. Это не признак того, что менеджеры не доверяют разработчикам, но просто признание того факта, что большинство программистов сосредоточены на технических проблемах, а не на задачах планирования. Менеджер должен помогать разработчикам учиться планировать, - но это тот навык, который может быть приобретен только опытом.  Объектно-ориентированный процесс разработки помогает выявить явные принципы калибровки. Метод итеративного развития позволяет в начале проекта найти множество промежуточных пунктов, которые менеджеры команды использовали бы для накопления данных о достижениях каждого разработчика, определения графиков работы и планирования встреч. При эволюционной разработке руководители коллектива со временем будут лучше понимать реальную продуктивность каждого своего разработчика, а разработчики смогут научиться более точно оценивать объем предстоящей работы. Те же выводы приложимы и к инструментам: архитектурные релизы уже на ранней стадии проекта стимулируют использование инструментов разработки, которые помогают своевременно проверить структурные ограничения.  **Просмотр**  Просмотр (walkthroughs) - общепринятая практика, которую нужно использовать каждой команде разработчиков. Как и планирование задач, просмотр программного обеспечения был введен независимо от объектно-ориентированной технологии. Однако при просмотре не объектно-ориентированных систем внимание обращается на другое.  Руководитель должен проводить просмотры с разумной частотой. За исключением самых ответственных и уязвимых для ошибок мест, просто неэкономично проверять каждую строчку программы. Следовательно, руководитель должен направить ограниченные ресурсы своей команды на рассмотрение проблем, опасных для стратегии разработки. Для объектно-ориентированных систем это означает большую формальность при проведении просмотров сценариев и архитектуры системы и менее формальную проверку тактических решений.  Как описано в предыдущей главе, сценарии являются первичным результатом объектно-ориентированного анализа. Они должны выражать требуемое поведение системы в терминах ее функциональных точек. Формальные просмотры сценариев проводятся аналитиками команды, вместе с экспертами предметной области или конечными пользователями при возможном участии других разработчиков. Лучше проводить такие просмотры на протяжении всей стадии анализа, чем ожидать выполнения одного глобального просмотра по завершении анализа, когда будет уже слишком поздно сделать что-нибудь полезное, перенаправив усилия аналитиков. Эксперименты показывают, что даже непрограммисты могут понять сценарии, представленные в виде текста или диаграмм объектов [Мы встречались с использованием этой системы обозначении в работе таких непрограммистских групп как астрономы, биологи, метеорологи, физики и банкиры]. В конечном счете просмотр помогает выработать общий словарь для разработчиков и пользователей системы. Привлечение к участию в просмотре других членов команды способствует уяснению ими реальных требований к системе на ранних этапах разработки.  Просмотр архитектуры должен охватывать всю систему, включая ее механизмы и структуру классов. Как и при просмотре сценариев, просмотр архитектуры (архитектором или другими проектировщиками) должен производиться на протяжении всего проекта. Сначала просмотр сосредоточен на общих архитектурных решениях, а позднее, возможно, он акцентируется на некоторых категориях классов или конкретных механизмах. Основная цель просмотра состоит в проверке архитектуры в начале жизненного цикла и выработке общего взгляда на нее. Вторичной целью является поиск повторяющихся шаблонов классов или взаимодействий, которые затем могут быть использованы для упрощения архитектуры.  Неформальный просмотр следует проводить еженедельно. На нем обычно рассматриваются некоторые группы классов или механизмы нижнего уровня. Цель - проверить тактические решения; побочная цель - дать возможность старшим разработчикам научить новичков.  7.2. Кадры  Распределение ресурсов  Один из наиболее замечательных аспектов управления объектно-ориентированными проектами - это тот факт, что в устойчивом состоянии обычно наблюдается сокращение необходимых ресурсов и изменяется график их расходования по сравнению с традиционными методами. Именно "в устойчивом состоянии". Вообще говоря, первый объектно-ориентированный проект, предпринятый организацией, потребует несколько больше ресурсов - главным образом, в соответствии с кривой обучения, описывающей адаптацию ко всякой новой технологии. Выгоды проявятся во втором или третьем проекте, когда разработчики наберутся опыта в проектировании классов, поиске общих абстракций и механизмов, а менеджеры освоятся с методом итеративного развития.  На стадии анализа потребность в ресурсах с переходом на объектно-ориентированные методы обычно мало изменяется. Однако, поскольку объектно-ориентированный процесс уделяет больше внимания архитектуре, мы стремимся привлекать архитекторов и других разработчиков как можно раньше, иногда начиная архитектурные эксперименты еще на последней стадии анализа. Во время эволюции, как правило, потребуется меньше ресурсов, потому что работа облегчится общими абстракциями и механизмами, изобретенными ранее при проектировании архитектуры или выпуске предварительных версий. Тестирование может также потребовать меньше ресурсов, потому что новые функции обычно добавляются к уже корректно ведущей себя структуре класса или механизму. Таким образом, тестирование начинается раньше и является скорее постоянным и постепенным, чем разовым действием. Интеграция обычно требует значительно меньших ресурсов по сравнению с традиционными методами, главным образом потому, что она тоже происходит постепенно, от релиза к релизу, а не одним броском. Таким образом, в устойчивом состоянии трудозатраты оказываются гораздо меньше, чем при традиционных подходах. Более того, если учесть эксплуатационные затраты, то окажется, что весь жизненный цикл объектно-ориентированных программ часто стоит дешевле, так как конечный продукт, скорее всего, будет лучшего качества и окажется более приспособленным к изменениям.  **Роли разработчиков**  Полезно помнить, что разработка программного продукта в конечном счете производится людьми. Разработчики - не взаимозаменяемые части, и успешное создание любой сложной системы требует уникальных и разнообразных навыков всех членов целеустремленного коллектива.  Эксперименты показывают, что объектно-ориентированная разработка требует несколько иного разделения труда по сравнению с традиционными методами. Мы считаем следующие три роли разработчиков важнейшими в объектно-ориентированном подходе:  архитектор проекта;  ответственные за подсистемы;  прикладные программисты.  