**1 Объектно-ориентированный подход основан на систематическом использовании моделей для языково-независимой разработки программной системы, на основе из ее прагматики.**

*Прагматика* определяется целью разработки программной системы: для обслуживания клиентов банка, для управления работой аэропорта и т.д. В формулировке цели участвуют предметы и понятия реального мира, имеющие отношение к разрабатываемой программной системе. При объектно-ориентированном подходе эти предметы и понятия заменяются их моделями, т.е. определенными формальными конструкциями, представляющими их в программной системе.

Модель содержит не все признаки и свойства представляемого ею предмета (понятия), а только те, которые существенны для разрабатываемой программной системы. Тем самым модель "беднее", а, следовательно, проще представляемого ею предмета (понятия). Но главное даже не в этом, а в том, что модель есть формальная конструкция: формальный характер моделей позволяет определить формальные зависимости между ними и формальные операции над ними. Это упрощает как разработку и изучение (анализ) моделей, так и их реализацию на компьютере. В частности, формальный характер моделей позволяет получить формальную модель разрабатываемой программной системы как композицию формальных моделей ее компонентов.

Таким образом, объектно-ориентированный подход помогает справиться с такими сложными проблемами, как

* уменьшение сложности программного обеспечения;
* повышение надежности программного обеспечения;
* обеспечение возможности модификации отдельных компонентов программного обеспечения без изменения остальных его компонентов;
* обеспечение возможности повторного использования отдельных компонентов программного обеспечения.

Объектно-ориентированная разработка программного обеспечения связана с применением объектно-ориентированных моделей при разработке программных систем и их компонентов. Говоря об объектно-ориентированной разработке, я имею в виду:

* объектно-ориентированные методологии (технологии) разработки программных систем;
* инструментальные средства, поддерживающие эти технологии.

Объектно-ориентированная разработка может начаться на самом первом этапе жизненного цикла; она не связана с языком программирования, на котором предполагается реализовать разрабатываемую программную систему: этот язык может и не быть объектно-ориентированным. На этапе разработки объекты - это некоторые формальные конструкции (например, прямоугольники с закругленными углами, с помощью которых они изображаются на схемах), никак пока не связанные с их будущей реализацией на одном из языков программирования.

Объектно-ориентированная разработка программного обеспечения связана с применением объектно-ориентированных методологий (Например, OMT). Обычно эти методологии поддерживаются инструментальными программными средствами, но и без таких средств они полезны, так как позволяют хорошо понять различные аспекты и свойства разрабатываемой программной системы, что в последующем существенно облегчает ее реализацию, тестирование, сопровождение, разработку новых версий и более существенную модификацию.

**Объектно-ориентированный язык программирования** (ОО-язык) — язык,построенный на принципах объектно-ориентированного программирования.

Неполный список объектно-ориентированных языков программирования:

|  |  |  |
| --- | --- | --- |
| * [C#](http://ru.wikipedia.org/wiki/C_Sharp) * [C++](http://ru.wikipedia.org/wiki/C%2B%2B) * [F#](http://ru.wikipedia.org/wiki/F_Sharp) * [Java](http://ru.wikipedia.org/wiki/Java) * [Delphi](http://ru.wikipedia.org/wiki/Delphi_(%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)) * [Eiffel](http://ru.wikipedia.org/wiki/Eiffel) * [Simula](http://ru.wikipedia.org/wiki/Simula) * [D](http://ru.wikipedia.org/wiki/D_(%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)) * [Io](http://ru.wikipedia.org/wiki/Io) * [Objective-C](http://ru.wikipedia.org/wiki/Objective-C) | * [Swift](http://ru.wikipedia.org/wiki/Swift_(%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)) * [Object Pascal](http://ru.wikipedia.org/wiki/Object_Pascal) * [VB.NET](http://ru.wikipedia.org/wiki/Visual_Basic_.NET) * [Visual DataFlex](http://ru.wikipedia.org/wiki/Visual_DataFlex) * [Perl](http://ru.wikipedia.org/wiki/Perl) * [PowerBuilder](http://ru.wikipedia.org/wiki/PowerBuilder) * [Python](http://ru.wikipedia.org/wiki/Python) * [Scala](http://ru.wikipedia.org/wiki/Scala_(%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)) * [ActionScript](http://ru.wikipedia.org/wiki/ActionScript) (3.0) | * [JavaScript](http://ru.wikipedia.org/wiki/JavaScript) * [JScript .NET](http://ru.wikipedia.org/wiki/JScript_.NET) * [Ruby](http://ru.wikipedia.org/wiki/Ruby) * [Smalltalk](http://ru.wikipedia.org/wiki/Smalltalk) * [Ada](http://ru.wikipedia.org/wiki/%D0%90%D0%B4%D0%B0_(%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)) * [Xbase++](http://ru.wikipedia.org/wiki/Xbase%2B%2B) * [X++](http://ru.wikipedia.org/wiki/X%2B%2B) * [Vala](http://ru.wikipedia.org/wiki/Vala) * [PHP](http://ru.wikipedia.org/wiki/PHP) * [Cyclone](http://ru.wikipedia.org/wiki/Cyclone) |

В современных ОО языках используются механизмы:

**3.** [***Наследование***](http://ru.wikipedia.org/wiki/%D0%9D%D0%B0%D1%81%D0%BB%D0%B5%D0%B4%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5))**. Создание нового класса объектов путём добавления новых элементов** ([методов](http://ru.wikipedia.org/wiki/%D0%9C%D0%B5%D1%82%D0%BE%D0%B4)). Некоторые ОО языки позволяют выполнять [множественное наследование](http://ru.wikipedia.org/wiki/%D0%9C%D0%BD%D0%BE%D0%B6%D0%B5%D1%81%D1%82%D0%B2%D0%B5%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BD%D0%B0%D1%81%D0%BB%D0%B5%D0%B4%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5), то есть объединять в одном классе возможности нескольких других классов.

**2.** [***Инкапсуляция***](http://ru.wikipedia.org/wiki/%D0%98%D0%BD%D0%BA%D0%B0%D0%BF%D1%81%D1%83%D0%BB%D1%8F%D1%86%D0%B8%D1%8F_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5))**. Сокрытие деталей реализации, которое позволяет вносить изменения в** части программы безболезненно для других её частей, что существенно упрощает сопровождение и модификацию ПО.

**4.** [***Полиморфизм***](http://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%BB%D0%B8%D0%BC%D0%BE%D1%80%D1%84%D0%B8%D0%B7%D0%BC_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5))**. При полиморфизме некоторые части (методы) родительского класса** заменяются новыми, реализующими специфические для данного потомка действия. Таким образом, [интерфейс классов](http://ru.wikipedia.org/w/index.php?title=%D0%98%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81_%D0%BA%D0%BB%D0%B0%D1%81%D1%81%D0%BE%D0%B2&action=edit&redlink=1) остаётся прежним, а реализация методов с одинаковым названием и набором параметров различается. С полиморфизмом тесно связано позднее связывание.

**2. Основные понятия объектно-ориентированного подхода. Понятие инкапсуляции**

Существует 3 принципа ООП: инкапсуляция, наследование, полиморфизм. Важным элементом ООП является абстракция. Остановимся на инкапсуляции.

Инкапсуляция — это механизм, который связывает код и данные, которыми он манипулирует. Один из возможных способов представления инкапсуляции — представление в виде защитной оболочки, которая предохраняет код и данные от произвольного доступа другого кода, находящегося снаружи оболочки. Доступ к коду и данным, находящимся внутри оболочки, строго контролируется тщательно определенным интерфейсом. Основа инкапсуляции — класс.



Пустые окружности — общедоступные переменные (не рекомендуется)

Пустые треугольники - общедоступные методы

Закрашенные треугольники — частные методы

Закрашенные окружности — приватные переменные экземпляра

**9. Объектная модель системы**

Объектная модель описывает структуру объектов, составляющих систему, их атрибуты, операции, взаимосвязи с другими объектами. В объектной модели должны быть отражены те понятия и объекты реального мира, которые важны для разрабатываемой системы. В объектной модели отражается прежде всего прагматика разрабатываемой системы, что выражается в использовании терминологии прикладной области, связанной с использованием разрабатываемой системы.

9. Объекты и классы. Атрибуты объектов.

По определению *объектом* можно называть понятие, абстракцию или любую вещь с четко очерченными границами, имеющую смысл в контексте рассматриваемой прикладной проблемы. Введение объектов преследует две цели:

* понимание прикладной задачи (проблемы);
* введение основы для реализации на компьютере.

Примеры объектов: форточка, Банк "Империал", Петр Сидоров, дело № 7461, сберкнижка и т.д.

Все объекты могут быть отличены один от другого: пусть у нас есть два яблока, имеющие одинаковый цвет, форму, вес и вкус; все равно это два яблока (а не одно), в чем легко убедиться, съев одно из них (другое останется). Между объектами можно установить отношение тождества: объекты, удовлетворяющие этому отношению, одинаковы (тождественны), как вышеупомянутые яблоки.

Два яблока из предыдущего примера принадлежат одному и тому же *классу* объектов (именно с этим связана их одинаковость). Цвет, форма, вес и вкус яблока - это его *атрибуты*: совокупность атрибутов и их значений (например, красное, овальное, стограммовое, кисло-сладкое) характеризует объект.

Все объекты одного и того же класса характеризуются одинаковыми наборами атрибутов. Однако объединение объектов в классы определяется не наборами атрибутов, а семантикой. Так, например, объекты конюшня и лошадь могут иметь одинаковые атрибуты: цена и возраст. При этом они могут относиться к одному классу, если рассматриваются в задаче просто как товар, либо к разным классам, что более естественно.

*Атрибут* - это значение, характеризующее объект в его классе. Примеры атрибутов: категория, баланс, кредит (атрибуты объектов класса счет); имя, возраст, вес (атрибуты объектов класса человек) и т.д.

Среди атрибутов различаются *постоянные атрибуты* (константы) и *переменные атрибуты*. Постоянные атрибуты характеризуют объект в его классе (например, номер счета, категория, имя человека и т.п.). Текущие значения переменных атрибутов характеризуют текущее *состояние* объекта (например, баланс счета, возраст человека и т.п.); изменяя значения этих атрибутов, мы изменяем состояние объекта.

**10. Множественное наследование.**

*Множественное наследование* позволяет классу иметь более одного суперкласса, наследуя свойства (атрибуты и операции) всех своих суперклассов. Класс, имеющий несколько суперклассов, называется объединенным классом. Свойства класса-предка, встречающегося более, чем один раз, в графе наследования, наследуются только в одном экземпляре. Конфликты между параллельными определениями порождают двусмысленности, которые должны разрешаться во время реализации. На практике следует избегать таких двусмысленностей или плохого понимания даже в тех случаях, когда конкретный язык программирования, выбранный для реализации системы, предоставляет возможность их разрешения, используя приоритеты или какие-либо другие средства.

В случае, если множественное наследование не поддерживается языком программирования, выбранным для реализации, оно может быть заменено одним из следующих способов.

Делегированием называется механизм реализации, в котором объект, ответственный за операцию, пересылает (делегирует) эту операцию другому объекту; в объектно-ориентированных языках делегирование реализуется путем присоединения методов непосредственно к объектам, а не к классам.

Возможны и другие способы замены множественного наследования. Во всех случаях при выборе способа замены множественного наследования нужно руководствоваться следующими правилами:

* если подкласс имеет несколько суперклассов, каждый из которых одинаково существен, лучше всего использовать делегирование
* если наиболее существенным является только один из суперклассов, а остальные не так важны, наилучшим способом является реализация множественного наследования через простое наследование и делегирование;
* если число возможных комбинаций групп наследуемых свойств невелико, можно использовать вложенное простое наследование; в случае большого числа комбинаций этот способ применять не следует;
* если один из суперклассов передает подклассу намного большее число свойств, чем остальные суперклассы, следует сохранить наследование по этому пути;
* если решено использовать вложенное простое наследование, то на первый уровень вложенности следует поместить наиболее существенный по передаче свойств суперкласс, затем наиболее существенный из оставшихся суперклассов и т.д.;
* следует избегать использования вложенного простого наследования , если это ведет к дублированию достаточно больших частей программы;
* следует помнить, что только вложенное простое наследование обеспечивает полную тождественность множественному наследованию.

**11. Операции и методы**

*Операция* - это функция (или преобразование), которую можно применять к объектам данного класса.

Все объекты данного класса используют один и тот же экземпляр каждой операции (т.е. увеличение количества объектов некоторого класса не приводит к увеличению количества загруженного программного кода). Объект, из которого вызвана операция, передается ей в качестве ее неявного аргумента (параметра).

Одна и та же операция может применяться к объектам разных классов: такая операция называется полиморфной, так как она может иметь разные формы для разных классов. Например, для объектов классов вектор и комплексное\_число можно определить операцию +; эта операция будет полиморфной, так как сложение векторов и сложение комплексных чисел разные операции.

Каждой операции соответствует метод - реализация этой операции для объектов данного класса. Таким образом, операция - это спецификация метода, метод - реализация операции. Например, в классе файл может быть определена операция печать (print). Эта операция может быть реализована разными методами: (а) печать двоичного файла; (б) печать текстового файла и др. Логически эти методы выполняют одну и ту же операцию, хотя реализуются они разными фрагментами кода.

Каждая операция имеет один неявный аргумент - объект к которому она применяется. Кроме того, операция может иметь и другие аргументы (параметры). Эти дополнительные аргументы параметризуют операцию, но не связаны с выбором метода. Метод связан только с классом и объектом.

Операция (и реализующие ее методы) определяется своей сигнатурой, которая включает, помимо имени операции, типы (классы) всех ее аргументов и тип (класс) результата (возвращаемого значения). Все методы, реализующие операцию должны иметь такую же сигнатуру, что и реализуемая ими операция.

Значения некоторых атрибутов объекта могут быть доступны только операциям этого объекта. Такие атрибуты называются *закрытыми.*

Запросы без аргументов (за исключением неявного аргумента - объекта, к которому применяется операция) могут рассматриваться как *производные атрибуты*. Значения производных атрибутов зависят от значений основных атрибутов. В этом их отличие от основных атрибутов, значения которых независимы. Следовательно, значения основных атрибутов объекта определяют как его состояние, так и значения его производных атрибутов. Так, например, длина, ширина и высота комнаты - ее основные атрибуты, а площадь и кубатура - производные атрибуты; такой атрибут как кубатура нужен для того, чтобы не вычислять кубатуру комнаты всякий раз, когда понадобится ее значение.

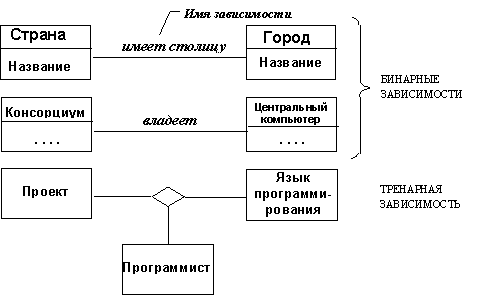
Выбор основных атрибутов объектов произволен, но в число основных атрибутов не следует включать такие атрибуты, значения которых определяются значениями других атрибутов, так что на самом деле они являются производными.

**12. Связь объектов с базой данных.**

В объектно-ориентированном проектировании мы имеем дело с множествами взаимосвязанных объектов. Каждый объект может рассматриваться как переменная или константа структурного типа (при таком рассмотрении методы, описываемые в объекте, трактуются как адреса функций, которые разрешено применять к этому объекту). Следовательно, множество объектов - это множество взаимосвязанных данных, т.е. нечто очень похожее на базу данных. Поэтому применение понятий баз данных часто оказывается полезным при объектно-ориентированном анализе и объектно-ориентированном проектировании прикладных программных систем.

**13. Зависимости между классами (объектами).**  
С каждым объектом связана структура данных, полями которой являются атрибуты этого объекта и указатели функций (фрагментов кода), реализующих операции этого объекта (отметим, что указатели функций в результате оптимизации кода обычно заменяются на обращения к этим функциям). Таким образом, объект - это некоторая структура данных, тип которой соответствует классу этого объекта.

Между объектами можно устанавливать *зависимости* по данным. Эти *зависимости* выражают *связи* или *отношения* между классами указанных объектов. Примеры таких зависимостей изображены на рисунке 2.6 (первые две зависимости - бинарные, третья зависимость - тренарная). Зависимость изображается линией, соединяющей классы над которой надписано имя этой зависимости, или указаны роли объектов (классов) в этой зависимости (указание ролей - наиболее удобный способ идентификации зависимости).



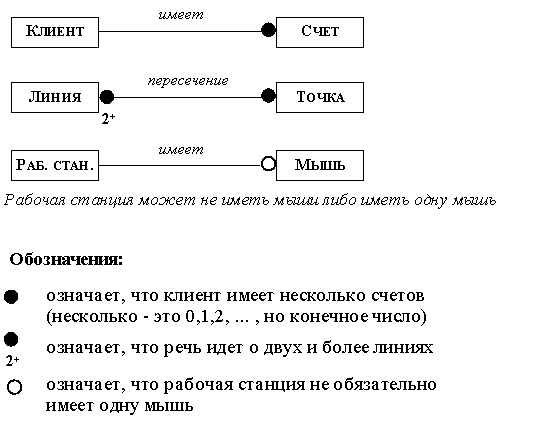
*Рис. 2.6. Зависимости между классами*

Зависимости между классами являются двусторонними: все классы в зависимости равноправны. Это так даже в тех случаях, когда имя зависимости как бы вносит направление в эту зависимость. Так, в первом примере на рисунке 2.6 имя зависимости имеет\_столицу предполагает, что зависимость направлена от класса страна к классу город (двусторонность зависимости вроде бы пропала); но следует иметь в виду, что эта зависимость двусторонняя в том смысле, что одновременно с ней существует и обратная зависимость является\_столицей. Точно таким же образом, во втором примере на рисунке 2.6 можно рассматривать пару зависимостей владеет-принадлежит. Подобных недоразумений можно избежать, если идентифицировать зависимости не по именам, а по наименованиям ролей классов, составляющих зависимость.

