# Объектно–ориентированное программное обеспечение

## Критерии качества программного обеспечения

Для ПО различают 2 различных типа качества. Может быть обнаружен пользователями (быстродействие, интерфейс и т.д.) – это внешние факторы качества. Такие характеристики как модульность, читаемость понятны профессионалам разработчикам и являются внутренними факторами качества. В конечном счёте только внешние факторы имеют значение, но ключ к достижению внешних факторов спрятан во внутренних факторах. Рассмотрим основные внешние факторы качества ПО.

**Correctness** (корректность) – способность ПО точно выполнять задачи так, как они определены в спецификации. Это важнейшее качество, если она не соблюдена, то все остальные качества не имеют особого значения.

Первый шаг к соблюдению корректности – это точная спецификация тех. Требований к системе.

Методы обеспечения корректности являются многоуровневыми:

1. Приложение
2. Транслятор
3. Операционная система
4. Железо

Корректность каждого уровня основывается на том, что ниже лежащие уровни корректны.

**Robustness** (устойчивость) **–** способность ПО соответствующим образом реагировать на аварийные ситуации. Устойчивость дополняет корректность.

Корректность – поведение системы в случаях определенной спецификации. Корректность определяет то, что происходит за пределами спецификации.

Роль требований устойчивость – удостовериться, что в аварийных случаях не произойдет непоправимой ситуации.

Возможные реакции

* Сообщение об ошибке
* Корректное завершение работы
* Переход в режим постепенного вывода из работы

**Extendibility** (расширяемость) – лёгкость адаптации ПО к изменениям спецификации.

Расширяемость – это проблема масштаба. Для небольших программ она несложна. Для больших программ возникают сложности. Для улучшения расширяемости важны 2 принципа:

1. Простота архитектуры
2. Децентрализация

Чем более автономны модули – тем больше шансов, что изменения затронут небольшое их количество.

**Reusability** (повторное использование) – способность элементов ПО служить для построения различных приложений.

Повторное использование влияет на остальные аспекты качества и позволяет прилагать больше усилий к улучшению других факторов.

**Compatibility** (совместимость) – лёгкость сочетания одних программных элементов с другими.

Ключ к совместимости заключается в стандартных соглашениях на коммуникацию между элементами ПО. Стандартные форматы файлов, структуры данных, пользовательский интерфейс.

**Efficiency** (эффективность) – способность ПО как моно меньше зависеть от ресурсов оборудования.

Существуют 2 подхода к эффективности:

1. Сразу тратить усилия на оптимизацию
2. “Сделай правильно, прежде чем делать быстро”

**Portability** (переносимость) – легкость переноса ПО в различные программные аппаратные среды.

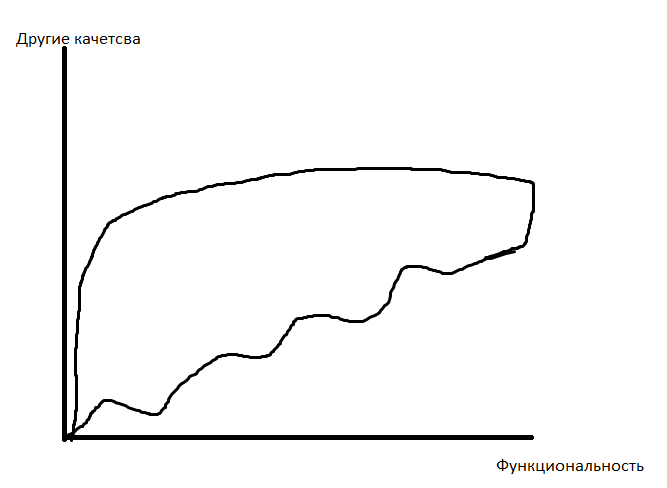
**Easy of use** (быстрота использования) – легкость, с которой с различными знаниями и квалификацией могут научить использовать программное обеспечение и применять его для решения задач. Сюда относится простота установки и текущего контроля.

**Functionality** (функциональность) – степень возможности обеспеченной системой.

Одна из самых сложных проблем – определение достаточной функциональности. Всегда существует желание добавить к проекту все новые и новые свойства.

Расширение системы приводит к 2 проблемам:

1. Потеря непротиворечивости
2. Слишком большое внимание к одним свойствам в ущерб другим качествам программы



*Кривые Осмонта*

**Timeliness** (своевременность) **–** выпуск ПО в нужный момент. Т.е. тогда или незадолго до того, когда у пользователей появилась соответствующая потребность.

**Verifiability** (верифицируемость) – Лёгкость подготовки процедур приёмки, процедур получения информации об отладке и трассировке ошибок.

