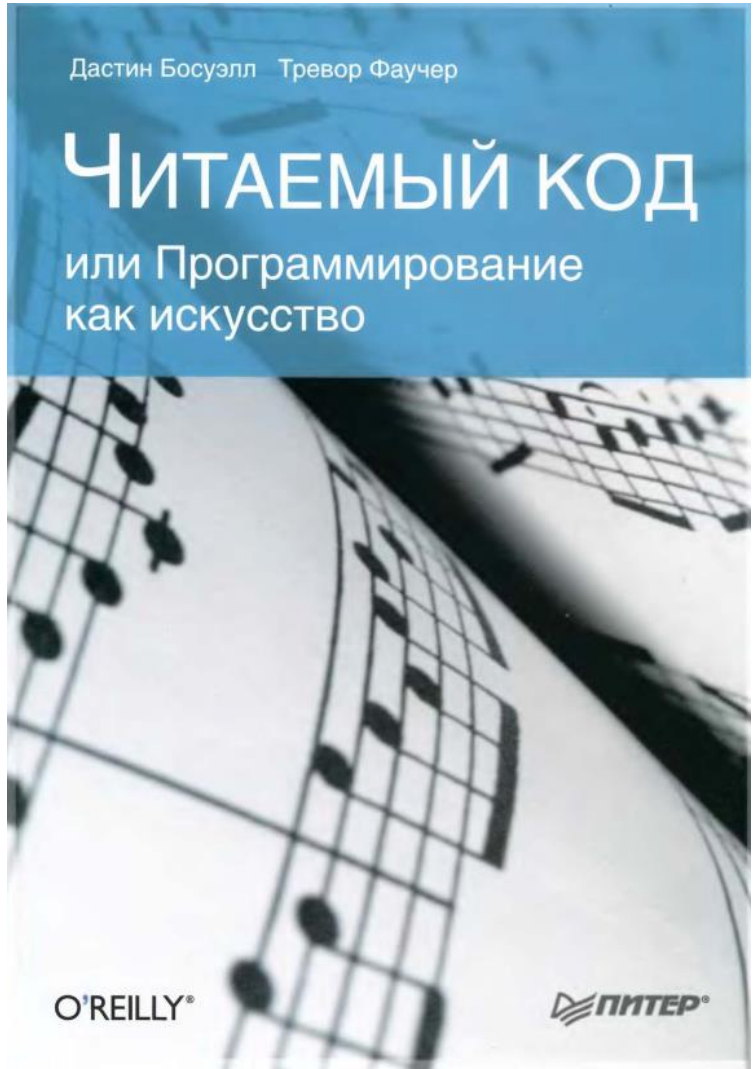
Читаемый код

Источник:



Код должен быть простым для понимания

Какой код понятнее ?

- Node* node = list->head;

```
if (node == NULL) return;
while (node->next != NULL) {
Print(node->data);
node = node->next;
}
if (node != NULL) Print (node->data)
```

- for (Node* node = list->head; node != NULL; node = node->next)
  Print(node->data);

(оба кода работают одинаково)

Какой код понятнее ?

1

```
return exponent >= 0 ? mantissa * (1 << exponent) : mantissa/  (1 << -exponent)
```

2

```
if (exponent >= 0){
  return mantissa * (1 << exponent);
  } else {
  return mantissa / (1 << -exponent);
 }
```

Фундаментальная теорема читаемости

Код должен быть написан так, чтобы можно было ***максимально быстро понять, как он работает.***

Меньше – значит лучше?

Чем меньше кода используется для решения проблемы, тем лучше!

Скорее всего, потребуется меньше времени для того, чтобы разобраться в 2000-строчном классе, чем в 5000-строчном.

Но меньшее количество строк не всегда делает код лучше:

1) assert((!(bucket = FindBuket(key))) || ! buket->IsOccupied());

2) bucket = FindBuket(key);
if (bucket != NULL) assert (! buket->IsOccupied());

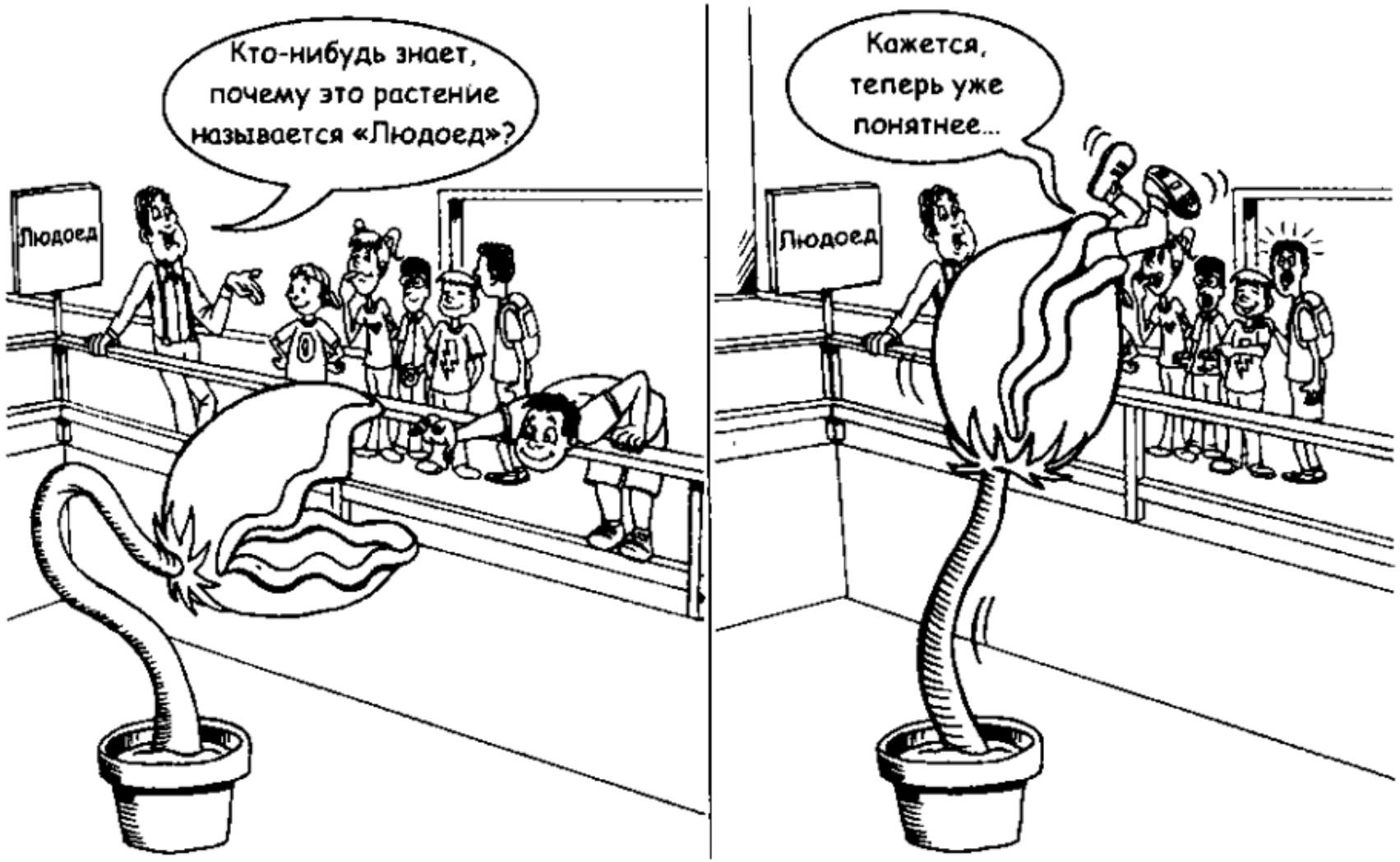
Выход: используйте комментарии:

//Версия кода “hash = (65599*hash) + C”, которая выполняется быстрее.
hash = (hash << 6) + (hash << 16) – hash + C;

Вывод: меньшее количество кода – это хорошая цель, однако важнее стремиться с сокращению времени-для-понимания

УЛУЧШЕНИЯ

Помещайте в имена полезную информацию:



УЛУЧШЕНИЯ

Когда вы придумываете имя для переменной, функции или класса, вы руководствуетесь примерно одинаковыми для всех программистов принципами:

- отнеситесь к имени как к небольшому комментарию;
- не используйте tmp где не надо;
- называйте вещи конкретно

(вместо stop() можно использовать kill() или pause())

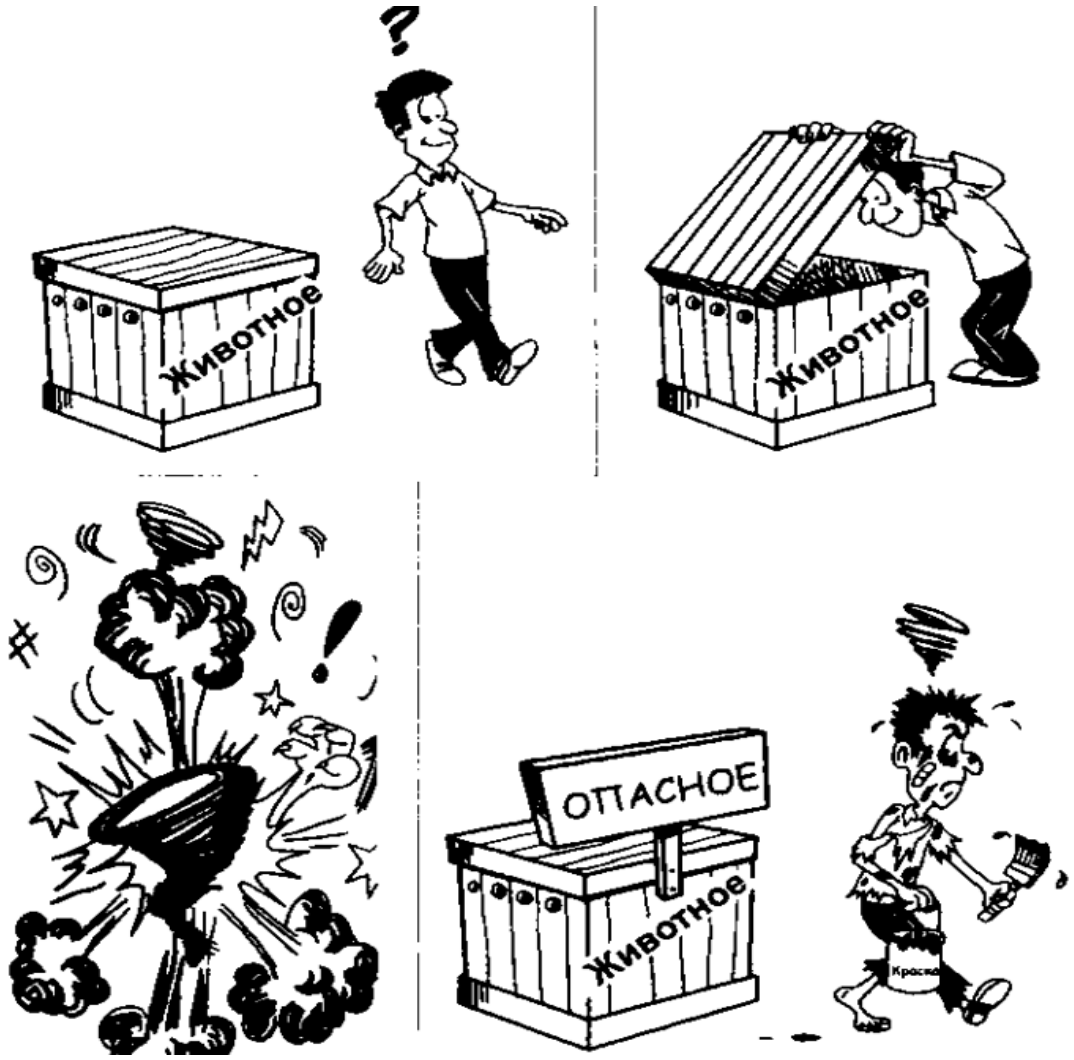
или

(make(создавать)/create/ setUp(устанавливать) / build(строить) / generate(генерировать) / compose(составлять) / add(добавлять)/ new);

УЛУЧШЕНИЯ

- вместо i, j, k лучше используйте более полное имя (i_users, j_answers);

Например, есть переменная,
содержащая строку в 16-формате:
string id; // например “af84ef845cd8”,
то следует использовать имя
string hex_id.



УЛУЧШЕНИЯ

- добавляйте единицы измерения (если переменная хранит результаты измерений, истекшее время или количество байт: delayMs, cacheSizeMb, imageSizePx);

В таблице приведены примеры случаев, когда возможно добавление важной информации к имени переменной:

Ситуация	Имя переменной	Более подходящее имя
Пароль хранится в открытом виде и должен быть зашифрован	password (пароль)	plaintext_password (незашифрованный пароль)
Пользовательский комментарий, который должен быть экранирован, прежде чем он буде отображен	comment (комментарий)	unescaped_comment (неэкранированный комментарий)
Байты переменной html преобразованы в кодировку UTF-8	html	html_utf8
URL входящих данных был закодирован	data	data_urlenc

УЛУЧШЕНИЯ

Венгерская нотация – это система образования имен, широко используемая компанией Microsoft. При такой нотации к имени каждой переменной в виде префикса добавляется ее тип.

Примеры:

Имя	Значение
pLast	Указатель (p) на последний элемент в какой-то структуре данных
pszBuffer	Указатель (p) на буфер строк(s), который заполнен нулями
Cch	Количество (c) символов (ch)
mpcopx	Ассоциативный контейнер(m), содержащий информацию об указателях на цвет (pco) и указателях на осевую длину(px)

УЛУЧШЕНИЯ

Для больших областей видимости используйте более длинные имена, избегайте использования одно- или двухбуквенных имен для переменных, которые встречаются на протяжении всего кода.

Более короткие имена предпочтительнее для переменных, которые встречаются всего несколько раз.

УЛУЧШЕНИЯ

Применяйте прописные буквы, подчеркивания и прочее форматирование, чтобы сделать код более понятным.

Например, можно добавить символ «_» к переменным, которые являются элементами класса, чтобы отличить их от локальных переменных.

Как гарантировать, что имена будут поняты правильно?

Что будет содержать переменная results после выполнения строки:

```
results = allObjects.filter("year < 2014");
```

- объекты, чье значение атрибута year <=2011?
- объекты, чье значение атрибута year не <=2011?

filter может иметь два значения. Не понятно, что именно требуется сделать: выбрать или удалить.

Лучше использовать select (для выбора) или exclude (для исключения)

УЛУЧШЕНИЯ

- применяйте префиксы min и max для (включающихся) границ;
- используйте имена границ слова first и last (begin и end);

- называйте булевы переменные:

рассмотрим на примере

bool read_password=true;

Как интерпретировать?

«Нужно прочесть пароль» или «пароль был уже прочитан».

Исправим, заменив имя read на **need_password** (необходим пароль) или **user_is_authenticated** (пользователь идентифицирован).

Совет: для названия булевых переменных используйте слова **is** и **has**.

Избегайте отрицательных конструкций:

вместо **bool disable_ss1=false;**

лучше использовать конструкцию:

bool use_ss1= true;

УЛУЧШЕНИЯ

- **get** всегда должен выполняться за константу и возвращать элемент класса.

Пример:

```
class StatisticsCollector{
…
public: double getMean(){
   // проходим по всем значениям и возвращаем
   // их сумму, деленную на количество
   }
};
```

Метод следует назвать **computeMean**(). Такое имя подразумевает, что эта операция будет выполняться довольно долго.

УЛУЧШЕНИЯ

- **List::size()** – размер списка.

Прочие контейнеры в C++ имеют мгновенный метод size(), возвращающий количество элементов контейнера.

А для **List** метод **size**() подсчитывает элементы списка один за другим, вместо того, чтобы возвращать предварительно вычисленное значение, что делает метод **ShrinkList**().

Если метод **size**() был назван **countSize**() (подсчитать размер) или **countElements**() (подсчитать количество элементов), то такой ошибки не произошло.

Скорей всего создатели стандартной библиотеки C++ хотели назвать метод **size**(), чтобы контейнер походил на другие, такие как **vector** или **map**. Из-за этого несоответствия программиста ошибочно считают код быстрым, таким как он реализован в других контейнерах.

УЛУЧШЕНИЯ

Почему красота имеет значение?