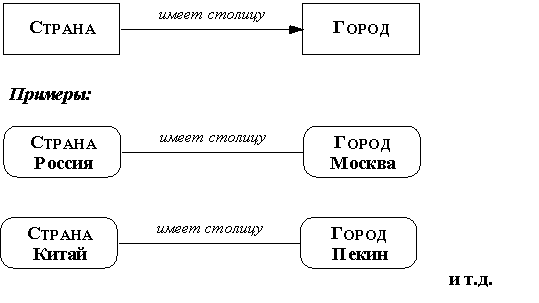
В языках программирования зависимости между классами (объектами) обычно реализуются с помощью ссылок (указателей) из одного класса (объекта) на другой. Представление зависимостей с помощью ссылок обнаруживает тот факт, что зависимость является свойством пары классов, а не какого-либо одного из них, т.е. зависимость - это отношение. Отметим, что хотя зависимости между объектами двунаправлены, их не обязательно реализовать в программах как двунаправленные, оставляя ссылки лишь в тех классах, где это необходимо для программы.

Дальнейшие примеры зависимостей между классами приведены на рисунке 2.7. Первый пример показывает зависимость между клиентом банка и его счетами. Клиент банка может иметь одновременно несколько счетов в этом банке, либо вовсе не иметь счета (когда он впервые становится клиентом банка). Таким образом, нужно изобразить зависимость между клиентом и несколькими счетами, что и сделано на рисунке 2.7. Второй пример показывает зависимость между пересекающимися кривыми (в частности, прямыми) линиями. Можно рассматривать 2, 3, и более таких линий, причем они могут иметь несколько точек пересечения. Наконец, третий пример показывает необязательную (optional) зависимость: компьютер может иметь, а может и не иметь мышь.

Зависимостям между классами соответствуют зависимости между объектами этих классов. На рисунке 2.8 показаны зависимости между объектами для первого примера рисунка 2.6; на рисунке 2.9 показаны зависимости между объектами для примеров, изображенных на рисунке 2.7.



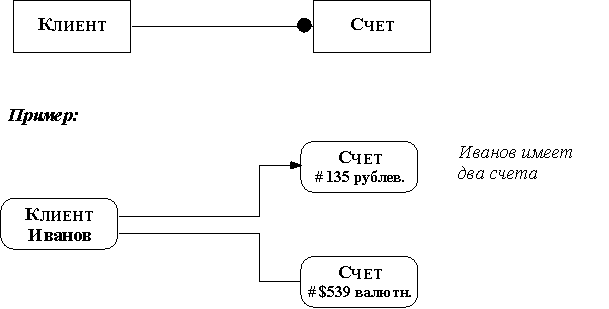
*Рис. 2.7. Дальнейшие примеры зависимостей. Обозначения*

**

*Рис. 2.8. Зависимости между объектами*

Отметим, что при изображении зависимостей между объектами мы, как правило, знаем количество объектов и не нуждаемся в таких обозначениях как "несколько", "два и более", "не обязательно".

При проектировании системы удобнее оперировать не объектами, а классами.



*Рис. 2.9. Более сложные зависимости между объектами*

Понятие зависимости перенесено в объектно-ориентированную технологию проектирования программных систем из технологии проектирования (и моделирования) баз данных, где зависимости используются с давних пор. Языки программирования, как правило, не поддерживают явного описания зависимостей. Тем не менее описание зависимостей очень полезно при разработке программных систем. Технология OMT использует зависимости при интерпретации диаграмм, описывающих систему.

**14. 1. Определение классов.**

Анализ внешних требований к проектируемой прикладной системе позволяет определить объекты и классы объектов, связанные с прикладной проблемой, которую должна решать эта система. Все классы должны быть осмыслены в рассматриваемой прикладной области; классов, связанных с компьютерной реализацией, как например список, стэк и т.п. на этом этапе вводить не следует.

Начать нужно с выделения возможных классов из письменной постановки прикладной задачи (технического задания и другой документации, предоставленной заказчиком). Следует иметь в виду, что это очень сложный и ответственный этап разработки, так как от него во многом зависит дальнейшая судьба проекта.

При определении возможных классов нужно постараться выделить как можно больше классов, выписывая имя каждого класса, который приходит на ум. В частности, каждому существительному, встречающемуся в предварительной постановке задачи, может соответствовать класс. Поэтому при выделении возможных классов каждому такому существительному обычно сопоставляется возможный класс.

Далее список возможных классов должен быть проанализирован с целью исключения из него *ненужных* классов. Такими классами являются:

* *избыточные классы:* если два или несколько классов выражают одинаковую информацию, следует сохранить только один из них;
* *нерелевантные* (не имеющие прямого отношения к проблеме) *классы*: для каждого имени возможного класса оценивается, насколько он необходим в будущей системе (оценить это часто бывает весьма непросто); нерелевантные классы исключаются;
* *нечетко определенные* (с точки зрения рассматриваемой проблемы) *классы* (см. примеры таких классов в п. [2.3.1](http://citforum.ru/programming/oop_rsis/glava2_3_1.shtml));
* *атрибуты*: некоторым существительным больше соответствуют не классы, а атрибуты; такие существительные, как правило, описывают свойства объектов (например, имя, возраст, вес, адрес и т.п.);
* *операции*: некоторым существительным больше соответствуют не классы, а имена операций (например, телефонный\_вызов вряд ли означает какой-либо класс);
* *роли*: некоторые существительные определяют имена ролей в объектной модели (например, владелец, водитель, начальник, служащий; все эти имена связаны с ролями в различных зависимостях объектов класса человек);
* *реализационные конструкции*: именам, больше связанным с программированием и компьютерной аппаратурой, не следует на данном этапе сопоставлять классов, так как они не отражают особенностей проектируемой прикладной системы; примеры таких имен: подпрограмма, процесс, алгоритм, прерывание и т.п.

После исключения имен всех ненужных (лишних) возможных классов будет получен предварительный список классов, составляющих проектируемую систему.

**14.2. Подготовка словаря данных.**

Приведем часть словаря данных, содержащую определения классов, используемых в проекте.

ATM (банкомат) - терминал, который дает возможность клиенту осуществлять свою собственную проводку, используя для идентификации свою карточку. ATM (банкомат) взаимодействует с клиентом, чтобы получить необходимую информацию для проводки, посылает информацию для проводки центральному\_компьютеру, чтобы он проверил ее и в дальнейшем использовал при выполнении проводки и выдает деньги и квитанцию клиенту. Предполагается, что ATM (банкомату) не требуется работать независимо от сети.

Банк - финансовая организация, которая содержит счета своих клиентов и выпускает карточки, санкционирующие доступ к счетам через сеть ATM (банкоматов).

Карточка - пластиковая карточка, врученная банком своему клиенту, которая санкционирует доступ к счетам через сеть ATM (банкоматов). Каждая карточка содержит код банка и номер карточки, закодированные в соответствии с национальными стандартами на банковские карточки. Код\_банка однозначно идентифицирует банк внутри консорциума. Номер\_карточки определяет счета, к которым карточка имеет доступ. Карточка не обязательно обеспечивает доступ ко всем счетам клиента. Каждой карточкой может владеть только один клиент, но у нее может существовать несколько копий, так что необходимо рассмотреть возможность одновременного использования одной и той же карточки с разных ATM (банкоматов).

Кассир - служащий банка, который имеет право осуществлять проводки с кассовых\_ терминалов, а также принимать и выдавать деньги и чеки клиентам. Проводки, деньги и чеки, с которыми работает каждый кассир должны протоколироваться и правильно учитываться.

Кассовый\_терминал - терминал, с которого кассир осуществляет проводки для клиентов. Когда кассир принимает и выдает деньги и чеки, кассовый\_терминал печатает квитанции. Кассовый\_терминал взаимодействует с компьютером\_банка, чтобы проверить и выполнить проводку.

Клиент - держатель одного или нескольких счетов в банке. Клиент может состоять из одного или нескольких лиц, или организаций. То же самое лицо, держащее счет и в другом банке рассматривается как другой клиент.

Компьютер\_банка - компьютер, принадлежащий банку, который взаимодействует с сетью ATM (банкоматов) и собственными кассовыми\_терминалами банка. Банк может иметь свою внутреннюю компьютерную сеть для обработки счетов, но здесь мы рассматриваем только тот компьютер\_банка, который взаимодействует с сетью ATM.

Консорциум - объединение банков, которое обеспечивает работу сети ATM (банкоматов). Сеть передает в консорциум проводки банков.

Проводка - единичный интегрированный запрос на выполнение некоторой последовательности операций над счетами одного клиента. Было сделано предположение, что ATM (банкоматы) только выдают деньги, однако для них не следует исключать возможности печати чеков или приема денег и чеков. Хотелось бы также обеспечить гибкость системы, которая в дальнейшем обеспечит возможность одновременной обработки счетов разных клиентов, хотя пока этого не требуется. Различные операции должны быть правильно сбалансированы.

Счет - единичный банковский счет, над которым выполняются проводки. Счета могут быть различных типов; клиент может иметь несколько счетов.

Центральный\_компьютер - компьютер, принадлежащий консорциуму, который распределяет проводки и их результаты между ATM (банкоматами) и компьютерами\_банков. Центральный\_компьютер проверяет коды банков, но не выполняет проводок самостоятельно.

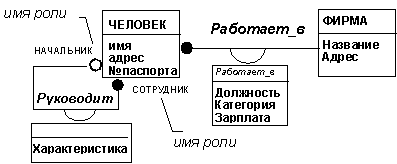
**15. Атрибуты зависимостей.**

Зависимости, как и классы, могут иметь атрибуты: например, при организации доступа пользователя к файлу разрешение\_на\_доступ является атрибутом зависимости доступен: см. рисунок 2.10, на котором атрибут зависимости обозначается прямоугольником, связанным дугой с прямой, изображающей зависимость. Такое обозначение атрибутов зависимостей принято в технологии OMT. Отметим, что разрешение на доступ связано как с пользователем, так и с файлом, и не может быть атрибутом ни пользователя, ни файла в отдельности.



*Рис. 2.10. Пример атрибута зависимости*

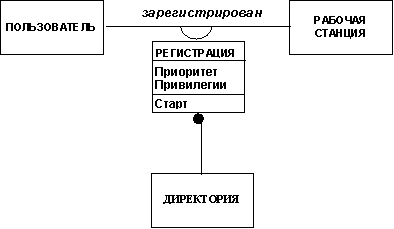
Еще один пример зависимостей, имеющих атрибуты, показан на рисунке 2.11. Из примера видно, что зависимость может иметь несколько атрибутов. Кроме того, на рисунке 2.11 указаны роли различных объектов в зависимости (подробнее см. ниже). Зависимость руководит на этом рисунке удобнее именовать как начальник-сотрудник.



*Рис. 2.11. Атрибуты двух зависимостей между одним и многими*

Иногда зависимости, имеющие много атрибутов, представляют с помощью классов. Такие зависимости в базах данных представляются временными таблицами, организуемыми в процессе обращения с базой данных. Пример зависимости, представленной через класс, показана на рисунке 2.12, на котором представлена информация о регистрации пользователей на рабочих станциях.

Пользователь может быть зарегистрирован на нескольких рабочих станциях, каждая регистрация содержит приоритет пользователя и его привилегии доступа (атрибуты зависимости). Пользователь может иметь свою директорию для каждой зарегистрированной рабочей станции, но одна и та же директория может принадлежать одновременно нескольким пользователям или нескольким рабочим станциям.



*Рис. 2.12. Представление зависимости в виде класса*

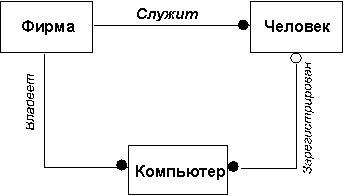
**16.1. Определение зависимостей.**

На следующем этапе построения объектной модели определяются зависимости между классами. Прежде всего из классов исключаются атрибуты, являющиеся явными ссылками на другие классы; такие атрибуты заменяются зависимостями. Смысл такой замены в том, что зависимости представляют собой абстракцию того же уровня, что и классы, и потому не оказывают непосредственного влияния на будущую реализацию (ссылка на класс лишь один из способов реализации зависимостей).

Аналогично тому, как имена возможных классов получались из существительных, встречающихся в предварительной постановке прикладной задачи, имена возможных зависимостей могут быть получены из *глаголов* или *глагольных оборотов*, встречающихся в указанном документе. Так обычно описываются: физическое положение (следует\_за, является\_частью, содержится\_в), направленное действие (приводит\_в\_движение), общение (разговаривает\_с), принадлежность (имеет, является\_частью) и т.п. Пример выделения явных и неявных глагольных оборотов из предварительной постановки конкретной прикладной задачи рассмотрен в п. [2.3.3.](http://citforum.ru/programming/oop_rsis/glava2_3_3.shtml)

Затем следует убрать ненужные или неправильные зависимости, используя следующие критерии:

* *зависимости между исключенными классами* должны быть исключены, либо переформулированы в терминах оставшихся классов (см. пример в п. [2.3.3](http://citforum.ru/programming/oop_rsis/glava2_3_3.shtml));
* *нерелевантные зависимости* и зависимости, связанные с реализацией, должны быть исключены (см. пример в п. [2.3.3](http://citforum.ru/programming/oop_rsis/glava2_3_3.shtml));
* *действия:* зависимость должна описывать структурные свойства прикладной области, а не малосущественные события (см. примеры в п. [2.3.3](http://citforum.ru/programming/oop_rsis/glava2_3_3.shtml));
* *тренарные зависимости:* большую часть зависимостей между тремя или большим числом классов можно разложить на несколько бинарных зависимостей, используя в случае необходимости квалификаторы (см. примеры в п. [2.3.3](http://citforum.ru/programming/oop_rsis/glava2_3_3.shtml)); в некоторых (очень редких) случаях такое разложение осуществить не удается; например, тренарная зависимость "Профессор читает курс в аудитории 628" не может быть разложена на бинарные без потери информации;
* *производные зависимости:* нужно исключать зависимости, которые можно выразить через другие зависимости, так как они избыточны (см. пример в п. [2.3.3](http://citforum.ru/programming/oop_rsis/glava2_3_3.shtml)); при исключении избыточных (производных) зависимостей нужно быть особенно осторожным, так как не все дублирующие одна другую зависимости между классами избыточны; в некоторых случаях другие зависимости позволяют установить только существование еще одной производной зависимости, но не позволяют установить кратность этой зависимости; например, в случае, представленном на рисунке 2.36, фирма имеет много служащих и владеет многими компьютерами; каждому служащему предоставлено для персонального использования несколько компьютеров, кроме того, имеются компьютеры общего пользования; кратность зависимости предоставлен\_для\_использования не может быть выведена из зависимостей служит и владеет; хотя производные зависимости и не добавляют новой информации, они часто бывают удобны; в этих случаях их можно указывать на диаграмме, пометив косой чертой.



*Рис. 2.36. Неизбыточные зависимости*

Удалив избыточные зависимости, нужно уточнить семантику оставшихся зависимостей следующим образом:

* *неверно названные зависимости:* их следует переименовать, чтобы смысл их стал понятен (см. пример в п. [2.3.3](http://citforum.ru/programming/oop_rsis/glava2_3_3.shtml));
* *имена ролей:* нужно добавить имена ролей там, где это необходимо; имя роли описывает роль, которую играет соответствующий класс в данной зависимости с точки зрения другого класса, участвующего в этой зависимости; если имя роли ясно из имени класса, его можно не указывать (см. пример в п. [2.3.3](http://citforum.ru/programming/oop_rsis/glava2_3_3.shtml));
* *квалификаторы:* добавляя квалификаторы там, где это необходимо, мы вносим элементы контекста, что позволяет добиться однозначной идентификации объектов; квалификаторы позволяют также упростить некоторые зависимости, понизив их кратность;
* *кратность:* необходимо добавить обозначения кратности зависимостей; при этом следует помнить, что кратность зависимостей может меняться в процессе дальнейшего анализа требований к системе;
* *неучтенные зависимости* должны быть выявлены и добавлены в модель.

**16.2. Уточнение атрибутов.**

На следующем этапе уточняется система атрибутов: корректируются атрибуты классов, вводятся, в случае необходимости, новые атрибуты. Атрибуты выражают свойства объектов рассматриваемого класса, либо определяют их текущее состояние.

Атрибуты обычно соответствуют существительным; например цвет\_автомобиля (свойство объекта), позиция\_курсора (состояние объекта). Атрибуты, как правило, слабо влияют на структуру объектной модели.

Не следует стремиться определить как можно больше атрибутов: большое количество атрибутов усложняет модель, затрудняет понимание проблемы. Необходимо вводить только те атрибуты, которые имеют отношение к проектируемой прикладной системе, опуская случайные, малосущественные и производные атрибуты.

Наряду с атрибутами объектов необходимо ввести и атрибуты зависимостей между классами (связей между объектами).

При уточнении атрибутов руководствуются следующими критериями:

* *Замена атрибутов на объекты*. Если наличие некоторой сущности важнее, чем ее значение, то это объект, если важнее значение, то это атрибут: например, начальник - это объект (неважно, кто именно начальник, главное, чтобы кто-то им был), зарплата - это атрибут (ее значение весьма существенно); город - всегда объект, хотя в некоторых случаях может показаться, что это атрибут (например, город как часть адреса фирмы); в тех случаях, когда нужно, чтобы город был атрибутом, следует определить зависимость (скажем, находится) между классами фирма и город.
* *Квалификаторы*. Если значение атрибута зависит от конкретного контекста, его следует сделать квалификатором (см. примеры в п. [2.3.4](http://citforum.ru/programming/oop_rsis/glava2_3_4.shtml)).
* *Имена*. Именам обычно лучше соответствуют квалификаторы, чем атрибуты объектов; во всех случаях, когда имя позволяет сделать выбор из объектов некоторого множества, его следует сделать квалификатором (см. примеры в п. [2.3.4](http://citforum.ru/programming/oop_rsis/glava2_3_4.shtml)).
* *Идентификаторы*. Идентификаторы объектов связаны с их реализацией. На ранних стадиях проектирования их не следует рассматривать в качестве атрибутов.
* *Атрибуты связей*. Если некоторое свойство характеризует не объект сам по себе, а его связь с другим объектом (объектами), то это атрибут связи, а не атрибут объекта.
* *Внутренние значения*. Атрибуты, определяющие лишь внутреннее состояние объекта, незаметное вне объекта, следует исключить из рассмотрения.
* *Несущественные детали*. Атрибуты, не влияющие на выполнение большей части операций, рекомендуется опустить.

