ТЕХНОСФЕРА

Лекция № 9

Углубленное программирование на языке C / C++

Алексей Петров

Лекция №9. Идиоматика C++. Основы рефакторинга и качество кода. Стандарты кодирования и методологии разработки ПО

1. Объектно-ориентированные идиомы языка C++ и управление памятью.
2. Статический анализ и рефакторинг исходного кода. Анти-шаблоны, логические и семантические ошибки, «технический долг» проекта.
3. Понятие, факторы и метрики качества исходного кода. Оформление и документирование кода.
4. Методологии промышленной разработки.
5. **Постановка задач к практикуму №7.**

Идиома языка программирования?..

Идиома — пригодная для многократного применения характерная конструкция языка высокого уровня, средство выражения семантики и самостоятельный инструмент программирования, обеспечивающий простую и удобную запись кода, требуемую структуру, гибкость реализации архитектуры системы и т.д.

В отличие от универсальных шаблонов, **идиомы** языков программирования **отражают особенности** их грамматики и семантики, а значит, позволяют добиться большего при выражении архитектурных идей, решении сложных задач, оптимизации и тонкой настройке кода. **Знание идиом** — признак **свободного владения** языком программирования.

Пространство идиом C++. Идиомы процедурного программирования

Идиомы C++ покрывают не вытекающие из его синтаксиса аспекты, придающие языку эффективность, совместимость с языком C и удобство использования, и допускают классификацию по таким признакам, как:

- **уровень сложности;**
- **степень переносимости** в другие языки (из других языков);
- **отношение к парадигме** программирования и т.д.

Из языка C в C++ вошли идиомы процедурного программирования:

```
for(;;);          // пустой бесконечный цикл
while(1) foo(); // цикл с тождественно истинным условием


// постфиксный декремент, адресная арифметика и др.
while(*cp1++ = *cp2++);
```

Пространство идиом C++. Объектно-ориентированные идиомы: обзор

Объектно-ориентированные идиомы C++ охватывают **все аспекты языковой поддержки** парадигмы объектно-ориентированного программирования, включая:

- систему статического контроля типов времени компиляции;
- систему динамической идентификации типов времени выполнения (RTTI);
- абстрактные и конкретные типы данных (структуры и классы);
- управление доступом;
- инициализацию и уничтожение объектов;
- инкапсуляцию, наследование, полиморфизм подклассов;
- перегрузку операций-функций (operator =, operator <<, operator >>).

Классы или структуры?

	Структура	Класс
Базовая семантика	Простой агрегат (данные, вложенные типы)	Пользовательский тип (без ограничений)
Наличие конструкторов и деструкторов	Редко	В большинстве случаев
Уровень доступа к членам по умолчанию	Открытые (`public`)	Закрытые (`private`)
Использование в шаблонах	Да	Да
Использование в иерархиях наследования	Да	Да
Примеры	`std::pair<T1, T2>` `boost::is_void<T>`	`std::vector<T>`

Объектно-ориентированные идиомы C++: примеры (1 / 2)

Тривиальный конструктор:

- формируется автоматически;
- не инициализирует значения POD-типов;
- рекурсивно вызывается для объектов базовых классов.

Конструктор (операция-функция) почленного копирования (присваивания) по умолчанию:

- формируются автоматически;
- выполняют поверхностную инициализацию (присваивание).

Отсутствие наследования конструкторов.

```
class A;
A a;              // вызов A::A(), если есть
// ...
A b(a);           // вызов A::A(A&), если есть
```

Объектно-ориентированные идиомы C++: примеры (2 / 2)

Инициализация и уничтожение объектов в автоматической и динамической памяти.

Связывание **инициализации** объекта с его **созданием**.

Неявный параметр нестатических методов классов — псевдопеременная this.

```
class A;
A *pa = new A(42);      // вызов A::A(int)
A *pa2 = new A[N];      // вызов A::A(), N раз
// ...
// освобождение памяти с вызовом A::~A()
delete [] pa2;
// освобождение памяти без вызова A::~A()
// delete pa2;
```

Идиомы повышенного уровня сложности

Перегруженные операции, в том числе:

- операция индексирования агрегатов: operator [] — допускает контекстно-зависимую перегрузку в зависимости от использования в лево- и праводопустимых выражениях;
- операция выделения памяти: operator new;
- операция освобождения памяти: operator delete;
- операция доступа по указателю: operator ->;
- операции приведения к POD-типам: operator double, operator char, др.

Функциональные объекты (функторы).

Подсчет ссылок и указателей — квазиавтоматическая сборка мусора из динамической памяти.

Объекты переменного размера и др.