Архитектор проекта - его творец, человек с сильно развитым воображением; он отвечает за эволюцию и сопровождение архитектуры системы. Для малых или средних систем архитектурное проектирование обычно выполняется одной, максимум двумя светлыми личностями. Для больших проектов эта обязанность может быть распределена в большом коллективе. Архитектор проекта - не обязательно самый главный разработчик, но непременно такой, который может квалифицированно принимать стратегические решения (как правило благодаря обширному опыту в построении систем такого типа). Благодаря опыту, разработчики интуитивно знают, какие общие архитектурные шаблоны уместны в данной предметной области и какие проблемы эффективности встают в определенных архитектурных вариантах. Архитекторы - не обязательно лучшие программисты, хотя они должны уметь программировать. Точно так же, как строительные архитекторы должны разбираться в строительстве, неблагоразумно нанимать архитектора программного обеспечения, который не является приличным программистом. Архитекторы проекта должны также быть сведущи в обозначениях и организации процесса объектно-ориентированной разработки, потому что они должны в конечном счете выразить свое архитектурное видение в терминах кластеров классов и взаимодействующих объектов.  Очень плохая практика - нанимать архитектора со стороны, который, образно выражаясь, въезжает на белом коне, провозглашает архитектурные принципы, а потом уматывает куда-то, в то время как другие пытаются справиться с последствиями его решений. Гораздо лучше привлечь архитектора к активной работе уже при проведении анализа и оставить его на как можно более длительный срок, даже на все время эволюции системы. Тогда он освоится с действительными потребностями системы и со временем испытает на себе последствия своих решений. Кроме того, сохраняя в руках одного человека или небольшой команды разработчиков ответственность за архитектурную целостность, мы повышаем шансы получить гибкую и простую архитектуру.  Ответственные за подсистемы - главные творцы абстракций проекта. Они отвечают за проектирование целых категорий классов или подсистем. Каждый ответственный в сотрудничестве с архитектором проекта разрабатывает, обосновывает и согласует с другими разработчиками интерфейс своей категории классов или подсистемы, а потом возглавляет ее реализацию, тестирование и выпуск релизов в течение всей эволюции системы.  Ответственные за подсистемы должны хорошо знать систему обозначений и организацию процесса объектно-ориентированной разработки. Обычно они программируют лучше чем архитекторы проекта, но не располагают обширным опытом последних. Лидеры подсистем составляют от трети до половины численности команды.  Прикладные программисты (инженеры) - младшие по рангу участники проекта. На них возложено выполнение двух обязанностей. Некоторые из них отвечают за реализацию категории или подсистемы под руководством ее ведущего. Эта деятельность может включать в себя проектирование некоторых классов, но в основном связана с реализацией и последующим тестированием классов и механизмов, разработанных проектировщиками команды. Другие отвечают за написание классов, спроектированных архитектором и ответственными за подсистемы, реализуя тем самым функциональные точки системы. В некотором смысле, эти программисты занимаются написанием маленьких программ на языке предметной области, определенном классами и механизмами архитектуры.  Инженеры разбираются в системе обозначений и в организации процесса разработки, но не слишком блестяще; зато они, как правило, являются очень хорошими программистами, знающими основные идиомы и слабые места выбранных языков программирования. Инженеры составляют половину команды или более того.  Разница в квалификации ставит проблему подбора кадров перед всеми организациями, которые обычно имеют несколько сильных проектировщиков и большее количество менее квалифицированного персонала. Социальная польза нашего подхода к кадровой политике состоит в том, что он открывает путь для карьеры начинающим сотрудникам: молодые разработчики работают под руководством более опытных. Когда они наберутся опыта в использовании хорошо определенных классов, они смогут сами проектировать классы. Вывод: не обязательно каждому разработчику быть экспертом по абстракциям, но каждый разработчик может со временем этому научиться.  В больших проектах могут потребоваться и другие роли. Большинство из них (например, роль специалиста в средствах разработки) явно не связаны с объектно-ориентированной технологией, но некоторые непосредственно вытекают из нее (такие, как инженер, отвечающей за повторное использование):    |  |  | | --- | --- | | ® Менеджер проекта | Отвечает за управление материалами проекта, заданиями, ресурсами и графиком работ. | | ® Аналитик | Отвечает за развитие и интерпретацию требований конечных пользователей; должен быть экспертом в проблемной области, однако его не следует изолировать от остальной команды разработчиков. | | ® Инженер по повторному использованию | Управляет хранилищем (репозитарием) материалов проекта; участвуя в просмотре и других действиях, активно ищет общее и добивается его использования; находит, разрабатывает или приспосабливает компоненты для общего использования в рамках конкретного проекта или целой организации. | | ® Контролер качества | Измеряет результаты процесса разработки; задает общее направление (на системном уровне) тестирования всех прототипов и релизов. | | ® Менеджер интеграции | Отвечает за сборку совместимых друг с другом версий категорий и подсистем в релизы; следит за их конфигурированием. | | ® Ответственный за документацию | Готовит документацию по выпускаемому продукту и его архитектуре для конечного пользователя. | | ® Инструментальщик | Отвечает за создание и адаптацию инструментов программирования, которые облегчают производство программ и (особенно) генерацию кода. | | ® Системный администратор | Управляет физическими компьютерными ресурсами в проекте. |   Конечно, не каждый проект требует всех этих ролей. Для небольших проектов обязанности могут совмещаться. С другой стороны, для очень больших проектов каждой из ролей может заниматься целая организация.  Опыт показывает, что объектно-ориентированная разработка может обойтись меньшим числом занятых в ней людей по сравнению с традиционными методами. На самом деле, чтобы за один год произвести высококачественную программу объемом в несколько сот тысяч строк достаточно 30-40 разработчиков. Однако мы согласны с Боемом, который считает, что "лучшие результаты получаются, когда разработчиков занято меньше, а квалификация их выше" [[6](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#7.6)]. К сожалению, если при разработке проекта пытаться обойтись меньшим количеством людей, чем считается необходимым, можно встретить известное сопротивление. Как отмечалось в предыдущей главе, такое отношение, возможно, вызвано попытками некоторых менеджеров построить империю. Другие менеджеры любят скрываться за множеством служащих, потому что большее количество людей означает больше власти. Кроме того, в случае провала проекта есть на кого свалить вину.  Применение самого изощренного метода проектирования или новейших инструментов не освобождает менеджера от ответственности за подбор проектировщиков, способных мыслить, и не является основанием для того, чтобы пустить проект на самотек [[7](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#7.7)].  