ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ

27 августа 2015

А.В. Якушин

Лаборатория ОИТ

Факультет ВМиК

МГУ им. М.В. Ломоносова

Что такое хорошая программа?

Качества хорошей программы

Правильно работает

Хорошо документирована

Можно модифицировать

Устойчиво работает

Длительное использование

Отвечает принципам

Уровни программиста

-1 может разработать программу по алгоритму

0 может написать программу, которой может пользоваться кто-то другой

Технологии в программировании

- Принципы
- Стиль оформления кода
- Чтение текста программ
- Анализ кода
- Повторное использование кода
- Системы контроля версий
- Методология SOLID
- Регулярные выражения

Принципы

- DRY
- KISS
- YAGNI
- Бритва Оккама
- SOLID

Чтение текста программ

- How not to program in C++
- Андрей Богатырев. Хрестоматия по программированию на Си в Unix
- В.В. Подбельский «Программирование на языке Си/С++»

Чтение текста программ

- Чтение кода это базовый навык.
- Программист всегда больше читает код, чем пишет его

Что делает эта программа?

```
Var
N, A, B, C: Integer;
Begin
Write ('Введите двузначное число > ');
Readln (N);
A:=N mod 10;
B:=N div 10;
C:=B-A;
Writeln ('Результат > ', C);
End.
```

Что делает эта программа?

- int i = 5;
- i = ++i + ++i;

Что делает эта программа?

```
1. int main()
2. {
3.  int i1 = 12; // A number
4.  int i2 = 3; // Another number

5.  if (i1 & i2)
6.  std::cout << "Both numbers are non-zero\n";
7.  else
8.  std::cout << "At least one number is zero\n";
9.  return (0);
10. }
```

Что делает эта программа?

1. var a: array[1..10]of integer;
2. i: integer;
3. Begin
4. for i :=1 to 10 do a[i]:=i;
5. for i :=1 to 10 do a[i+1]:=a[i];
6. end.

Анализ кода

Виды анализа кода

Динамический анализ

- Тестирование спецификаций (black box)
- Структурное тестирование (white box)
- Интеграционное тестирование
- Приемочное тестирование

Виды анализа кода

Статический анализ кода

- Формальная верификация
- Инспекция кода

Преимущества статического анализа

- Может использоваться на незавершенной программе
- Более эффективен чем тестирование.
- Формальная верификация более надежна чем тестирование.

Преимущества динамического анализа

- Статический анализ проверяет, что код делает, а не что он должен делать.
- Динамический анализ проверяет соответствие кода заявленному функционалу.
- Заказчик может оценить степень соответствия ПО своим ожиданиям.

Формальная верификация

Формальная верификация является доказательством того, что программа соответствует спецификации.

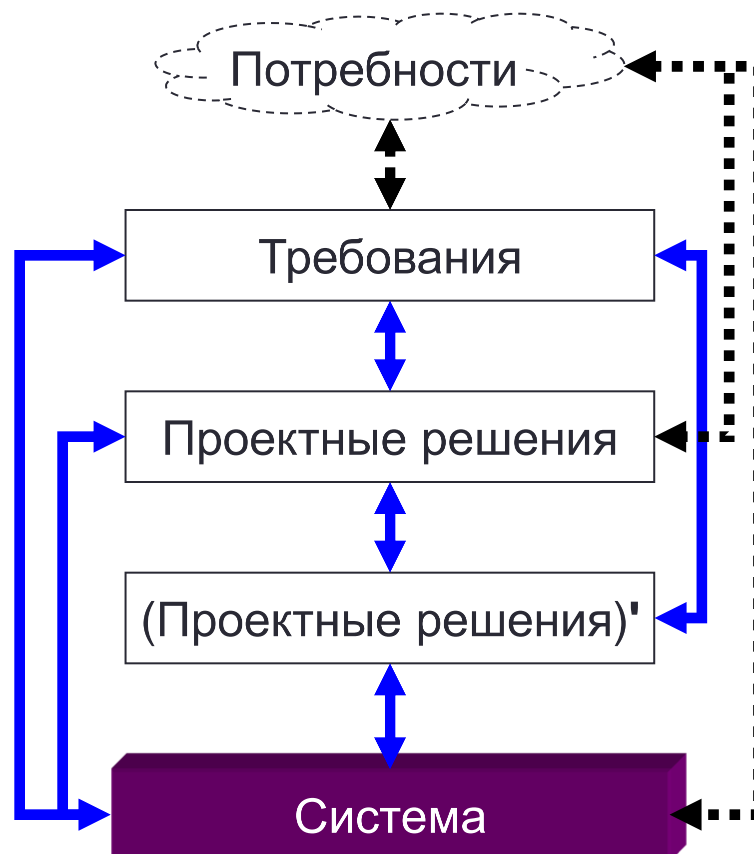
- высокоформализована
- требует специальной подготовки
- сложна в эксплуатации