**Integrity** (целостность) – способность ПО защищать свои компоненты от несанкционированного доступа и модификации.

**Reparability** (восстанавливаемость) – способность устранять наличие дефектов.

**Economy** (экономичность) – способность проекта завершиться, оставаясь в рамках бюджета.

### Компромиссы

Некоторые элементы конфликтуют друг с другом. Например, целостность с простотой использования, экономичность и функциональность, эффективность и переносимость. Повторное использование расширяет границы заданной спецификации и т.д. Поэтому приходится прибегать к различного рода компромиссам. Компромиссы могут затрагивать все факторы качества, кроме одного – корректности.

## Парадигма программирования

Парадигма программирования – это способ построения программ, основанный на определенных принципах программирования.

В структурной или процедурной парадигме основной абстракцией является алгоритм.

В логической парадигме основной абстракцией являются цели, обычно выраженные в терминах исчисления предикатов.

В объектно-ориентированной парадигме основной абстракцией являются классы и объекты.

Для объектной парадигмы концептуальная база - объектная модель.

Из 7 основных принципов, которые 4 являются главными (абстрагирование, инкапсуляция, модульность и иерархия) и 3 дополнительными (типизация, параллелизм и сохраняемость)

### Абстрагирование

Является одним из основных методов используемых в решениях сложных задач

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

Абстрагирование концентрирует внимание на внешних особенностях объекта и позволяет отделить самые существенные особенности объектов.

Клиентом называется любой объект, использующий ресурсы другого объекта (сервера)

Такой подход концентрирует внимание на внешних проявлениях объекта и приводит к идеи контрактной модели программирования, внешнее проявление объекта рассматривается с точки зрения его контракта с другими объектами в соответствии с этим должно быть выполнено и его внутреннее устройство.

Контракт фиксирует все обязательства, которые объект-сервер имеет перед объектом-клиентом.

Каждая операция, предусмотренная контрактом, однозначно определяется формальными параметрами и типом возвращаемого значения.

Полный набор операций вместе с правильным порядком вызова, называется протоколом.

Протокол отражает все возможные способы, которыми объект может действовать или подвергаться воздействию полностью определяя внешнее поведение абстракции со статической и динамической точек зрения.

Центральная идея абстракции является понятие инварианта.

Инвариант – это некоторое логическое условие, значение которого должно сохранятся

Для каждой операции объекта можно задать предусловие (инварианты предполагаемой итерации) и постусловие (инварианты которым удовлетворяет операция).

Изменение инварианта нарушает контракт связанный с итерацией. Если нарушено условие, то клиент не соблюдает свои обязательства и сервер не может выполнить задачу правильно, если нарушено постусловие, то свои обязательства нарушил сервер.

При нарушении какого-то условия вызывается исключительная ситуация.

Все абстракции обладают как статическими, так и динамическими свойствами

Например, файл имеет имя, размер и определённое содержание, эти атрибуты являются статическими, а конкретные значение этих атрибутов динамичны и меняются в процессе выполнения объекта.

Абстракция и инкапсуляция дополняют друг друга. Абстрагирование направлено на наблюдаемое поведение объекта, а инкапсуляция занимается внутренним устройством. Чаще всего инкапсуляция выполняется по средствам скрытия информации или маскировкой всех внутренних деталей, не влияющих на поведение объекта. Обычно вскрывают и внутреннюю структуру объекта, и реализацию его методов.

Инкапсуляция – это процесс отделение друг от друга элементов объекта, определяющие его устройство и поведение.

Инкапсуляция служит для того, чтобы изолировать контрактные обязательства абстракции от их реализации.

ПРИМЕР

Одной из ключевых абстракций в тепличном хозяйстве является нагреватель, можно ограничиться 3 действиями с этим объектом (включение/выключение/запрос состояния). Нагреватель не должен отвечать за поддержание температуры, эта абстракция более высокого уровня, совместно реализуемая датчиком, нагревателем и регулятором.

Во многих языках попытка обращения к закрытым данным выявляется на этапе компиляции, но путем манипуляции адресами, можно забраться во внутрь закрытой части объекта.

«Инкапсуляция защищает от ошибок, но не от жульничества»

## Инкапсуляция в Java script

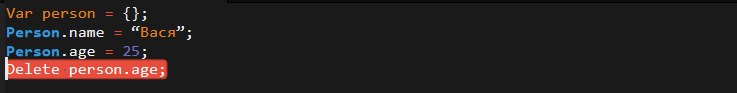
Пустой объект можно создать 2 способами:



В любой момент после создания объекта можно добавить к нему свойства путем простого присваивания.