Идиома №1. Контекстно–зависимая перегрузка индексирования (1 / 2)

```
class ARef {   // фиктивный класс
public:
    ARef(A &a, int i) : _a(a), _ix(i) { }
    // запись t в _a
    ARef& operator= (T t)      { return *this; }
    // чтение из _a в _t
    operator T()               { return _t; }

private:
    A&  _a;     // ссылка на агрегат
    int _ix;    // индекс в агрегате
    T   _t;     // T - гл. обр. базовый тип
};
```

Идиома №1. Контекстно–зависимая перегрузка индексирования (2 / 2)

```
class A {                   // агрегат
friend class ARef;          // для повышения производительности
public:
    A() { /* ... */ }
    ARef operator[] (int ix) { return ARef(*this, ix); }
};

// в точке использования
A a;            int i, j;               T t;

// ...
a[i] = t; // (a.operator[](i)).operator=(t);
t = a[j]; // t = (a.operator[](j)).operator T();
```

Идиома №2. «Автоматическая» сборка мусора. Подсчет ссылок (1 / 4)

Операции создания и уничтожения объектов **могут подвергаться оптимизации** для повышения быстродействия и эффективности использования памяти отдельными классами

Наиболее распространенные механизмы подобной оптимизации реализуют идиомы **общего (разделяемого) представления** и **подсчета ссылок**, которые:

- обобщаются для любых классов с копированием экземпляров;
- представляют особую ценность при динамическом выделении и освобождении памяти.

Идиома №2. «Автоматическая» сборка мусора. Подсчет ссылок (2 / 4)

```
class StringRep {                   // класс-представление
friend class String;
private:
    StringRep(const char *s) : _count(1) {
        strcpy(_rep = new char[std::strlen(s) + 1], s);
    }
    ~StringRep() { delete [] _rep; }

private:
    char*       _rep;               // общее представление
    int         _count;             // количество ссылок
};
```

Идиома №2. «Автоматическая» сборка мусора. Подсчет ссылок (3 / 4)

```
class String {          // класс-строка
public:
    String()    { rep = new StringRep(""); }
    String(const String &s) { rep = s.rep; rep->_count++; }

    String& operator= (const String &s) {
        s.rep->_count++;
        if(--rep->_count <= 0) delete rep;
        rep = s.rep;
        return *this;
    }

    ~String() { if(--rep->_count <= 0) delete rep; }
```

Идиома №2. «Автоматическая» сборка мусора. Подсчет ссылок (4 / 4)

```
// class String
    String(const char *s) { rep = new StringRep(s); }
    int length() const {       // делегирование
        return strlen(rep->_rep);
    }

private:
    StringRep *rep;            // общее представление
};
```

Идиома №3. Управление памятью заменой операций new и delete (1 / 2)

Вариант №1. Полный контроль над выделением и освобождением памяти (например, по соображениям эффективности или в силу отсутствия базовой поддержки) достигается переопределением глобальных операций-функций:

```
void* operator new(std::size_t size);
void  operator delete(void *ptr);

// для "размещающего" new: T *pt = new (buf) T(val);
void* operator new(std::size_t, void *p);
```

Примечание: При переопределении ::new и ::delete может потребоваться знание адреса начала области динамической памяти HEAP_BASE_ADDRESS и прочих сопутствующих констант.

Идиома №3. Управление памятью заменой операций new и delete (2 / 2)

Вариант №2. Контроль над выделением и освобождением памяти под объектами класса требует специализированной реализации (перегрузки) операций распределения памяти на уровне самого класса (или одного из его предков).

Преимущества перегрузки операций-функций operator new и operator delete проявляются:

- для классов, создающих мелкие объекты (от 4 до 10 байт);
- в операционных системах с примитивами управления памятью и без них;
- в виде повышения производительности и сокращения накладных расходов на распределение отдельных блоков (от 4 до 8 байт).

С учетом нужд приложения память объекта может выделяться **из пула статического или динамического размера**.

Создание объектов в пуле динамического размера (1 / 3)

```
class String {
public:
    String()                    { /* _rep = ... */ }
    String(const char *s)       { /* _rep = ... */ }
    ~String()                   { delete[] _rep; }
    void* operator new(std::size_t);
    void  operator delete(void*);
private:
    static String *_list;       // список свободных блоков
    // указатель на следующий блок или представление строки
    union {
        String *_freeptr; char *_rep;
    };
};
```

Создание объектов в пуле динамического размера (2 / 3)

```
String *String::_list = nullptr;

void String::operator delete(void *p) {
    String *s = (String*)p;
    s->_freeptr = _list;
    _list = s;
}
```

Создание объектов в пуле динамического размера (3 / 3)

```
void* String::operator new(std::size_t /* size */) {
    if(!_list) {
        int i = 0;
        for(_list = (String*)new char[POOL_SIZE *
                    sizeof(String)]; i < POOL_SIZE - 1; i++)
            _list[i]->_freeptr = &(_list[i + 1]);
        _list[i]->_freeptr = nullptr;
    }

    String *aList = _list;
    _list = _list->_freeptr;
    return aList;
}
```

Идиома №4. «Конверт/письмо» и делегированный полиморфизм

Идиома **«конверт/письмо»** — частный случай шаблона проектирования «мост», используемый, **когда необходимо**:

- работать с группой взаимозаменяемых классов и интерпретировать все объекты как принадлежащие к одному типу;
- преодолеть ограничения системы контроля типов, состоящие в жесткой привязке символических имен переменных (идентификаторов) к объектам (адресам в памяти);
- идентифицировать фактический класс объекта на стадии выполнения и разрешить переменным «менять свои типы».