Плюсы

- программы доказуемы
- новый часто основывается на проверенном
- ошибки малочисленны

Верификация

- Верификация – проверка корректности ***результатов*** некоторого этапа разработки по отношению к ***требованиям***, сформулированным на предыдущих этапах
- Валидация



Примеры формальных языков

- Логические
  - Логика 1 порядка
  - Лямбда-исчисление
- Операционные
  - Машина Тьюринга
  - Сети Петри

Принципы формальной верификации

- preconditions
- postconditions
- loop variants
- loop invariants
- class invariants

Пример 1

```
from S
-- Setup Code
invariant I
-- Invariant
variant v
-- Variant
until C
-- Stopping Condition
loop B
-- Loop Body
end
```

Реализация

- i = 0;
- s = a[0];
- while (i < a.length - 1) {
- /* * Invariant: s is the smallest element in the set * {a[0], a[1], ..., a[i]} * Variant: a.length - i - 1 */
- i++;
- s = min(s, a[i]); }

from
i := a.lower
s := a.item (i)
invariant
-- s is the smallest element in the set
-- {a.item (a.lower), ..., a.item (i)}
variant
a.upper - i
until
i = a.upper
loop
i := i + 1 s := s.min (a.item (i)) end

Статический анализ

- Приведение типов данных
- Инициализация переменных
- Неиспользуемые переменные
- Размер и сложность кода
- И т.п.

Рефакторинг

- Рефакторинг — это процесс улучшения написанного ранее кода путем такого изменения его внутренней структуры, которое не влияет на внешнее поведение.

Повторное использование кода

- Повторное использование кода (англ. code reuse) — методология проектирования компьютерных и других систем, заключающаяся в том, что система (компьютерная программа, программный модуль) частично либо полностью должна составляться из частей, написанных ранее компонентов и/или частей другой системы, и эти компоненты должны применяться более одного раза (если не в рамках одного проекта, то хотя бы разных). Повторное использование — основная методология, которая применяется для сокращения трудозатрат при разработке сложных систем.

Технология

Существующий модуль 1

Существующий модуль 2

Новая программа

Существующий модуль 4

Существующий модуль 3

Системы контроля версий

Зачем нужен контроль версий?