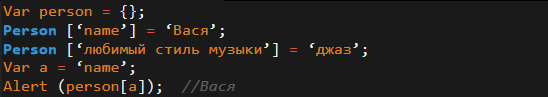


Можно удалить свойство:

Существует возможность обратиться по чтению к отсутствующему свойству, при этом ошибка не возвращается.

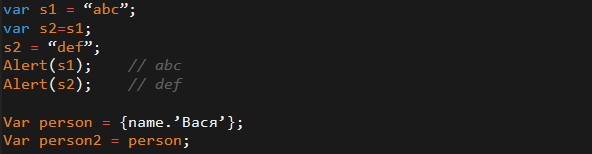


Существует альтернативный способ работы со свойствами использующий квадратные скобки.

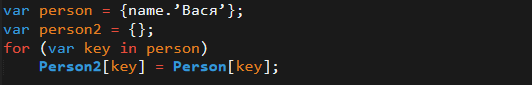


Объект можно заполнить значениями при создании:

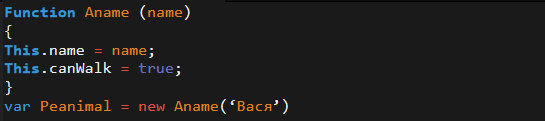
Объекты в отличии от других примитивов (строки, числа) копируются по ссылке:



Следует создать клон объекта



Часто необходимо создать группу одинаковых объектов для этого используют функции конструктора. Конструктором становиться любая функция, запущенная оператором new.



Функция, запущенная через new:

1. Создаётся пустой объект
2. This получает ссылку на этот объект
3. Функция выполняется
4. Возвращается this

В функции конструктора можно объявлять локальные переменные и вложенные функции, которые будут видны только изнутри.

## Иерархия

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

Иерархия – это упорядочение абстракций и расположение их по уровням.

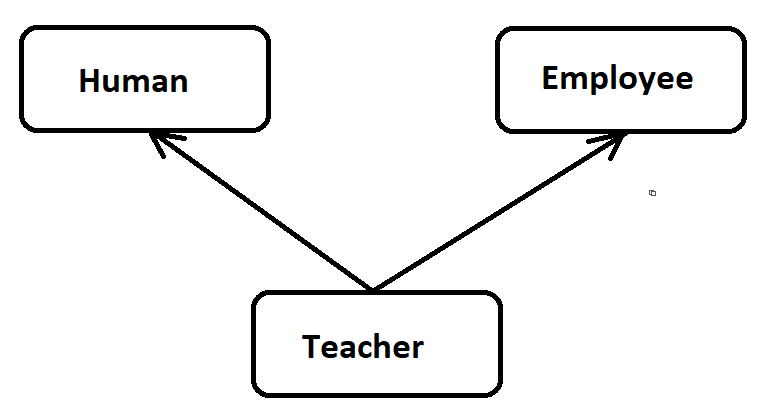
Существует 2 основных вида иерархических структур:

1. Иерархия классов (структура классов)
2. Иерархия объектов (структура объектов)

Важным элементом объектно-ориентированных систем и основным видом иерархии ИСЭ, является концепция наследования.

Наследование означает такое отношение между классами, когда один класс заимствует структурную и функциональные части одного или нескольких других классов. Иными словами, наследование создает такую абстракцию, в которой под классы наследуют строение от одного или нескольких суперклассов, изменяя и(или) добавляя новые элементы.

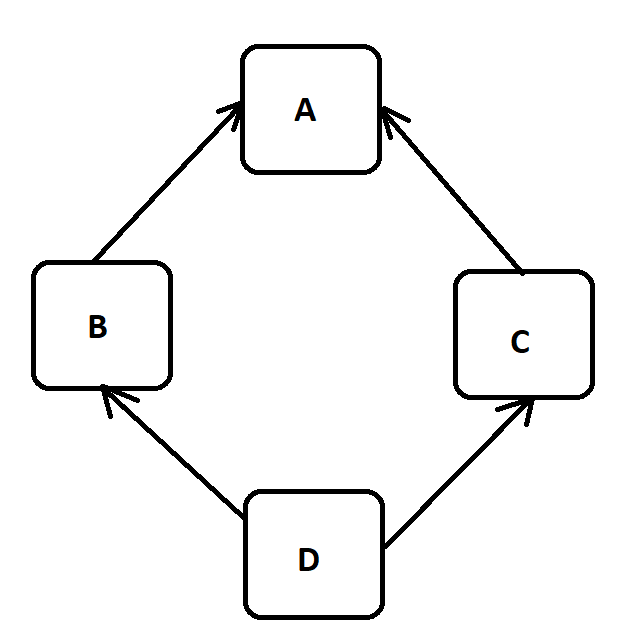
Суперклассы отображают более общие, а подклассы более специализированные абстракции. Принципы инкапсуляции и иерархии находятся в некотором противоречии. Инкапсуляция скрывает внутреннее строение объекта, а принцип наследования требует открыть туда доступ (компромиссом является protected(в плюсах)).