**17. Объектная модель системы. Имена ролей, квалификаторы.**

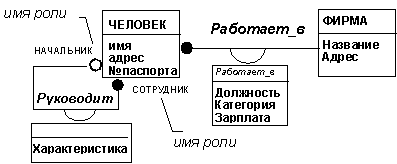
#### Объектная модель системы

Объектная модель описывает структуру объектов, составляющих систему, их атрибуты, операции, взаимосвязи с другими объектами. В объектной модели должны быть отражены те понятия и объекты реального мира, которые важны для разрабатываемой системы. В объектной модели отражается прагматика разрабатываемой системы, что выражается в использовании терминологии прикладной области, связанной с использованием разрабатываемой системы.

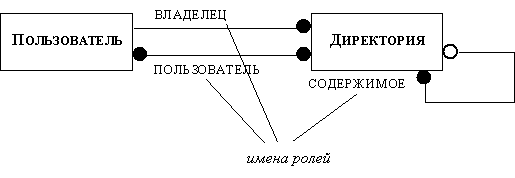
Имена ролей, квалификаторы

Роль определяет одну сторону зависимости. В бинарной зависимости определены две роли. Имя роли однозначно определяет одну сторону зависимости. Роли дают возможность рассматривать бинарную зависимость как связь между объектом и множеством зависимых объектов: каждая роль является обозначением объекта или множества объектов, связанных зависимостью с объектом на другом конце зависимости. Имя роли можно рассматривать как производный атрибут, множеством значений которого является множество связанных с этой ролью объектов. В бинарной зависимости пара имен ролей может использоваться для идентификации этой зависимости.

На рисунке [2.11](http://citforum.ck.ua/programming/oop_rsis/glava2_1_5.shtml#r2_11) имена начальник и сотрудник в зависимости руководит - это имена ролей; как уже было отмечено, эту зависимость удобнее назвать начальник-сотрудник.



Еще один пример имен ролей показан на рисунке 2.13.

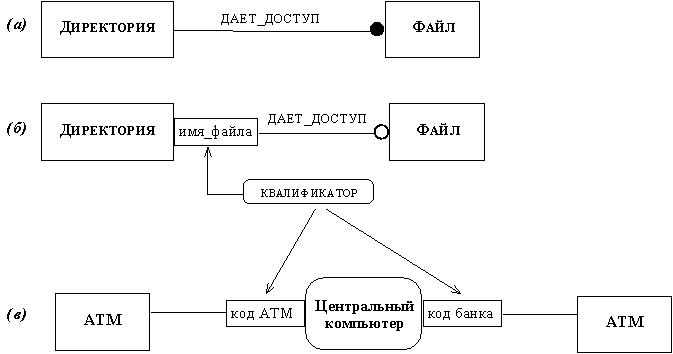


*Рис. 2.13. Имена ролей*

Пользователь может быть либо владельцем, либо зарегистрированным пользователем директории; директория может содержать в себе другие директории.

Имена ролей следует обязательно указывать в тех случаях, когда зависимость устанавливается между объектами одного и того же класса (как в случаях, показанных на рисунках [2.11](http://citforum.ck.ua/programming/oop_rsis/glava2_1_5.shtml#r2_11) и 2.13). Имена ролей должны быть уникальны, так как они используются для различения объектов, участвующих в зависимости.

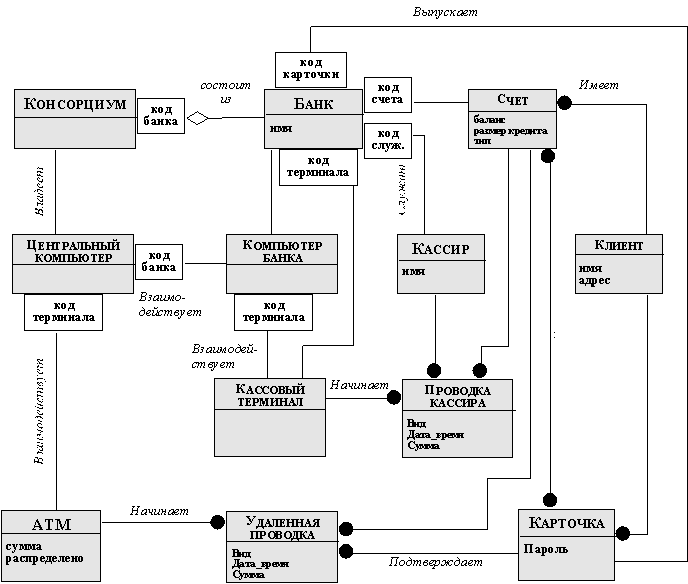
*Квалификатором* называется некоторый атрибут, который позволяет снизить эффективную кратность зависимости. Квалификаторы применяются в зависимостях типов "один ко многим" или "много ко многим". Так в примере, показанном на рисунке 2.14, использование квалификатора имя файла позволяет привести зависимость дает доступ от вида, приведенного на рисунке 2.14(а), к виду, показанному на рисунке 2.14(б), сократив число зависимых объектов до одного. Еще один пример использования квалификатора показан на рисунке 2.14(в): использование квалификаторов и здесь позволяет сократить кратность зависимости до одного объекта. Квалификаторы указываются на схемах в прямоугольничках, пририсованных к прямоугольнику, изображающему соответствующий класс.

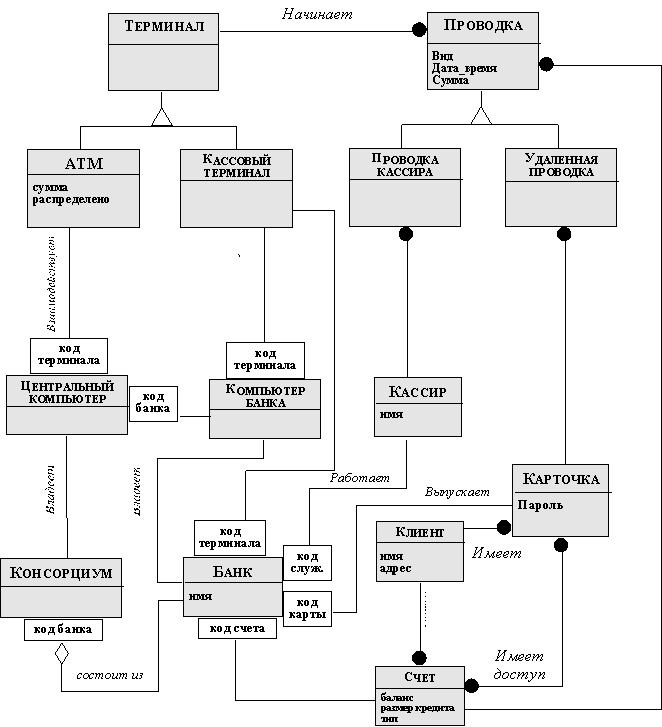


*Рис. 2.14. Использование квалификаторов*

Рисунок 2.14(в) может быть проинтерпретирован следующим образом: центральныйкомпьютер + код ATM определяют конкретную ATM (отметим, что код ATM - имя одного из атрибутов класса ATM, а не класса центральный компьютер); аналогично центральный компьютер + код банка определяют конкретный компьютер банка. Использование квалификаторов повышает точность описания семантики и наглядность описания зависимостей.

#### 18.Построение объектной модели. Организация системы классов, используя наследование.

 Организация системы классов с использованием наследования  
  
В рассматриваемом примере естественно определить суперклассы для объектов, определяющих различные терминалы: кассовый терминал и БМ (банкомат), и для объектов, определяющих проводки: проводка кассира и удаленная проводка (с банкомата).   
  
Внеся соответствующие изменения, получим объектную диаграмму, представленную на рисунке 1.4.   
  
  
  
Рисунок 1.3 - Объектная диаграмма для банковской сети после уточнения атрибутов и добавления квалификаторов

  
  
Рисунок 1.4 - Объектная диаграмма для банковской с учетом наследования

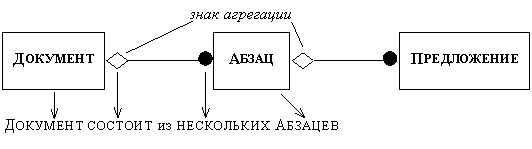
19.Объектная модель системы Агрегация.

#### Объектная модель системы

Объектная модель описывает структуру объектов, составляющих систему, их атрибуты, операции, взаимосвязи с другими объектами. В объектной модели должны быть отражены те понятия и объекты реального мира, которые важны для разрабатываемой системы. В объектной модели отражается прежде всего прагматика разрабатываемой системы, что выражается в использовании терминологии прикладной области, связанной с использованием разрабатываемой системы.

#### *Агрегация*

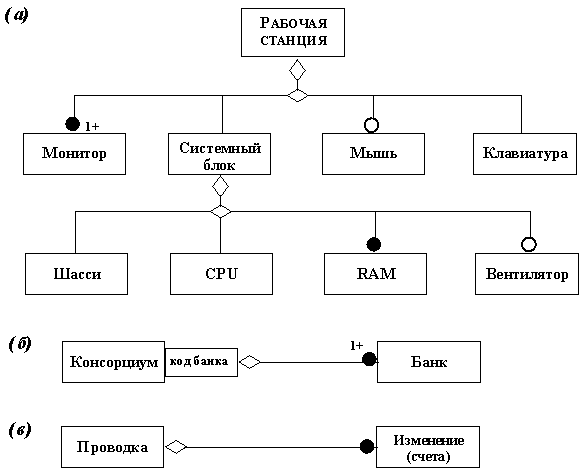
*Агрегация* - это зависимость между классом составных объектов и классами, представляющими компоненты этих объектов (отношение "целое"-"часть"). Агрегация обозначается ромбиком: на рисунке 2.15 приведен пример агрегации; этот пример интерпретируется следующим образом: документ состоит из нескольких (нуля, или более) абзацев; каждый абзац состоит из нескольких (нуля, или более) предложений.



*Рис. 2.15. Агрегация*

Наиболее важным свойством отношения агрегации является его *транзитивность* (если A есть часть B, а B есть часть C, то A есть часть C): так, из рисунка 2.15 можно заключить, что документ состоит из нескольких (нуля, или более) предложений. Легко видеть, что отношение агрегации *антисимметрично* (если A есть часть B, то B не есть часть A). Отметим также, что часть свойств целого может быть перенесена и на его части, возможно, с несущественными изменениями (например, контекст каждого оператора некоторой функции совпадает с внутренним контекстом всей функции).

Дальнейшие примеры агрегации показаны на рисунке 2.16. Отметим, что обе агрегации, показанные на рисунке 2.16(а), следует рассматривать не как зависимости между пятерками классов, а как четверки зависимостей между парами классов. Только при таком рассмотрении можно говорить о транзитивности и антисимметричности отношения агрегации.



*Рис. 2.16. Примеры агрегации*

**20. Пример объектной модели. Дальнейшее исследование и усовершенствование модели.**

***Дальнейшее исследование и усовершенствование модели***

Лишь в очень редких случаях построенная объектная модель сразу же оказывается корректной. Модель должна быть исследована и отлажена. Некоторые ошибки могут быть найдены при исследовании модели без компьютера, другие - при ее интерпретации совместно с динамической и функциональной моделями на компьютере (эти модели строятся после того, как объектная модель уже построена).

Здесь мы рассмотрим приемы бескомпьютерного поиска и исправления ошибок в объектной модели. В их основе лежат внешние признаки, по которым можно находить ошибки в модели; эти признаки могут быть объединены в следующие группы.

***Признаки пропущенного объекта (класса):***

* несимметричности связей и обобщений (наследований); для исправления ошибки необходимо добавить пропущенные классы;
* несоответствие атрибутов и операций у класса; для исправления ошибки необходимо расщепить класс на несколько других классов, так чтобы атрибуты и операции новых классов соответствовали друг другу;
* обнаружена операция, не имеющая удовлетворительного целевого класса; для исправления ошибки необходимо добавить пропущенный целевой класс;
* обнаружено несколько зависимостей с одинаковыми именами и назначением; для исправления ошибки необходимо сделать обобщение и добавить пропущенный суперкласс.

***Признаки ненужного (лишнего) класса:***

* нехватка атрибутов, операций и зависимостей у некоторого класса; для исправления ошибки необходимо подумать, не следует ли исключить такой класс.

***Признаки пропущенных зависимостей:***

* отсутствуют пути доступа к операциям; для исправления ошибки необходимо добавить новые зависимости, обеспечивающие возможности обслуживания соответствующих запросов.

***Признаки ненужных (лишних) зависимостей:***

* избыточная информация в зависимостях; для исправления ошибки необходимо исключить зависимости, не добавляющие новой информации, или пометить их как производные зависимости;
* не хватает операций, пересекающих зависимость; для исправления ошибки необходимо подумать, не следует ли исключить такую зависимость.

***Признаки неправильного размещения зависимостей:***

* имена ролей слишком широки или слишком узки для их классов; для исправления ошибки необходимо переместить зависимость вверх или вниз по иерархии классов.

***Признаки неправильного размещения атрибутов:***

* нет необходимости доступа к объекту по значениям одного из его атрибутов; для исправления ошибки необходимо рассмотреть нужно ли ввести квалифицированную зависимость.

**21. *Первая фаза – анализ требований***

Как известно, проектирование прикладной программной системы начинается с анализа требований, которым она должна будет удовлетворять. Такой анализ проводится с целью понять назначение и условия эксплуатации системы настолько, чтобы суметь составить ее предварительный проект.

При объектно-ориентированном подходе анализ требований к системе сводится к разработке моделей этой системы. *Моделью системы* (или какого-либо другого объекта или явления) мы называем формальное описание системы, в котором выделены основные объекты, составляющие систему, и отношения между этими объектами. Построение моделей - широко распространенный способ изучения сложных объектов и явлений. В модели опущены многочисленные детали, усложняющие понимание. Моделирование широко распространено и в науке, и в технике.

Модели помогают:

* проверить работоспособность разрабатываемой системы на ранних этапах ее разработки;
* общаться с заказчиком системы, уточняя его требования к системе;
* вносить (в случае необходимости) изменения в проект системы (как в начале ее проектирования, так и на других фазах ее жизненного цикла).

В настоящее время существует несколько технологий объектно-ориентированной разработки прикладных программных систем, в основе которых лежит построение и интерпретация на компьютере моделей этих систем. Мы подробно ознакомимся с одной из таких технологий - *OMT (Object Modeling Techniques).* Эта технология оказала большое влияние на других разработчиков объектно-ориентированных технологий, а книга, в которой она описана, является одной из наиболее часто цитируемых книг по данному направлению. Более того, система обозначений (графический язык) для описания моделей, предложенная в этой книге, широко применяется в других технологиях и в статьях по объектно-ориентированной разработке программных систем.

В технологии OMT проектируемая программная система представляется в виде трех взаимосвязанных моделей:

* объектной модели, которая представляет статические, структурные аспекты системы, в основном связанные с данными;
* динамической модели, которая описывает работу отдельных частей системы;
* функциональной модели, в которой рассматривается взаимодействие отдельных частей системы (как по данным, так и по управлению) в процессе ее работы.

Эти три вида моделей позволяют получить три взаимно-ортогональных представления системы в одной системе обозначений. Совокупность моделей системы может быть проинтерпретирована на компьютере (с помощью инструментального программного обеспечения), что позволяет продемонстрировать заказчику характер работы с будущей системой и существенно упрощает согласование предварительного проекта системы.

Модели, разработанные и отлаженные на первой фазе жизненного цикла системы, продолжают использоваться на всех последующих его фазах, облегчая программирование системы, ее отладку и тестирование, сопровождение и дальнейшую модификацию.

Как будет показано в дальнейшем, модели системы не связаны с языком программирования, на котором будет реализована система.

**22. Структу́рное программи́рование** — методология разработки [программного обеспечения](http://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B5_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D0%B5), в основе которой лежит представление программы в виде иерархической структуры [блоков](http://ru.wikipedia.org/wiki/%D0%91%D0%BB%D0%BE%D0%BA_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)). Предложена в 70-х годах XX века [Э. Дейкстрой](http://ru.wikipedia.org/wiki/%D0%94%D0%B5%D0%B9%D0%BA%D1%81%D1%82%D1%80%D0%B0,_%D0%AD%D0%B4%D1%81%D0%B3%D0%B5%D1%80_%D0%92%D0%B0%D0%B9%D0%B1), разработана и дополнена [Н. Виртом](http://ru.wikipedia.org/wiki/%D0%92%D0%B8%D1%80%D1%82,_%D0%9D%D0%B8%D0%BA%D0%BB%D0%B0%D1%83%D1%81).

В соответствии с данной методологией

1. Любая программа представляет собой структуру, построенную из трёх типов базовых конструкций:
   * **последовательное исполнение** — однократное выполнение операций в том порядке, в котором они записаны в тексте программы;
   * **ветвление** — однократное выполнение одной из двух или более операций, в зависимости от выполнения некоторого заданного условия;
   * **цикл** — многократное исполнение одной и той же операции до тех пор, пока выполняется некоторое заданное условие (условие продолжения цикла).

В программе базовые конструкции могут быть вложены друг в друга произвольным образом, но никаких других средств управления последовательностью выполнения операций не предусматривается.

1. Повторяющиеся фрагменты программы (либо не повторяющиеся, но представляющие собой логически целостные вычислительные блоки) могут оформляться в виде т. н. [подпрограмм](http://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%B4%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B0) ([процедур](http://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D0%B4%D1%83%D1%80%D0%B0) или [функций](http://ru.wikipedia.org/wiki/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5))). В этом случае в тексте основной программы, вместо помещённого в подпрограмму фрагмента, вставляется инструкция**вызова подпрограммы**. При выполнении такой инструкции выполняется вызванная подпрограмма, после чего исполнение программы продолжается с инструкции, следующей за командой вызова подпрограммы.
2. Разработка программы ведётся пошагово, методом «сверху вниз».