УЛУЧШЕНИЯ

- **Эстетичность**.

Хороший исходный код также должен быть приятным на вид:

- необходимо применять постоянный шаблон, к элементам которого читатель может легко привыкнуть;
- похожие строки кода следует оформлять одинаково;
- строки кода, решающие одну и ту же задачу, нужно объединять в блоки;
- ровные границы и столбцы позволяют читателям быстрее просматривать текст программы.

УЛУЧШЕНИЯ

```
class StatsKeeper{
public:
// класс для отслеживания
//последовательности числе в
//формате double

 void Add(double); //и методы,
//необходимые для получения
//статической информации о них

private: int count; /*подсчитываем их
количество*/

public:
   double Average();
private: double minimum;
list<double>
   past_items
   ; double maximum;
};
```

```
// класс для отслеживания
//последовательности чисел в формате
//double и методы, необходимые для
//получения статической информации о них

class StatsKeeper{
public:
   void Add(double);
   double Average();

private:
   list<double> past_items;
   int count;  // подсчитываем их количество

 double minimum;
 double maximum;
};
```

УЛУЧШЕНИЯ

- Разбиваем на блоки

```
DatabaseConnection()
{ string error;
assert(ExpandFullName(database_connection,”Do
ug Adams”,&error)==“Mr. Douglas Adams”);
assert(error == “ ”);
assert(ExpandFullName(database_connection,”
Jake Broun”,&error)==“Mr. Jacob Broun III”);
assert(error == “ ”);
assert(ExpandFullName(database_connection,”No
Such Guy”,&error)==“”);
assert(error == “no match foun”);
assert(ExpandFullName(database_connection,”Jo
hn”,&error)==“”);
assert(error == “more that one result”);
}
```

Добавим вспомогательный метод:

```
DatabaseConnection(){
CheckFullName(”Doug Adams”, “Mr. Douglas Adams”,””);
CheckFullName(” Jake Broun”, “Mr. Jacob Broun III”, ” ”);
CheckFullName(”No Such Guy”,””, “no match foun”);
CheckFullName(”John”,””, “more that one result”);
}
void CheckFullName(string partial_name,
                   string expected_full_name,
                    string expected_error){
string error;
string full_name=ExpandFullName(database_connection,
                                partial_name, &error);
assert(error== expected_error);
 assert(full_name== expected_full_name);
}
```

Мораль такова – приведение кода в «опрятный вид» часто сулит не только внешние улучшения, оно помогает качественно его структурировать

УЛУЧШЕНИЯ

- Разбиваем код на абзацы

```
DatabaseConnection(){
   CheckFullName( ”Doug Adams”   ,  “Mr. Douglas Adams”      , ” ”);
   CheckFullName( ” Jake Broun”   ,  “Mr. Jacob Broun III”    , ” ”);
   CheckFullName( ”No Such Guy”  , ” ”                        , “no match foun”);
   CheckFullName( ”John”         , ” ”                        , “more that one result”);
}
```

Ровные границы столбцов позволяют быстрее просматривать код:

```
commands[] = {
    ...
    { "timeout",          NULL,                cmd_spec_timeout },
    { "timestamping",     &opt.timestamping,   cmd_boolean },
    { "tries",            &opt.ntry,           cmd_number_inf },
    { "useproxy",         &opt.use_proxy,      cmd_boolean },
    { "useragent",        NULL,                cmd_spec_useragent },
    ...
};
```

УЛУЧШЕНИЯ

- Выберите определенный порядок и придерживайтесь его.
  Если в одном месте кода упоминается А, Б и В, не меняйте их расстановку (например: Б, В и А) в другом.
  Выберите осмысленный порядок и придерживайтесь его (расположите от наиболее до наименее важного или в алфавитном порядке).

УЛУЧШЕНИЯ

- Используйте пустые строки для того, чтобы разбить большие блоки кода на логические абзацы.

```
class FrontendServer{
public:
FrontendServer();
void ViewProfile(HttpRequest* request);
void OpenDatabase(string location, string user);
void SaveProfile(HttpRequest* request);
string ExtractQueryParam(HttpRequest* request, string param);
void ReplyOk(HttpRequest* request, string html);
void FindFriends(HttpRequest* request);
void ReplyNotFound(HttpRequest* request, string error);
void CloseDatabase(string location);
~FrontendServer();
};
```

```
class FrontendServer{
 public:
      FrontendServer();
      ~FrontendServer();

// обработчики
void ViewProfile(HttpRequest* request);
void SaveProfile(HttpRequest* request);
void FindFriends(HttpRequest* request);

//утилиты запроса/ответа
string ExtractQueryParam(HttpRequest* request, string param);
void ReplyOk(HttpRequest* request, string html);
void ReplyNotFound(HttpRequest* request, string error);

// вспомагательные функции работы с базами данных
void OpenDatabase(string location, string user);
void CloseDatabase(string location);
};
```

УЛУЧШЕНИЯ

- Что выбрать персональный стиль или единообразие?

```
class Logger{

};
```

или

```
class Logger
{

};
```

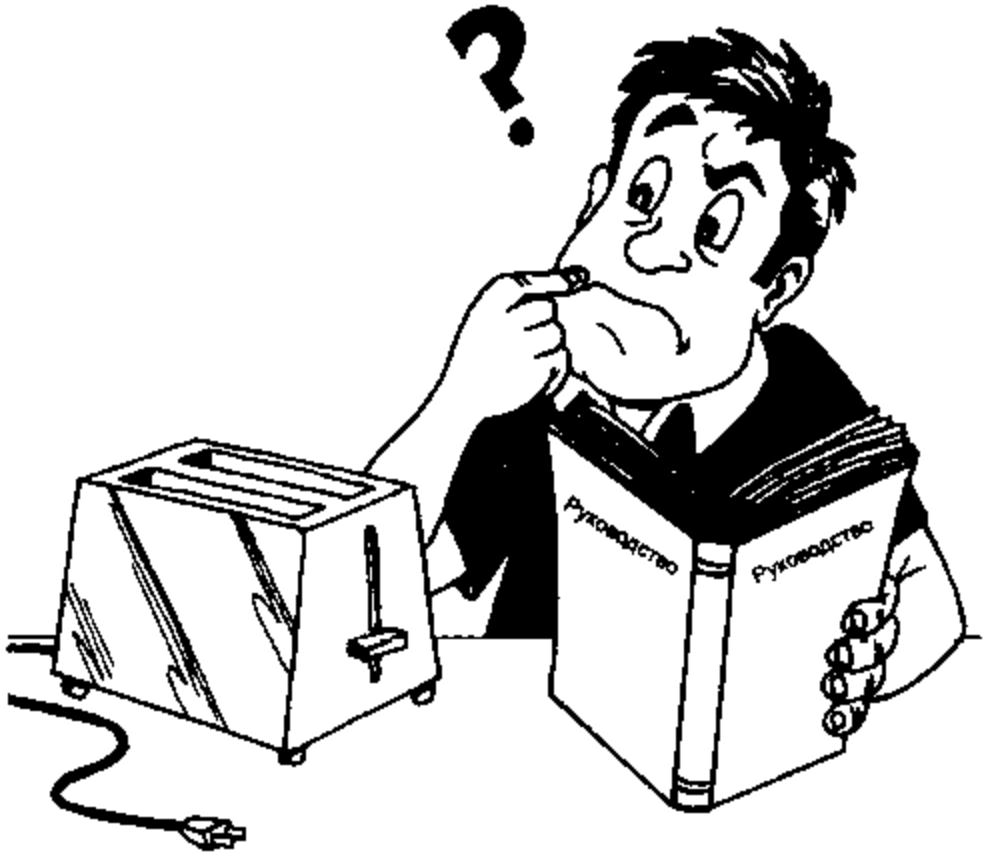
Предпочтение одного из этих стилей другому мало повлияет на читаемость кода. Не нужно смешивать два стиля, т.к. это затрудняет читаемость кода.

Итог: последовательный стиль важнее «правильного стиля».

Комментарии

РУКОВОДСТВА ПОЛЬЗОВАТЕЛЯ

ТРЕБУЮТСЯ!!