*#include* <string>  
  
*using namespace* std;  
  
*class* Human{  
 string m\_name;  
 *int* m\_age;  
*public*:  
 Human(string name, *int* age){  
 m\_name = name;  
 m\_age = age;  
 }  
 *int* getName(){  
 *return* m\_name;  
 }  
 *int* getAge(){  
 *return* m\_age;  
 }  
};  
  
Class Employee{  
 string m\_employez;  
 *double* m\_wage;  
*public*:  
 Employe(String employes, *double* wage){  
 m\_employez = employez;  
 m\_wage = wage;  
 }  
   
 string getEmployez(){  
 *return* m\_employez;  
 }  
   
 *double* getWage(){  
 *return* m\_wage;  
 }  
};  
  
Class Teacher: *public* Human, *public* Employee{  
 *int* m\_teachersGrade;  
*public*:  
 Teacher(string name, *int* age, string employez, *double* wage, *int* TeachersGrade) :  
 Human(name, age), Employee(employez, wage), m\_teachersGrade(teachersGrade){}  
};

Множественное наследование может привести к ряду проблем, которые могут заметно увеличить сложность программы и сделать кошмаром дальнейшую поддержку кода.

1. Может возникнуть неоднозначность, когда несколько родительских классов имеют одноименные методы.
2. Ромбовидное наследование или “Алмаз смерти”



## Наследование в JavaScript

### Функциональное наследование

В JS наследование можно реализовать 2 способами 1 из которых функциональное наследование или наложение конструкторов. Рассмотрим пример, определением базовый класс машина и определим кофеварку.



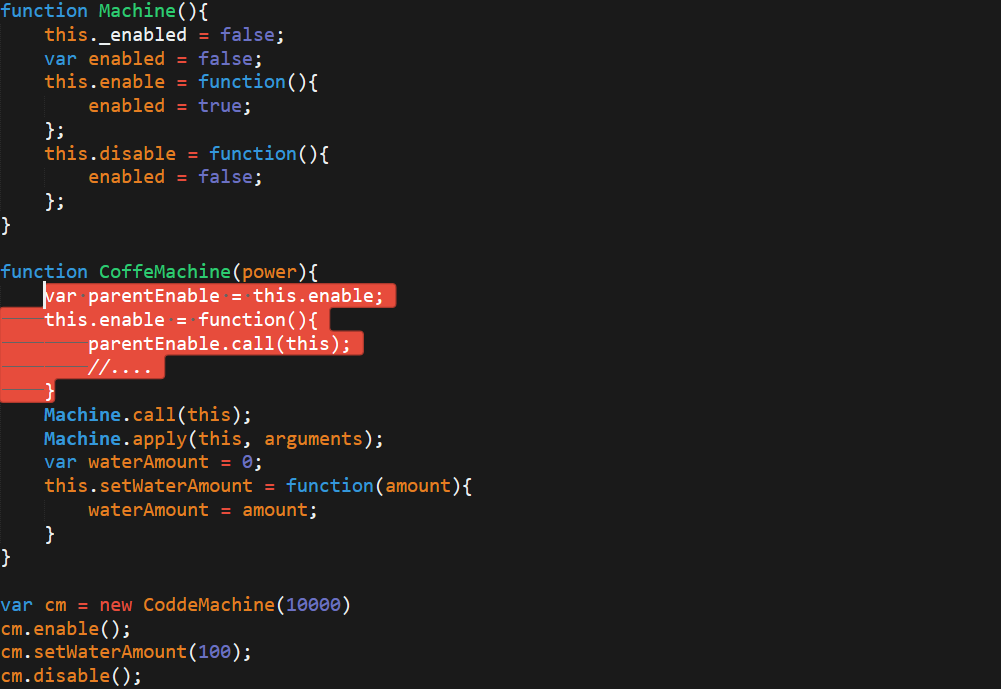
В данном примере есть проблема, наследник не имеет доступа к приватному свойству “enabled” в классе машина. Чтобы получить доступ к свойству – его нужно записать в this, а, чтобы подчеркнуть, что свойство является внутренним его имя, начинают с подчеркивания. Подчеркивание – общепринятый знак что свойство является внутренним предназначенным лишь для доступа из самого объекта и его наследников.