Сначала пишется текст основной программы, в котором, вместо каждого связного логического фрагмента текста, вставляется вызов подпрограммы, которая будет выполнять этот фрагмент. Вместо настоящих, работающих подпрограмм, в программу вставляются «заглушки», которые ничего не делают. Полученная программа проверяется и отлаживается. После того, как [программист](http://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%81%D1%82) убедится, что подпрограммы вызываются в правильной последовательности (то есть общая структура программы верна), подпрограммы-заглушки последовательно заменяются на реально работающие, причём разработка каждой подпрограммы ведётся тем же методом, что и основной программы. Разработка заканчивается тогда, когда не останется ни одной «затычки», которая не была бы удалена. Такая последовательность гарантирует, что на каждом этапе разработки программист одновременно имеет дело с обозримым и понятным ему множеством фрагментов, и может быть уверен, что общая структура всех более высоких уровней программы верна. При сопровождении и внесении изменений в программу выясняется, в какие именно процедуры нужно внести изменения, и они вносятся, не затрагивая части программы, непосредственно не связанные с ними. Это позволяет гарантировать, что при внесении изменений и исправлении ошибок не выйдет из строя какая-то часть программы, находящаяся в данный момент вне зоны внимания программиста.

**23. Компонентно ориентированное программирование** ([англ.](http://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *component-oriented programming, COP*) — [парадигма программирования](http://ru.wikipedia.org/wiki/%D0%9F%D0%B0%D1%80%D0%B0%D0%B4%D0%B8%D0%B3%D0%BC%D0%B0_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F), ключевой фигурой которой является[компонент](http://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D0%BE%D0%BD%D0%B5%D0%BD%D1%82_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)).

Компонентно ориентированное программирование включает в себя набор ограничений, налагаемых на механизм [объектно ориентированного программирования](http://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) (далее ООП). Это было сделано для повышения надежности больших программных комплексов. Проблема [хрупких базовых классов](http://ru.wikipedia.org/wiki/%D0%A5%D1%80%D1%83%D0%BF%D0%BA%D0%B8%D0%B9_%D0%B1%D0%B0%D0%B7%D0%BE%D0%B2%D1%8B%D0%B9_%D0%BA%D0%BB%D0%B0%D1%81%D1%81) возникает при изменении реализации типа-[предка](http://ru.wikipedia.org/wiki/%D0%A1%D1%83%D0%BF%D0%B5%D1%80%D0%BA%D0%BB%D0%B0%D1%81%D1%81_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)). В этом случае в классическом ООП возможна ситуация, когда изменить реализацию типа-предка невозможно, не нарушив корректность функционирования типов-[потомков](http://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%B4%D0%BA%D0%BB%D0%B0%D1%81%D1%81_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) (даже при прежних [интерфейсах](http://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81_(%D0%BE%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) его [методов](http://ru.wikipedia.org/wiki/%D0%9C%D0%B5%D1%82%D0%BE%D0%B4_(%D1%8F%D0%B7%D1%8B%D0%BA%D0%B8_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)))[[1]](http://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D0%BE%D0%BD%D0%B5%D0%BD%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5#cite_note-1).

Ситуация, когда возникает новая парадигма программирования в качестве ограничения прежней, уже была, когда возникло [структурное программирование](http://ru.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) для использования структур управления, исключающих неупорядоченные переходы управления с помощью оператора [GOTO](http://ru.wikipedia.org/wiki/GOTO), затрудняющие анализ алгоритма программы.

**Прототипное программирование** — [стиль](http://ru.wikipedia.org/wiki/%D0%9F%D0%B0%D1%80%D0%B0%D0%B4%D0%B8%D0%B3%D0%BC%D0%B0_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F) [объектно-ориентированного программирования](http://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5), при котором отсутствует понятие [класса](http://ru.wikipedia.org/wiki/%D0%9A%D0%BB%D0%B0%D1%81%D1%81_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)), а [наследование](http://ru.wikipedia.org/wiki/%D0%9D%D0%B0%D1%81%D0%BB%D0%B5%D0%B4%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) производится путём[клонирования](http://ru.wikipedia.org/wiki/%D0%9A%D0%BB%D0%BE%D0%BD%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)) существующего [экземпляра объекта](http://ru.wikipedia.org/wiki/%D0%AD%D0%BA%D0%B7%D0%B5%D0%BC%D0%BF%D0%BB%D1%8F%D1%80_%D0%BE%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%B0) — **прототипа**

## Сравнение с [класс-ориентированным подходом](http://ru.wikipedia.org/wiki/%D0%9A%D0%BB%D0%B0%D1%81%D1%81-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)

В языках, основанных на понятии «класс», все объекты разделены на два основных типа — [*классы*](http://ru.wikipedia.org/wiki/%D0%9A%D0%BB%D0%B0%D1%81%D1%81_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) и [*экземпляры*](http://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)). *Класс* определяет структуру и функциональность (*поведение*), одинаковую для всех экземпляров данного класса. *Экземпляр* является носителем данных — то есть обладает [*состоянием*](http://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D1%81%D1%82%D0%BE%D1%8F%D0%BD%D0%B8%D0%B5), меняющимся в соответствии с поведением, заданным классом.

Сторонники прототипного программирования часто утверждают, что языки, основанные на классах, приводят к излишней концентрации на [таксономии](http://ru.wikipedia.org/wiki/%D0%A2%D0%B0%D0%BA%D1%81%D0%BE%D0%BD%D0%BE%D0%BC%D0%B8%D1%8F) классов и на отношениях между ними. В противоположность этому, прототипирование заостряет внимание на поведении некоторого (небольшого) количества «образцов», которые затем классифицируются как «базовые» объекты и используются для создания других объектов. Многие прототип-ориентированные системы поддерживают изменение прототипов во время выполнения программы, тогда как лишь небольшая часть класс-ориентированных систем (например, [Smalltalk](http://ru.wikipedia.org/wiki/Smalltalk), [Ruby](http://ru.wikipedia.org/wiki/Ruby)) позволяет динамически изменять классы.

Хотя подавляющее большинство прототип-ориентированных систем основаны на интерпретируемых языках с динамической типизацией, технически возможно добавить прототипирование и в языки со статической проверкой типов. Язык [Omega](http://ru.wikipedia.org/w/index.php?title=Omega_(%D1%8F%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)&action=edit&redlink=1) является одним из примеров такой системы.

## Конструирование объектов

В класс-ориентированных языках новый экземпляр создаётся через вызов [конструктора класса](http://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BD%D1%81%D1%82%D1%80%D1%83%D0%BA%D1%82%D0%BE%D1%80_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) (возможно, с набором параметров). Получившийся экземпляр имеет структуру и поведение, жёстко заданные его классом.

В прототип-ориентированных системах предоставляется два метода создания нового объекта: *клонирование* существующего объекта, либо создание объекта *«с нуля»*. Для создания объекта с нуля программисту предоставляются синтаксические средства добавления [свойств](http://ru.wikipedia.org/wiki/%D0%A1%D0%B2%D0%BE%D0%B9%D1%81%D1%82%D0%B2%D0%BE_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) и [методов](http://ru.wikipedia.org/wiki/%D0%9C%D0%B5%D1%82%D0%BE%D0%B4_(%D1%8F%D0%B7%D1%8B%D0%BA%D0%B8_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)) в объект. В дальнейшем, с получившегося объекта может быть получена полная копия — клон. В процессе клонирования копия наследует все характеристики своего прототипа, но с этого момента она становится самостоятельной и может быть изменена. В некоторых реализациях копии хранят ссылки на объекты-прототипы, [делегируя](http://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%82%D0%BE%D1%82%D0%B8%D0%BF%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5#.D0.94.D0.B5.D0.BB.D0.B5.D0.B3.D0.B8.D1.80.D0.BE.D0.B2.D0.B0.D0.BD.D0.B8.D0.B5) им часть своей функциональности; при этом изменение прототипа может затронуть все его копии. В других реализациях новые объекты полностью независимы от своих прототипов. Ниже рассмотрены оба этих случая.

### Делегирование

В прототип-ориентированных языках, использующих *делегирование*, среда исполнения способна выполнять диспетчеризацию вызовов методов (или поиск нужных данных) просто следуя по цепочке делегирующих указателей (от объекта к его прототипу), до совпадения. В отличие от отношения «класс — экземпляр», отношение «прототип — потомки» не требует, чтобы объекты-потомки сохраняли структурное подобие со своим прототипом. С течением времени они могут адаптироваться и улучшаться, но при этом нет нужды переделывать прототип. Важно, что добавлять/удалять/модифицировать можно не только данные, но и функции, при этом функции тоже оказываются *объектами первого уровня*. Вследствие этого большинство прототип-ориентированных языков называют данные и методы объекта «слотами» (ячейками).

### Каскадирование

При «чистом» прототипировании — именуемом также *каскадным* и представленном в [Kevo](http://ru.wikipedia.org/w/index.php?title=Kevo&action=edit&redlink=1) — клонированные объекты не хранят ссылок на свои прототипы. Прототип копируется один-в-один, со всеми методами и атрибутами, и копии присваивается новое имя (ссылка). Это напоминает [митоз](http://ru.wikipedia.org/wiki/%D0%9C%D0%B8%D1%82%D0%BE%D0%B7) биологических клеток.

В число преимуществ данного подхода входит тот факт, что создатель копии может менять её, не опасаясь побочных эффектов среди других потомков своего предка. Также радикально снижаются вычислительные затраты на диспетчеризацию, так как нет необходимости обходить всю цепочку возможных делегатов в поисках подходящего слота (метода или атрибута).

В число недостатков можно включить трудности с распространением изменений в системе: модификация прототипа не влечёт за собой немедленное и автоматическое изменение всех его потомков. Тем не менее,[Kevo](http://ru.wikipedia.org/w/index.php?title=Kevo&action=edit&redlink=1) предоставляет дополнительные средства для публикации изменений среди множества объектов, причём на основании их подобия («семейного сходства»), а не по наличию общего предка, что типично для моделей с делегированием.

Другой недостаток в том, что простейшие реализации этой модели приводят к увеличенному (по сравнению с моделью делегирования) расходу памяти, так как каждый клон, пока он не изменён, будет содержать копию данных своего прототипа. Однако эта проблема разрешима оптимальным разделением неизменённых данных и применением «[ленивого копирования](http://ru.wikipedia.org/w/index.php?title=%D0%9B%D0%B5%D0%BD%D0%B8%D0%B2%D0%BE%D0%B5_%D0%BA%D0%BE%D0%BF%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5&action=edit&redlink=1)» — что и было использовано в Kevo.

**24.1. ДОСТОИНСТВА ООП**

От любого метода программирования мы ждем, что он поможет нам в решении наших проблем. Но одной из самых значительных проблем в программировании является сложность. Чем больше и сложнее программа, тем важнее становится разбить ее на небольшие, четко очерченные части. Чтобы побороть сложность, мы должны абстрагироваться от мелких деталей. В этом смысле классы представляют собой весьма удобный инструмент.

Классы позволяют проводить конструирование из полезных компонент, обладающих простыми инструментами, что дает возможность абстрагироваться от деталей реализации.

Данные и операции вместе образуют определенную сущность и они не «размазываются» по всей программе, как это нередко бывает в случае процедурного программирования.

Локализация кода и данных улучшает наглядность и удобство сопровождения программного обеспечения.

Инкапсуляция информации защищает наиболее критичные данные от несанкционированного доступа.

ООП дает возможность создавать расширяемые системы (extensible systems). Это одно из самых значительных достоинств ООП и именно оно отличает данный подход от традиционных методов программирования. Расширяемость (extensibility) означает, что существующую систему можно заставить работать с новыми компонентами, причем без внесения в нее каких-либо изменений. Компоненты могут быть добавлены на этапе выполнения.

Расширение типа (type extension) и вытекающий из него полиморфизм переменных оказываются полезными преимущественно в следующих ситуациях.

Обработка разнородных структур данных. Программы могут работать, не утруждая себя изучением вида объектов. Новые виды могут быть добавлены в любой момент.

Изменение поведения во время выполнения. На этапе выполнения один объект может быть заменен другим. Это может привести к изменению алгоритма, в котором используется данный объект.

Реализация родовых компонент. Алгоритмы можно обобщать до такой степени, что они уже смогут работать более, чем с одним видом объектов.

Доведение полуфабрикатов. Компоненты нет надобности подстраивать под определенное приложение. Их можно сохранять в библиотеке в виде полуфабрикатов (semifinished products) и расширять по мере необходимости до различных законченных продуктов.

Расширение каркаса. Независимые от приложения части предметной области могут быть реализованы в виде каркаса и в дальнейшем расширены за счет добавления частей, специфичных для конкретного приложения.

Многоразового использования программного обеспечения на практике добиться не удается из-за того, что существующие компоненты уже не отвечают новым требованиям. ООП помогает этого достичь без нарушения работы уже имеющихся клиентов, что позволяет нам извлечь максимум из многоразового использования компонент.

Мы сокращаем время на разработку, которое с выгодой может быть отдано другим проектам.

Компоненты многоразового использования обычно содержат гораздо меньше ошибок, чем вновь разработанные, ведь они уже не раз подвергались проверке.

Когда некая компонента используется сразу несколькими клиентами, то улучшения, вносимые в ее код, одновременно оказывают свое положительное влияние и на множество работающих с ней программ.

Если программа опирается на стандартные компоненты, то ее структура и пользовательский интерфейс становятся более унифицированными, что облегчает ее понимание и упрощает ее использование.

2. НЕДОСТАТКИ ООП

Объектно-ориентированное программирование требует знания четырех вещей.

(1) Необходимо понимать базовые концепции, такие как классы, наследование и динамическое связывание. Для программистов, уже знакомых с понятием модуля и с абстрактными типами данных, это потребует минимальных усилий. Для тех же, кто никогда не использовал инкапсуляцию данных, это может означать изменения мировоззрения и может отнять на изучение значительное количество времени.

(2) Многоразовое использование требует от программиста познакомиться с большими библиотеками классов. А это может оказаться сложнее, чем даже изучение нового языка программирования. Библиотека классов фактически представляет собой виртуальный язык, который может включать в себя сотни типов и тысячи операций. В языке Smalltalk, к примеру, до того, как перейти к практическому программированию, нужно изучить значительную часть его библиотеки классов. А это тоже требует времени.

(3) Проектирование классов — задача куда более сложная, чем их использование. Проектирование класса, как и проектирование языка, требует большого опыта. Это итеративный процесс, где приходится учиться на своих же ошибках.

(4) Очень трудно изучать классы, не имея возможности их «пощупать». Только с приобретением мало-мальского опыта можно уверенно себя почувствовать при работе с использованием ООП.

Как мы видели, усилия на освоение базовых концепций невелики, но вот в случае библиотек классов и их использования они могут быть очень существенными.

Поскольку детали реализации классов обычно неизвестны, то программисту, если он хочет разобраться в том или ином классе, нужно опираться на документацию и на используемые имена. И время, которое было сэкономлено на том, что удалось обойтись без написания собственного класса, должно быть отчасти потрачено (особенно вначале освоения) на то, чтобы разобраться в существующем классе.

Документирование классов — задача более трудная, чем это было в случае процедур и модулей. Поскольку любой метод может быть переопределен, в документации должно говориться не только о том, что делает данный метод, но также и о том, в каком контексте он вызывается. Ведь переопределенные методы обычно вызываются не клиентом, а самим каркасом. Таким образом, программист должен знать, какие условия выполняются, когда вызывается данный метод. Для абстрактных методов, которые пусты, в документации должно даже говориться о том, для каких целей предполагается использовать переопределяемый метод.

В сложных иерархиях классов поля и методы обычно наследуются с разных уровней. И не всегда легко определить, какие поля и методы фактически относятся к данному классу. Для получения такой информации нужны специальные инструменты вроде навигаторов классов. Если конкретный класс расширяется, то каждый метод обычно сокращают перед передачей сообщения базовому классу. Реализация операции, таким образом, рассредоточивается по нескольким классам, и чтобы понять, как она работает, нам приходится внимательно просматривать весь код.

Методы, как правило, короче процедур, поскольку они осуществляют только одну операцию над данными. Зато количество методов намного выше. Короткие методы обладают тем преимуществом, что в них легче разбираться, неудобство же их связано с тем, что код для обработки сообщения иногда «размазан» по многим маленьким методам.

Абстракция данных ограничивает гибкость клиентов. Клиенты могут лишь выполнять те операции, которые предоставляет им тот или иной класс. Они уже лишены неограниченного доступа к данным. Причины здесь аналогичны тем, что вызвали к жизни использование высокоуровневых языков программирования, а именно, чтобы избежать непонятных программных структур.

Абстракцией данных не следует злоупотреблять. Чем больше данных скрыто в недрах класса, тем сложнее его расширять. Отправной точкой здесь должно быть не то, что клиентам не разрешается знать о тех или иных данных, а то, что клиентам для работы с классом этих данных знать не требуется.

Часто можно слышать, что ООП является неэффективным. Как же дело обстоит в действительности? Мы должны четко проводить грань между неэффективностью на этапе выполнения, неэффектиностью в смысле распределения памяти и неэффективностью, связанной с излишней универсализацией.

(1) Неэффективность на этапе выполнения. В языках типа Smalltalk сообщения интерпретируются во время выполнения программы путем осуществления поиска их в одной или нескольких таблицах и за счет выбора подходящего метода. Конечно, это медленный процесс. И даже при использовании наилучших методов оптимизации Smalltalk-программы в десять раз медленнее оптимизированных C-программ [Cha92].

В гибридных языках типа Oberon-2, Object Pascal и C++ посылка сообщения приводит лишь к вызову через указатель процедурной переменной. На некоторых машинах сообщения выполняются лишь на 10% медленнее, чем обычные процедурные вызовы. И поскольку сообщения встречаются в программе гораздо реже других операций, их воздействие на время выполнения влияния практически не оказывает.

Однако, существует другой фактор, который затрагивает время выполнения: это абстракция данных. Абстракция запрещает непосредственный доступ к полям класса и требует, чтобы каждая операция над данными выполнялась через методы. Такая схема приводит к необходимости выполнения процедурного вызова при каждом доступе к данным. Однако, когда абстракция используется только там, где она необходима (т.е. не из одной лишь прихоти), то замедление вполне приемлемое.

(2) Неэффективность в смысле распределения памяти. Динамическое связывание и проверка типа на этапе выполнения требуют по ходу работы информации о типе объекта. Такая информация хранится в дескрипторе типа, и он выделяется один на класс. Каждый объект имеет невидимый указатель на дескриптор типа для своего класса. Таким образом, в объектно-ориентированных программах требуемая дополнительная память выражается в одном указателе для объекта и в одном дескрипторе типа для класса.

(3) Излишняя универсальность. Неэффективность может также означать, что программа имеет ненужные возможности. В библиотечном классе часто содержится больше методов, чем это реально необходимо. А поскольку лишние методы не могут быть удалены, то они становятся мертвым грузом. Это не воздействует на время выполнения, но влияет на возрастание размера кода.