НЕ ТРЕБУЮТСЯ

Комментарии

Смысл комментирования заключается в том, чтобы помочь читателю получить столько же информации, сколько имеет разработчик.

```
//описание класса Account
class Account{
   public:
      //конструктор
          Account();

   // устанавливаем новое значение члена profit
   void SetProfit(double profit);


   // возвращаем значение profit этого объекта класса Account
   double GetProfit();
};
```

Не комментируйте строки, предназначение которых можно легко и быстро понять и без комментариев.

Комментарии

- Не нужно комментировать каждую строчку.

Пример бесполезного комментария:

```
//находим узел в заданном поддереве, с заданным именем,
//опускаясь на заданную глубину
Node* FindNodeInSubtree(Node* subtree, string name, int depth);
```

Если хотите оставить комментарий, то следует обратить внимание на следующие важные вещи:

```
// Находим узел с заданным именем name  или возвращаем NULL.
// Если глубина <= 0, исследуем только поддерево subtree
// Если глубина = 0, исследуем только поддерево subtree и N уровней под ним.
Node* FindNodeInSubtree(Node* subtree, string name, int depth);
```

Комментарии

- Не комментируйте плохие имена, лучше исправьте их.

```
//Применяем к параметру reply ограничения, указанные в параметре request,
//такие как количество возвращаемых элементов, количество байт и т.д.
void ClearReply(Request request, Reply reply);
```

Исправим: фразу «применим ограничения» (enforce limits)поместим в имя функции:

```
// Убедимся, что параметр reply соответствует ограничениям по количеству
//элементов, размеру и т.д., заложенным в параметр request
void EnforceLimits (Request  request, Reply reply);
```

Хорошее имя лучше хорошего комментария потому, что оно видно в любом месте кода, где используется функция.

Комментарии

- Добавляйте «комментарий режиссера».

Пример:

```
// Удивительно для этого типа данных, но бинарное дерево оказалось
// на 40 % быстрее, чем хэш-таблица.
// На вычисление хэша потребовалось больше времени, чем для сравнений право/лево.
```

Комментарий может пояснить, почему приведенный код не идеален:

```
// Содержание этого класса становится все более путанным.
// Возможно стоило создать подкласс ResourseNode для обеспечения лучшей организации
```

Этот комментарий подтверждает, что код не идеален. Без него множество читателей не рискнули бы трогать этот беспорядочный код.

Комментарии

- Комментируйте недостатки кода.

Таблица специальных соглашений

Обозначение	Значение
TODO:	Задуманное, но не реализованное
FIXME:	Известно, что здесь есть проблема
HACK:	Неэлегантное решение проблемы
XXX:	Внимание! Серьезная проблема

Пример:

```
//TODO: использовать более быстрый алгоритм
```

Комментарии

- Комментируйте константы.

Пример:

```
NUM_THEADRS = 8
```

Может, эта строка и не требует комментариев, но исправив:

```
NUM_THEADRS = 8 // значение является подходящим, пока оно >= 2* num_processors.
```

Тогда читатель знает как можно изменить значение (например, 1 слишком маленькое, а значение 50 - слишком большое).

Иногда константа содержит значение, которое было тщательно выверено, и, возможно, его не следует изменять:

```
image_quality = 0.72      // пользователи считают, что 0.72 – это лучшее
                          // соотношение размера и качества.
```

Комментарии

- Поставьте себя на место читателя – опередите часто задаваемые вопросы.

```
struct Recorder{
        vector<float> data;
        …
        void Clear(){
         vector<float>().swap(data);
           }
};
```

Может использовать data.clear(), вместо того, чтобы меняться значениями с пустым вектором?

Оказывается, это единственный способ передать память, принадлежащую контейнеру vector, распределителю памяти.

Следует прокомментировать последнюю строку:

```
// Принуждаем вектор передать свою память
// (для получения более подробной информации прочтите о трюке
// с использованием обмена для STL).
```

Комментарии

- Документируйте любое неожиданное поведение, которое может смутить среднестатистического читателя;
- используйте комментарии, проясняющие общую картину на уровне класса/ файла, а также объясните, как именно эти объекты собираются воедино;
- подводите итого блоков кода комментариями, чтобы читатель не запутался, разбираясь в деталях.

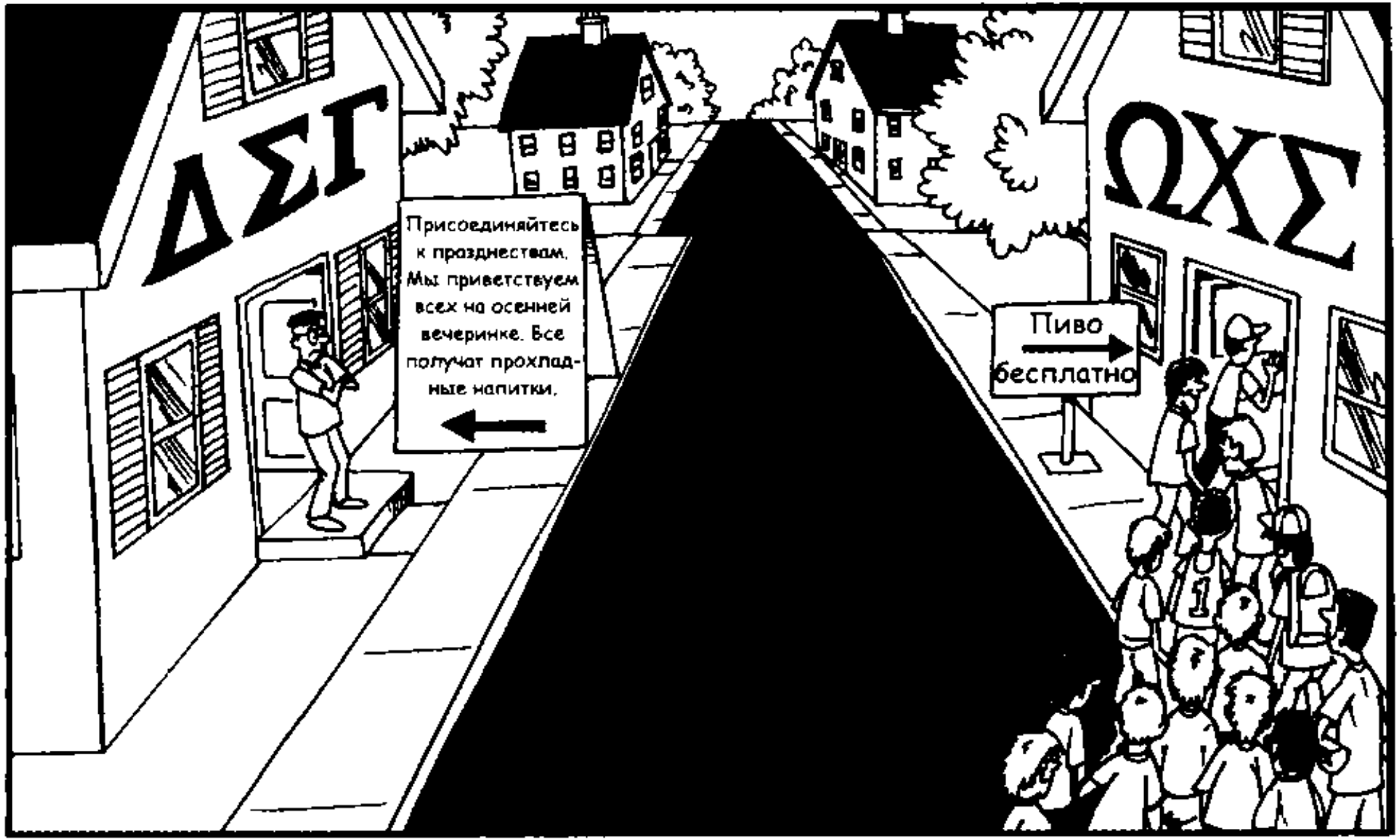
ЧТО ИМЕННО СЛЕДУЕТ КОММЕНТИРОВАТЬ: ОБЪЕКТ, ПРИЧИНУ ИЛИ СПОСОБ?

Возможно, вы слышали совет вроде «необходимо комментировать причину, а не объект (или способ)»?

Мы советуем писать комментарии, помогающие читателю быстрее понять код. При необходимости вы можете прокомментировать объект, причину или способ (или все вместе).

Комментарии

- Комментарии должны быть четкими («не лейте воду»).