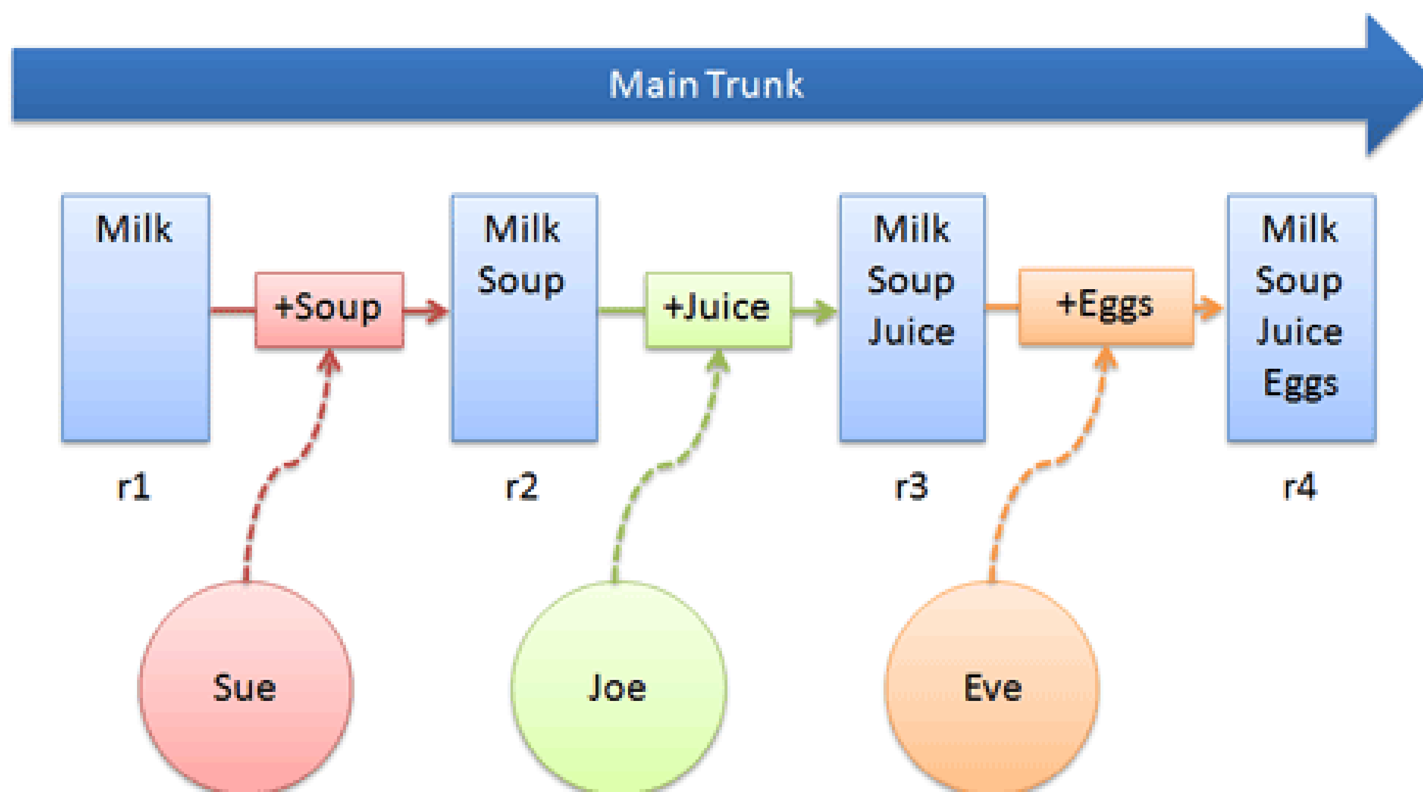
Системы контроля версий (VCS) это инструмент для управления программным кодом, позволяющий:

- *Отслеживать и хранить изменения в коде*
- *Объединять изменения*
- Создавать несколько версий проекта
- И другое.