Одно из возможных решений — строить базовый класс с минимальным числом методов, а затем уже реализовывать различные расширения этого класса, которые позволят нарастить функциональность.

Другой подход — дать возможность компоновщику удалять лишние методы.

Такие интеллектуальные компоновщики уже доступны для различных языков и операционных систем.

Oberon избрал третий путь избавления от излишней универсальности.

Программные части могут добавляться на этапе выполнения. Таким образом, нет надобности загружать всю программу целиком, а можно обойтись лишь теми ее частями, которые в данный момент необходимы. Как показала практика, это экономит гораздо больше кода, чем можно добиться при удалении лишних методов.

Таким образом, нельзя утверждать, что ООП вообще неэффективно.

Если классы используются лишь там, где это действительно необходимо, то потеря эффективности и на этапе выполнения и в смысле памяти сводится практически на нет.

3. БУДУЩЕЕ ООП

Выживет ли объектно-ориентированное программирование, или оно лишь модное поветрие, которое скоро исчезнет?

Классы нашли свое место в большинстве современных языков программирования. Одно лишь это говорит о том, что им суждено остаться. Классы в самом ближайшем будущем войдут в стандартный набор концепций для каждого программиста, точно так же, как многие сегодня применяют динамические структуры данных и рекурсию, которые двадцать лет назад были также в диковинку. В то же время классы — это просто еще одна новая конструкция наряду с остальными. Нам нужно узнать, для каких ситуаций они подходят, и только здесь мы и будем их использовать. Правильно выбрать инструмент для конкретной задачи — обязательно для каждого мастерового и в еще большей степени для каждого инженера.

ООП ввергает многих в состояние эйфории. Пестрящая тут и там реклама сулит нам невероятные вещи, и даже некоторые исследователи, похоже, склонны рассматривать ООП как панацею, способную решить все проблемы разработки программного обеспечения. Со временем эта эйфория постепенно уляжется. И после периода разочарования люди, быть может, перестанут уже говорить об ООП, точно также как сегодня вряд ли от кого можно услышать о структурном программировании. Но классы будут использовать как нечто само собой разумеющееся, и мы сможем, наконец, понять, что они собой представляют: просто компоненты, которые помогают строить модульное и расширяемое программное обеспечение.

**25.Объектно-ориентированные языки программирования. Свойства объекта.**

Поля предназначены для использования внутри класса. Однако класс должен каким-либо образом взаимодействовать с другими классами или программными элементами приложения. В подавляющем большинстве случаев класс должен выполнить с некоторыми данными определенные действия и представить результат.

Для получения и передачи данных в классе применяются свойства.

Свойства представляют собой атрибуты, которые составляют индивидуальность объекта и помогают описать его. Например, обычная кнопка в окне приложения обладает такими свойствами, как цвет, размеры, положение.

Так как свойство обеспечивает обмен данными с внешней средой, то для доступа к его значению используются специальные методы класса. Поэтому обычно свойство определяется двумя элементами: двумя методами, которые осуществляют его чтение/запись.

Свойства реализованы таких языках как C#,Delphi,java. В С++ свойств нет.

**26. Объектно-ориентированные языки программирования. Векторные свойства.**

**На примере Delphi**

Свойство может быть и векторным; в этом случае оно внешне выглядит как массив:

propertyAPoints[Index : Integer]:TPoint read GetPoint write SetPoint;

На самом деле в классе может и не быть соответствующего поля — массива. Напомним, что вся обработка обращений к внутренним структурам класса может быть замаскирована.

Для векторного свойства необходимо описать не только тип элементов массива, но также имя и тип индекса. После ключевых слов read и write в этом случае должны стоять имена методов — использование здесь полей массивов недопустимо. Метод, читающий значение векторного свойства, должен быть описан как функция, возвращающая значение того же типа, что и элементы свойства, и имеющая единственный параметр того же типа и с тем же именем, что и индекс свойства:

functionGetPoint(Index:Integer):TPoint;

Аналогично, метод, помещающий значения в такое свойство, должен первым параметром иметь индекс, а вторым — переменную нужного типа (которая может быть передана как по ссылке, так и по значению):

procedureSetPoint(Index:Integer; NewPoint:TPoint);

У векторных свойств есть еще одна важная особенность. Некоторые классы в Delphi (списки т-List, наборы строк TStrings) "построены" вокруг основного векторного свойства (см. гл. 7). Основной метод такого класса дает доступ к некоторому массиву, а все остальные методы являются как бы вспомогательными. Специально для облегчения работы в этом случае векторное свойство может быть описано с ключевым словом default:

type

TMyObject = class;

property Strings[Index: Integer]: string read Get write Put; default;

  end;

Если у объекта есть такое свойство, то можно его не упоминать, а ставить индекс в квадратных скобках сразу после имени объекта:

varAMyObject: TMyObject;

begin

...

AMyObject.Strings[1] := 'First'; {первыйспособ}

 AMyObject[2] := 'Second'; (второйспособ}

...

end.

Будьте внимательны, применяя зарезервированное слово default, — как мы увидели, для обычных и векторных свойств оно употребляется в разных случаях и с различным синтаксисом.

**27.Полиморфизм. Статические, виртуальные, динамические методы.**

Полиморфизм — это свойство системы использовать объекты с одинаковым интерфейсом без информации о типе и внутренней структуре объекта.

[Статический](http://msdn.microsoft.com/ru-ru/library/98f28cdx.aspx) класс в основном такой же, что и нестатический класс, но имеется одно отличие: нельзя создавать экземпляры статического класса. Другими словами, нельзя использовать ключевое слово [new](http://msdn.microsoft.com/ru-ru/library/51y09td4.aspx) для создания переменной типа класса. Поскольку нет переменной экземпляра, доступ к членам статического класса осуществляется с использованием самого имени класса. Например, если имеется статический класс, называемый UtilityClass, имеющий открытый метод, называемый MethodA, вызов метода выполняется, как показано в следующем примере.

C#

UtilityClass.MethodA();

Принципиально отличаются от статических виртуальные и динамические методы. Они должны быть объявлены путем добавления соответствующей директивы virtual или dynamic. Обе эти категории существовали и в прежних версиях языка Pascal. С точки зрения наследования методы этих двух видов одинаковы: они могут быть перекрыты в дочернем классе только одноименными методами, имеющими тот же тип.

Если задуматься над рассмотренным выше примером, становится ясно, что у компилятора нет возможности определить класс объекта, фактически переданного в процедуру showData. Нужен механизм, позволяющий определить это прямо во время выполнения. Такой механизм называется поздним связыванием (latebinding).

Естественно, что этот механизм должен быть каким-то образом связан с передаваемым объектом. Для этого используются таблица виртуальных методов (VirtualMethodTable, VMT) и таблица динамических методов (DynamicMethodTable, DMT).

Разница между виртуальными и динамическими методами заключается в особенности поиска адреса. Когда компилятор встречает обращение к виртуальному методу, он подставляет вместо прямого вызова по конкретному адресу код, который обращается к VMT и извлекает оттуда нужный адрес.

Такая таблица есть для каждого класса (объектного типа). В ней хранятся адреса всех виртуальных методов класса, независимо от того, унаследованы ли они от предка или перекрыты в данном классе. Отсюда и достоинства, и недостатки виртуальных методов: они вызываются сравнительно быстро, однако для хранения указателей на них в таблице VMT требуется большое количество памяти.

Динамические методы вызываются медленнее, но позволяют более экономно расходовать память. Каждому динамическому методу системой присваивается уникальный индекс. В таблице динамических методов класса хранятся индексы и адреса только тех динамических методов, которые описаны в данном классе. При вызове динамического метода происходит поиск в этой таблице; в случае неудачи просматриваются таблицы DMT всех классов-предков в порядке иерархии и, наконец, класс TObject, где имеется стандартный обработчик вызова динамических методов. Экономия памяти налицо.

**28.Абстрактные методы. Директива override.**

Для перекрытия и виртуальных, и динамических методов служит директива override, с помощью которой (и только с ней!) можно переопределять оба этих типа методов.

**Абстра́ктныйме́тод** (или **чистый виртуальный метод** (**purevirtualmethod** - часто неверно переводится как **чисто виртуальный метод**)) — в [объектно-ориентированном программировании](http://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5), метод [класса](http://ru.wikipedia.org/wiki/%D0%9A%D0%BB%D0%B0%D1%81%D1%81_%28%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5%29), реализация для которого отсутствует. Класс, содержащий абстрактные методы, также принято называть [абстрактным](http://ru.wikipedia.org/wiki/%D0%90%D0%B1%D1%81%D1%82%D1%80%D0%B0%D0%BA%D1%82%D0%BD%D1%8B%D0%B9_%D0%BA%D0%BB%D0%B0%D1%81%D1%81) (там же и пример). Абстрактные методы зачастую путают с [виртуальными](http://ru.wikipedia.org/wiki/%D0%92%D0%B8%D1%80%D1%82%D1%83%D0%B0%D0%BB%D1%8C%D0%BD%D1%8B%D0%B9_%D0%BC%D0%B5%D1%82%D0%BE%D0%B4). Абстрактный метод подлежит определению в классах-наследниках, поэтому его можно отнести к виртуальным, но не каждый виртуальный метод является абстрактным.

**41 Обработка исключительных ситуаций. Реализация на уровне объектов.**

**Общее понятие исключительной ситуации**

Во время выполнения программы могут возникать ситуации, когда состояние данных, устройств ввода-вывода или компьютерной системы в целом делает дальнейшие вычисления в соответствии с базовым алгоритмом невозможными или бессмысленными. Классические примеры подобных ситуаций приведены ниже.

* Целочисленное деление на ноль. Результата у операции быть не может, поэтому ни дальнейшие вычисления, ни попытка использования результата деления не приведут к решению задачи.
* Ошибка при попытке считать данные с внешнего устройства. Если данные не удаётся ввести, любые дальнейшие запланированные операции с ними бессмысленны.
* Исчерпание доступной памяти. Если в какой-то момент система оказывается не в состоянии выделить достаточный для прикладной программы объём оперативной памяти, программа не сможет работать нормально.
* Появление сигнала аварийного отключения электропитания системы. Прикладную задачу, по всей видимости, решить не удастся, в лучшем случае (при наличии какого-то резерва питания) прикладная программа может позаботиться о сохранении данных.
* Появление на входе коммуникационного канала данных, требующих немедленного считывания. Чем бы ни занималась в этот момент программа, она должна перейти к чтению данных, чтобы не потерять поступившую информацию.

**Виды исключительных ситуаций**

Исключительные ситуации, возникающие при работе программы, можно разделить на два основных типа: синхронные и асинхронные, принципы реакции на которые существенно различаются.

* **Синхронные исключения** могут возникнуть только в определённых, заранее известных точках программы. Так, ошибка чтения файла или коммуникационного канала, нехватка памяти — типичные синхронные исключения, так как возникают они только в операции чтения из файла или из канала или в операции выделения памяти соответственно.
* **Асинхронные исключения** могут возникать в любой момент времени и не зависят от того, какую конкретно инструкцию программы выполняет система. Типичные примеры таких исключений: аварийный отказ питания или поступление новых данных.

Некоторые типы исключений могут быть отнесены как к синхронным, так и к асинхронным. Например, инструкция деления на нуль формально должна приводить к синхронному исключению, так как логически оно возникает именно при выполнении данной команды, но на некоторых платформах за счёт глубокой конвейеризации исключение может фактически оказаться асинхронным.

## Обработчики исключений

### Общее описание

В отсутствие собственного механизма обработки исключений для прикладных программ наиболее общей реакцией на любую исключительную ситуацию является немедленное прекращение выполнения с выдачей пользователю сообщения о характере исключения. Можно сказать, что в подобных случаях единственным и универсальным обработчиком исключений становится операционная система. Например, в операционную систему [Windows](http://dic.academic.ru/dic.nsf/ruwiki/754819) встроена утилита [Dr. Watson](http://dic.academic.ru/dic.nsf/ruwiki/1267722), которая занимается сбором информации о необработанном исключении и её отправкой на специальный сервер компании [Microsoft](http://dic.academic.ru/dic.nsf/ruwiki/7585).

Возможно игнорирование исключительной ситуации и продолжение работы, но такая тактика опасна, так как приводит к ошибочным результатам работы программ или возникновению ошибок впоследствии. Например, проигнорировав ошибку чтения из файла блока данных, программа получит в своё распоряжение не те данные, которые она должна была считать, а какие-то другие. Результаты их использования предугадать невозможно.

Обработка исключительных ситуаций самой программой заключается в том, что при возникновении исключительной ситуации управление передаётся некоторому заранее определённому *обработчику* — блоку кода, процедуре, функции, которые выполняют необходимые действия.

Существует два принципиально разных механизма функционирования обработчиков исключений.

* **Обработка с возвратом** подразумевает, что обработчик исключения ликвидирует возникшую проблему и приводит программу в состояние, когда она может работать дальше по основному алгоритму. В этом случае после того, как выполнится код обработчика, управление передаётся обратно в ту точку программы, где возникла исключительная ситуация (либо на команду, вызвавшую исключение, либо на следующую за ней, как в некоторых старых диалектах языка BASIC) и выполнение программы продолжается. Обработка с возвратом типична для обработчиков асинхронных исключений (которые обычно возникают по причинам, не связанным прямо с выполняемым кодом), для обработки синхронных исключений она малопригодна.
* **Обработка без возврата** заключается в том, что после выполнения кода обработчика исключения управление передаётся в некоторое, заранее заданное место программы, и с него продолжается исполнение. То есть, фактически, при возникновении исключения команда, во время работы которой оно возникло, заменяется на [безусловный переход](http://dic.academic.ru/dic.nsf/ruwiki/803812) к заданному оператору.

Существует два варианта подключения обработчика исключительных ситуаций к программе: структурная и неструктурная обработка исключений.

### Неструктурная обработка исключений

Неструктурная обработка исключений реализуется в виде механизма регистрации [функций](http://dic.academic.ru/dic.nsf/ruwiki/7543) или команд-обработчиков для каждого возможного [типа](http://dic.academic.ru/dic.nsf/ruwiki/4898) исключения. [Язык программирования](http://dic.academic.ru/dic.nsf/ruwiki/1465) или его системные [библиотеки](http://dic.academic.ru/dic.nsf/ruwiki/10141) предоставляют программисту как минимум две стандартные процедуры: регистрации обработчика и разрегистрации обработчика. Вызов первой из них «привязывает» обработчик к определённому исключению, вызов второй — отменяет эту «привязку». Если исключение происходит, выполнение основного кода программы немедленно прерывается и начинается выполнение обработчика. По завершении обработчика управление передаётся либо в некоторую наперёд заданную точку программы, либо обратно в точку возникновения исключения (в зависимости от заданного способа обработки — с возвратом или без). Независимо от того, какая часть программы в данный момент выполняется, на определённое исключение всегда реагирует последний зарегистрированный для него обработчик. В некоторых языках зарегистрированный обработчик сохраняет силу только в пределах текущего блока кода (процедуры, функции), тогда процедура разрегистрации не требуется. Ниже показан условный фрагмент кода программы с неструктурной обработкой исключений:

УстановитьОбработчик(ОшибкаБД, ПерейтиНаОшБД)

// На исключение "ОшибкаБД" установлен обработчик - команда "ПерейтиНаОшБД"

... // Здесь находятся операторы работы с БД

ПерейтиНаСнятьОшБД // Команда безусловного перехода - обход обработчика исключений

ОшБД: // метка - сюда произойдёт переход в случае ошибки БД по установленному обработчику

... // Обработчик исключения БД

СнятьОшБД:

// метка - сюда произойдёт переход, если контролируемый код выполнится без ошибки БД.

СнятьОбработчик(ОшибкаБД)

// Обработчик снят

Неструктурная обработка — практически единственный вариант для обработки асинхронных исключений, но для синхронных исключений она неудобна: приходится часто вызывать команды установки/снятия обработчиков, всегда остаётся опасность нарушить логику работы программы, пропустив регистрацию или разрегистрацию обработчика.

### Структурная обработка исключений

Структурная обработка исключений требует обязательной поддержки со стороны языка программирования — наличия специальных синтаксических конструкций. Такая конструкция содержит блок контролируемого кода и обработчик (обработчики) исключений. Наиболее общий вид такой конструкции (условный):

НачалоБлока

... // Контролируемый код

...

если (условие) то СоздатьИсключение Исключение2

...

Обработчик Исключение1

... // Код обработчика для Исключения1

Обработчик Исключение2

... // Код обработчика для Исключения2

ОбработчикНеобработанных

... // Код обработки ранее не обработанных исключений

КонецБлока

Здесь «НачалоБлока» и «КонецБлока» — ключевые слова, которые ограничивают блок контролируемого кода, а «Обработчик» — начало блока обработки соответствующего исключения. Если внутри блока, от начала до первого обработчика, произойдёт исключение, то произойдёт переход на обработчик, написанный для него, после чего весь блок завершится и исполнение будет продолжено со следующей за ним команды. В некоторых языках нет специальных ключевых слов для ограничения блока контролируемого кода, вместо этого обработчик (обработчики) исключений могут быть встроены в некоторые или все синтаксические конструкции, объединяющие несколько операторов. Так, например, в языке Ада любой составной оператор (begin — end) может содержать обработчик исключений.

«ОбработчикНеобработанных» — это обработчик исключений, которые не соответствуют ни одному из описанных выше в данном блоке. Обработчики исключений в реальности могут описываться по-разному (один обработчик на все исключения, по одному обработчику на каждый тип исключение), но принципиально они работают одинаково: при возникновении исключения находится первый соответствующий ему обработчик в данном блоке, его код выполняется, после чего выполнение блока завершается. Исключения могут возникать как в результате программных ошибок, так и путём явной их генерации с помощью соответствующей команды (в примере — команда «СоздатьИсключение»). С точки зрения обработчиков такие искусственно созданные исключения ничем не отличаются от любых других.

Блоки обработки исключений могут многократно входить друг в друга, как явно (текстуально), так и неявно (например, в блоке вызывается процедура, которая сама имеет блок обработки исключений). Если ни один из обработчиков в текущем блоке не может обработать исключение, то выполнение данного блока немедленно завершается, и управление передаётся наближайший подходящий обработчик более высокого уровня иерархии. Это продолжается до тех пор, пока обработчик не найдётся и не обработает исключение или пока не выйдет из обработчиков заданных программистом и не будет передано системному обработчику по умолчанию, аварийно закроющему программу.