Делегированный полиморфизм в действии (1 / 3)

```
struct BaseConstructor{ BaseConstructor(int = 0) {} };

class Base {   // класс-конверт
public:
    Base()              { _rep = new DerivedOne; }
    Base(T t)           { _rep = new DerivedOne(t); }
    Base(T t, U u)      { _rep = new DerivedTwo(t, u); }
    Base(Base &b)       { /* ... */ }
    Base& operator=(Base &b) { /* ... */ }
    virtual Base& operator+(const Base &b) {
        return _rep->operator+(b);
    }
```

Делегированный полиморфизм в действии (2 / 3)

```
// class Base
    void Redefine(Base *pb) {
        if(!(--_rep->_refCount)) delete _rep;
        _rep = pb;
    }

protected:
    Base(BaseConstructor) : _refCount(1) { }

private:
    Base*       _rep;           // адрес экземпляра письма
    int         _refCount;      // количество ссылок
};
```

Делегированный полиморфизм в действии (3 / 3)

```
class DerivedOne : public Base {
public:
    DerivedOne(T t) : Base(BaseConstructor()), _refCount(1)
    { /* ... */ }

    Base& operator+(const Base &b) { /* ... */ }
    // ...

private:
    // ...
};
```

Рефакторинг исходного кода

Рефакторинг — систематическая (технологичная) деятельность по изменению внутренней структуры ПО, **цель** которой:

- облегчить понимание работы исходного кода, — а значит, облегчить обнаружение ошибок;
- упростить модификацию кода без изменения наблюдаемого поведения;
- улучшить композицию ПО и ускорить написание кода.

Преимущества:

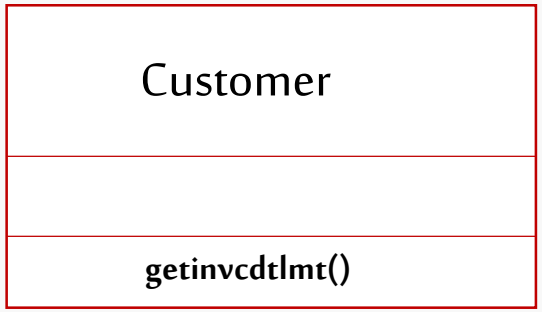
- предсказуемость результата каждого шага;
- продолжение проектирования во время разработки (сопровождения);
- повышение скорости внесения изменений и реализации новых функций;
- поддержание качества при продолжении разработки.

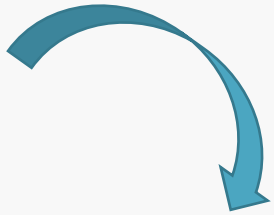
Ключевые риски:

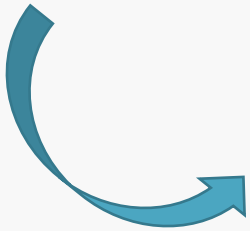
- необходимость внесения изменений в работающий код;
- необходимость (в ряде случаев) изменения интерфейсов.

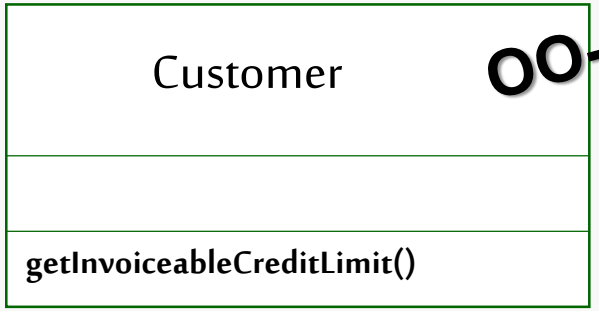
Пример рефакторинга (UML)

Наименование: «переименование метода».
Источник: Мартин Фаулер, Rename Method
Причина: текущий вариант имени не раскрывает назначение **метода**.









Рефакторинг невозможен без понимания правил и принципов ОО-проектирования

При подготовке сл. 26 – 27, 29 – 31, 66 использованы материалы семинара «Как измерить архитектуру ПО?» («Академия информационных систем», 2014).

Рефакторинг ОО-кода и правила ОО-проектирования

Б. Фут (Brian Foote) и У. Опдайк (William Opdyke) в работе *Life Cycle and Refactoring Patterns that Support Evolution and Reuse* (1995) указали на ряд правил ОО-проектирования, многие из которых перекликаются с каталогом методов рефакторинга из книги М. Фаулера:

- DR1. Use Consistent Names
- DR2. Eliminate Case Analysis
- DR3. Reduce the Number of Arguments
- DR4. Reduce the Number of Methods
- DR7. Minimize Access to Variables
- DR8. Subclasses Should Be Specializations
- DR9. Split Large Classes
- DR11. Separate Methods That Do not Communicate

Статический анализ и инспекция кода (1 / 2)

Статический анализ кода:

- предшествует и сопровождает его рефакторинг;
- в отличие от традиционных практик тестирования проводится без реального выполнения объекта исследования, вручную или специальными инструментами.