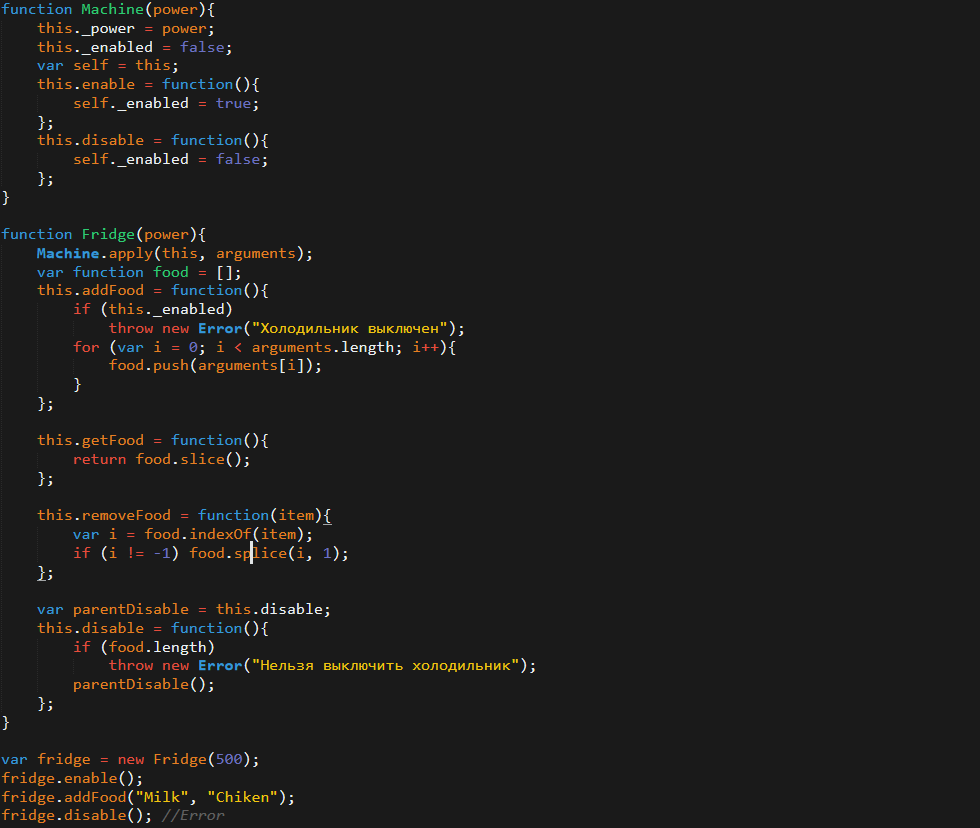
Предположим, что необходимо переопределить метод родителя, например, для кофеварки необходимо переопределить метод enable.



Но чаще необходимо доопределить метод.

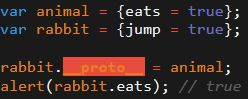


Рассмотрим более крупный пример “Холодильник”



### Наследование в прототипном стиле

Объекты JS можно организовывать в цепочки так, чтобы свойство, ненайденное в 1 объекте автоматически искалось по-другому. Связующим звеном выступает специальное свойство \_\_proto\_\_. Если 1 объект имеет ссылку \_\_proto\_\_ на другой объект, то при чтении свойства из него, если свойство отсутствует в самом объекте, оно ищется в объекте \_\_proto\_\_.

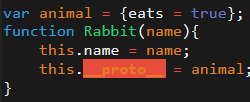


Объект, на который указывает ссылка \_\_proto\_\_ является прототипом.

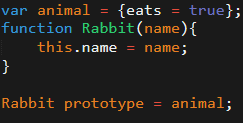
Прототип используется исключительно при чтении. Запись значения работает напрямую с объектом.

Используя \_\_proto\_\_ можно строить достаточно длинные цепочки объектов.