Иногда бывает неудобно завершать обработку исключения в текущем блоке, то есть желательно, чтобы при возникновении исключения в текущем блоке обработчик выполнил какие-то действия, но исключение продолжило бы обрабатываться на более высоком уровне (обычно так бывает, когда обработчик данного блока не полностью обрабатывает исключение, а лишь частично). В таких случаях в обработчике исключений генерируется новое исключение или возобновляется с помощью специальной команды ранее появившееся. Код обработчиков не является защищённым в данном блоке, поэтому созданное в нём исключение будет обрабатываться в блоках более высокого уровня.

### Блоки с гарантированным завершением

Помимо блоков контролируемого кода для обработки исключений, языки программирования могут поддерживать блоки с гарантированным завершением. Их использование оказывается удобным тогда, когда в некотором блоке кода, независимо от того, произошли ли какие-то ошибки, необходимо перед его завершением выполнить определённые действия. Простейший пример: если в процедуре динамически создаётся какой-то локальный объект в памяти, то перед выходом из этой процедуры объект должен быть уничтожен (чтобы избежать утечки памяти), независимо от того, произошли после его создания ошибки или нет. Такая возможность реализуется блоками кода вида:

НачалоБлока

... // Основной код

Завершение

... // Код завершения

КонецБлока

Заключённые между ключевыми словами «НачалоБлока» и «Завершение» операторы (основной код) выполняются последовательно. Если при выполнении их не возникает исключений, то затем выполняются операторы между ключевыми словами «Завершение» и «КонецБлока» (код завершения). Если же при выполнении основного кода возникает исключение (любое), то сразу же выполняется код завершения, после чего весь блок завершается, а возникшее исключение продолжает существовать и распространяться до тех пор, пока его не перехватит какой-либо блок обработки исключений более высокого уровня.

Принципиальное отличие блока с гарантированным завершением от обработки — то, что он не обрабатывает исключение, а лишь гарантирует выполнение определённого набора операций перед тем, как включится механизм обработки. Стоит заметить, что блок с гарантированным завершением легко реализуется с помощью команд «возбудить исключение» и «структурный обработчик исключения».

**42 Стадии разработки ПО.**

В процессе создания любой программы можно выделить несколько этапов.

1. *Постановка задачи —* выполняется специалистом в предметной области на естественном языке (русском, английском и т. д.). Необходимо определить цель задачи, ее содержание и общий подход к решению. Возможно, что задача решается точно*(аналитически),* и без компьютера можно обойтись. Уже на этапе постановки надо учитывать эффективность алгоритма решения задачи на ЭВМ, ограничения, накладываемые *аппаратным и программным обеспечением* (АО и ПО).

2. *Анализ задачи и моделирование —* определяются исходные данные и результат, выявляются ограничения на их значения, выполняется формализованное описание задачи и построение (выбор) математической модели, пригодной для решения на компьютере.

3. *Разработка или выбор алгоритма решения задачи —* выполняется на осно­ве ее математического описания. Многие задачи можно решить различными способами. Программист должен выбрать оптимальное решение. Неточности в постановке, анализе задачи или разработке алгоритма могут привести к *скрытой ошибке —* программист получит неверный результат, считая его правильным.

4. *Проектирование общей структуры программы —* формируется модель решения с последующей детализацией и разбивкой на подпрограммы, определяется "архитектура" программы, способ хранения информации (набор переменных, массивов и т. п.).

5. *Кодирование —* запись алгоритма на языке программирования. Современные системы программирования позволяют ускорить процесс разработки программы, автоматически создавая часть ее текста, однако творческая работа по-прежнему лежит на программисте. Для успешной реализации целей проекта программисту необходимо использовать *методы структурного программирования.*

6. *Отладка и тестирование программы.* Под *отладкой* понимается устранение ошибок в программе. *Тестирование* позволяет вести их поиск и, в конечном счете, убедиться в том, что полностью отлаженная программа дает правильный результат. Для этого разрабатывается *система тестов —*специально подобранных контрольных примеров с такими наборами па­раметров, для которых решение задачи известно. Тестирование должно охватывать все возможные ветвления в программе, т. е. *проверять все ее инструкции,*и включать такие исходные данные, для которых решение *невозможно.* Проверка особых, *исключительных ситуаций,* необходима для анализа корректности. Например, программа должна отказать клиенту банка в просьбе выдать сумму, отсутствующую на его счете. В ответст­венных проектах большое внимание уделяется так называемой "защите от дурака" подразумевающей устойчивость программы к неумелому обращению пользователя. Использование специальных программ — *отладчиков,* которые позволяют выполнять программу по отдельным шагам, просматривая при этом значения переменных, значительно упрощает этот этап.

7. *Анализ результатов —* если программа выполняет моделирование какого-либо известного процесса, следует сопоставить результаты вычислений с результатами наблюдений. В случае существенного расхождения необ­ходимо изменить модель.

8. *Публикация* результатов работы, *передача заказчику* для эксплуатации.

9. *Сопровождение программы —* включает консультации представителей заказчика по работе с программой и обучение персонала. Недостатки и ошибки, замеченные в процессе эксплуатации, должны устраняться.

**42. Проектирование модели состояний программной системы.**

Объектная модель представляет статическую структуру проектируемой системы (подсистемы). Однако знания статической структуры недостаточно, чтобы понять и оценить работу подсистемы. Необходимо иметь средства для описания изменений, которые происходят с объектами и их связями во время работы подсистемы. Одним из таких средств является динамическая модель подсистемы. Она строится после того, как объектная модель подсистемы построена и предварительно согласована и отлажена. Динамическая модель подсистемы состоит из диаграмм состояний ее объектов и подсистем.

Текущее состояние объекта характеризуется совокупностью текущих значений его атрибутов и связей. Во время работы системы составляющие ее объекты взаимодействуют друг с другом, в результате чего изменяются их состояния. Единицей влияния является событие: каждое событие приводит к смене состояния одного или нескольких объектов в системе, либо к возникновению новых событий. Работа системы характеризуется последовательностью происходящих в ней событий.

*События*

Событие происходит в некоторый момент времени (нередко оно используется для определения соответствующего момента времени). Примеры событий: старт ракеты, старт забега на 100 м, начало проводки (в банковской сети), выдача денег и т.п. Событие не имеет продолжительности (точнее, оно занимает пренебрежимо малое время).

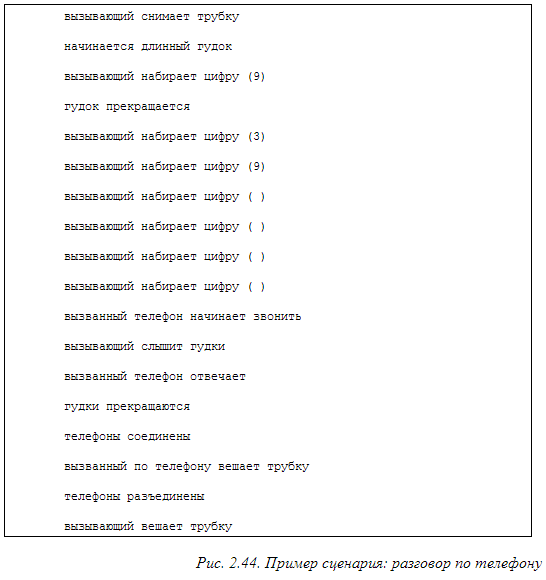
Одно из событий может логически предшествовать другому, либо следовать за другим, либо они могут быть независимыми; примерами логически (причинно) связанных событий являются старт и финиш одного забега, начало проводки и выдача денег клиенту (в результате этой проводки), примерами независимых событий - старт ракеты и финиш забега, проводки, запущенные с разных ATM, и т.п. Если события не имеют причинной связи (т.е. они логически независимы), они называются *независимыми*(concurrent); такие события не влияют друг на друга. Независимые события не имеет смысла упорядочивать, так как они могут происходить в произвольном порядке. Модель распределенной системы обязательно должна содержать независимые события и активности.

События передают информацию с одного объекта на другой. Существуют классы событий, которые просто сигнализируют о том, что что-то произошло или происходит (примеры: загорание лампочки лифта, гудок в телефонной трубке). В программировании рассматриваются *исключительные события*(иногда их называют исключениями), которые сигнализируют о нарушениях работы аппаратуры, либо программного обеспечения.

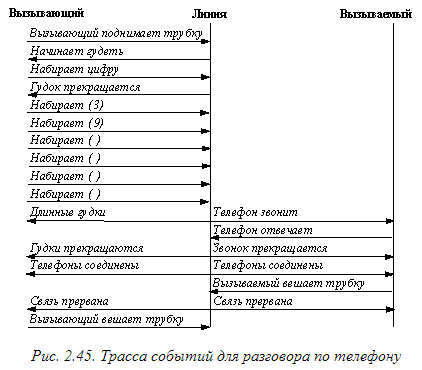
*Сценарии и трассы событий*

Сценарием называется последовательность событий, которая может иметь место при конкретном выполнении системы. Сценарии могут иметь разные области влияния: они могут включать все события, происходящие в системе, либо только события, возникающие и влияющие только на определенные объекты системы.

На рисунке 2.44 приведен пример сценария пользования телефонной линией. Каждое событие в этом сценарии передает информацию с одного объекта на другой; например событие начинается длинный гудок передает сигнал от телефонной линии к вызывающему (пользователю). При анализе динамики работы системы необходимо составить и рассмотреть несколько сценариев, отражающих типичные варианты ее работы.



Следующим этапом после разработки и анализа сценариев является определение объектов, генерирующих и принимающих каждое событие сценария. Последовательности событий с привязкой к объектам проектируемой системы удобно представлять на диаграммах, называемых трассами событий. Пример трассы событий для разговора по телефону представлен на рисунке 2.45. Вертикальные линии изображают на этой диаграмме объекты, а горизонтальные стрелки - события (стрелка начинается в объекте, генерирующем событие, и заканчивается в объекте, принимающем событие). Более поздние события помещены ниже более ранних, одновременные - на одном уровне.

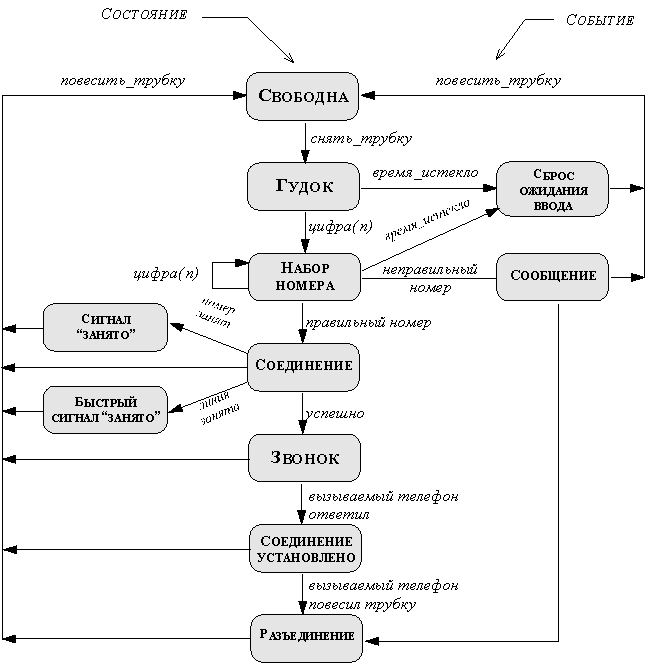


*Состояния*

Состояние определяется совокупностью текущих значений атрибутов и связей объекта. Например, банк может иметь состояния платежеспособный и неплатежеспособный (когда большая часть банков одновременно оказывается во втором состоянии, наступает банковский кризис).Состояние определяет реакцию объекта на поступающее в него событие (в том, что реакция различна нетрудно убедиться с помощью банковской карточки: в зависимости от состояния банка обслуживание (реакция банка на предъявление карточки) будет разным). Реакция объекта на событие может включать некоторое действие и/или перевод объекта в новое состояние.Объект сохраняет свое состояние в течение времени между двумя последовательными событиями, которые он принимает: события представляют моменты времени, состояния - отрезки времени; состояние имеет продолжительность, которая обычно равна отрезку времени между двумя последовательными событиями, принимаемыми объектом, но иногда может быть больше.При определении состояний мы не рассматриваем тех атрибутов, которые не влияют на поведение объекта, и объединяем в одно состояние все комбинации значений атрибутов и связей, которые дают одинаковые реакции на события.

*Диаграммы состояний*

Диаграмма состояний связывает события и состояния. При приеме события следующее состояние системы зависит как от ее текущего состояния, так и от события; смена состояния называется *переходом*. Диаграмма состояний - это граф, узлы которого представляют состояния, а направленные деги, помеченные именами соответствующих событий, - переходы. Диаграмма состояний позволяет получить последовательность состояний по заданной последовательности событий.



*Рис. 2.46. Диаграмма состояний телефонной линии*

**43 Проектирование функциональной модели программной системы.**

Функциональная модель показывает как вычисляются значения в системе и как они зависят одно от другого. Для конструирования функциональной модели необходимо выполнить следующее:

* определить входные и выходные значения;
* построить ДПД, показывающие функциональные зависимости;
* описать функции;
* описать ограничения;
* сформулировать критерии оптимизации;
* уточнить набор операций в объектной модели.

Функциональная модель описывает вычисления в системе. Она показывает, каким образом выходные данные вычисляются по входным данным, не рассматривая порядок и способ реализации вычислений. Функциональная модель состоит из набора диаграмм потока данных, которые показывают потоки значений от внешних входов через операции и внутренние хранилища данных к внешним выходам. Функциональная модель описывает смысл операций объектной модели и действий динамической модели, а также ограничения на объектную модель. Неинтерактивные программы (например, компиляторы) имеют тривиальную динамическую модель: их цель состоит в вычислении значения некоторой функции. Основной моделью таких программ является функциональная модель (хотя если программа имеет нетривиальные структуры данных, для нее важна и объектная модель).

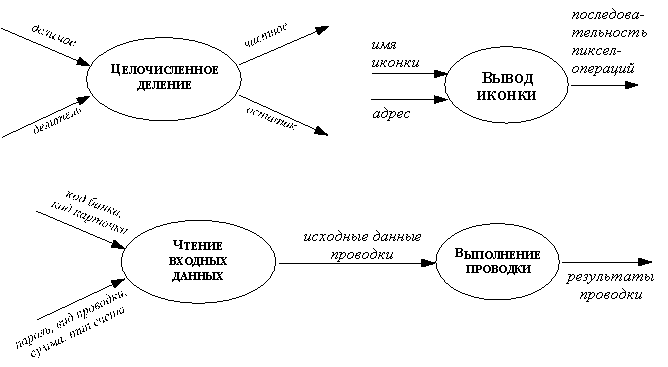
Функциональная модель представляет собой набор диаграмм потоков данных (далее - ДПД), которые описывают смысл операций и ограничений. ДПД отражает функциональные зависимости значений, вычисляемых в системе, включая входные значения, выходные значения и внутренние хранилища данных. ДПД - это граф, на котором показано движение значений данных от их источников через преобразующие их процессы к их потребителям в других объектах.

ДПД содержит процессы, которые преобразуют данные, потоки данных, которые переносят данные, активные объекты, которые производят и потребляют данные, и хранилища данных, которые пассивно хранят данные.

*Процессы*

Процесс преобразует значения данных. Процессы самого нижнего уровня представляют собой функции без побочных эффектов (примерами таких функций являются вычисление суммы двух чисел, вычисление комиссионного сбора за выполнение проводки с помощью банковской карточки и т.п.). Весь граф потока данных тоже представляет собой процесс (высокого уровня). Процесс может иметь побочные эффекты, если он содержит нефункциональные компоненты, такие как хранилища данных или внешние объекты.

На ДПД процесс изображается в виде эллипса, внутри которого помещается имя процесса; каждый процесс имеет фиксированное число входных и выходных данных, изображаемых стрелками (см. рисунок 2.60).

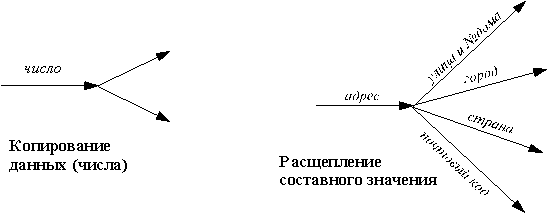


*Рис. 2.60. Примеры процессов*

Процессы реализуются в виде методов (или их частей) и соответствуют операциям конкретных классов.

*Потоки данных*

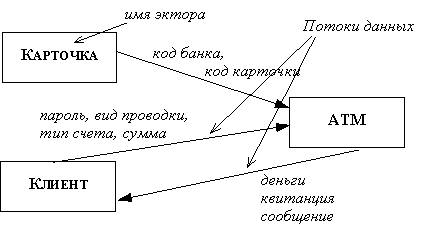
Поток данных соединяет выход объекта (или процесса) со входом другого объекта (или процесса). Он представляет промежуточные данные вычислений. Поток данных изображается в виде стрелки между производителем и потребителем данных, помеченной именами соответствующих данных; примеры стрелок, изображающих потоки данных, представлены на рисунке 2.61. На первом примере изображено копирование данных при передаче одних и тех же значений двум объектам, на втором - расщепление структуры на ее поля при передаче разных полей структуры разным объектам.



*Рис. 2.61. Потоки данных*

*Активные объекты*

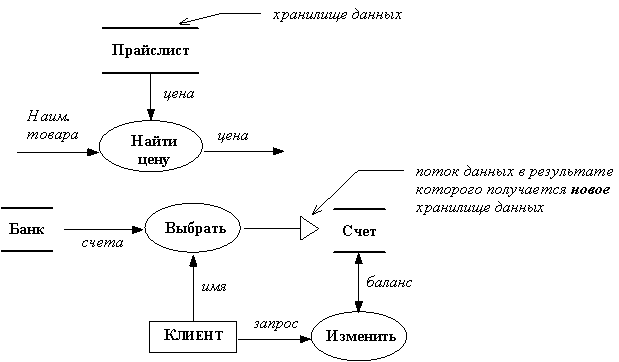
Активным называется объект, который обеспечивает движение данных, поставляя или потребляя их. Активные объекты обычно бывают присоединены к входам и выходам ДПД. Примеры активных объектов показаны на рисунке 2.62. На ДПД активные объекты обозначаются прямоугольниками.