К числу **ошибок**, выявляемых при статическом анализе кода, относятся:

- неверное или неопределенное поведение — обращение к неинициализированным переменным, «пустым» указателям и др.;
- использование небезопасных функций (например, gets());
- переполнение буфера;
- нарушения кроссплатформенности;
- нарушения зон ответственности классов;
- вызов функций (методов) как процедур и т.д.

Статический анализ и инспекция кода (2 / 2)

Персональная оценка (инспекция) — статический анализ кода без применения инструментальных средств для определения

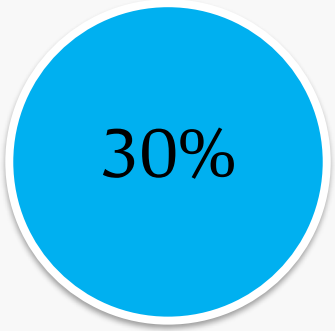
- эффективности (использования ресурсов, вычислительной сложности);
- удобства сопровождения (анализа, проверки, внесения изменений);
- надежности (зрелости, способности к восстановлению после сбоев) ;
- иных структурных показателей качества (напр., по ГОСТ Р ИСО 9126).

Основными правилами персональной оценки являются:

- **регулярность** — предоставление отчета о проведении на каждом техническом совете (напр. еженедельно);
- **независимость** — назначение рецензентов из числа членов команды, не являющихся первоначальными владельцами кода;
- **вовлечение** — распространение знаний о каждом (не)удачном фрагменте кода среди всех членов команды;
- **управление** — назначение ответственности и сроков исправления недостатков.

Эффективность статического анализа

Совокупная эффективность пересмотров дизайна и инспекции кода иногда превышает **90%**



Снижение расходов и сокращение периода разработки благодаря пересмотрам, инспекции / статическому анализу и технологиям виртуализации

Статический анализ позволяет избежать возникновения «периода хаоса» в начале эксплуатации и обнаруживать дефекты на тех стадиях разработки, когда они возникают.

Пересмотры и эффективность снижения дефектоемкости кода

Вид пересмотра	Мин., %	Медиана, %	Макс., %
Пересмотр архитектуры верхнего уровня	30	40	60
Детальный пересмотр «функциональной архитектуры»	30	45	65
Детальный пересмотр «логической архитектуры»	35	55	75
Статический анализ / Инспекция кода	35	60	85

По данным обзора: Jones, C. Software Quality in 2010: A Survey of the State of the Art. URL: http://www.sqgne.org/presentations/2010-11/Jones-Nov-2010.pdf

Типовые анти-шаблоны и «грязные техники» (1 / 2)

Инспекция кода способствует обнаружению неудачных решений задач проектирования, проявляющих себя как использование так называемых **«анти-шаблонов»** дизайна и **«грязных техник»** (англ. dirty coding) программирования, например (начало):

- **«божественный» объект** (англ. God object) — объект (класс) с чрезмерной функциональной нагрузкой; монолит, замыкающий на себе чересчур много каналов взаимодействия прочих элементов архитектуры;
- **магические числа** (англ. magic numbers) — константы с трудно постижимой семантикой;
- **«жесткий код»** (англ. hard code) — имена, адреса и пр. числовые и символьные литералы, наличие которых затрудняет или делает невозможным конфигурирование системы;
- **загадочный код** (англ. cryptic code) — умышленное или неумышленное несоблюдение принципа самодокументируемости исходного кода.

Типовые анти-шаблоны и «грязные техники» (2 / 2)

Инспекция кода способствует обнаружению неудачных решений задач проектирования, проявляющих себя как использование так называемых **«анти-шаблонов»** дизайна и **«грязных техник»** (англ. dirty coding) программирования, например (окончание):

- **проверка типа вместо интерфейса** (англ. checking type instead of interface) — нарушение принципа Programming to Interfaces;
- **«мертвый» или пустой код** (англ. dead or empty code) — кодовые фрагменты, которые не используются в текущей сборке (версии) приложения, устарели или сделаны «про запас»;
- **архитектурно необоснованные заглушки** (англ. stub code) — методы или функции, не выполняющие роль пустых неабстрактных методов, шаблонных методов (GoF) или операций-«зацепок»;
- **код с непредсказуемым поведением** (англ. unpredictable code) — обращение к неинициализированным переменным, «трюки» в управлении памятью, неконтролируемое переполнение буферов, пр.