Рассмотрим задание прототипа при создании объекта через функцию конструктора

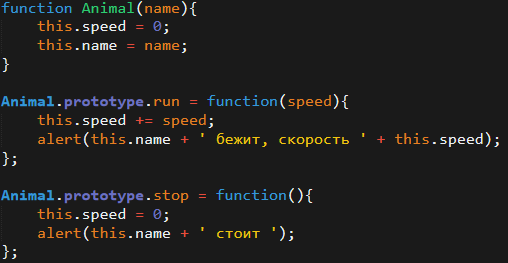


С этим способом были проблемы в Internet explorer. Поэтому есть 2 вариант



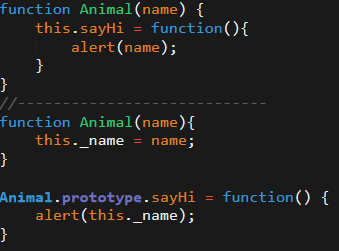


Функциональный стиль



Прототипный стиль

В результате в объекте хранятся только свойства, а общие методы в прототипе (достоинства портативного стиля). Функциональный стиль записывает свойства и методы, а прототипный только свойства. Поэтому прототипный стиль стребует меньше ресурсов. При создании методов через прототип теряется возможность использовать локальные переменные как приватные свойства. У них нет общей видимости с конструктором.



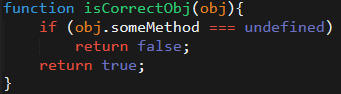
Если иерархия is a определяет отношение обобщения специализация, то отношение part of вводит иерархию агрегация. Агрегация есть во всех языках, использующих структуры и записи, состоящие из разнотипных данных. Агрегация позволяет физически сгруппировать. Логически связанные структуры, а наследование копирует эти общие группы в различные абстракции. В связи с агрегацией возникает проблема владения или принадлежности объектов.

Рассмотрим огород и растущие в нем растения. При уничтожении огорода возможны 2 решения.

1. Уничтожить растения
2. Растения можно пересадить в этом случае огород и растения имеют свои отдельные и независимые сроки жизни.

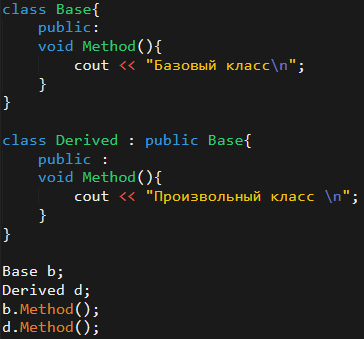
## Типизация

Это способ защититься от использования объектов 1 класса вместо другого или управлять таким использованием. Языком программирования может иметь сильную или слабую типизацию, оставаясь объектно-ориентированным. В сильнотипизированных языках нарушение согласования типов обнаруживается во время трансляции, в слаботипизированных обычно проявляется ошибка исполнения (или не проявляется).



Следует различать сильную и статическую типизации. Сильная типизация следит за соответствием типов. А статическая определяет время, когда имена связываются с типами. Статическая связь означает, что типы всех переменных и выражений известны при компиляции, а динамическое (позднее) связывание означает, что типы не известны до времени выполнения программы.

Пример

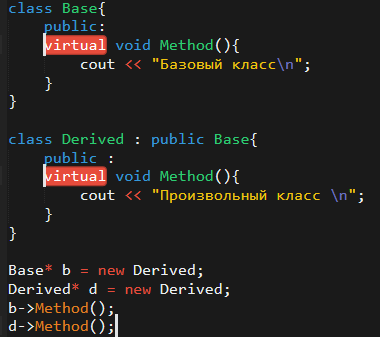


На этапе компиляции память выделяется для 2 версий функции Method и адреса привязываются к именам Base::Method() и Derived::Method().



Вывод будет таким же.

Чтобы сработало позднее связывание нужно функцию член класса метод объявить как виртуальную.



Теперь дважды будет выведен произвольный класс.

Для виртуальных методов память выделяется так же на этапе компиляции, но адреса их не привязываются к именам. Когда в базовом классе объявляется хотя бы одна виртуальная функция, то для всей иерархии создается таблица виртуальных функций. Это одномерный массив указателей на функции, количество элементов массива = количеству виртуальных функций в классе. При создании объекта таблица заполняется адресами виртуальных методов в соответствии с созданным типом.

Способность родительского класса представлять любого из своих наследников называется полиморфизмом. Полиморфизм возникает там, где взаимодействует наследование и динамическое связывание.