Комментарии

- Комментируйте компактно.

```
// типом int обладает параметр CategoryType
// первый параметр с типом float во внутренней паре называется score
// второй – weight
 typedef hash_map<int.pair<float.float>> ScoreMap;
```

Можно описать комментарий одной строкой:

```
// CategoryType ->(score.weight)
 typedef hash_map<int. pair<float. float> > ScoreMap;
```

Комментарии

- Избегайте двусмысленных местоимений и указательных слов.

Пример:

```
// Добавим фрагмент данных в кэш, но сначала проверим,
// не слишком ли он велик
```

К чему относиться «он» к фрагменту данных или к кэшу.

Исправим:

```
// Добавим фрагмент данных в кэш, но сначала проверим,
// не слишком ли велик этот фрагмент
```

или

```
// Если фрагмент данных достаточно мал, вставляем его в кэш
```

Комментарии

- Описывайте поведение функции четко и с практической точки зрения.

```
//Возвращает количество строк в заданном файле.
int CountLines(string filename) { … }
```

“” (пустой файл) - вернет 0 или 1?
“привет” - вернет 0 или 1?
“привет\n” - вернет 1 или 2?
“привет\n мир” - вернет 1 или 2?
“привет\n\r жестокий\n мир\r” - вернет 1, 2, 3 или 4?

Исправим:

```
// Подсчитывает количество символов ‘\n’ в файле
int CountLines(string filename) { … }
```

Комментарии

- Иллюстрируйте комментарии хорошо подобранными примерами.

```
// …
// Пример: Partition([8 5 9 8 2], 8) вернет 1
// и расставит элементы следующим образом: [5 2 | 8 9 8].
int Partition(vector<int>* v, int pivot);
```

Комментарии

- Описывайте свои намерения, а не очевидные детали.

```
void DisplayProducts(list<Product> products){
products.sort(CompareProductByPrice);

// Проходим от конца списка к началу
for(list<Product>::reverse_iterator it = products.rbegin();
        it != products.rend(); ++it)
DisplayPrice(it->price);
}
```

или

```
// Отображаем все цены, от наибольшей до наименьшей
for(list<Product>::reverse_iterator it = products.rbegin();…)
```

Комментарии

- Используйте внутристрочные комментарии, чтобы объяснить значения таинственных аргументов функции.

```
Connect(/* timeout_ms = */ 10, /* use_encryption = */ false);
```

Комментарии

- Делайте комментарии более компактными, используя слова, содержащие большой объем информации.

```
// Этот класс содержит ряд членов, включающих ту же информацию,
// что и база данных, они сохраняются здесь для ускорения работы.
// Когда этот класс считывается позже, проверяется наличие этих членов.
// Если они существуют, то функция их возвращает. В противном случае
// информация считывается из базы данных и заносится в эти поля на будущее.
```

или

```
// Этот класс ведет себя как кэширующий слой базы данных
```

Упрощаем код, или как сделать поток команд управления удобочитаемым

Упрощение кода

Порядок аргументов в условных конструкциях:

- if (length >= 10)
- if (10 <= length)

- while(bytesReceived < bytesExpected)
- while(bytesExpected > bytesReceived)

- if (min < a && a < max) // min< a< max
- if (a > min && a < max)

При сравнении 2 параметров лучше поместить изменяющееся значение слева, а более постоянное справа.

Этот принцип соответствует принципу построения предложений в обычной речи. Сравните фразу «Если вам есть 18 лет» и «Если 18 лет – это ваш возраст или меньше», первая звучит естественней.

Упрощение кода

«Не думайте о розовом слоне!»

```
if(a == b)
{//код_1
} else {
//код_2
}
```

```
if(a != b)
{//код_2
} else {
//код_1
}
```

Совет:

сначала располагайте следующие инструкции

- положительного блока вместо отрицательного (например if (debug) вместо if (!debug));
- более простого блока, чтобы далее уже не думать о нем;
- более интересного или заметного блока.

Упрощение кода

Упрощение кода

Старайтесь *не использовать* тернарный оператор.

- return exponent >= 0 ? mantissa * (1 << exponent) : mantissa/ (1 << -exponent)

- if (exponent >= 0) return mantissa * (1 << exponent);
  else return mantissa / (1 << -exponent);

Совет: **вместо того, чтобы уменьшать количество строк, сократите время, требуемое для понимания кода:**
- по умолчанию используйте конструкцию if-else,
- тернарный оператор ?: следует применять только в простейших случаях.

Старайтесь ***не использовать*** цикл do while.

Упрощение кода

Используйте ранний выход из метода

```
public boolean Contains(String str, String substr) {
    if (str == null || substr == null) return false;
    if (subtr.equals("")) return true;
    ...
}
```

Упрощение кода

Старайтесь **не использовать** оператор goto.

Безобидный вариант:

```
void fun(/*…*/){
    If ( p == NULL) goto exit;
    …
    exit:
            fclose(f1);
            fclose(f2);
    return;
}
```

Сложности могут возникнуть в ситуации, когда задействуется *несколько* операторов goto, особенно если их «пути» пересекаются. В частности, переход *вверх* с использованием goto может привести к страшно запутанной программе (так называемый «спагетти-код»). Такой код вполне можно заменить структурированными циклами.

Упрощение кода

Создавайте более линейный код, избегайте чрезмерной вложенности.

Вложенный код труден для понимания:

```
if (user_result == SUCCESS) {
      if (permission_result != SUCCESS){
        reply.WriteErrors("Error reading permissions");
        reply.Done();
        return;
      }
      reply.WriteErrors("");
} else {
            reply.WriteError(user_result);
}
reply.Done();
```

```
if (user_result != SUCCESS) {
     reply.WriteError(user_result);
     reply.Done();
     return;
}
 if (permission_result != SUCCESS){
    reply.WriteErrors("Error reading permissions");
   reply.Done();
  return;
}
  reply.WriteErrors("");
  reply.Done();
```

Подобной вложенности кода можно избежать при помощи оперативной обработки случаев отказа и возврата из функции (не откладывать).

Разбивка длинных выражений

Разбивка длинных выражений

Используйте поясняющие переменные.

Самый простой способ разбить длинное выражение – это ввести новую переменную (*поясняющую*), которая будет содержать меньшее подвыражение.

- if line.split(':')[0].strip() == "root";

- username = line.split(':')[0].strip();
  if username == "root";

Разбивка длинных выражений

Используйте итоговые переменные.

Даже если выражение *не нуждается* в объяснении (поскольку вы и так можете понять, что оно означает), все равно может быть полезно хранить результат вычисления этого выражения в новой переменной *(итоговой summary variable).*

```
if (request.user.ID == document.ownerID){
    //могу менять
}
...
if (request.user.ID != document.ownerID){
     //а тут не могу менять
}
```

Добавим итоговую переменную:

```
boolean userOwnsDocument =
       (request.user.ID == document.ownerID);
if (userOwnsDocument){
      //могу менять
}
...
if (!userOwnsDocument){
     //а тут не могу менять
 }
```

Разбивка длинных выражений

Применяйте законы Де Моргана – это позволяет переписывать булево выражение в более прозрачной форме.

- не (*a* или *b* или *c*) ⇔ (не *a*) и (не *b*) и (не *c*);
- не (*a* и *b* и *c*) ⇔ (не *a*) или (не *b*) или (не *c*).

$$\overline{A \vee B} = \overline{A} \wedge \overline{B}$$

$$\overline{A \wedge B} = \overline{A} \vee \overline{B}$$

- if (! (fileExists && ! fileProtected)) { error … }

перепишем:

- if (!fileExists || fileProtected) { error … }

Злоупотребление упрощенной логикой

В большинстве языков программирования булевы операторы выполняют упрощенные вычисления.

Например: утверждение if (a || b) не вычисляет b, если значение a равно true. Такое поведение удобно, но иногда им злоупотребляют, пытаясь создать более сложную логику.

```
assert ( ( ! ( bucket = FindBucket(key))) || !bucket->IsOccupied());
```

Если перевести на русский: «Получить сегмент памяти для этого ключа. Если значение не равно нулю, убедитесь, что ключ не занят». Перепишем:

```
bucket = FindBucket(key);
If (bucket != NULL) assert ( !bucket->IsOccupied());
```

Совет: остерегайтесь «заумных» фрагментов кода – они часто запутывают тех, кто будет читать код позднее.