Типовые логико-семантические ошибки в программном коде

К числу ошибок в исходном коде на любом объектно-ориентированном языке, в том числе C++, следует отнести:

- **неверную обработку краевых условий** (например, замену <= на <);
- **неверную реализацию применяемых алгоритмов;**
- **неверный выбор базовых типов;**
- **неверный способ передачи параметров;**
- **опечатки в именах** перегружаемых методов;
- **неправильную обработку исключений** (в том числе пустые обработчики);
- **несоблюдение принципов транзакционной обработки** данных (англ. ACID — atomicity, consistency, isolation, [and] durability);
- **допущение нежелательного параллельного доступа** нескольких потоков к разделяемым данным;
- **компрометацию уровней доступа** к членам классов и уровней приложения;
- **неверную обработку «пустых» указателей**.

Актуальность проблем качества ПО

Актуальность проблематики качества ПО обусловлена рядом объективных факторов эволюции индустрии разработки ПО, в том числе:

- развитием новых итеративных методов **разработки;**
- распространением методов **обеспечения и контроля качества** ПО на все этапы разработки продуктов;
- распространением методов **объектно-ориентированного анализа, проектирования и разработки;**
- широким применением формальных **языков моделирования** (UML) и **CASE-технологий**.

Понятие качества. Что такое «качественное ПО»?

Согласно стандарту ГОСТ Р ИСО 9000, **качество** — это «степень соответствия присущих характеристик (отличительных свойств) изделия или продукта потребностям, ожиданиям».

В программной инженерии различают **качество ПО** и **качество исходного кода** (с точки зрения человека или машины?!).

Необходимость управления качеством обусловлена потребностью управлять (повышать качество управления) рисками и затратами на всех этапах жизненного цикла ПО.

Функциональные и структурные показатели качества кода

Различают функциональные и структурные показатели качества:

- **функциональные** — отражают степень соответствия продукта требованиям к его функции, техническому проекту, спецификации;
- **структурные** — описывают соответствие ПО требованиям к архитектуре (организации) и характеризуют надежность, удобство сопровождения и т.д.

Квалифицированная оценка «структурного качества» ПО предполагает высокоуровневый статический анализ **архитектуры** продукта (в том числе **компонентной структуры**, используемой **платформы** и **схемы БД**), а также **исходного программного кода**.

Модели качества ПО

Модель качества ПО — это упорядоченная система атрибутов, вместе и по отдельности значимых для заинтересованных сторон проекта разработки ПО (представителей заказчика, пользователей, разработчиков, специалистов по сопровождению и т.д.).

Наибольшую известность на сегодняшний день приобрели:

- модель **Дж. МакКола** и др.;
- модель **Б. Боэма;**
- модель **ISO 9126 / ISO 250*x*0.**

Качество ПО по МакКолу

Предложенная в 1977 г. Дж. МакКолом (J. McCall), П. Ричардсом (P. Richards) и Дж. Уолтерсом (G. Walters) модель качества ПО подразделяет атрибуты качества на три категории:

- **факторы** (англ. factors) – описывают ПО с точки зрения пользователя, определяются требованиями и группируются по видам деятельности заинтересованных лиц;
- **критерии** (англ. criteria) – числовые уровни факторов: описывают ПО с точки зрения разработчика, задаются как цели при разработке;
- **метрики** (англ. metrics) – служат для количественного описания и измерения качества (оценки от 0 до 10).

Факторы качества ПО. Треугольник МакКола

Модель МакКола: как это работает?

Каждая метрика качества ПО влияет на оценку нескольких факторов.

Числовое выражение фактора f_i — это линейная комбинация (взвешенных) значений влияющих метрик m_j:

$$f_i = \sum_j w_{ij} m_j$$

Коэффициенты w_{ij} определяются сугубо индивидуально для различных:

- моделей, методологий и методов разработки;
- предприятий-заказчиков;
- проектных офисов или групп разработки и пр.

Качество ПО по Боэму

В 1978 г. Б. Боэм (B. Boehm) расширил модель МакКола и предложил собственную, содержащую ряд дополнительных атрибутов качества ПО.

В модели Боэма атрибуты качества группируются **по способу использования** ПО.

Промежуточные атрибуты (англ. intermediate constructs, всего 19):

- включают 11 факторов качества по МакКолу;
- разделяются на **примитивные** атрибуты (англ. primitive constructs), которые могут быть оценены на основе метрик.

Качество ПО согласно ISO 9126

Стандартная на сегодняшний день модель качества ПО принята
в 1991 г. и закреплена стандартом ISO 9126:

- Введен в РФ как ГОСТ Р ИСО/МЭК 9126-93 «Информационная технология. Оценка программной продукции. Характеристики качества и руководства по их применению».

Модель ISO 9126 оперирует 3 категориями:

- **цели** (англ. goals) — ожидания от ПО;
- **атрибуты** (англ. attributes) — свойства ПО: показывают близость к достижению целей;
- **метрики** (англ. metrics) — количественные оценки меры наличия атрибутов.

В модели ISO 9126 выделено **6 целей**, достижение которых определяется **21 атрибутом**.