7.3. Управление релизами  Интеграция  Промышленные программные проекты требуют создания семейств программ. В процессе разработки создаются прототипы и релизы. Очень часто каждый разработчик имеет свое собственное представление о разрабатываемой системе.  В предыдущей главе объяснялось, что в процессе объектно-ориентированной разработки интеграция редко проводится за один раз. Обычно происходит множество мелких интеграции, каждая из которых соответствует созданию нового прототипа или архитектурного релиза. Каждый новый релиз поступательно развивает предыдущие стабильные релизы. "При итеративной разработке сначала строится программный продукт, удовлетворяющий нескольким конечным требованиям, но конструктивно выполненный так, чтобы облегчить затем удовлетворение всех требований и достичь таким образом большей адаптируемости" [[8](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#7.8)]. С точки зрения конечного пользователя, поток релизов проистекает из макропроцесса. Каждый следующий релиз поддерживает все больше функций, и в конечном счете они развиваются в готовую систему. С точки зрения человека, наблюдающего процесс изнутри, создается много больше релизов, но только некоторые из них будут "заморожены", чтобы стабилизировать важнейшие интерфейсы системы, и взяты за основу для дальнейшей работы. Такая стратегия дает возможность снизить риск разработки, ускоряет выявление проблем в архитектуре и узких мест уже на ранних стадиях.  Для больших проектов организация может готовить внутренние релизы системы каждые несколько недель, а релизы для заказчика - раз в несколько месяцев, в соответствии с потребностями проекта. В стабильном состоянии релиз состоит из множества совместимых подсистем вместе с соответствующей документацией. Приступать к построению релиза можно при условии, что главные подсистемы проекта достаточно стабильны, а их совместное взаимодействие достаточно слажено, чтобы обеспечить новый уровень функциональности.  Управление конфигурацией и версиями  Рассмотрим поток релизов с точки зрения разработчика, занятого созданием некоторой подсистемы. Он имеет текущую версию этой подсистемы. Следовательно, в его распоряжении должны быть как минимум интерфейсы всех импортируемых подсистем. Когда рабочая версия стабилизируется, она передается команде, занимающейся интеграцией, которая отвечает за сборку совместимых подсистем в целую систему. В конце концов набор подсистем замораживается, берется за точку отсчета и входит во внутренний релиз. Внутренний релиз становится текущим оперативным релизом, доступным всем разработчикам, которым нужно провести дальнейшую доработку своих частей реализации. Наш разработчик в это время может трудиться над новой версией своей подсистемы. Таким образом, разработка может вестись параллельно и оставаться устойчивой благодаря четко определенным и защищенным интерфейсам подсистем.  В этой модели неявно подразумевается, что единицей контроля версий является не отдельный класс, а группа классов. Опыт показывает, что управление версиями классов слишком безнадежное дело, так как они слишком зависимы друг от друга. Лучше выпускать версии связанных групп классов, точнее говоря - подсистем, поскольку кластеры классов логической модели системы отображаются в подсистемы физической модели.  В любой момент разработки системы могут существовать несколько версий ее подсистем: версия для текущего разрабатываемого релиза, версия для текущего внутреннего релиза, версия для последующего релиза, предназначенного для заказчика и т.д. Это обостряет потребность в достаточно мощных средствах управления конфигурацией и версиями.  Под понятие "исходный код" подпадает не только текст программ, но и все остальные продукты объектно-ориентированного развития: технические требования, диаграммы классов, объектов, модулей и процессов.  Тестирование  Принцип непрерывной интеграции приложим и к тестированию, которое также производится в течение всего процесса разработки. В контексте объектно-ориентированной архитектуры тестирование должно охватывать как минимум три направления:  Тестирование модулей.  Предполагает тестирование отдельных классов и механизмов; является обязанностью инженера, который их реализовал.  Тестирование подсистем.  Предполагает тестирование целых категорий или подсистем; является обязанностью ответственного за подсистему; тесты подсистем могут использоваться регрессивно для каждой вновь выпускаемой версии подсистемы.  Тестирование системы  Предполагает тестирование системы как целого; является обязанностью контролеров качества; тестирование системы, как правило, тоже происходит регрессивно.  Тестирование должно фокусироваться на внешнем поведении системы; его побочная цель - определить границы системы чтобы понять, как она может выходить из строя при определенных условиях.  7.4. Повторное использование  Элементы повторного использования  Любой программный продукт (текст программы, архитектура, сценарий или документация) может быть использован повторно. Как сказано в главе 3, в объектно-ориентированных языках программирования первичным лингвистическим средством повторного использования являются классы: класс может порождать подклассы, специализирующие или дополняющие его. Далее, в главе 4 говорилось о повторном использовании шаблонов классов, объектов и элементов проектирования в форме идиом, механизмов и сред разработки. Повторное использование шаблонов находится на более высоком уровне абстракции по сравнению с использованием индивидуальных классов и дает больший выигрыш (хотя оно труднее достижимо).  Не следует доверять цифрам, характеризующим повторное использование [[9](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#7.9)]. В удачных проектах, с которыми мы сталкивались, количество повторно использованных элементов доходило до 70% (то есть почти три четверти программного обеспечения системы было взято без изменений из некоторого другого источника), но бывало и нулевым. Не следует думать, что повторное использование должно достичь некоторой обязательной величины; возможность повторного использования сильно зависит от предметной области и нетехнических факторов, таких, например, как степень напряженности рабочего графика, природа отношений с субподрядчиками и соображения безопасности.  Безусловно, любой процент повторного использования лучше, чем нулевой, так как экономит ресурсы, которые иначе пришлось бы потратить еще раз.  Как осуществить повторное использование?  Повторное использование в пределах проекта или даже целой организации не должно протекать по воле случая. Нужно специально выискивать возможности и поощрять успехи. Именно поэтому мы включили поиск повторяющихся шаблонов в макропроцесс.  Лучше всего поручить повторное использование кому-то лично. Как описывалось в предыдущей главе, надо искать возможные общности, обычно выявляемые при просмотре архитектуры, реализовывать их, создавая новые или приспосабливая старые компоненты, а потом отстаивать их перед другими разработчиками. Даже простые формы поощрения, такие, как равное признание автора первоначального кода и первооткрывателя возможности заимствования, оказывают стимулирующее воздействие. Можно придумать что-нибудь посущественнее - обед в ресторане или путешествие на выходные для двоих - и присуждать эти поощрения тем разработчикам, чьи решения были заимствованы чаще всего, или тем, которые заимствовали наибольшую часть кода за заданное время [Близкие к разработчикам люди часто терпят некоторый моральный урон в заключительной горячке разработки, и такая компенсация им будет весьма кстати].  