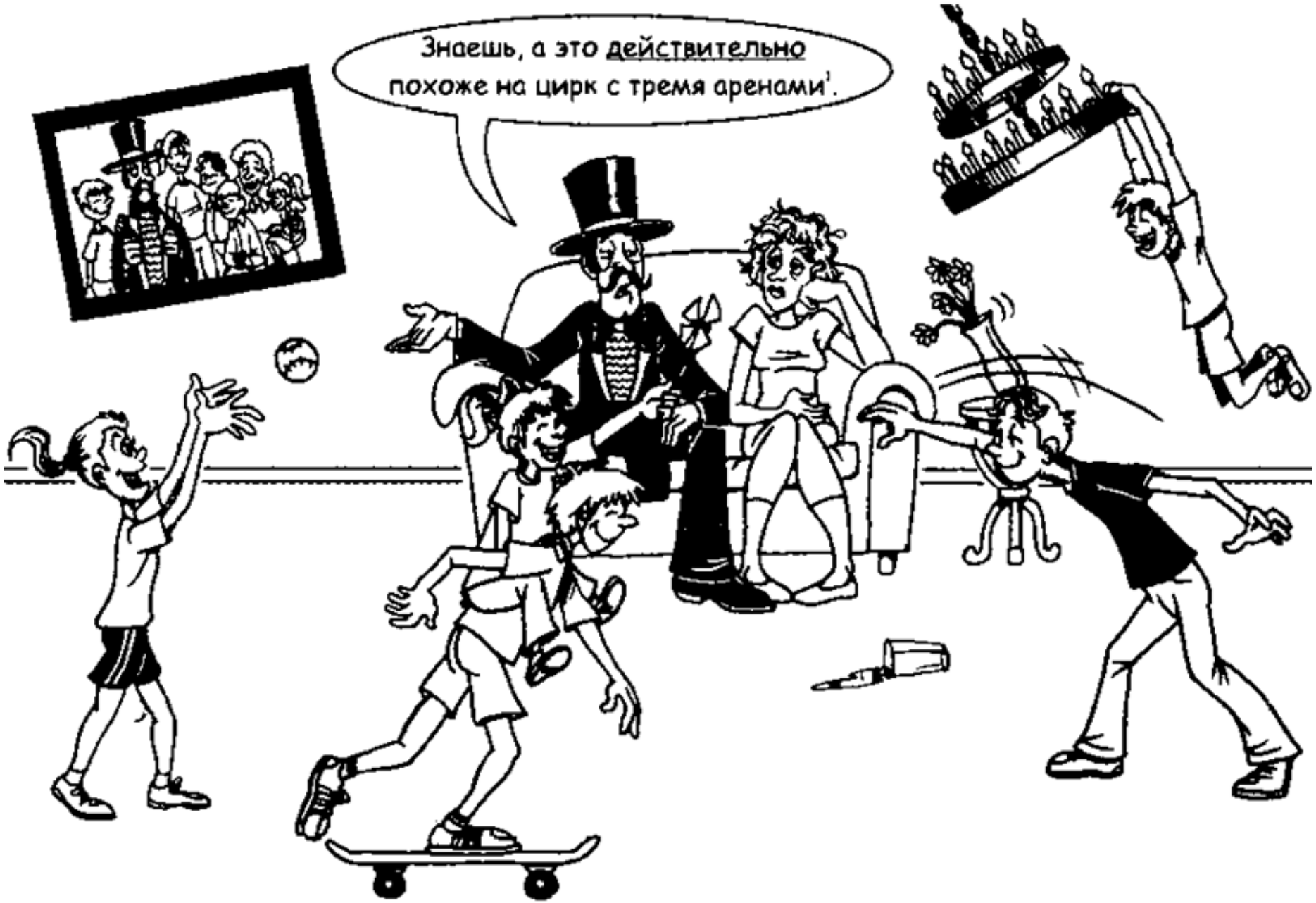
Еще один творческий способ упрощения выражений

Рассмотрим еще один пример кода, написанный на языке C++:

```
void AddStats(const Stats& add_from, Stats* add_to){
  add_to->set_total_memory(add_from.total_memory() + add_to->total_memory());
  add_to->set_free_memory(add_from.free_memory() + add_to->free_memory());
  add_to->set_swap_memory(add_from.swap_memory() + add_to->swap_memory());
  add_to->set_status_string(add_from.status_string() + add_to->status_string());
  add_to->set_num_process(add_from.num_process() + add_to->num_process());
…
} // используем макрос:
      void AddStats(const Stats& add_from, Stats* add_to){
      #define ADD_FIELD(field) add_to->set_##field(add_from.field() + add_to->field ())
          ADD_FIELD(total_memory);
          ADD_FIELD(free_memory);
          ADD_FIELD(swap_memory);
          ADD_FIELD(status_string);
          ADD_FIELD(num_process); …
      #undef ADD_FIELD
      }
```

Обычно стараются обходиться без макроса, но иногда, как в этом случае, простые макросы могут значительно улучшить читаемость кода.

Дом конферансье цирка



Переменные и читаемость

Небрежное использование переменных может сделать программу трудно читаемой.

Рассмотрим три основные проблемы:

- чем больше переменных, тем сложнее отслеживать их все;
- чем больше область видимости переменной, тем дольше ее придется отслеживать;
- чем чаще используется значение переменной, тем сложнее отследить ее текущее значение.

Бесполезные временные переменные

Пример:

```
now = datetime.datetime.now();
root_message.last_view_time = now;
```

Переменная now не нужна,

во-первых она не разбивает сложное выражение;

во-вторых она ничего не проясняет выражение datetime.datetime.now() и так довольно прозрачно;

в-третьих она использована только однажды. Без нее код является таким же простым:

```
root_message.last_view_time= datetime.datetime.now();
```

Избавляемся от промежуточных результатов

Переменные и читаемость

Избавляйтесь от переменных, содержащих промежуточные результаты.

```
var remove_one = function (array, value_to_remove){
  var index_to_remove =null;
for(var i=0; i<array.length; i+=1){
        if (array[i] == value_to_remove) {
                index_to_remove = i;
                break;
        }
 }
   if ( index_to_remove !==null){
        array.splice( index_to_remove, 1);
   }
};
```

```
var remove_one = function (array, value_to_remove){
      for(var i = 0;  i< array.length; i += 1){
            if (array[i] == value_to_remove) {
                  array.splice( i, 1);
                  return;
            }
      }
};
```

Переменная **index_to_remove** используется только для того, чтобы хранить промежуточный результат. Предусмотрев в коде возможность досрочного возврата, можно избавиться от переменной **index_to_remove** и несколько упростить код.

Переменные и читаемость

Уменьшайте область видимости каждой переменной, делая ее минимальной (перемещайте каждую переменную в те области кода, где их будет видеть как можно меньшее количество строк программы).

```
class LargeClass{
string str_;

void Metod1(){
        str_= …;
        Metod2();
}
void Metod2(){
// используется  str_
}
// множество методов, не использующих str_
};
```

p.s. “С глаз долой из сердца вон”

```
class LargeClass{
void Metod1(){
                string str = …;
                Metod2( str );
        }
        void Metod2(string str){
        // используется  str
        }
// множество методов, теперь не видят str
};
```