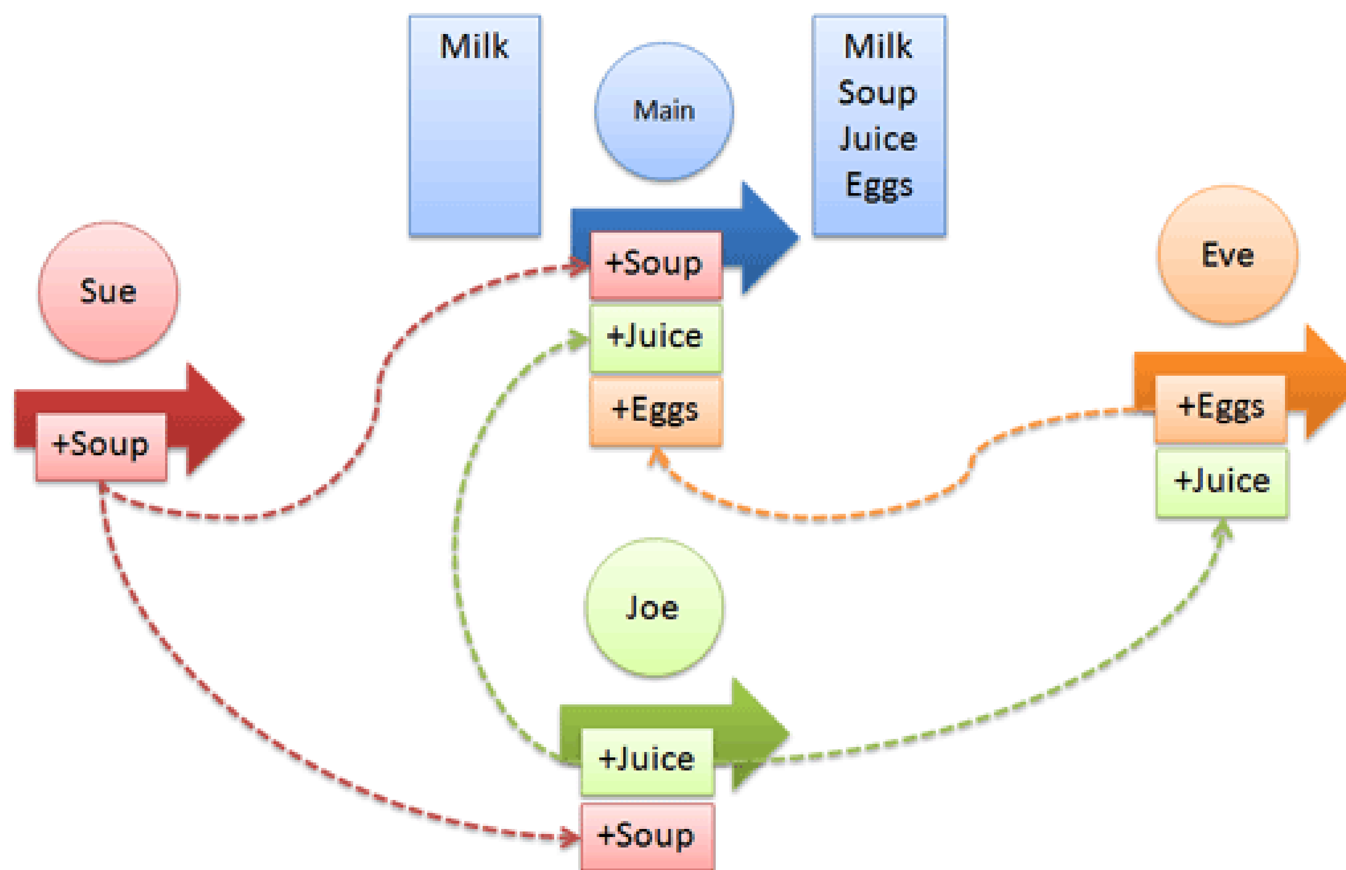
Виды систем контроля версий

- Централизованные
- Распределенные

Централизованные (CVS, Subversion)



Распределенные (Git, Mercurial)



Принципы SOLID

- Р. Мартин «Чистый код»

- Принцип единственной ответственности (Single responsibility)
- Принцип открытости/закрытости (Open-closed)
- Принцип подстановки Барбары Лисков (Liskov substitution)
- Принцип разделения интерфейса (Interface segregation)
- Принцип инверсии зависимостей (Dependency Invertion)

SINGLE RESPONSIBILITY PRINCIPLE

Just Because You Can, Doesn't Mean You Should

Принцип единственной ответственности (Single responsibility)

- *На каждый объект должна быть возложена одна единственная обязанность*

Множественная ответственность

- function getSumm:integer;
- Begin
- Readln(a,b);
- getSumm:=a+b;
- End;

Божественные функции/классы/методы

```
public class OrderService
{
    public Order Get(int orderId) { ... }
    public Order Save(Order order) { ... }
    public Order SubmitOrder(Order order) { ... }
    public Order GetOrderByName(string name) { ... }
    public void CancelOrder(int orderId) { ... }
    public void ProcessOrderReturn(int orderId) {...}
    public IList<Order> GetAllOrders { ... }
    public IList<Order> GetShippedOrders { ... }
    public void ShipOrder { ... }
}
```

OPEN CLOSED PRINCIPLE
Open Chest Surgery Is Not Needed When Putting On A Coat

Принцип открытости/закрытости (Open-closed)

- программные сущности должны быть открыты для расширения, но закрыты для модификации