Повторное использование может и не принести краткосрочных выгод, но окупается в долгосрочной перспективе. Этим имеет смысл заниматься в организации, которая имеет обширные, далеко идущие планы разработки программного обеспечения и смотрит дальше интересов текущего проекта.  7.5. Качество и измерения  Качество программного продукта  Шульмейер и МакМанус определяют качество программного продукта как "пригодность к использованию" [[10](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#7.10)]. Качество программы не должно быть делом случая. Качество должно гарантироваться процессом разработки. На самом деле, объектно-ориентированной технология не порождает качества автоматически: можно написать сколь угодно плохие программы на любом объектно-ориентированном языке программирования.  Вот почему в процессе объектно-ориентированной разработки мы придаем такое значение архитектуре программной системы. Качество закладывается благодаря простой, гибкой архитектуре и осуществляется естественными и последовательными тактическими решениями.  Контроль качества программного продукта - это "систематические действия, подтверждающие пригодность к использованию программного продукта в целом" [[11](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#7.11)]. Цель контроля качества - дать нам количественные меры качества программной системы. Многие традиционные количественные меры непосредственно приложимы и к объектно-ориентированным системам.  Как описывалось выше, разбор и просмотр не теряют своей значимости в объектно-ориентированных системах, позволяя предсказать качество системы и влиять на него. Возможно, самым главным количественным критерием качества является количество обнаруженных ошибок. Во время эволюции системы мы учитываем программные ошибки в соответствии с их весом и расположением. График обнаружения ошибок отображает зависимость количества обнаруженных ошибок от времени. Как указывает Доббинс, "не так важно действительное число ошибок, как наклон этого графика" [[12](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#7.12)]. Для управляемого процесса этот график имеет форму горба, с вершиной примерно в середине периода тестирования, а дальше эта кривая падает до нуля. Неуправляемому процессу соответствует неубывающая со временем или медленно убывающая кривая.  Одно из достоинств макропроцесса в объектно-ориентированной разработке состоит в том, что он позволяет вести непрерывный сбор данных о количестве обнаруженных ошибок уже на ранних стадиях разработки. Для каждого нового релиза мы можем провести тестирование системы и нарисовать график зависимости количества ошибок от времени. У "здорового" проекта горбовидная форма этого графика наблюдается для каждого релиза, начиная с самых ранних.  Другая количественная мера - плотность ошибок. Количество обнаруженных ошибок на килостроку программного текста (KSLOC - Kilo Source Lines Of Code) является традиционным показателем, применимым, в частности, к объектно-ориентированным системам. В "здоровых" проектах плотность ошибок "имеет тенденцию достигать стабильного значения при просмотре примерно 10 KSLOC. Просматривая код далее, мы не должны наблюдать увеличения этого показателя" [[13](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#7.13)].  Мы полагаем, что в объектно-ориентированных системах полезно также измерять число ошибок на категорию классов или на класс. При этом правило 80/20 считается приемлемым: 80% выявленных ошибок в программе сосредоточено в 20% классов системы [[14](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#7.14)].  В дополнение к этим более формальным подходам к накоплению получаемой при тестировании информации об ошибках, мы считаем полезным устраивать "охоту за ошибками", в которой все желающие могут экспериментировать с релизом в течение ограниченного промежутка времени, после чего награждается призами тот, кто обнаружил наибольшее количество ошибок, и тот, кто отыскал самую незаметную ошибку. Призы не должны быть экстравагантными: для награждения бесстрашных охотников годятся кофейная кружка, талоны на обед, билеты в кино или даже футболка.  Объектно-ориентированные меры  Наверное, самый скверный способ оценить сделанную работу, каким может воспользоваться управляющий, - сосчитать количество написанных строк текста программы. Число строк, попавших во фрагмент кода, абсолютно никак не связано с его завершенностью и сложностью. В дополнение к другим недостаткам этого неандертальского подхода, в нем слишком просто играть с цифрами, что приводит к оценкам производительности, отличающимся друг от друга более, чем на два порядка. Например, что такое строка программы (особенно на Smalltalk)? Считаются ли физические строки или точки с запятой? Как учесть несколько операторов на одной строке и операторы, которые занимают более одной строки? А как измерить количество затраченного труда? Считать код работой всего персонала или, может быть, только программистов, написавших реализацию? Рабочий день должен считаться восьмичасовым или время, проведенное за утренней раскачкой, тоже должно учитываться? Традиционные меры сложности более подходят для первых поколений языков программирования, они не являются показателями завершенности и сложности объектно-ориентированной системы.  Например, цикломатическая метрика МакКэйба не будет сколько-нибудь полезной мерой сложности, если ее применить к объектно-ориентированной системе в целом, потому что она слепа к структуре классов системы и механизмам. Однако, мы находим полезным применять цикломатическую метрику к отдельным классам, - она дает некоторое представление об их сложности и может быть использована для определения наиболее подозрительных классов, которые, вероятно, содержат больше всего ошибок.  Мы склонны измерять прогресс разработки числом готовых и работающих классов (логический аспект), или количеством функционирующих модулей (физический аспект). Как говорилось в предыдущей главе, другой мерой прогресса является стабильность ключевых интерфейсов (то есть насколько часто они подвергаются изменениям). Сначала интерфейсы всех ключевых абстракций изменяются ежедневно, если не ежечасно. Через некоторое время стабилизируются наиболее важные из ключевых интерфейсов, следом - вторые по важности и т.д. К концу жизненного цикла разработки только несколько несущественных интерфейсов нуждаются в доработке, так как основной упор делается на то, чтобы заставить готовые классы и модули работать вместе. Иногда в ключевые интерфейсы требуется внести некоторые изменения, но они обычно остаются совместимыми снизу вверх. Причем, эти изменения производятся только после того, как будет тщательно продумано их влияние. Эти изменения могут быть внесены в разрабатываемую систему при подготовке нового релиза.  