Желаемые структурные характеристики ПО

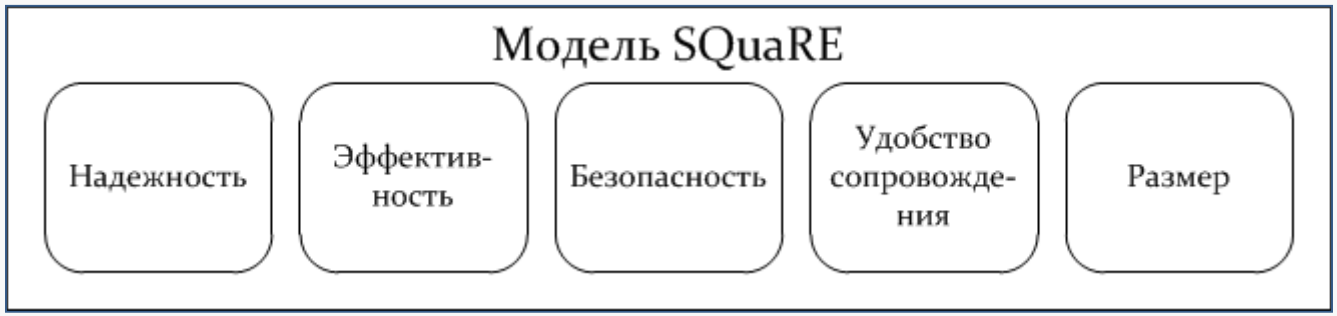
Посвященный вопросам качества продуктов стандарт ISO/IEC 9126-3 дополняют стандарты ISO/IEC 250*x*0, которые вводят в рассмотрение практическую **модель качества SQuaRE**:

- ISO/IEC 9126-3 "Software Engineering — Product Quality";
- ISO/IEC 25000:2014 "Systems and Software Engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Guide to SQuaRE";
- ISO/IEC 25010:2011 "Systems and Software Engineering — Systems and Software Quality Requirements and Evaluation (SQuaRE) — System and software quality models".

Согласно модели SQuaRE, рекомендуется принимать во внимание пять основных **желаемых структурных характеристик** ПО (ср. с целями модели ISO 9126).

Цели в модели качества ISO 9126 и характеристики SQuaRE

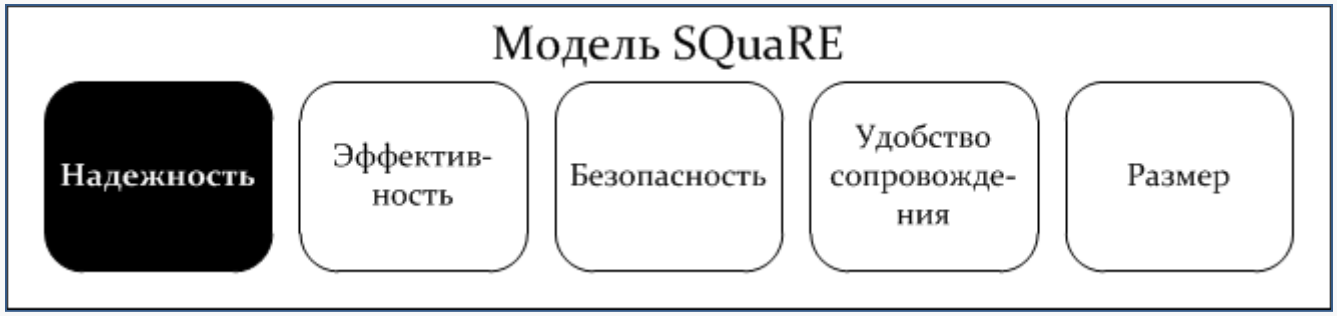
- **Efficiency** — эффективность.
- **Functionality** — функциональность.
- **Maintainability** — удобство сопровождения.
- **Portability** — переносимость.
- **Reliability** — надежность.
- **Usability** — удобство использования.



Модель SQuaRE: надежность

Надежность — показатель прочности и устойчивости системы, отражающий степень риска, связанного с использованием системы, и вероятность возникновения в ней сбоев.

Оценка и мониторинг надежности позволяют предотвратить простои системы (или сократить продолжительность таковых), а также снизить частоту проявления ошибок, влияющих напрямую на пользователей.



Модель SQuaRE: эффективность

Эффективность — один из ключевых показателей для высокопроизводительных и высоконагруженных систем с высокой алгоритмической сложностью и массовыми транзакциями. Оценка эффективности предполагает рассмотрение, в том числе:

- **производительности операций доступа к данным и управления данными;**
- **управления памятью, сетевыми и дисковыми ресурсами;**
- **соблюдения правил кодирования: ЯВУ, SQL.**

Модель SQuaRE: безопасность

Критерий **безопасности** показывает, насколько вероятно обнаружение критических уязвимостей и нарушение (взлом) защиты системы по причине некачественного написания кода или неудачной архитектуры, и учитывает:

- **соблюдение правил кодирования наиболее уязвимых мест** (проверка ввода, защита от SQL-инъекций, доступ к системным функциям и т.д.);
- **корректную обработку ошибок и исключений**.