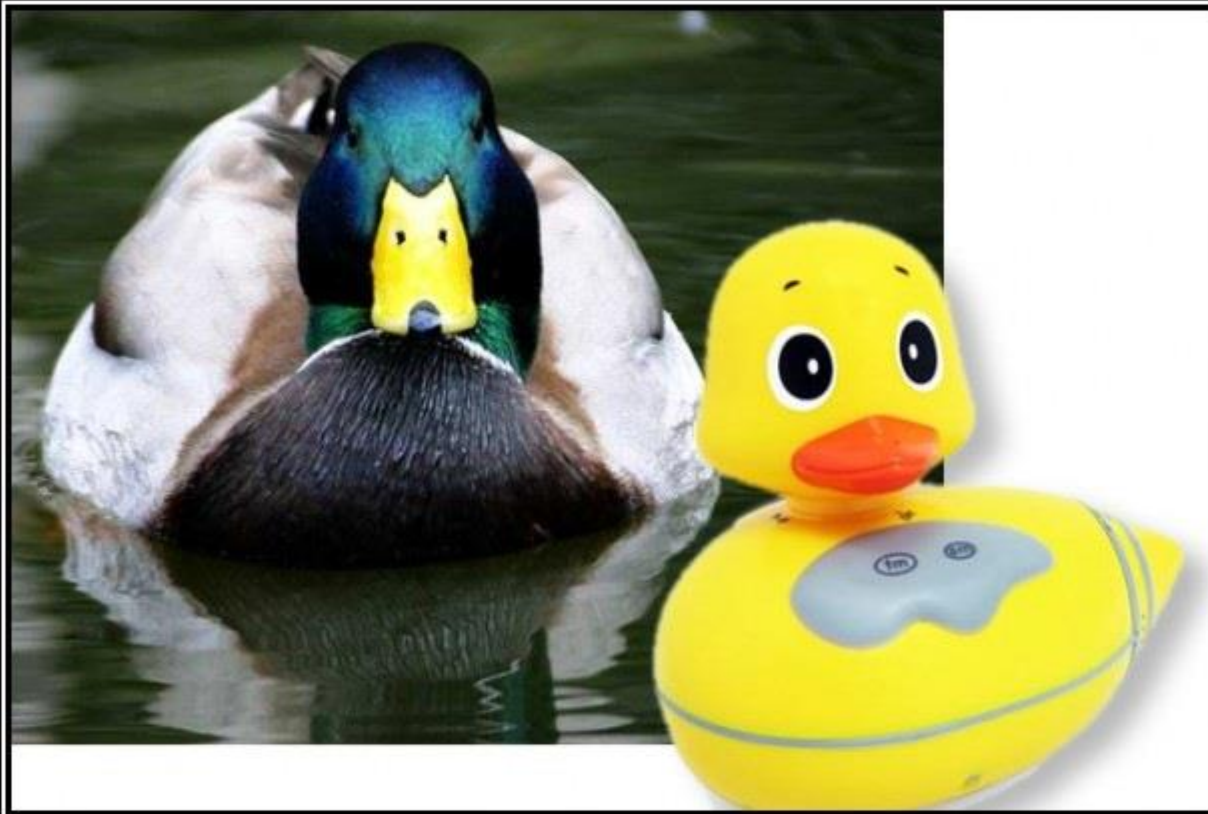
Пример

- Функция проверяет ответ пользователя

- Function checkResponse(response:string):boolean;
- Begin
-   if (response=“Yes”)or(response=“Y”) then return true;
-   if (response=“No”)or(response=“N”) then return false;
- End;

Решение проблемы

- Function checkResponse(
- trueAns,
- falseAns:array of string;
- response:string):boolean;
- Begin
- if checkTrue(trueAns,response) then return true;
- if checkTrueFalse(FalseAns,response) then return false;
- End;


LISKOV SUBSTITUTION PRINCIPLE
If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction

Принцип замещения Барбары Лисков (Liskov substitution)

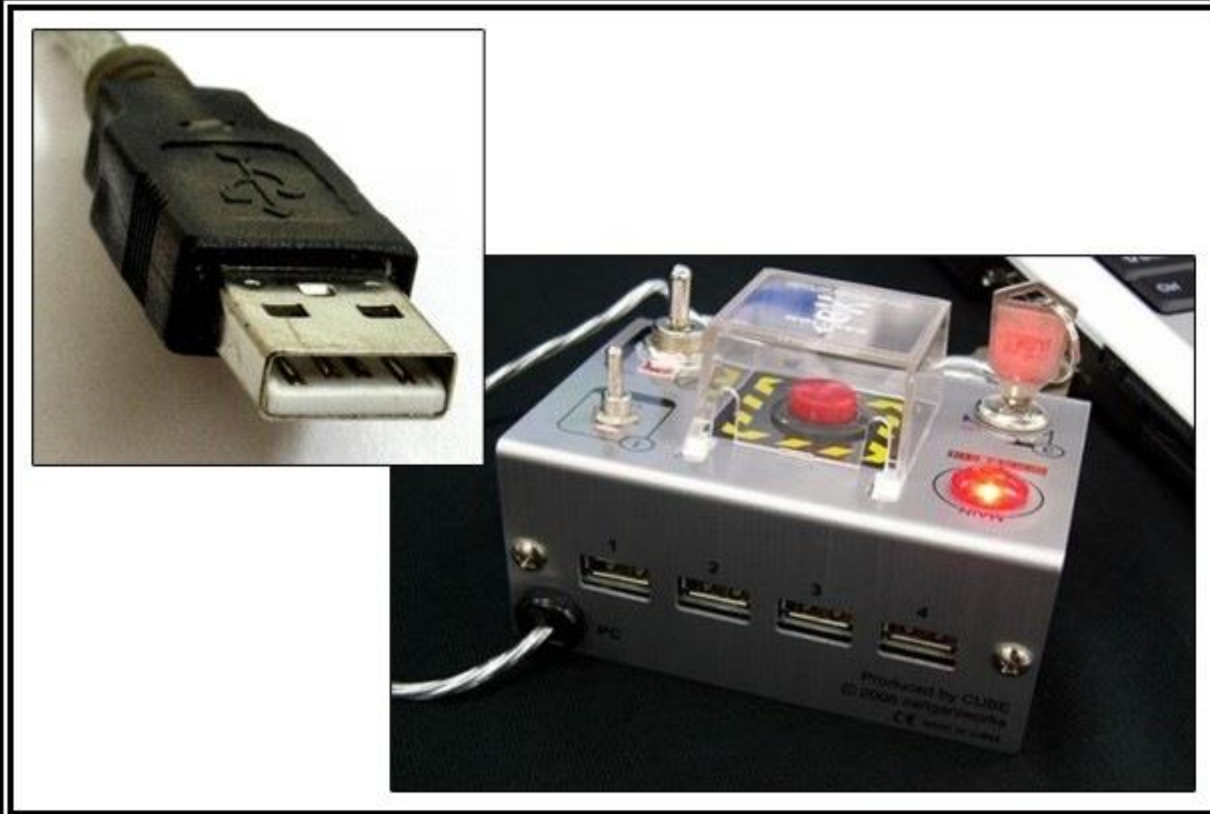
- если для каждого объекта o_1 типа S существует объект o_2 типа T, который для всех программ P определен в терминах T, то поведение P не изменится, если o_2 заменить на o_1 при условии, что S является подтипом T.