## Параллелизм

Есть задачи, в которых много событий должны обрабатываться одновременно. В других случаях при вычислении единственной задачи требуется большая вычислительная мощность. В каждой из этих задач естественно использовать несколько компьютеров, либо многозадачность на одном компьютере. Процесс(потокоуправление) это фундаментальная единица действия в системе. Каждая программа имеет по крайней мере 1 поток управления, параллельная система – имеет множество потоков. Реальная многозадачность достигается только на многопроцессорных системах. А в однопроцессорных только имитируется за счёт алгоритмов деления выполнения. Следует различать “тяжелую” и “легкую” потребность. Тяжелые задачи выполняются независимо друг от друга, под них выделяется защищенное адресное пространство. Легкие сосуществуют в едином адресном пространстве. Параллелизм основное внимание уделяет абстрагированию и синхронизации процесса. Каждый объект, полученный из абстракции реального мира может представлять собой отдельный поток управления(абстракцию процесса). Такой объект называют активным. Для систем, построенных на основе ООД (д-дизайн(проектирование)) – мир может быть представлен как совокупность взаимодействующих объектов, часть з которых может быть активной и выступает в роли независимых вычислительных центров. На этой основе можно дать определение параллелизма. Параллелизм позволяет различным объектам действовать одновременно. Параллелизм – это свойство, отличающее активные объекты от пассивных. ООД есть 3 подхода к параллелизму:

1. Параллелизм – внутреннее свойство языков программирования.
2. Использовать библиотеку классов, реализующих параллелизм.
3. Реализация многозадачности с использованием прерывания.

Как только в систему введен параллелизм возникает проблема синхронизации объектов.

## Сохраняемость

Любой программный объект существует в памяти и живет во времени. Время жизни может быть совершенно различным. Например: Промежуточные результаты вычисления выражения, локальные переменные подпрограмм, глобальные переменные и динамические данные. Данные, сохраняющиеся между сеансами выполнения программы, данные, сохраняемые при переходе на новую версию программы, данные, которые переживают программу.

Сохраняемость – способность объекта существовать во времени, переживать породившие его процессы или перемещаясь из 1 адресного пространства в другое.

## Качество классов и объектов

При проектировании процесс выделения классов и объектов является последовательным, итерационным. За исключением самых простых задач с 1 раза не удается окончательно выделить и описать классы. Но очень важно с самого начала по возможности приблизиться к правильным решениям. Для оценки качества классов и объектов, выделяемых в системе можно предложить следующие критерии:

1. *Зацепление*. Его можно определить, как степень глубины связи между отдельными модулями. Система сильного зацепления сложнее воспринимается и модифицируется. Кроме зацепления между модулями существенно зацепление между классами и объектами. Существуют противоречия между зацеплением и наследованием. С одной стороны, желательно избегать сильного зацепления классов, с другой стороны механизм наследования помогает выгодно использовать сходства абстракции.
2. *Связность*. Это степень взаимодействия между элементами отдельного модуля. Наименее желательна связность по случайному принципу, когда в 1 модуле собираются совершенно независимые абстракции. Наиболее желательной является функциональная связность, при которой все элементы модуля тесно взаимодействуют при достижении определенной цели.
3. *Достаточность*. Под достаточностью понимается наличие в классе или модуле всего необходимого для реализации логичного и эффективного поведения. Например, класс множества содержит операцию удаления элемента, но было бы ошибкой не включить в класс операцию добавления элемента.
4. *Полнота*. Под полнотой подразумевается наличие в интерфейсной части класса всех характеристик абстракций. Полнота – субъективный фактор и разработчики иногда ошибочно выносят наверх операции, которые можно было реализовать на более низком уровне.
5. *Примитивность*. Примитивными являются такие реализации, которые имеют доступ к внутренней реализации абстракции. Например операция добавления элемента в множество – примитивна, а операция добавления нескольких элементов не примитивна, т.к. реализуется через операцию добавления 1 элемента.

## Ключевые абстракции и механизмы

Ключевая абстракция – класс или объект, который входит в словарь предметной области. Главная ценность ключевых абстракций в том, что они определяют границы проблем. Определяют, что важно и устраняют лишнее. Определение ключевых абстракций включает в себя 2 процесса:

* Открытие – мы открываем абстракции, слушая специалистов по предметной области.
* Изобретение – изобретение абстрактных классов и объектов, полезных при проектировании и реализации системы.

Желательно по возможности при выделении ключевых абстракций сводить задачи к уже имеющимся классам и объектам.

Определив новые абстракции необходимо найти им место среди существующих классов и объектов. Либо найдется место в существующих иерархиях, либо придется строить дополнительную новую иерархию.

Иногда потребуется перестройка существующих иерархий. Иногда, найдя важный класс следует переместить его вверх по иерархии, увеличивая степень повторности использования кода (это называется продвижением класса).

Иногда наоборот оказывается, что класс слишком обобщен и необходима его детализация. Очень важно в обозначении классов и методов отражать сущность описываемых ими предметов.

## Идентификация механизмов

Механизмами называют структуры, посредством которых объекты взаимодействуют друг с другом и ведут себя так, как требуется. Механизмы служат для задания поведения совокупности объектов и представляют собой шаблоны поведения.