Чидамбер и Кемерер предлагают несколько мер, которые непосредственно применимы к объектно-ориентированным системам [[15](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#7.15)]:  взвешенная насыщенность класса методами;  глубина дерева наследования;  число потомков;  зацепление объектов;  отклик на класс;  недостаток связности в методах.  Взвешенная насыщенность класса дает относительную меру его сложности; если считать, что все методы имеют одинаковую сложность, то это будет просто число методов в классе. Вообще, класс, который имеет большее количество методов среди классов одного с ним уровня, является более сложным; скорее всего, он специфичен для данного приложения и содержит наибольшее количество ошибок.  Глубина дерева наследования и число потомков - количественные характеристики формы и размера структуры классов. Как обсуждалось в главе 3, хорошо структурированная объектно-ориентированная система чаще бывает организована как лес классов, чем как одно высоченное дерево. Мы советуем строить сбалансированные по глубине и ширине структуры наследования: обычно - не глубже, чем 7Ѓ2 уровня, и не шире, чем 7Ѓ2 ветви.  Зацепление объектов - мера их взаимозависимости. Так же, как и при традиционном программировании, мы стремимся спроектировать незацепленные (то есть слабо связанные) объекты, поскольку они имеют больше шансов на повторное использование.  Отклик на класс - количество методов, которые могут вызываться экземплярами класса. Связность методов - мера насыщенности абстракции. Класс, который может вызывать существенно больше методов, чем равные ему по уровню классы, является более сложным. У класса с низкой связностью можно подозревать случайную или неподходящую абстракцию: такой класс должен, вообще говоря, быть переабстрагирован в несколько классов или его обязанности должны быть переданы другим существующим классам.  7.6. Документация  Наследие разработки  Разработка программной системы включает в себя гораздо больше, чем просто написание кода. Некоторые аспекты проекта должны быть постоянно доступны менеджерам разработки и внешним пользователям. Мы хотим также сохранить сведения о решениях, принятых при анализе и проектировании для тех, кто будет заниматься сопровождением системы. Как отмечалось в главе 5, результаты объектно-ориентированной разработки обычно содержат диаграммы классов, объектов, модулей и процессов. В совокупности эти диаграммы предоставляют возможность проследить их появление непосредственно из начальных требований к системе. Диаграммы процессов соответствуют программам, которые являются корневыми модулями диаграммы модулей. Каждый модуль представляет реализацию некоторой комбинации классов и объектов, которые, в свою очередь, можно найти на диаграммах классов и объектов соответственно. Наконец, диаграммы объектов представляют сценарии, определяемые требованиями, а диаграммы классов демонстрируют ключевые абстракции, которые формируют словарь предметной области.  Содержание документации  Документация по архитектуре системы важна, но ее составление не должно быть двигателем процесса разработки: документация - существенный, но не самый главный результат. Важно помнить, что документация - живой продукт, и ей надо позволить эволюционировать вместе с релизами проекта. Вместе с текстом программ сопровождающая документация служит основой для проведения многих формальных и неформальных просмотров.  Что должно быть документировано? Очевидно, что документация, представляемая конечному пользователю, должна включать инструкции по установке и использованию каждого релиза. Кроме того, должны быть документированы результаты анализа, чтобы зафиксировать семантику функциональных точек системы в последовательности сценариев. Должна также вестись документация по архитектуре и реализации для согласования в команде разработчиков общего видения системы и деталей архитектуры, а также для того, чтобы сохранить информацию обо всех стратегических решениях - это несомненно облегчает эволюцию и адаптацию системы.  Документация по архитектуре и реализации должна описывать:  архитектуру системы верхнего уровня;  ключевые абстракции и механизмы архитектуры;  сценарии, иллюстрирующие важнейшие аспекты предусмотренного поведения системы.  Наихудшая документация, которую можно создать для объектно-ориентированной системы - это расписанные по классам объяснения смысла каждого метода в отдельности. При таком подходе получаются горы бесполезной бумаги, которую никто не читает и не доверяет ей, причем оказываются потеряны наиболее важные архитектурные решения, выходящие за рамки отдельных классов и проявляющиеся в сотрудничестве классов и объектов. Гораздо лучше документировать эти структуры верхнего уровня на языке диаграмм в описанной выше системе обозначений, но без явных операторов языка программирования, и сделать ссылки для разработчиков на интерфейсы важнейших классов для уточнения тактических деталей.  7.7. Инструменты  В предыдущих поколениях языков программирования команде разработчиков достаточно было иметь минимальный набор инструментов: редактор, компилятор, компоновщик и загрузчик - вот все, что обычно требовалось (и, чаще всего, все, что имелось). Особо благополучная команда могла обзавестись кодовым отладчиком. Сложные задачи в корне изменили картину: попытка построить большую программную систему с минимальным набором инструментов эквивалентна намерению возвести многоэтажное здание вручную.  Объектно-ориентированный подход также изменил многое. Традиционные инструменты разработки сосредоточены только на тексте программы, но объектно-ориентированные анализ и проектирование выдвигают на первый план ключевые абстракции и механизмы, и нам нужны инструменты, работающие с более богатой семантикой. Кроме того, для ускорения движимого макропроцессом объектно-ориентированного развития, требуются инструменты, позволяющие ускорить циклы разработки, особенно цикл редактирование/компиляция/выполнение/отладка.  Важно выбрать хорошо масштабируемые инструменты. Инструмент, который удобен для программиста-одиночки, занятого написанием небольшого приложения, не всегда подходит для производства сложных систем. На самом деле, для каждого инструмента существует порог, за которым его возможности оказываются превышены: его достоинства перевешиваются неуклюжестью и нерасторопностью.  Виды инструментов  Мы выделяем по крайней мере семь различных видов инструментов для объектно-ориентированной разработки. Первый инструмент - система с графическим интерфейсом, поддерживающая объектно-ориентированную систему обозначений, представленную в главе 5. Такой инструмент может быть использован при анализе, чтобы зафиксировать семантику сценариев, на ранних стадиях разработки, чтобы передать стратегические и тактические решения, принятые при проектировании, а также для координирования действий проектировщиков. Подобный инструмент будет полезен на протяжении всего жизненного цикла и при сопровождении системы. В частности, мы считаем возможным восстанавливать по тексту программы многие важные аспекты объектно-ориентированной системы. Такая способность инструментальной системы очень важна: в традиционных CASE-средствах разработчик может создавать замечательные картинки только для того, чтобы обнаружить, что они устарели, потому что программисты манипулируют реализацией, не отраженной в проекте. Восстановление проекта по коду снижает вероятность того, что документация будет идти не в ногу с реализацией.  