Смысл этого принципа

- Если потомком четрырехугольника является бублик, то вы используете неверную абстракцию

- Ключ->Открыть()
- Открыть()=Вставить()+Повернуть()+Вытащить()

- Магнитная карта создает проблемы!



INTERFACE SEGREGATION PRINCIPLE

You Want Me To Plug This In, Where?

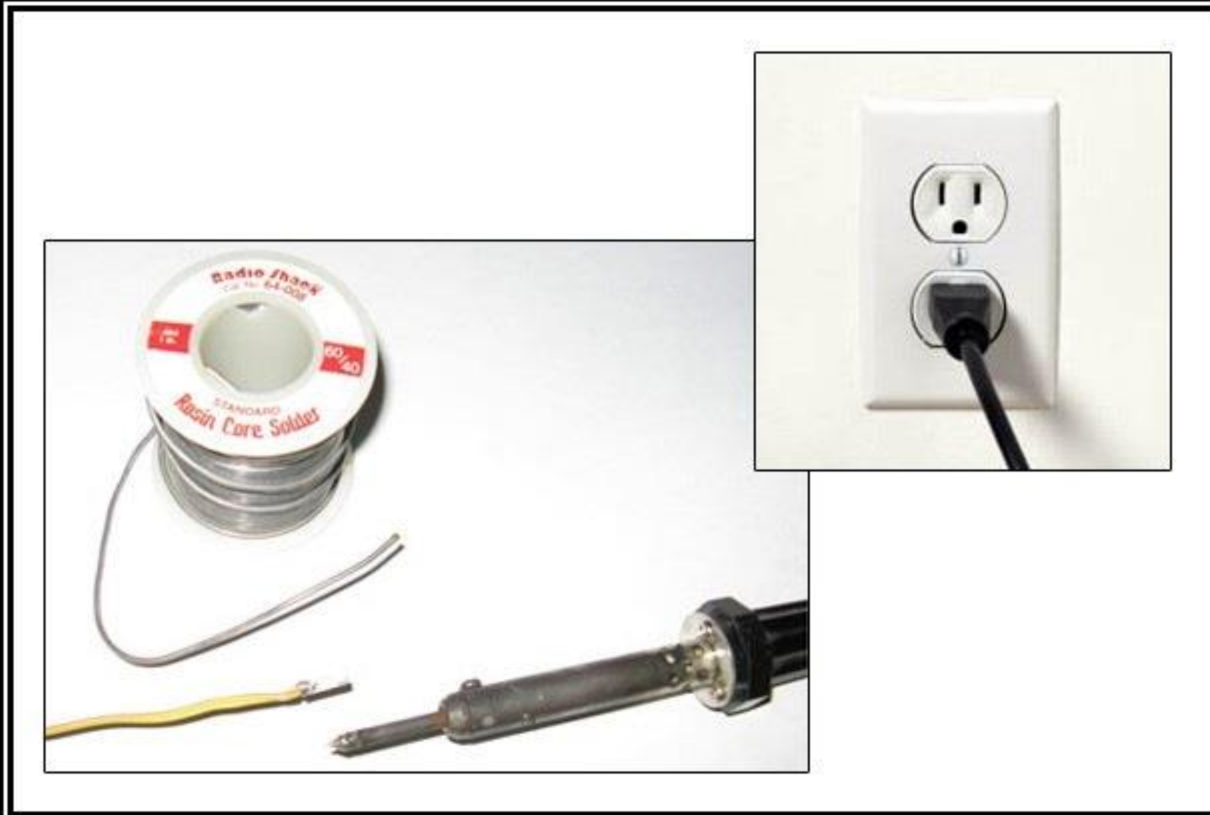
Принцип разделения интерфейса (Interface segregation)