Модель SQuaRE: удобство сопровождения

Критерий **удобства сопровождения** дает интегральную оценку возможности адаптации, переноса системы на другую платформу и передачи проекта между командами разработчиков. Его оценка предполагает, в числе прочего, анализ:

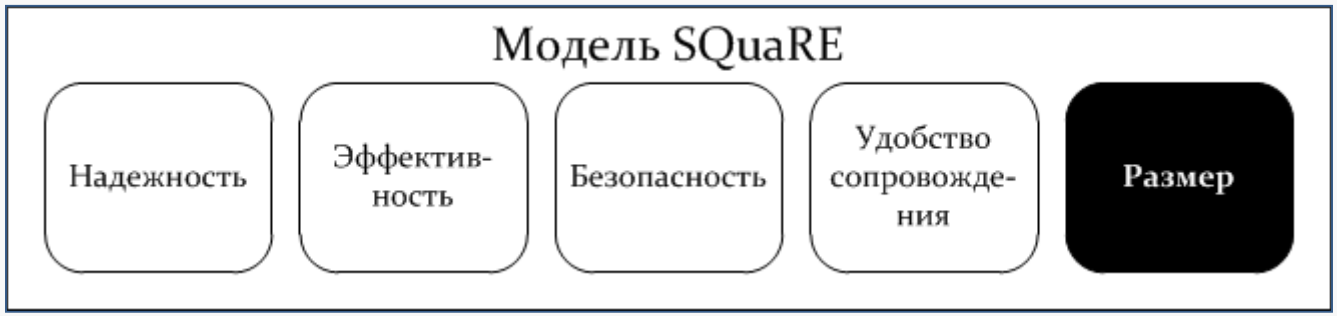
- **наличия документации и удобства чтения исходного кода;**
- **сложности транзакций и алгоритмов;**
- **применения «грязных» техник;**
- **связанности и переносимости кода.**

Модель SQuaRE: размер кода

Не будучи в буквальном смысле показателем качества, **размер исходного кода** влияет на возможность сопровождения системы и позволяет оценить трудозатраты на разработку в ретроспективе («что сделано?») и перспективе («что предстоит сделать?»).

Размер продукта оценивается, к примеру:

- **по количеству строк исходного кода** (SLOC — source line of code);
- **по количеству артефактов** (файлов, пакетов, классов, методов, таблиц и т.д.).



Метрики в модели качества ISO 9126

В ч. 2 и ч. 3 стандарта ISO 9126 введены метрики качества ПО, например:

- **полнота и корректность реализации функций** (пригодность);
- **отношение количества найденных дефектов к прогнозному** (завершенность);
- **отношение количества проведенных тестов к общему их числу** (завершенность);
- **отношение количества доступных проектных документов к указанному в реестре** (удобство анализа).

В трактовке ISO 9126 качество ПО можно повысить, не внося в него изменений.

Сложность как атрибут качества ПО

Сложность как атрибут качества ПО допускает множество различных интерпретаций, при этом **с трудом поддается количественной оценке**, а многие оценки представляют скорее академический интерес.

Например:

- **объем ресурсов**, расходуемых системой (компьютером, человеком) при взаимодействии с программой на решение задачи (исполнение кода, хранение данных, кодирование, отладку и т.д.) (Basili, 1980);
- линейные функции **количества операндов и операторов** (Halstead, 1977).

Примером «удачной» метрики сложности является оценка **цикломатической сложности** управляющего графа программы (McCabe, 1976).

Управление сложностью исходного кода. Влияние модульности

Практическую ценность представляет **опосредованное восприятие сложности** ПО участниками проекта по разработке через набор легко вычислимых или наглядных метрик и артефактов:

- количество, размер и связность единиц трансляции;
- соблюдение общепринятых правил ОО-проектирования;
- соблюдение соглашений о моделировании, кодировании.

«Несложный» код отличают:

- лаконичность;
- модульность;
- слабая связанность;
- использование архитектурных шаблонов;
- соблюдение правил оформления кода;
- систематическая обработка ошибок.

Предварительное проектирование и сложность кода

Снижению сложности ПО способствует **предварительное проектирование с целью разработки архитектуры** (дизайна) в соответствии с заданными критериями качества (см. ниже) и с учетом ее реализуемости на выбранном языке.

Критерии качества архитектуры, как правило, обеспечивают:

- возможность повторного использования;
- гибкость настройки;
- расширяемость и переносимость;
- структурированность и модульность;
- понятность и простоту (в том числе взаимодействия компонентов).

«Технический долг»: его накопление и снижение

«Технический долг» (англ. technical debt) — метафора, введенная У. Каннингемом (Ward Cunningham) для обозначения:

- временных архитектурных решений;
- применения устаревших или устаревающих технологий;
- требующих устранения ошибок и «мертвого» кода;
- нереализованных тестов;
- не выполненных работ по рефакторингу продукта.

Накоплению технического долга способствуют:

- длительность разработки (несколько лет);
- раздробленность коллектива разработчиков на небольшие команды.

Стратегии снижения технического долга предполагают:

- формирование выделенной команды;
- введение «технического налога».