## Процесс ООД

### Процесс ООД делится на 2 части:

* Микропроцесс проектирования
* Макропроцесс проектирования

### Микропроцесс проектирования

Микропроцесс ООД приводится в движение потоком сценариев и решений, которые порождаются и последовательно уточняются макропроцессе.

С точки зрения программиста микропроцесс – это принятие большого числа ежедневных тактических решений, которые являются частью процесса создания и подгонки архитектуры системы.

С точки зрения архитектора микропроцесс является основой для развития архитектуры и опробования альтернатив.

В микропроцессе традиционные фазы анализа и проектирования перемешаны.

### Выявление связей между классами и объектами

Цель выявления связи между классами и объектами – уточнить границы обнаруженных ранее абстракций и опознать все сущности, с которыми она взаимодействует. Данный этап необходим, чтобы специфицировать взаимодействие и осуществить группирование классов и модулей в подсистеме. Основными результатами данного шага являются диаграммы классов, объектов и модулей. Нет необходимости создавать исчерпывающий набор диаграмм, необходимо сосредоточиться на самых важных отношениях между абстракций. В случае использования параллельного программирования на данном шаге принимаются решения о распределении процессов п процессорам. Данная фаза считается завершенной, когда достаточно полно определены семантика и связи абстракций, чтобы на следующем шаге можно было приступить к началу реализации. Меры качества – связность, зацепление и полнота.

### Реализация классов и объектов

На этапе анализа реализация классов и объектов необходима, чтобы довести существующие абстракции до уровня, достаточного для обнаружения новых классов и объектов на новом уровне абстракции. Они сами будут в дальнейшем поданы на новую итерацию микропроцесса. При проектировании с целью реализации создаются осязаемые представления абстракций путем выпуска последовательных исполнимых версий системы (макропроцесс). На данном шаге принимаются решения о представлении каждой абстракции и об отображении этих абстракций в физическую модель.

С реализацией связан выбор структур и алгоритмов, которые представляют семантику определенных ранее микропроцессом абстракций. В отличии от первых 3 стадий сосредоточенных на внешних представлениях абстракций данный этап акцентирует внимание на их внутреннее представление. Данный этап считается завершенным, когда получена исполнимая абстрактная модель абстракций. Главные критерий – простота сложной реализации свидетельствует о недостатках абстракции.

## Макропроцесс проектирования

Макропроцесс является контролирующим по отношению к микропроцессу. Макропроцесс описывает ряд измеряемых результатов и действий, которые позволяют команде разработчиков оценить риск, вести заблаговременно изменения в микропроцесс и сосредоточиться на коллективном анализе и корректировании. Таким образом макропроцесс – это деятельность коллектива в масштабе от недель до месяцев, многие элементы макропроцесса относятся к практике менеджмента программных продуктов и выполняются одинаково для ОО и других систем. Например, управления организации, контроля качества, составления документации. Заказчика не волнуют нюансы программирования, заказчик более обеспокоен сроками, качеством, полнотой и правильностью программы. Поэтому макропроцесс концентрирован на управление рисками и выявление общей архитектуры.

### Концептуализация

Концептуализация должна установить основные требования к системе. Первичными продуктами концептуализации являются прототипы системы. Необязательно, что прототипы будут эволюционировать в готовую систему. Концептуализация – творческая деятельность и она не должна быть скована жесткими правилами разработки. Когда для исследования выбрано конкретное направление. Типичен следующий порядок действий.

1. Определить основные цели и решить какие критерии того, что достигнут благополучный исход.
2. Создать подходящую команду для разработки прототипа.
3. Оценить готовый прототип и принять решении о проектировании конечного продукта или о дальнейшем исследовании.

Концептуализация не содержит ничего от ОО но разработка прототипа может происходить быстрее в ОО среде.

Важно, чтобы при оценке прототипа были установлены критерии. Работу над прототипом чаще планируют по срокам, чем по требованиям.

### Анализ

Цель анализа – дать описание задаче. Описание должно быть полным, непротиворечивым, пригодным для чтения всеми заинтересованными сторонами. Реально проверяемым. Таким образом, цель анализа – представить модель системы, объяснить, что делает система, но не то, как она это делает.

На стадии анализа семантика функциональных точек передается через сценарии. Результатом анализа должно быть описание назначения системы, сопровождаемое ее характеристиками и перечислениями требуемых ресурсов.

В ОО проектировании реализуют такие описания с помощью сценариев. Каждый сценарий представляет одну функциональную точку. Побочный результат анализа – оценка риска и выявление опасных мест. Данная фаза будет завершена, когда появится уточненный и подписанный сценарий для всех типов поведения системы.