Следующий важный для объектно-ориентированной разработки инструмент - броузер, который показывает структуру классов и архитектуру модулей системы. Иерархия классов может сделаться настолько сложной, что трудно даже отыскать все абстракции, которые были введены при проектировании или выявить кандидатов для повторного использования [[16](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#7.16)]. При изучении фрагмента программы разработчику может понадобиться посмотреть определение класса некоторого объекта. Найдя этот класс, ему вероятно придется заглянуть в описание какого-нибудь из его суперклассов. Рассматривая этот суперкласс, разработчик может захотеть внести изменения в интерфейс класса, но при этом придется изучить поведение всех его клиентов. Этот анализ будет особенно громоздким, если он применяется к файлам, которые представляют физические, а не логические аспекты проекта. По этой причине броузер оказывается очень важным инструментом объектно-ориентированного анализа и проектирования. Броузеры есть, например, в среде Smalltalk. Подобные средства, хотя и разного качества, имеются и для других объектно-ориентированных языков.  Другой вид инструментов, который очень важен, если не абсолютно необходим - инкрементный компилятор. Метод эволюционной разработки, который применяется в объектно-ориентированном программировании, нуждается в компиляторе, который мог бы компилировать отдельные объявления и операторы. Мейровиц замечает, что "UNIX, с ее ориентацией на пакетное компилирование больших программных файлов в библиотеки, которые потом компонуются с другими фрагментами кода, не предоставляет необходимой поддержки для объектно-ориентированного программирования. Совершенно недопустимо тратить десять минут на компиляцию и компоновку, чтобы изменить реализацию метода и тратить целый час на компиляцию и компоновку только для того, чтобы добавить новое поле в суперкласс верхнего уровня! Для быстрой отладки методы и определения полей должны компилироваться инкрементно" [[17](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#7.17)]. Такие компиляторы существуют для многих языков, описанных в приложении. К сожалению, большинство реализации содержит традиционные пакетно-ориентированные компиляторы.  Мы также полагаем, что нетривиальные проекты нуждаются в отладчиках, которые могут работать с семантикой классов и объектов. При отладке программ часто требуется справка о переменных экземпляра и переменных класса для данного объекта. Традиционные отладчики для необъектных языков ничего не знают о классах и объектах. Так, пытаясь использовать стандартный отладчик языка С для программы на C++, разработчик не сможет получить всею информацию, необходимую для отладки объектно-ориентированной программы. Ситуация особенно критична для объектно-ориентированных языков, поддерживающих несколько потоков управления. При выполнении такой программы в один и тот же момент времени могут быть запущены несколько активных процессов. Естественно требуется отладчик, который позволял бы контролировать каждый отдельный поток и взаимодействие объектов между потоками.  В категорию отладочных средств мы включаем и такие инструменты, как стрессовые тестеры, которые испытывают программы в критических условиях ограниченности ресурсов, и инструменты для анализа памяти, которые распознают нарушения доступа к памяти (запись в неразрешенные участки памяти, чтение из неинициализированных участков, чтение или запись за границами массива).  Для больших проектов требуются инструменты управления конфигурацией и контроля версий. Как упоминалось ранее, для управления конфигурацией лучшими единицами являются категории классов и подсистемы.  Другой инструмент, который мы считаем важным для объектно-ориентированной разработки, - это библиотекарь классов. Многие из языков, о которых упоминается в этой книге, поставляются со стандартными библиотеками или библиотеки для них можно купить отдельно. По мере развития проекта библиотека классов растет, дополняясь новыми предметно-специфическими программными компонентами, годными для повторного использования. Библиотека быстро разрастается до таких размеров, что разработчики не могут отыскать понадобившийся класс. Одна из причин быстрого роста библиотеки состоит в том, что класс может иметь несколько реализации с различными временными и пространственными семантиками. Если ожидаемая стоимость (обычно преувеличенная) отыскания компоненты выше, чем ожидаемая стоимость (обычно преуменьшенная) создания этой компоненты заново, пропадает всякий смысл повторного использования. По этой причине бывает важно иметь хоть какой-нибудь минимальный библиотечный инструмент, который позволял бы разработчику искать классы и модули, удовлетворяющие различным критериям, и заносить в библиотеку полезные классы и модули по мере их разработки.  Еще один тип инструмента, который мы считаем полезным для некоторых систем - генератор графического интерфейса пользователя. Для систем, в которых велик объем взаимодействия с пользователем, лучше иметь специальный инструмент для интерактивного создания диалогов и окон, чем программировать все с нуля. Код, сгенерированный такой системой, может быть потом связан с остальной объектно-ориентированной системой и, если необходимо, вручную подкорректирован.  Организационные выводы  Потребность в мощных инструментах приводит к появлению двух специальных должностей в штате организации, занимающейся разработкой программных систем: инженер по повторному использованию и инженер-инструментальщик. Среди прочего, обязанности первого состоят в сопровождении библиотеки классов проекта. Без активных усилий она может стать необозримым пустырем превратившихся в хлам классов, в который ни один из разработчиков не захочет заглядывать. Часто бывает, что разработчиков надо побуждать к заимствованию существующих компонентов или предотвращать изобретение велосипеда; это тоже входит в задачи инженера по повторному использованию. В обязанности инженера-инструментальщика входят создание новых предметно-зависимых инструментов и переделка существующих инструментов для текущих нужд. Например, может потребоваться целая система тестов для проверки некоторых аспектов пользовательского интерфейса или более специализированный броузер классов. Инструментальщик облегчает свою работу тем, что разрабатывает и развивает инструменты, используя компоненты, уже помещенные в библиотеку классов. Их можно задействовать и в новых разработках.  Менеджер, ограниченный в средствах, может жаловаться, что хорошие инструменты, инженеры по повторному использованию и инструментальщики - непозволительная роскошь. Для некоторых проектов - да. Однако часто все это так или иначе делается, но стихийным, неэффективным образом. Мы стоим за явные затраты на средства и персонал, чтобы сделать эту деятельность более концентрированной и эффективной и увеличить ценность общей организации.  