Переменные и читаемость

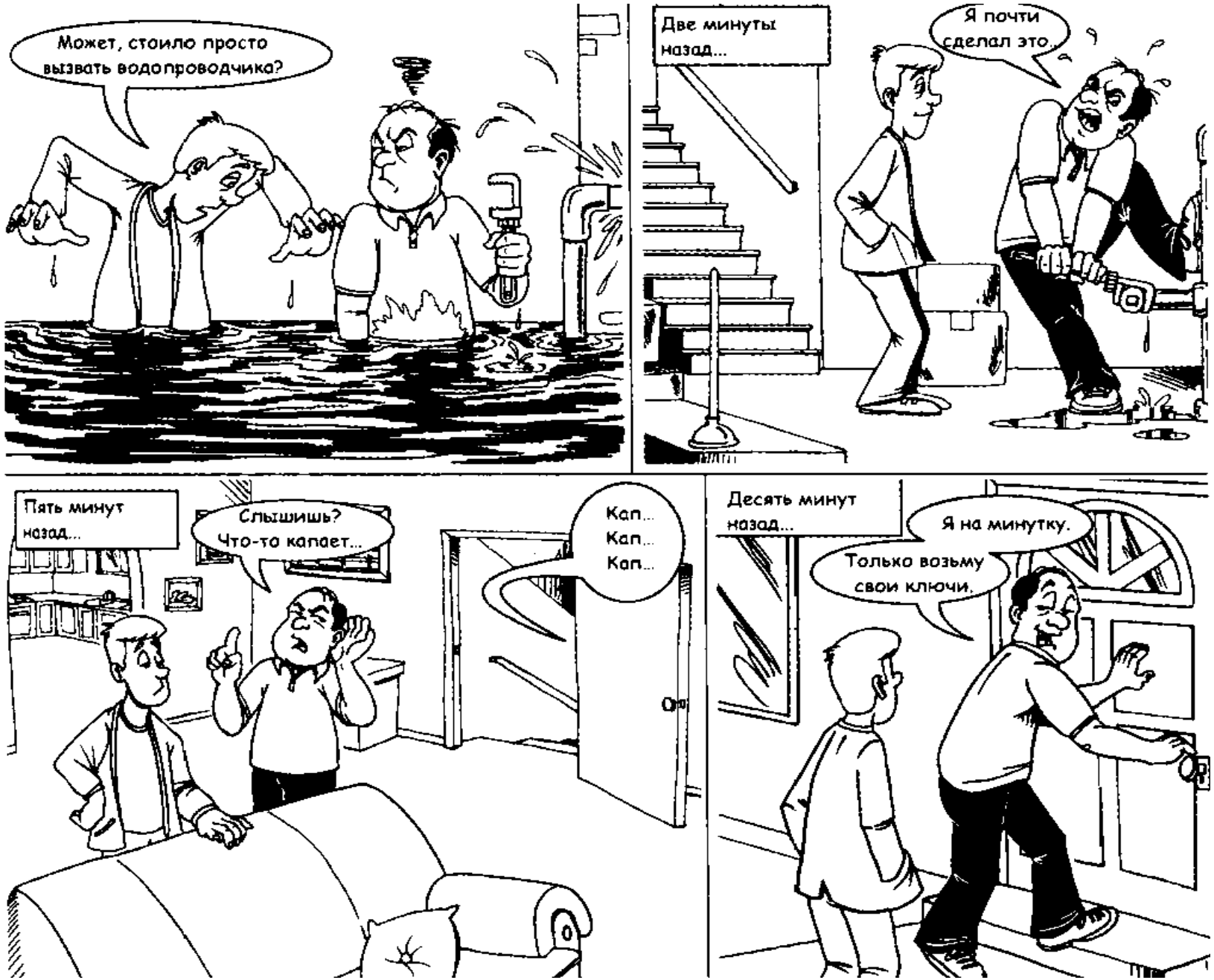
Используйте переменные, значение которых меняется только один раз. Такие переменные (или же константы const, final и прочие конструкции) делают код более понятным.

Реорганизация кода

Рассмотрим три способа реорганизации кода:

- извлечение побочных подзадач, которые не связаны с первоначальной целью программы;
- перекомпоновку кода таким образом, чтобы он выполнял лишь одну задачу в каждый момент времени;
- поиск более прозрачного решения задачи (для этого сначала опишем код словами, а затем используем данное описание в качестве опоры для поиска решения).

Выделяем подзадачи



Выделяем подзадачи

Отделяйте код, предназначенный для решения общих задач, от кода вашего проекта.

Тогда окажется, что большая часть кода является универсальной. Создавая множество библиотек и вспомогательных функций, предназначенных для решения общих задач, можно уменьшить размер проекта.

Основная задача этого приема – позволить программисту сосредоточиться на небольших, хорошо описанных задачах, которые отделены от остального проекта. В результате решения подзадач будут более детальными и верными. Их можно также использовать их в дальнейшем.

Выделяем подзадачи

Существует основной набор простых задач, которые решаются в большинстве программ: например, манипулирование строками, использование хэш-таблиц, а также чтение и запись файлов.

Часто эти «базовые функции» реализуются встроенными библиотеками языка. Например, нужно прочесть все содержимое файла, в языке PHP - функция file_get_contents(“filename”), в языке Python – open(“filename”).read().

В языке C++ такой функции нет, ее нужно описать:

```
ifstream file(file_name);
// рассчитываем размер файла и выделяем буфер такого размера
file.seekg(0, ios::end);
   const int file_size=file.tellg();
    char* file_buf=new char [file_size];
// считываем все содержимое файла в этот буфер
 file.seekg(0, ios::end);
   file.read(file_buf, file_size);
     file.close();
```

Выделяем подзадачи

Универсальный код восхитителен, поскольку он полностью отделен от остального проекта. Подобный код легче менять, тестировать и понимать.

Существуют много мощных библиотек и систем, таковы, например, базы данных SQL, библиотеки JavaScript, а также система шаблонов HTML. Не нужно беспокоится об их содержимом – эти базы кода полностью изолированы от вашего проекта.

Чем большую часть проекта вы сможете разбить на изолированные библиотеки, тем лучше, поскольку ваш код будет короче и проще для понимания.

Итог: отделяйте код, предназначенный для решения общих задач, от кода вашего проекта.

Одна задача в любой момент времени

Одна задача в любой момент времени

Основная идея:

Код должен быть реорганизован таким образом, чтобы он выполнял только одно задание в каждый момент времени.

Если у вас трудночитаемый код, попробуйте перечислить все задания, которые он выполняет. Некоторые из них вполне могут стать отдельными функциями (или классами). Другие же могут сталь логическими «абзацами» внутри одной функции. Тонкости того, как вы реализуете подобное разделение, важны не так, как сам факт их разделения. Сложность заключается в том, чтобы описать все мелкие задачи, которые решает программа.

Одна задача в любой момент времени

Пример. Пусть в каком-нибудь блоге есть виджет для голосования, с помощью которого пользователь может проголосовать «За»(Up) или «Против»(Down). Рейтинг комментария (score) получается из суммы всех голосов: +1 «За»; -1 «Против».

При нажатии одной из кнопок (чтобы проголосовать/изменить свой голос), вызывается следующая функция, написанная на JavaScript;

```
vote_changed(old_vote, new_vote); // Каждый голос может быть «За»
                                  // «Против» или «»
```

Одна задача в любой момент времени

```
vote_changed(old_vote, new_vote){
      var score = get_ score();

      if (new_vote !==old_vote){
        if( new_vote == 'Up'){
          score += (old_vote==='Down' ? 2 : 1);
        } else  if( new_vote == 'Down'){
                    score -= (old_vote==='Up' ? 2 : 1);
                  } else if( new_vote == ' '){
                              score += (old_vote==='Up' ? -1 : 1);
                            }
      }
     set_score(score);
};
```