- *Много специализированных интерфейсов лучше, чем один универсальный*
- **клиенты не должны зависеть от методов, которые они не используют**
- **Сходные действия выполняются сходными способами, различные действия выполняются различными способами**

Пример

- Продукт
  - Материал
  - Цена
  - Стоимость
  - Скидка

- Сок
- DVD диск
- Книга
- Морковь



DEPENDENCY INVERSION PRINCIPLE

Would You Solder A Lamp Directly To The Electrical Wiring In A Wall?

Принцип инверсии зависимостей (Dependency Invertion)

- *Зависимости внутри системы строятся на основе абстракций. Модули верхнего уровня не зависят от модулей нижнего уровня. Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций*
- *зависимости должны строится относительно абстракций, а не деталей*

Пример

- A -> B ->C ->A

Для чего нужны «Регулярные выражения»

- **Регулярные выражения** – это один из способов поиска подстрок (соответствий) в строках.
- Применение регулярных выражений дает значительное увеличение производительности
- Обычно с помощью регулярных выражений выполняются три действия:
  - Проверка наличия соответствующей шаблону подстроки.
  - Поиск и выдача пользователю соответствующих шаблону подстрок.
  - Замена соответствующих шаблону подстрок.

В каких языках реализованы «Регулярные выражения»

- Наибольшее развитие регулярные выражения получили в Perl, где их поддержка встроена непосредственно в интерпретатор.
- В VBScript и JScript используется объект RegExp, в C/C++ можно использовать библиотеки Regex++ и PCRE (Perl Compatible Regular Expression).
- Для Java существует целый набор расширений – ORO , RegExp, Rex и gnu.regexp.
- Microsoft Visual Studio.Net класс RegExp.

Типы «машин регулярных выражений»

- **DFA** (Deterministic Finite-state Automaton – детерминированные конечные автоматы)
- **Традиционные NFA-машины** (NonDeterministic Finite-state Automaton – недетерминированные конечные автоматы)
- **POSIX NFA** - машины похожи на традиционные NFA-машины.

Какие бывают «опции»

- I Поиск без учета регистра.
- M Многострочный режим, позволяющий находить совпадения в начале или конце строки, а не всего текста.
- R Ищет справа налево.

Метасимволы

\ - считать следующий метасимвол как обычный символ.

^- начало строки

. - один произвольный символ. Кроме '\n' - конец строки.

$ - конец строки

| - альтернатива (или)

() - группировка

[] - класс символов

Метасимволы

\w Слово. То же, что и [a-zA-Z_0-9].

\W Все, кроме слов. То же, что и [^a-zA-Z_0-9].

\s Любое пустое место. То же, что и [\f\n\r\t\v].

\S Любое непустое место. То же, что и [^ \f\n\r\t\v].

\d Десятичная цифра. То же, что и [0-9].

\D Не цифра. То же, что и [^0-9].

Метасимволы для последовательностей

\w+ - слово

\d+ - целое число

[+-]?\d+ - целое со знаком

[+-]?\d+\.?\d* - число с точкой

Мнимые метасимволы

\b - граница слова

\B - не граница слова

\A - начало строки

\Z - конец строки

\G - конец действия m//g

Квантификаторы, они же умножители (Quantifiers)

- * Соответствует 0 или более вхождений предшествующего выражения. Например, 'zo*' соответствует "z" и "zoo".
- + Соответствует 1 или более предшествующих выражений. Например, "zo+" соответствует "zo" and "zoo", но не "z".
- ? Соответствует 0 или 1 предшествующих выражений. Например, 'do(es)?' соответствует "do" в "do" or "does".
- {n} n – неотрицательное целое. Соответствует точному количеству вхождений. Например, 'o{2}' не найдет "o" в "Bob",но найдет два "o"' в "food".
- {n,} n – неотрицательное целое. Соответствует вхождению, повторенному не менее n раз. Например, 'o{2,}' не находит "o" в "Bob", зато находит все "o" в "foooood". 'o{1,}' эквивалентно 'o+'. 'o{0,}' эквивалентно 'o*'.
- {n,m} m и n – неотрицательные целые числа, где n <= m. Соответствует минимум n и максимум m вхождений. Например, 'o{1,3} находит три первые "o" в "fooooood". 'o{0,1}' эквивалентно 'o?'. Пробел между запятой и цифрами недопустим.