7.8. Специальные вопросы  Узко-специфические проблемы  Мы считаем, что некоторые предметные области заслуживают специального архитектурного рассмотрения.  Проектирование эффективного пользовательского интерфейса - скорее искусство, чем наука. В этой области абсолютно необходимо использование прототипов. Как можно раньше следует установить интенсивную обратную связь с конечными пользователями, чтобы выявить характерные движения рук, реакцию на ошибки и другие парадигмы взаимодействия. При анализе пользовательского интерфейса в высшей степени эффективно составление сценариев.  Некоторые приложения имеют собственную базу данных, для других приложений может требоваться интеграция с существующей базой данных, схема которой не может быть изменена (обычно из-за того, что база уже заполнена большим количеством данных - разновидность проблемы унаследованной сложности). В таких случаях хорошо работает принцип разделения обязанностей: лучше всего инкапсулировать доступ к таким базам данных в небольшом количестве четко определенных интерфейсов классов. Этот принцип особенно важен при совместном использовании объектно-ориентированной декомпозиции и технологии реляционных баз данных (РБД). Объектно-ориентированные базы данных (ООБД) лучше стыкуются с объектными программами, но мы должны помнить, что ООБД больше подходят для обеспечения продолжительности жизни объектов и меньше - для хранения больших объемов данных.  Коснемся также систем реального времени. Понятие реальное время в различном контексте воспринимается по-разному. В диалоговых системах оно может означать отклик в течение менее чем одной секунды, а в системах сбора данных и управления может требоваться отклик быстрее чем за одну микросекунду. Важно ясно понимать, что даже при очень жестких требованиях ко времени не каждая компонента должна (или может) быть оптимизирована. Для сложных систем наибольший риск состоит в том, будет или нет система завершена, а не в том, будет ли она удовлетворять требованиям эффективности. По этой причине мы предостерегаем от преждевременной оптимизации. Сосредоточьтесь на создании простой архитектуры, а узкие места выявятся в процессе эволюции системы сами собой (причем достаточно рано), и вы сможете принять меры.  Термин "унаследованная сложность" относится к приложениям, в которые были сделаны большие капиталовложения и от которых по экономическим причинам или по соображениям безопасности нельзя отказаться. Однако, такие системы могут иметь неподъемную стоимость сопровождения, отчего их и приходится со временем заменять. К счастью, совладать с унаследованной сложностью можно почти также, как с базами данных: инкапсулировать доступ к их услугам в контексте четко определенных интерфейсов классов, а потом постепенно, функцию за функцией, вытеснять их объектно-ориентированной архитектурой. Конечно, необходимо представлять, себе конечный результат, чтобы система в процессе неуправляемых изменений не превратилась в лоскутное одеяло.  Переход на объектные технологии  Как считает Кемпф, "Изучение объектно-ориентированного программирования может оказаться гораздо более трудной задачей, чем просто освоение очередного языка. В данном случае надо научиться другому стилю программирования, а не просто запомнить синтаксис. То есть учить приходится не язык, а способ мышления" [[18](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#7.18)]. Как развить объектно-ориентированное мировоззрение? Мы рекомендуем:  Обеспечить формальное обучение разработчиков и менеджеров элементам объектной модели.  Начать с проектов с низким уровнем риска и позволить команде делать ошибки; по мере приобретения опыта можно направлять участников в другие команды в качестве наставников.  Продемонстрировать разработчикам и менеджерам примеры хорошо структурированных объектно-ориентированных систем.  Хорошими кандидатами для первых проектов являются инструментальные средства или предметно-ориентированные библиотеки классов, которые можно потом использовать как ресурсы для следующих проектов.  Согласно нашему опыту, профессиональному разработчику требуется несколько недель, чтобы освоиться с синтаксисом и семантикой нового языка программирования. Потребуется в несколько раз больше времени, чтобы тот же разработчик начал понимать важность и мощь классов и объектов. Наконец, может потребоваться более шести месяцев экспериментов, чтобы разработчик созрел до хорошего проектировщика классов. Это не обязательно плохо - обучение мастерству всегда требует времени.  Мы считаем, что обучение на примерах часто оказывается эффективным и целесообразным подходом. Когда организация накопит критическую массу приложений, выполненных в объектно-ориентированном стиле, станет гораздо легче обучать этому делу новых разработчиков и менеджеров. Разработчики начинают как программисты-исполнители, применяя уже готовые хорошо структурированные абстракции. Через некоторое время разработчики, которые изучали и использовали эти компоненты под наблюдением более опытных сотрудников, сами приобретают достаточный опыт, чтобы проектировать классы.  7.9. Выгоды и опасности объектно-ориентированной разработки  Выгоды  Приверженцы объектно-ориентированной технологии обычно называют два ее главных преимущества. Во-первых, большая конкурентоспособность благодаря предсказуемости, сокращению времени на разработку и большой гибкости продукта. Во-вторых, разрабатываемые задачи могут быть настолько сложными, что не остается альтернативных решений.  В главе 2 говорилось, что использование объектной модели позволяет перенести в программу пять свойств хорошо структурированных сложных систем. Объектная модель формирует концептуальный каркас системы обозначений и процесса объектно-ориентированной разработки; таким образом, и эти выгоды мы получаем непосредственно благодаря методу. Отмечались и преимущества, вытекающие из того, что объектная модель (а значит и процесс разработки):  использует выразительную мощь объектно-ориентированных языков программирования;  стимулирует повторное использование программных компонент;  приводит к созданию более гибких, легко изменяемых систем;  сокращает риск разработки;  лучше воспринимается человеческим сознанием.  Изучение многочисленных случаев из практики подкрепляет эти выводы; особенно часто указывается на то, что объектный подход может сократить время разработки и размер кода [[19](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#7.19), [20](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#7.20), [21](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#7.21)].  