*Рис. 2.62. Активные объекты (экторы)*

*Хранилища данных*

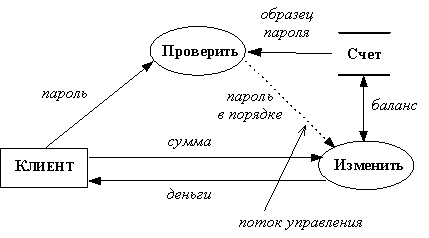
Хранилище данных - это пассивный объект в составе ДПД, в котором данные сохраняются для последующего доступа. Хранилище данных допускает доступ к хранимым в нем данным в порядке, отличном от того, в котором они были туда помещены. Агрегатные хранилища данных, как например, списки и таблицы, обеспечивают доступ к данным в порядке их поступления, либо по ключам. Примеры хранилищ данных приведены на рисунке 2.63.



*Рис. 2.63. Хранилища данных*

*Потоки управления*

ДПД показывает все пути вычисления значений, но не показывает в каком порядке значения вычисляются. Решения о порядке вычислений связаны с управлением программой, которое отражается в динамической модели. Эти решения, вырабатываемые специальными функциями, или предикатами, определяют, будет ли выполнен тот или иной процесс, но при этом не передают процессу никаких данных, так что их включение в функциональную модель необязательно. Тем не менее иногда бывает полезно включать указанные предикаты в функциональную модель, чтобы в ней были отражены условия выполнения соответствующего процесса. Функция, принимающая решение о запуске процесса, будучи включенной в ДПД, порождает в ДПД поток управления (он изображается пунктирной стрелкой).



*Рис. 2.64. Поток управления*

На рисунке 2.64 изображен пример потока управления: клиент, желающий снять часть своих денег со счета в банке, вводит в ATM пароль и требуемую сумму, однако фактическое снятие и выдача денег происходит только в том случае, когда введенный пароль совпадает с его образцом.

Несмотря на то, что потоки управления иногда оказываются весьма полезными, следует иметь в виду, что включение их в ДПД приводит к дублированию информации, входящей в динамическую модель.

**45)В результате анализа мы получаем три модели: объектную, динамическую и функциональную.** При этом объектная модель составляет базу, вокруг которой осуществляется дальнейшая разработка. При построении объектной модели в ней не всегда указываются операции над объектами, так как с точки зрения объектной модели объекты это, прежде всего, структуры данных. Поэтому разработка системы начинается с сопоставления действиям и активностям динамической модели и процессам функциональной модели операций и внесения этих операций в объектную модель. С этого начинается процесс разработки программы, реализующей поведение, которое описывается моделями, построенными в результате анализа требований к системе.

Поведение объекта задается его диаграммой состояния; каждому переходу на этой диаграмме соответствует применение к объекту одной из его операций; можно каждому событию, полученному объектом, сопоставить операцию над этим объектом, а каждому событию, посланному объектом, сопоставить операцию над объектом, которому событие было послано. Активности, запускаемой переходом на диаграмме состояний, может соответствовать еще одна (вложенная) диаграмма состояний.

Результатом этого этапа проектирования является уточненная объектная модель, содержащая все классы проектируемой программной системы, в которых специфицированы все операции над их объектами.

**46)Третья фаза жизненного цикла программной системы состоит в реализации разработанных** программных единиц (классов, функций, библиотек), которые в совокупности составляют разрабатываемую программную систему. Реализация каждой программной единицы может осуществляться как на объектно-ориентированном, так и на не объектно-ориентированном языке программирования, с использованием ранее разработанных программ, библиотек и баз данных.

Каждый язык программирования имеет средства для выражения трех сторон спецификации разрабатываемой прикладной системы: структур данных, потоков управления и функциональных преобразований

**47)Правильно разработанные программы должны не только удовлетворять своим функциональным** требованиям, но и обладать такими свойствами, как:

* повторная используемость;
* расширяемость;
* устойчивость к неправильным данным;
* системность.

Правильный объектно-ориентированный стиль программирования обеспечивает наличие этих свойств. Поясним это на примере свойства системности.

Программа обладает свойством *системности*, если она применима в качестве обобщенного оператора при "крупноблочном программировании". *Крупноблочное программирование* - это систематическое использование ранее разработанных крупных программных единиц (таких, как классы, подсистемы, или модули) при разработке новых программных систем.

Следующие рекомендации могут помочь разрабатывать классы, обладающие свойством системности.

* Методы должны быть хорошо продуманы.
* Методы должны быть понятны всем, кто их прочитает.
* Методы должны быть легко читаемы.
* Имена должны быть такими же, как и в модели.
* Методы должны быть хорошо документированы.
* Спецификации методов должны быть доступны.

**48. Содержание основных программных документов. (из интернета)**



**49.Проектирование модульной структуры .**

Первое, что необходимо сделать, начиная этап разработки системы, определить ее разбиение на некоторое количество компонентов - модулей. Модуль не является ни объектом, ни функцией; модуль - это набор (пакет) классов и отдельных объектов, подсистем, зависимостей, операций, событий и ограничений, которые взаимосвязаны и имеют достаточно хорошо определенный и по возможности небольшой интерфейс с другими модулями. Часто модуль включает одну подсистему, являясь ее реализацией. Модуль (подсистема) обычно определяется через службы, которые он обеспечивает. Службой называется набор взаимосвязанных функций, которые совместно обеспечивают какую-либо функциональность, например, выполнение ввода-вывода, отрисовку картинок, выполнение арифметических действий. Подсистема определяет согласованный способ рассмотрения одной из сторон прикладной задачи, для решения которой разрабатывается рассматриваемая система. Например, система файлов - подсистема операционной системы; она обеспечивает набор взаимосвязанных абстрактных операций, которые в большой степени (но не полностью) независимы от абстрактных операций, обеспечиваемых другими подсистемами. Эта подсистема может быть реализована в виде отдельного модуля.

Как уже отмечалось, каждая подсистема имеет хорошо определенный (внешний) интерфейс с остальной частью системы (другими подсистемами). Этот интерфейс определяет форму всех взаимодействий с подсистемой и все потоки данных через ее границы, но не специфицирует внутреннюю структуру и внутреннее окружение подсистемы, а также особенности ее реализации. Поэтому каждая подсистема может разрабатываться независимо от остальных подсистем.

Подсистемы должны определяться таким образом, чтобы большая часть взаимодействий оставалась внутри подсистем, для уменьшения глобальных потоков данных и сокращения зависимостей между подсистемами. Подсистем должно быть не очень много (в пределах десятка). Некоторые подсистемы могут быть в свою очередь подразделены на подсистемы.

Две подсистемы могут взаимодействовать друг с другом либо как клиент и поставщик (клиент-сервер), либо как равноправные партнеры (сопрограммы). В первом случае клиент вызывает сервер, который выполняет некоторый запрос клиента и возвращает результат; клиент должен знать интерфейс сервера, но сервер может не знать интерфейсов клиента, так как все взаимодействия инициируются клиентом. В случае сопрограммного взаимодействия обе подсистемы вызывают друг друга. Обращение подсистемы к другой подсистеме не обязательно связано с немедленным получением ответа. Сопрограммное взаимодействие является более сложным, так как обе подсистемы должны знать интерфейсы друг друга. Поэтому нужно стараться, чтобы большая часть подсистем взаимодействовала как клиент и сервер.

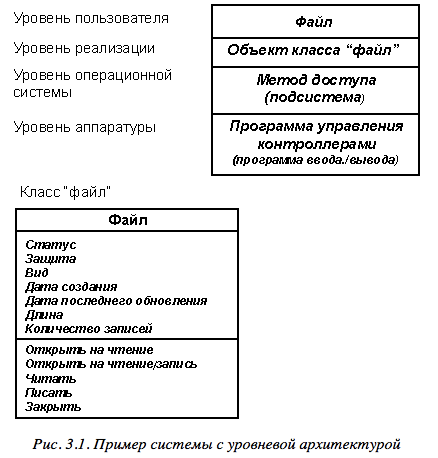
Подсистемы (и реализующие их модули) могут образовывать в системе уровни, либо разделы.

Уровни

Уровневая система может рассматриваться как упорядоченное множество виртуальных миров, каждый из которых построен на основе понятий, поддерживаемых его подсистемами; подсистемы одного уровня обеспечивают реализацию подсистем следующего уровня. Объекты каждого уровня могут быть независимыми, хотя нередко объекты разных уровней могут соответствовать друг другу. Каждая подсистема знает о подсистемах более низких уровней и ничего не знает о более высоких уровнях. Зависимость клиент-сервер существует между более верхним (клиент) и более нижними уровнями (серверы). При этом каждый уровень может иметь свой собственный набор классов и операций. Каждый уровень реализуется через операции объектов и подсистем более нижних уровней. Уровневые архитектуры бывают двух видов: открытые и замкнутые. В замкнутой архитектуре каждый уровень строится на базе непосредственно следующего за ним уровня. Это сокращает зависимости между уровнями и упрощает внесение изменений. В открытой архитектуре каждый уровень строится на базе всех следующих за ним уровней. Это уменьшает потребность в переопределении операций на каждом уровне и приводит к более эффективному и компактному коду. Однако открытая архитектура не удовлетворяет принципу упрятывания информации, поскольку изменения в какой-либо подсистеме могут потребовать соответствующих изменений в подсистемах более высоких уровней.

Обычно лишь подсистемы самого верхнего и самого нижнего уровней могут быть выведены из постановки задачи: самый верхний уровень - это требуемая система, а самый нижний уровень - это доступные ресурсы: аппаратура, операционная система, имеющиеся библиотеки. Промежуточные уровни вводятся разработчиком системы.

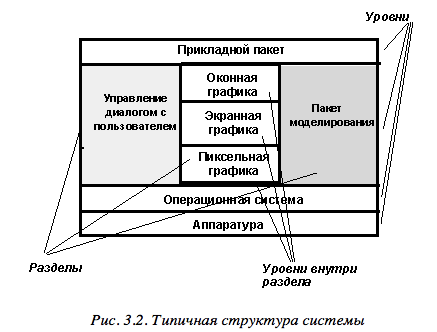
Система с уровневой архитектурой при переносе на другую платформу требует переписывания только одного (самого нижнего) уровня. Пример системы с уровневой архитектурой представлен на рисунке 3.1.



Разделы

Разделы подразделяют систему на несколько независимых или слабо связанных модулей (подсистем), каждая из которых обеспечивает один из видов услуг. Например, операционная система компьютера включает систему файлов, управление процессами, управление виртуальной памятью и управление устройствами.

Обычно система подразделяется на модули (подсистемы) с использованием обоих способов разбиения в самых разных комбинациях: уровни делятся на разделы, разделы содержат в себе уровни. Пример такой структуры системы показан на рисунке 3.2.





Топология системы

Когда все модули и подсистемы всех уровней названы, необходимо показать информационные потоки между модулями и подсистемами, построив ДПД (примеры см. в п. 3.1.8). Это позволит понять топологию системы. Топология системы определяется совокупностью потоков информации в системе; например, у компилятора конвейерная топология (рисунок 3.3), можно представить себе топологию звезды (рисунок 3.4) и т.п.; нужно стремиться, чтобы топология системы была как можно проще.

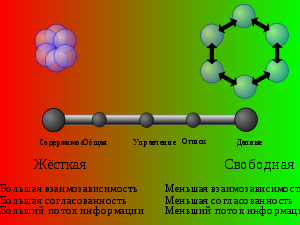
(http://citforum.ru/programming/oop\_rsis/glava3\_1\_1.shtml)

**50.Связанность и цельность модулей**

**Связанность** (или **зависимость)** — характеристика взаимосвязи модуля с другими модулями. Это степень, в которой каждый программный модуль полагается на другие модули.

Связанность обычно противопоставляется [связности](http://ru.wikipedia.org/wiki/%D0%A1%D0%B2%D1%8F%D0%B7%D0%BD%D0%BE%D1%81%D1%82%D1%8C_%28%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5%29) (Связность — характеристика *внутренней* взаимосвязи между частями *одного модуля. Иное название связности -* ***цельность***). Слабая связанность часто сочетается с сильной связностью и наоборот. [Метрика качества ПО](http://ru.wikipedia.org/wiki/%D0%9C%D0%B5%D1%82%D1%80%D0%B8%D0%BA%D0%B0_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D1%8F) связанности и связности была придумана [Ларри Константином](http://en.wikipedia.org/wiki/Larry_Constantine), изначальным разработчиком Структурного Дизайна, который был также ранним сторонником таких концепций (см. также [SSADM](http://en.wikipedia.org/wiki/Structured_Systems_Analysis_and_Design_Methodology)). Слабая связанность часто является признаком хорошо структурированной компьютерной системы и признаком хорошего дизайна, и, когда она комбинируется с сильной связностью, соответствует общим показателям хорошей читаемости и поддерживаемости.

## Типы связанности



*Концептуальная модель связанности*

Связанность может быть «низкой» (также «свободной» и «слабой») или «высокой» (также «несвободной» или «сильной»). Некоторые типы связанности по направлению от наивысшей к наинизшей связанности, следующие:

Связанность содержимого (высокая)

Связанность содержимого это когда один модуль изменяет или полагается на внутреннюю кухню другого модуля (например, доступ к локальным данным другого модуля).

Следовательно, изменение способа, которым второй модуль производит данные (расположение, тип, время) приведёт к необходимости изменять зависимый модуль.

Общая связанность

Общая связанность это когда два модуля делят между собой один и тот же глобальный набор данных (например, глобальную переменную). Изменение общего ресурса подразумевает изменение всех модулей, использующих его.

Внешняя связанность

Внешняя связанность возникает когда два модуля делят между собой определённый извне формат данных, протокол связи, или интерфейс устройства. Это обычно основано на коммуникации со внешними средствами и устройствами.

Связанность управления

Связанность управления это когда один модуль управляет исполнением другого путём передачи ему информации по поводу того, что делать (например, передачей флага «что-нужно-сделать»).

Связанность по отпечатку в структуре данных (stamp coupling)

Это когда модули делят между собой составную структуру данных, и каждый используют только её часть, по возможности даже не ту же самую часть (например, передача полной структуры в функцию, которой нужно только одно поле этой структуры). Это может привести к изменению способа чтения модулем этой структуры, когда изменится поле, которое модуль не использует.

Связанность данных

Связанность данных это когда модули делят общие данные через, скажем, параметры. Каждая порция данных это элементарный фрагмент, и это только те данные, которые используются совместно (например, передача целого числа функции, вычисляющей квадратный корень).

Связанность сообщений (низкая)

Это наислабейший тип связанности. Он может быть достигнут децентрализацией состояний (как в объектах) и коммуникацией компонентов через параметры или передачу сообщений.

Нет связанности

Модули не общаются между собой вовсе.

### *Связанность В ООП:*

Связанность подклассов

Описывает взаимоотношения между дочерним и родительским элементами. Дочерний элемент связан с родительским, но родительский не связан с дочерним.

Временнáя связанность

Когда два действия связаны вместе в один модуль только потому, что они происходят в одно и то же время.

**51. Критерии качества ПО**

**Качество программного обеспечения** — способность программного продукта при заданных условиях удовлетворять установленным или предполагаемым потребностям; весь объем признаков и характеристик программ, который относится к их способности удовлетворять установленным или предполагаемым потребностям.

ГОСТ 28195-89 содержит следующий список комплексных показателей качества программного продукта:

*1) надежность* – это его способность с достаточно большой вероятностью безотказно выполнять определенные функции при заданных условиях и в течение заданного периода времени;

*2) корректность*

*3) удобство применения -* эргономичность программы, легкость, с которой пользователь может использовать программу для своих целей;

*4) сопровождаемость*

*5) эффективность* – соотношение уровня услуг, предоставляемых ПО пользователю при заданных условиях, и объема используемых для этого ресурсов. К числу таких ресурсов могут относиться требуемые аппаратные средства, время выполнения программ, затраты на подготовку данных и интерпретацию результатов;

*6) универсальность.*

В международном стандарте ISO 9126 прописаны следующие показатели:

*1) эффективность -* количество ресурсов системы, потребляет программа (время процессора, размер памяти, внешняя память, ширина канала сети, а также взаимодействия с пользователем). Чем меньше ресурсов потребляется, тем лучше;

*2) надежность -* это его способность безотказно выполнять определенные функции при заданных условиях в течение заданного периода времени с достаточно большой вероятностью. При этом под отказом в ПС понимают проявление в нем ошибки. Надежное ПС не исключает наличия в нем ошибок - важно лишь, чтобы эти ошибки при практическом применении этого ПС в заданных условиях проявлялись достаточно редко.

*3) сопровождаемость -* это характеристики ПО, которые позволяют минимизировать усилия по внесению изменений для устранения в нем ошибок и по его модификации в соответствии с изменяющимися потребностями пользователей.

*4) функциональность -* это способность ПО выполнять набор функций, удовлетворяющих заданным или подразумеваемым потребностям пользователей. Набор указанных функций определяется во внешнем описании ПО.

*5) практичность*

*6) мобильность -* это способность ПС быть перенесенным из одной среды (окружения) вдругую, в частности, с одного компьютера на другой.

Комплексные показатели называют также факторами качества или характеристиками качества.

Каждому из комплексных показателей соответствует определенный набор критериев качества. В свою очередь, каждый из критериев определяется своими метриками, которые составляются из оценочных элементов, определяющих заданное в метрике свойство. На разных этапах ЖЦ для разных классов программных продуктов применяются разные критерии качества.

Показатели качества программного продукта можно разделить на две категории: *внутренние* и *внешние* . Внешние факторы качества (корректность, надежность, эффективность, удобство использования) могут быть определены, исходя из поведения программного продукта, и являются видимыми пользователю. В отличие от них внутренние показатели качества (переносимость, понятность, модифицируемость, сопровождаемость) для пользователя не видны и не важны. Однако эти показатели чрезвычайно важны для разработчиков, поскольку в конечном итоге именно они определяют экономические затраты на разработку требуемого программного продукта. Внутренние показатели более важны для проекта, чем внешние, поскольку внешние характеристики практически всегда от них зависят.