«Жадность»

- Важной особенностью квантификаторов '*' и '+' является их всеядность. Они находят все, что смогут – вместо того, что нужно.
- Излечить квантификатор от жадности можно, добавив '?'.

Вариации и группировка

- Символ '|' можно использовать для перебора нескольких вариантов. Использование этого символа совместно со скобками – '(...|...|...)' – позволяет создать группы вариантов.

Квантификаторы

- *? - станет 0 и более
- +? - 1 и более
- ?? - 0 или 1 раз
- {n}? - точно n раз
- {n,}? - не меньше n раз
- {n,m}? - больше или равно n и меньше m раз

Дополнительные переменные

$1, $2, …

$+ - обозначает последнее совпадение

$& - все совпадение

$` - все до совпадения

$' - все после совпадения

Правила регулярного выражения

- Любой символ обозначает себя самого, если это не метасимвол. Если вам нужно отменить действие метасимвола, то поставьте перед ним '\'.
- Строка символов обозначает строку этих символов.
- Множество возможных символов (класс) заключается в квадратные скобки '[]', это значит, что в данном месте может стоять один из указанных в скобках символов. Если первый символ в скобках это '^' - значит ни один из указанных символов не может стоять в данном месте выражения. Внутри класса можно употреблять символ '-', обозначающий диапазон символов. Например, a-z - один из малых букв латинского алфавита, 0-9 - цифра и т.д.
- Все символы, включая специальные, можно обозначать с помощью '\' как в языке С.
- Альтернативные последовательности разделяются символом '|' Заметьте что внутри квадратных скобок это обычный символ.
- Внутри регулярного выражения можно указывать "подшаблоны" заключая их в круглые скобки и ссылаться на них как '\номер' Первая скобка обозначается как '\1'.

Например,

- $str=~/perl/; проверяет, есть ли в строке $str подстрока "perl"
- $str=~/^perl/; проверяет, начинается ли строка с подстроки "perl"
- $str=~/perl$/; проверяет, заканчивается ли строка на подстроку "perl"
- $str=~/c|g|i/; проверяет, содержит ли строка символ 'c' или 'g' или 'i'
- $str=~/cg{2,4}i/; проверяет, содержит ли строка символ 'c', следующие сразу за ним 2-4 символа 'g', за которыми следует символ 'i'

Например,

- $str=~/cg*i/; проверяет, содержит ли строка символ 'c', слудующие за ним 0 или больше символа 'g', за которыми следует символ 'i'
- $str=~/c..i/; проверяет, содержит ли строка символ 'c', и символ 'i', разделенные двумя любыми буквами
- $str=~/[cgi]/; проверяет, содержит ли строка один из символов 'c', 'g' или 'i'
- $str=~/\d/; проверяет, содержит ли строка цифру
- $str=~/\W/; проверяет, содержит ли строка символы, не являющиеся буквами латинского алфавита и цифрами

Пример

```
$string="chmod 755 test.cgi";
 if($string=~/[a-z]+\s\d+\s.*/) {
        print "верно";
        }

#    'chmod' -- [a-z]+   - 1 или больше букв
#        ' ' -- \s       - 1 пробел
#      '755' -- \d+      - 1 или больше цифр
#        ' ' -- \s       - 1 пробел
#  'test.cgi' -- .*      - дальше идут любые символы
```

Пример: проверка e-mail адреса

```
$string="billgates@microsoft.com";
 if($string=~/\w+\@\w+\.\w+/) {
        print "верно";
        }

# 'billgates' -- \w+       - 1 или больше букв/цифр
#         '@' -- \@        - символ @
# 'microsoft' -- \w+       - 1 или больше букв/цифр
#         '.' -- \.        - символ .
#       'com' -- \w+       - 1 или больше букв/цифр
```

Операции

- Поиск шаблона – match
- Замена шаблона – replace
- Разбиение по шаблону - split