Опасности  Говоря о теневой стороне объектно-ориентированной технологии, нужно рассмотреть два вопроса: производительность и начальные затраты. По сравнению с процедурными языками, объектно-ориентированные языки определенно вносят дополнительные накладные расходы на пересылку сообщения от одного объекта другому. Как указывалось в главе 3, при вызове методов, которые не найдены и не связаны статически во время компиляции, выполняемая программа должна динамически искать нужный метод по классу объекта-получателя. Исследования показывают, что, в худшем случае, на вызов метода тратится в 1.75-2.5 раза больше времени чем на обычный вызов процедуры [[22](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#7.22), [23](http://www.helloworld.ru/texts/comp/other/oop/liter.htm#7.23)]. С другой стороны, наблюдения показывают, что при вызове методов динамический поиск действительно необходим примерно в 20% случаев. В строго типизированных языках компилятор часто может определять, какие вызовы могут быть связаны статически и сгенерировать для них вызов процедуры вместо динамического поиска.  Другая причина снижения производительности кроется не столько в природе объектно-ориентированных языков, сколько в способе их использования в процессе объектно-ориентированной разработки. Как говорилось уже много раз, объектно-ориентированная технология порождает многослойные системы абстракций. Одно из следствий этого расслоения в том, что каждый метод оказывается очень маленьким, так как он строится на методах нижнего уровня. Другое следствие расслоения: иногда методы служат лишь для того, чтобы получить доступ к защищенным атрибутам объекта. В результате происходит слишком много вызовов. Вызов метода на высшем уровне абстракции обычно влечет каскад других вызовов; методы верхних уровней вызывают методы нижних уровней и т.д. Для приложений, в которых время - ограниченный ресурс, недопустимо слишком большое количество вызовов методов. Опять же, с позитивной стороны такое слоение способствует пониманию системы; к некоторым сложным системам невозможно даже подступиться, если не начать с проектирования слоев. Мы рекомендуем сначала проектировать систему с желаемыми функциональными свойствами, а потом определять узкие места. Часто их можно "расшить", объявив соответствующие методы как подстановки (выигрывая тем самым время), "подчистив" иерархию классов или нарушив инкапсуляцию атрибутов класса.  Похожий риск для потери производительности происходит из-за большого количества наследуемого кода. Класс, лежащий глубоко в структуре наследования, может иметь много суперклассов; при компоновке программы должно быть подгружено описание их всех. Для маленьких приложений это практически может означать, что нужно избегать глубоких иерархий классов, потому что они требуют чрезмерного количества объектного кода. Проблему можно несколько смягчить, используя высокоразвитые компиляторы и компоновщики, которые могли бы устранять бесполезные участки программы.  Еще один источник проблем для производительности объектно-ориентированных программ - их поведение в системе со страничной организацией памяти. Большинство компиляторов выделяет память сегментами, размещая каждый компилируемый модуль (обычно файл) в одном или более сегментах. Это подразумевает локальность большинства ссылок: процедуры в одном сегменте вызывают в основном процедуры в том же сегменте. В больших объектно-ориентированных системах все не так. Код каждого класса размещается в отдельном файле, а раз его методы интенсивно используют методы других (особенно вышестоящих) классов, при их выполнении происходит интенсивное переключение сегментов. Это поведение противоречит тому, чего большинство компиляторов ожидает от программ, особенно в системах с конвейерным процессором и страничной организацией памяти. Это еще один пример того, почему нужно разделять логические и физические аспекты проекта. Если программа работает слишком медленно из-за чрезмерно частого переключения страниц, можно попробовать изменить физическое расположение классов в модулях. Это проектное решение, касающееся физической модели системы - на логику программы оно не повлияет.  Наконец, еще одна составляющая риска - динамическое размещение и уничтожение объектов. Динамическое выделение памяти из "кучи" сопряжено с дополнительными вычислительными расходами по сравнению с размещением данных в стеке и (конечно) статическим резервированием при компиляции. Во многих системах это не вызывает никаких практических проблем, но иногда дополнительные накладные расходы непозволительны. Существуют простые решения: откажитесь от динамического создания объектов и разместите их все заранее, или замените стандартный алгоритм выделения памяти на специально приспособленный для ваших нужд.  И опять о хорошем: положительные свойства объектно-ориентированных систем часто окупают все перечисленные выше неприятности. Например, Руссо и Каплан сообщают, что производительность программы на C++ часто бывает выше, чем ее функционального эквивалента на С. Выигрыш, как они полагают, связан с использованием виртуальных функций, благодаря которым можно сэкономить на проверке типов и опустить многие управляющие конструкции. Согласно нашему опыту, код объектно-ориентированной программы и в самом деле обычно короче.  Для некоторых проектов начальные затраты на переход к объектно-ориентированной технологии могут оказаться непреодолимыми. Как при всякой смене технологии, придется вкладывать деньги в покупку новых инструментальных средств разработки. Кроме того, если какой-либо объектно-ориентированный язык используется организацией впервые, то нет и предметно-специфического задела для повторного использования. Короче говоря, приходится начинать все сначала или найти какой-то способ использовать существующие программы в объектно-ориентированном окружении. Наконец, первая попытка объектно-ориентированной разработки наверняка провалится, если не было проведено соответствующее обучение. Объектная ориентация - это не "еще один" язык программирования, который можно выучить на трехдневных курсах или по книжке. Как мы многократно указывали, требуется время, чтобы освоить объектно-ориентированное проектирование как новое мировоззрение, которое должно быть усвоено как разработчиками, так и менеджерами.  Выводы  Успешная разработка и внедрение сложных программных систем - это нечто больше, чем просто программирование.  Многие приемы традиционного менеджмента программных разработок, например, просмотр, применимы и в объектно-ориентированной технологии.  В стабильном состоянии объектно-ориентированные проекты требуют меньших ресурсов, а роли, необходимые для управления этими ресурсам, несколько отличаются от традиционных.  В процессе объектно-ориентированной разработки нельзя производить интеграцию всего сразу и за один раз; структурными единицами управления для релизов должны быть категории классов и подсистемы, а не отдельные файлы и классы.  Повторным использованием надо заниматься специально.  График числа обнаруженных ошибок за определенное время и плотность ошибок - полезные количественные меры качества объектно-ориентированных программ. Существует ряд полезных количественных характеристик, ориентированных на классы.  Документация никогда не должна ставиться во главу угла при разработке.  Объектно-ориентированная разработка требует иного инструментария по сравнению с традиционными методами.  Переход организации на объектно-ориентированные технологии - это смена мировоззрения, а не просто изучение нового языка программирования.  Объектно-ориентированные технологии связаны как с выгодами, так и с опасностями, но опыт показывает, что выгод много больше. |

Додатки