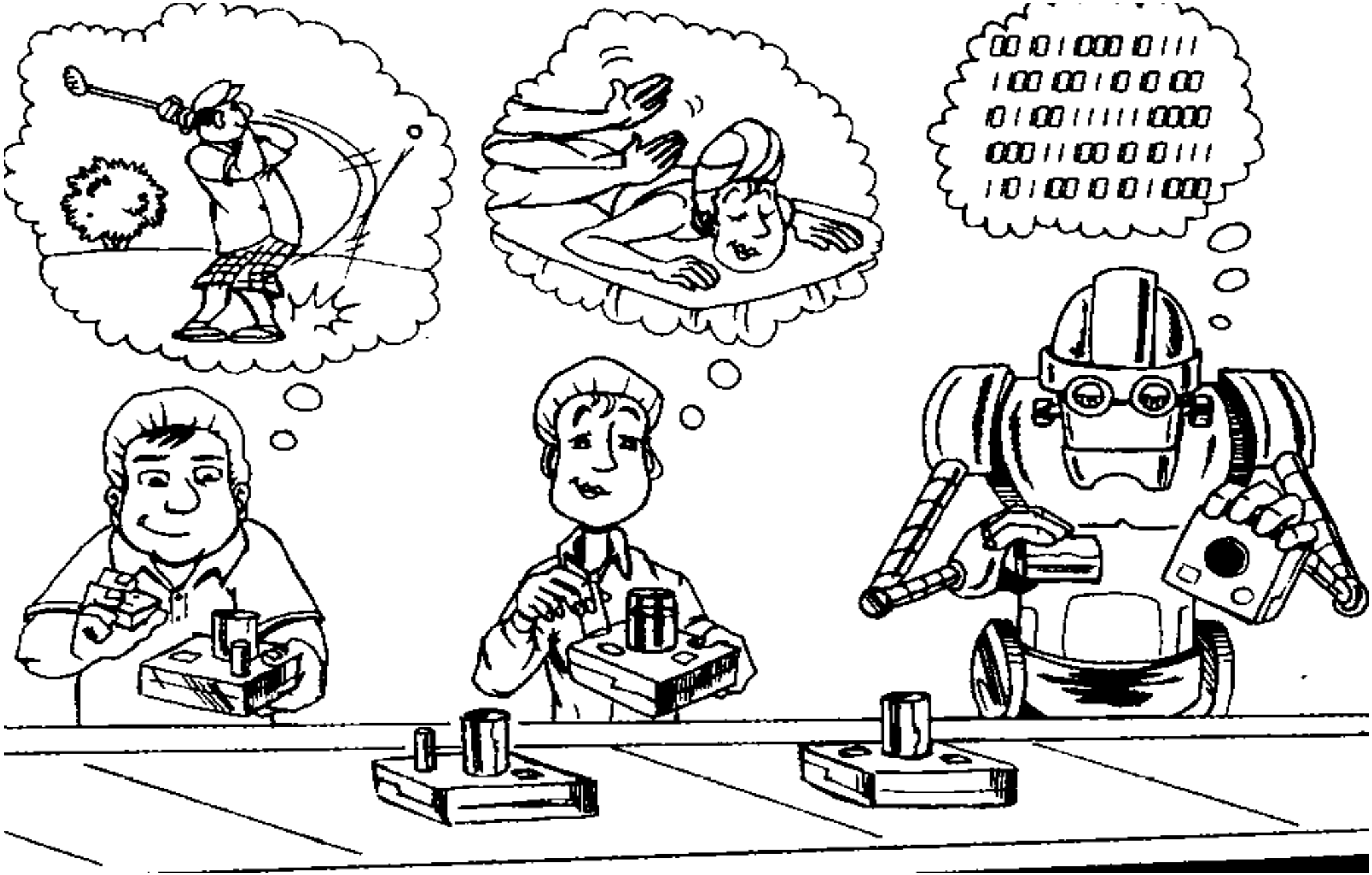
```
// преобразуем «голос»  в числовое значение
var vote_value=function(vote){
            if (vote == 'Up'){
              return +1;
             }
             if( vote == 'Down'){
              return -1;
            }
 return 0;
};

// обновляем параметр score
vote_changed = function (old_vote, new_vote){
            var score = get_ score();

   score -= vote_vote(old_vote); // удаляем прежний голос
  score += vote_vote(old_vote); // добавляем новый голос

set_score(score);
};
```

Код выполняет два задания:

- параметры и score «преобразуются» в числовые значения;
- обновляется параметр score.

Можно сделать код проще, разделив решения каждой из задач.

Превращаем мысли в код



Превращаем мысли в код

Пример: фрагмент кода веб-страницы, написанный на языке PHP. Это код верхней части защищенной страницы, он проверяет, авторизован ли пользователь и имеет ли он право просматривать эту страницу. Если нет, то код перенаправляет посетителя на страницу, которая сообщает ему о том, что он не авторизован.

```
$is_admin = is_admin_request();
    if($document){
        if(!$is_admin &&($document[‘username’] != $_SESSION[‘username’])) {
        return not_authoriezed();
       }
    } else {
              if (!$is_admin) {
                    return not_authorized();
              }
    }
    // продолжаем отображать страницу
```

Превращает мысли в код

Опишем логику кода понятным языком:

есть две ситуации, в которых вы можете быть авторизованы:

1) вы администратор;
2) вы – владелец текущего документа (если такой существует).

В противном случае вы не авторизованы.

```
if(is_admin_request()){
        // авторизован
 } else   if($is_admin &&($document['username'] != $_SESSION['username'])) {
        // авторизован
      } else {
          return not_authorized();
        }
            // продолжаем отображать страницу
```

Код стал короче, а логика проще, поскольку избавились от отрицания.

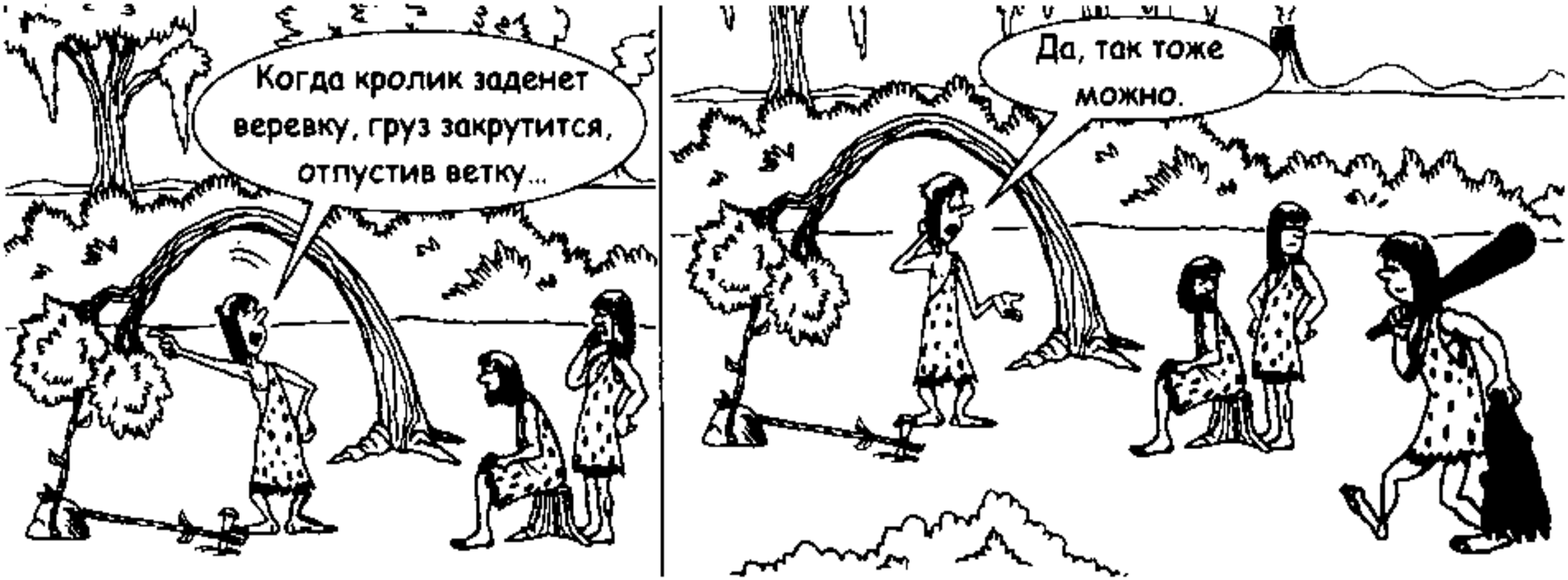
Превращает мысли в код

Процесс описания простым языком можно применять не только при написании кода.

Например, политика одной из компьютерных лабораторий гласит, что в тех случаях, когда студенту требуется помощь в отладке программы, он сначала должен объяснить свою проблему плюшевому мишке, который сидит в углу. Удивительно, но простое описание проблемы вслух часто помогает студенту придумать решение. Этот прием называется «*метод утенка*».

Иначе говоря, если вы не можете описать проблему словами, значит, возможно, вы что-то упустили или не определили. Описание словами программы (или идеи) действительно помогает претворить ее в жизнь.

Пишите меньше кода



Изучайте стандартные библиотеки

Пишите меньше кода

Чем больше кода в вашей базе, чем “тяжелее” она становиться и тем труднее с ней работать.

Советы как избавиться от написания нового кода:

- исключать некритические особенности из продукта и, так сказать, “не перехимичивать”;
- многократно обдумывать поставленные требования (это делается для того, чтобы решить самую простую возможную задачу и при этом реализовать первоначальный замысел);
- уточнять требования (может нужно решить более простую задачу);
- изучать стандартные библиотеки.

СПАСИБО ЗА ВНИМАНИЕ!