**52. Тестирование ПО. Методы «Белого ящика»**

Белый ящик - то, содержимое чего известно.

*«Белый ящик»* — тестирование кода на предмет логики работы программы и корректности её работы с точки зрения компилятора того языка, на котором она писалась.

Техника тестирования по принципу Белого ящика, также называемая *техникой тестирования, управляемой логикой программы,* позволяет проверить внутреннюю структуру программы. Исходя из этой стратегии тестировщик получает тестовые данные путеманализа логики работы программы.

Методы белого ящика основаны на знаниях о внутреннем устройстве проги, т.е. используя этот метод, тестируются алгоритмы, система управления кодом, сам код и др. внутренние механизмы.

Методы белого ящика начинают применять, когда приложение еще не закончено, т.е. в процессе разработки ПО.

Методы белого ящика направлены на локализацию ошибок, которые сложнее выявить, найти и зафиксировать. С их помощью можно *обнаружить логические ошибки и проверить степень покрытия тестами*.

Тестирование посредством белого ящика, как правило, включает в себя стратегию *модульного тестирования*, при котором тестирование ведется на модульном или функциональном уровне и работы по тестированию направлены на исследование внутреннего устройства модуля. На этом уровне тестирования проверяется управляющая логика, проявляющаяся на модульном уровне. Тестовые драйверы используются для того, чтобы все пути в данном модуле были проверены хотя бы один раз, все логические решения рассмотрены во всевозможных условиях, циклы были выполнены с использованием верхних и нижних границ и проконтролированы структуры внутренних данных.

Методы тестирования на основе стратегии белого ящика:

Ввод неверных значений. При вводе неверных значений тестировщик заставляет коды возврата показывать ошибки и смотрит на реакцию кода. Это хороший способ моделирования определенных событий, например переполнения диска, нехватки памяти и т.д.

Модульное тестирование. При создании кода каждого модуля программного продукта проводится модульное тестирование для проверки того, что код работает верно и корректно реализует архитектуру. При модульном тестировании новый код проверяется на соответствие подробному описанию архитектуры; обследуются пути в коде, устанавливается, что экраны, ниспадающие меню и сообщения должным образом отформатированы; проверяются диапазон и тип вводимых данных, а также то, что каждый блок кода, когда нужно, генерирует исключения и возвращает ошибки (еггог returns). Тестирование каждого модуля программного продукта проводится для того, чтобы проверить корректность алгоритмов и логики и то, что программный модуль удовлетворяет предъявляемым требованиям и обеспечивает необходимую функциональность. По итогам модульного тестирования фиксируются ошибки, относящиеся к логике программы, перегрузке и выходу из диапазона, времени работы и утечке памяти.

Тестирование обработки ошибок. При использовании этого метода признается, что нереально на практике проверить каждое возможное условие возникновения ошибки. По этой причине программа обработки ошибок может сгладить последствия при возникновении неожиданных ошибок. Тестировщик обязан убедиться в том, что приложение должным образом выдает сообщение об ошибке. Так, приложение, которое сообщает о системной ошибке, возникшей из-за промежуточного программного обеспечения представляет небольшую ценность, как для конечного пользователя, так и для тестировщика.

Утечка памяти. При тестировании утечки памяти приложение исследуется с целью обнаружения ситуаций, при которых приложение не освобождает выделенную память, в результате чего снижается производительность или возникает тупиковая ситуация.

Комплексное тестирование. Целью комплексного тестирования является проверка того, что каждый модуль программного продукта корректно согласуется с остальными модулями продукта. При комплексном тестировании может использоваться технология обработки сверху вниз и снизу вверх, при которой каждый модуль, являющийся листом в дереве системы, интегрируется со следующим модулем более низкого или более высокого уровня, пока не будет создано дерево программного продукта. Эта технология тестирования направлена на проверку не только тех параметров, которые передаются между двумя компонентами, но и на проверку глобальных параметров и, в случае объектно-ориентированного приложения, всех классов верхнего уровня.

Тестирование цепочек. Тестирование цепочек подразумевает проверку группы модулей, составляющих функцию программного продукта. Эти действия известны еще как модульное тестирование, с его помощью обеспечивается адекватное тестирование компонентов системы. Данное тестирование выявляет, достаточно ли надежно работают модули для того, чтобы образовать единый модуль, и выдает ли модуль программного продукта точные и согласующиеся результаты.

Исследование покрытия. При выборе инструмента для исследования покрытия важно, чтобы группа тестирования проанализировала тип покрытия, необходимый для приложения. Исследование покрытия можно провести с помощью различных технологий. Метод покрытия операторов часто называют С1, что также означает покрытие узлов. Эти измерения показывают, был ли проверен каждый исполняемый оператор. Данный метод тестирования обычно использует программу протоколирования (profiler) производительности.

Покрытие решений. Метод покрытия решений направлен на определение (в процентном соотношении) всех возможных исходов решений, которые были проверены с помощью комплекта тестовых процедур. Метод покрытия решений иногда относят к покрытию ветвей и называют С2. Он требует: чтобы каждая точка входа и выхода в программе была достигнута хотя бы единожды, чтобы все возможные условия для решений в программе были проверены не менее одного раза и чтобы каждое решение в программе хотя бы единожды было протестировано при использовании всех возможных исходов.

Покрытие условий. Покрытие условий похоже на покрытие решений. Оно направлено на проверку точности истинных или ложных результатов каждого логического выражения. Этот метод включает в себя тесты, которые проверяют выражения независимо друг от друга. Результаты этих проверок аналогичны тем, что получают при применении метода покрытия решений, за исключением того, что метод покрытия решений более чувствителен к управляющей логике программы.

**53. Тестирование ПО. Методы «Черного ящика»**

**Тестирование чёрного ящика** или **поведенческое тестирование** — стратегия (метод) тестирования функционального поведения объекта (программы, системы) с точки зрения внешнего мира, при котором не используется знание о внутреннем устройстве тестируемого объекта.

Под «чёрным ящиком» понимается объект исследования, внутреннее устройство которого неизвестно. Манипулируя только лишь со входами и выходами, можно проводить определенные исследования.

Принципы тестирования чёрного ящика

В этом методе программа рассматривается какчерный ящик (т.е. то, реализацию, внутреннее строение чего мы не знаем). Целью тестирования ставится выяснение обстоятельств, в которых поведение программы не соответствует спецификации. Для обнаружения всехошибок в программе необходимо выполнить исчерпывающее тестирование, то есть тестирование на всевозможных наборах данных. Для большинства программ такое невозможно, поэтому применяют разумное тестирование, при котором тестирование программы ограничивается небольшим подмножеством всевозможных наборов данных. При этом необходимо выбирать наиболее подходящие подмножества, подмножества с наивысшей вероятностью обнаружения ошибок.

### Свойства правильно выбранного теста

1. Уменьшает более, чем на одно число других тестов, которые должны быть разработаны для разумного тестирования.
2. Покрывает значительную часть других возможных тестов, что в некоторой степени свидетельствует о наличии или отсутствии ошибки до и после ограниченного множества тестов.

### Приёмы тестирования чёрного ящика

1. Эквивалентное разбиение.
2. Анализ граничных значений.
3. Анализ причинно-следственных связей.
4. Предположение об ошибке.

Рассмотрим подробнее каждый из этих методов:

#### Эквивалентное разбиение

Основу метода составляют два положения:

1. Исходные данные необходимо разбить на конечное число классов эквивалентности. В одном классе эквивалентности содержатся такие тесты, что, если один тест из класса эквивалентности обнаруживает некоторую ошибку, то и любой другой тест из этого класса эквивалентности должен обнаруживать эту же ошибку.
2. Каждый тест должен включать, по возможности, максимальное количество классов эквивалентности, чтобы минимизировать общее число тестов.

Разработка тестов этим методом осуществляется в **два этапа**: выделение классов эквивалентности и построение теста. Классы эквивалентности выделяются путём выбора каждого входного условия, которые берутся с помощью технического задания или спецификации и разбиваются на две и более группы.

Выделение классов эквивалентности является эвристическим способом, однако существует ряд правил:

1. Если входное условие описывает область значений, например «Целое число принимает значение от 0 до 999», то существует один правильный класс эквивалентности и два неправильных.
2. Если входное условие описывает число значений, например «Число строк во входном файле лежит в интервале (1..6)», то также существует один правильный класс и два неправильных.
3. Если входное условие описывает множество входных значений, то определяется количество правильных классов, равное количеству элементов в множестве входных значений. Если входное условие описывает ситуацию «должно быть», например «Первый символ должен быть заглавным», тогда один класс правильный и один неправильный.
4. Если есть основание считать, что элементы внутри одного класса эквивалентности могут программой трактоваться по-разному, необходимо разбить данный класс на подклассы. На этом шаге тестирующий на основе таблицы должен составить тесты, покрывающие собой все правильные и неправильные классы эквивалентности. При этом составитель должен минимизировать общее число тестов.

Определение тестов:

1. Каждому классу эквивалентности присваивается уникальный номер.
2. Если еще остались не включенные в тесты правильные классы, то пишутся тесты, которые покрывают максимально возможное количество классов.
3. Если остались не включенные в тесты неправильные классы, то пишут тесты, которые покрывают только один класс.

#### Анализ граничных значений

**Граничные условия** — это ситуации, возникающие на высших и нижних границах входных классов эквивалентности.

**Анализ граничных значений отличается от эквивалентного разбиения следующим:**

1. Выбор любого элемента в классе эквивалентности в качестве представительного осуществляется таким образом, чтобы проверить тестом каждую границу этого класса.
2. При разработке тестов рассматриваются не только входные значения (пространство входов), но и выходные (пространство выходов).

Метод требует определённой степени творчества и специализации в рассматриваемой задаче.

**Существует несколько правил:**

1. Построить тесты с неправильными входными данными для ситуации незначительного выхода за границы области значений. Если входные значения должны быть в интервале [-1.0 .. +1.0], проверяем −1.0, 1.0, −1.000001, 1.000001.
2. Обязательно писать тесты для минимальной и максимальной границы диапазона.
3. Использовать первые два правила для каждого из входных значений (использовать пункт 2 для всех выходных значений).
4. Если вход и выход программы представляет упорядоченное множество, сосредоточить внимание на первом и последнем элементах списка.

Анализ граничных значений, если он применён правильно, позволяет обнаружить большое число ошибок. Однако определение этих границ для каждой задачи может являться отдельной трудной задачей. Также этот метод не проверяет комбинации входных значений.

#### Анализ причинно-следственных связей

**Этапы построения теста:**

1. Спецификация разбивается на рабочие участки.
2. В спецификации определяются множество причин и следствий. Под причиной понимается отдельное входное условие или класс эквивалентности. Следствие представляет собой выходное условие или преобразование системы. Здесь каждой причине и следствию присваивается номер.
3. На основе анализа семантического (смыслового) содержания спецификации строится таблица истинности, в которой последовательно перебираются всевозможные комбинации причин и определяются следствия для каждой комбинации причин.

Таблица снабжается примечаниями, задающими ограничения и описывающими комбинации, которые невозможны. Недостатком этого подхода является плохое исследование граничных условий.

#### Предположение об ошибке

Тестировщик с большим опытом выискивает ошибки без всяких методов, но при этом он подсознательно использует метод предположения об ошибке. Данный метод в значительной степени основан на интуиции. Основная идея метода состоит в том, чтобы составить список, который перечисляет возможные ошибки и ситуации, в которых эти ошибки могли проявиться. Потом на основе списка составляются тесты

**54. Отладка ПО.**

Хотя слова «отладка» и «тестирование» часто используются как синонимы, под ними подразумеваются разные виды деятельности. Тестирование — это процесс выполнения его программ на некотором наборе данных, для которого заранее известен результат применения или известны правила поведения этих программ; отладка направлена на установление точной природы известной ошибки, а затем — на исправление этой ошибки. Эти два вида деятельности связаны — результаты тестирования являются исходными данными для отладки.

Таким образом, отладку можно представить в виде многократного повторения трех процессов: тестирования, в результате которого может быть констатировано наличие в ПО ошибки, поиска места ошибки в программах и документации ПО и редактирования программ и документации с целью устранения обнаруженной ошибки. Другими словами:

Отладка = Тестирование + Поиск ошибок + Редактирование. Поэтому важно составлять хорошие тесты для успешного выявления ошибок и их исправления.

Место отладки в цикле разработки программы:

Типичный цикл разработки, за время жизни программы многократно повторяющийся, выглядит примерно так:

1. Программирование — внесение в программу новой функциональности, исправление ошибок в имеющейся.
2. Тестирование (ручное или автоматизированное; программистом, тестером или пользователем; «дымовое», в режимечерного ящика илимодульное...) — обнаружение факта ошибки.
3. Воспроизведение ошибки — выяснение условий, при которых ошибка случается. Это может оказаться непростой задачей при программировании параллельных процессов и при некоторых необычных ошибках.
4. **Отладка** — обнаружение причины ошибки.

Выявленные программные ошибки, как правило, делятся на три вида:

1. ***Синтаксическая ошибка*** *.* Неправильное употребление синтаксических конструкций.
2. ***Семантическая ошибка*** *.* Нарушение семантики той или иной конструкции, например передача функции параметров, не соответствующих ее аргументам.
3. ***Логическая ошибка*** *.* Нарушение логики программы, приводящее к неверному результату. Это наиболее трудный для "отлова" тип ошибки, ибо подобного рода ошибки, как правило, кроются в алгоритмах и требуют тщательного анализа и всестороннего тестирования.
4. Поскольку безошибочное программирование почти невозможно, а ручная отладка немыслима, необходимы средства поиска ошибки (иногда это не так просто) и ее исправления. В каждой современной системе программирования существует специальное средство отладки программ — *отладчик* (debugger), который позволяет в режиме интерпретации установить контрольные точки, выполнить отдельные участки программы и посмотреть результаты работы операторов.

Ниже приводятся рекомендации по организации отладки в форме заповедей.

*Заповедь 1*. Считайте тестирование ключевой задачей разработки ПС, поручайте его самым квалифицированным и одаренным программистам; нежелательно тестировать свою собственную программу.

*Заповедь 2*. Хорош тот тест, для которого высока вероятность обнаружить ошибку, а не тот, который демонстрирует правильную работу программы.

*Заповедь 3*. Готовьте тесты как для правильных, так и для неправильных данных.

*Заповедь 4*. Документируйте пропуск тестов через компьютер; детально изучайте результаты каждого теста; избегайте тестов, пропуск которых нельзя повторить.

*Заповедь 5.* Каждый модуль подключайте к программе только один раз; никогда не изменяйте программу, чтобы облегчить ее тестирование.

*Заповедь 6.* Пропускайте заново все тесты, связанные с проверкой работы какой-либо программы ПС или ее взаимодействия с другими программами, если в нее были внесены изменения (например, в результате устранения ошибки).

**55.CASE-технологии разработки ПО**

CASE (англ. Computer-Aided Software Engineering) — набор инструментов и методов программной инженерии для проектирования программного обеспечения, который помогает обеспечить высокое качество программ, отсутствие ошибок и простоту в обслуживании программных продуктов. Также под CASE понимают совокупность методов и средств проектирования информационных систем с использованием CASE-инструментов.

Средства автоматизации разработки программ (CASE-средства) — инструменты автоматизации процессов проектирования и разработки программного обеспечения для системного аналитика, разработчика ПО и программиста. Первоначально под CASE-средствами понимались только инструменты для упрощения наиболее трудоёмких процессов анализа и проектирования, но с приходом стандарта ISO/IEC 14102 CASE-средства стали определять как программные средства для поддержки процессов жизненного цикла ПО.

Общая информация.

Основной целью CASE-технологии является разграничение процесса проектирования программных продуктов от процесса кодирования и последующих этапов разработки, максимально автоматизировать процесс разработки. Для выполнения поставленной цели CASE-технологии используют два принципиально разных подхода к проектированию: структурный и объектно-ориентированный.

Структурный подход предполагает декомпозицию (разделение) поставленной задачи на функции, которые необходимо автоматизировать. В свою очередь, функции также разбиваются на подфункции, задачи, процедуры. В результате получается упорядоченная иерархия функций и передаваемой информацией между функциями.

Структурный подход подразумевает использование определенных общепринятых методологий при моделировании различных информационных систем:

SADT (Structured Analysis and Design Technique);

DFD (Data Flow Diagrams);

ERD (Entity-Relationship Diagrams).

Существует 3 основных типа моделей, используемых при структурном подходе:

Функциональные модели;

Информационные модели;

Динамические модели.

Программные средства для реализации структурного подхода:

BPWin;

ERWin;

Business Studio;

IBM WEBsphere business modeler;

Sybase PowerDesigner;

Основным инструментом объектно-ориентированного подхода является язык UML — унифицированный язык моделирования, который предназначен для визуализации и документирования объектно-ориентированных систем с ориентацией их на разработку программного обеспечения. Данный язык включает в себя систему различных диаграмм, на основании которых может быть построено представление о проектируемой системе.

Примерами программных средств, реализующих данный подход, являются:

Rational Rose;

ARIS.

Классификация[править | править исходный текст]

В функции CASE входят средства анализа, проектирования и программирования программных средств, проектирования интерфейсов, документирования и производства структурированного кода на каком-либо языке программирования.[4]

CASE-инструменты классифицируются по типам и категориям.

Классификация по типам отражает функциональную ориентацию средств на те или иные процессы жизненного цикла разработки программного обеспечения, и, в основном, совпадают с компонентным составом крупных интегрированных CASE-систем, и включает следующие типы:

средства анализа — предназначены для построения и анализа модели предметной области;

средства проектирования баз данных;

средства разработки приложений;

средства реинжиниринга процессов;

средства планирования и управления проектом;

средства тестирования;

средства документирования.

Классификация по категориям определяет степень интегрированности по выполняемым функциям и включают — отдельные локальные средства, решающие небольшие автономные задачи, набор частично интегрированных средств, охватывающих большинство этапов жизненного цикла и полностью интегрированных средств, охватывающий весь жизненный цикл информационной системы и связанных общим репозиторием.

Типичными CASE-инструментами являются:

инструменты управления конфигурацией;

инструменты моделирования данных;

инструменты анализа и проектирования;

инструменты преобразования моделей;

инструменты редактирования программного кода;

инструменты рефакторинга кода;

генераторы кода;

инструменты для построения UML-диаграмм. (http://ru.wikipedia.org/wiki/CASE)