Стандарты и стили кода

Стандарты и руководства по стилю оформления объектно-ориентированного исходного программного кода:

- **закрепляют эмпирические правила** организации исходного кода;
- **отражают многолетний опыт** практического программирования многих специалистов;
- **описывают рекомендуемое применение** идиоматики языка программирования;
- **обеспечивают единый стиль** оформления классов;
- **помогают справиться со сложностью** классов в процессе развития программы.

Соглашение о кодировании и его роль в командной разработке ПО

Соглашение о кодировании (англ. coding standard) — документ:

- регламентирующий подходы к оформлению исходного кода на языке высокого уровня или ручной верстки на языке разметки;
- (опционально) имеющий статус локального правового акта организации;
- действующий в рамках организации или проектного офиса (реже — отдельного проекта или проектов).

Преимущества заключения соглашения о кодировании:

- легкость включения в проект разработки новых специалистов;
- удобство чтения и простота понимания и инспекции исходного кода;
- формирование важных в командной разработке навыков и привычек оформления результатов труда;
- легкость трассировки изменений и ручного контроля версий.

Правила организации и способы записи исходного кода на языке C++

Эмпирические **правила организации исходного кода** на языке C++ — **результат многолетнего опыта** промышленной разработки специалистов по всему миру:

- объявления классов, как правило, хранятся в заголовочных файлах с именами <имя класса>.{h | hpp}, исходный код реализации в основном хранится в исходных файлах с именами <имя класса>.{c | C | cpp};
- члены класса перечисляются в порядке назначенных им уровней доступа: public, protected, private;
- подставляемые функции обычно выделяются из интерфейса и со спецификатором inline размещаются в заголовочном файле после объявления класса (оптимальное разделение достигается размещением определений подставляемых функций в отдельном заголовочном файле);
- полностью заголовочный файл помещается в директивы условной компиляции во избежание повторного включения в сборку при вложенных директивах #include.

Ортодоксальная каноническая форма класса (1 / 2)

Ортодоксальная каноническая форма (ОКФ) класса — одна из **важнейших идиом** C++, согласно которой класс должен содержать:

- конструктор по умолчанию: `T::T()`;
- конструктор копирования: `T::T(const T&)`;
- операцию-функцию присваивания: `T& T::operator=(const T&)`;
- деструктор: `T::~T()`.

ОКФ **обеспечивает**:

- единый стиль оформления классов;
- помогает справиться со сложностью классов в процессе развития программы.

Ортодоксальная каноническая форма класса (2 / 2)

ОКФ класса **следует использовать, когда**:

- необходимо обеспечить поддержку присваивания для объектов класса или передачу их по значению в параметрах функций;
- объект создает указатели на объекты, для которых применяется подсчет ссылок;
- деструктор класса вызывает operator delete для атрибута класса.

ОКФ класса **желательно использовать для всех классов**, не ограничивающихся агрегированием данных аналогично структурам языка C.

Отклонения от ОКФ позволяют реализовать нестандартные аспекты поведения класса.

Комментирование и документирование кода

Одним из показателей качества исходного кода является его **самодокументируемость** (англ. self-descriptiveness). Соблюдение принципа самодокументирования **обеспечивает**:

- понятность кода без обращения к проектной документации;
- соответствие исходного кода «внутренней программной документации».

Самодокументируемости исходного кода **способствуют**:

- единообразие нотации;
- значимость (осмысленность) идентификаторов (противоположное — анти-шаблон «загадочный код» (англ. cryptic code));
- наличие аннотаций для артефактов (переменных, классов, методов и т.д.) и комментариев в местах, трудных для понимания;
- модульность решения, отсутствие монолитных артефактов большого размера.

Жизненный цикл разработки ПО (1 / 2)

Жизненный цикл (ЖЦ, англ. life cycle) — ряд стадий развития программной системы от замысла до разработки, производства, эксплуатации и вывода из последней. Смена стадий ЖЦ сопровождается сменой основного вида работ (ср. анализ и испытания) и точки зрения на систему.

Частным случаем ЖЦ программной системы является **ЖЦ ее производства** (англ. software development process).

Для эффективной коммуникации идеи ЖЦ используется **модель жизненного цикла** (англ. life cycle model) — основа процессов и действий, относящихся к ЖЦ, которая служит общей ссылкой для установления связей и взаимопонимания сторон (см. ГОСТ Р ИСО/МЭК 15288–2005. Информационная технология. Системная инженерия. Процессы жизненного цикла систем).

Жизненный цикл разработки ПО (2 / 2)

Наглядным примером простейшего **хронологического представления** ЖЦ является графический язык описания, используемый в ISO/IEC TR 19760.

Концепция [Concept]	**Разработка** [Development]	**Производство** [Production]	**Использование** [Utilization]	**Поддержка** [Support]	**Изъятие из эксплуатации** [Retirement]

За десятилетия существования программная инженерия породила **значительное число признанных моделей разработки ПО**, что означает отсутствие на сегодняшний день единственной модели…

- принятой во всем мире;
- подходящей для любой ситуации.

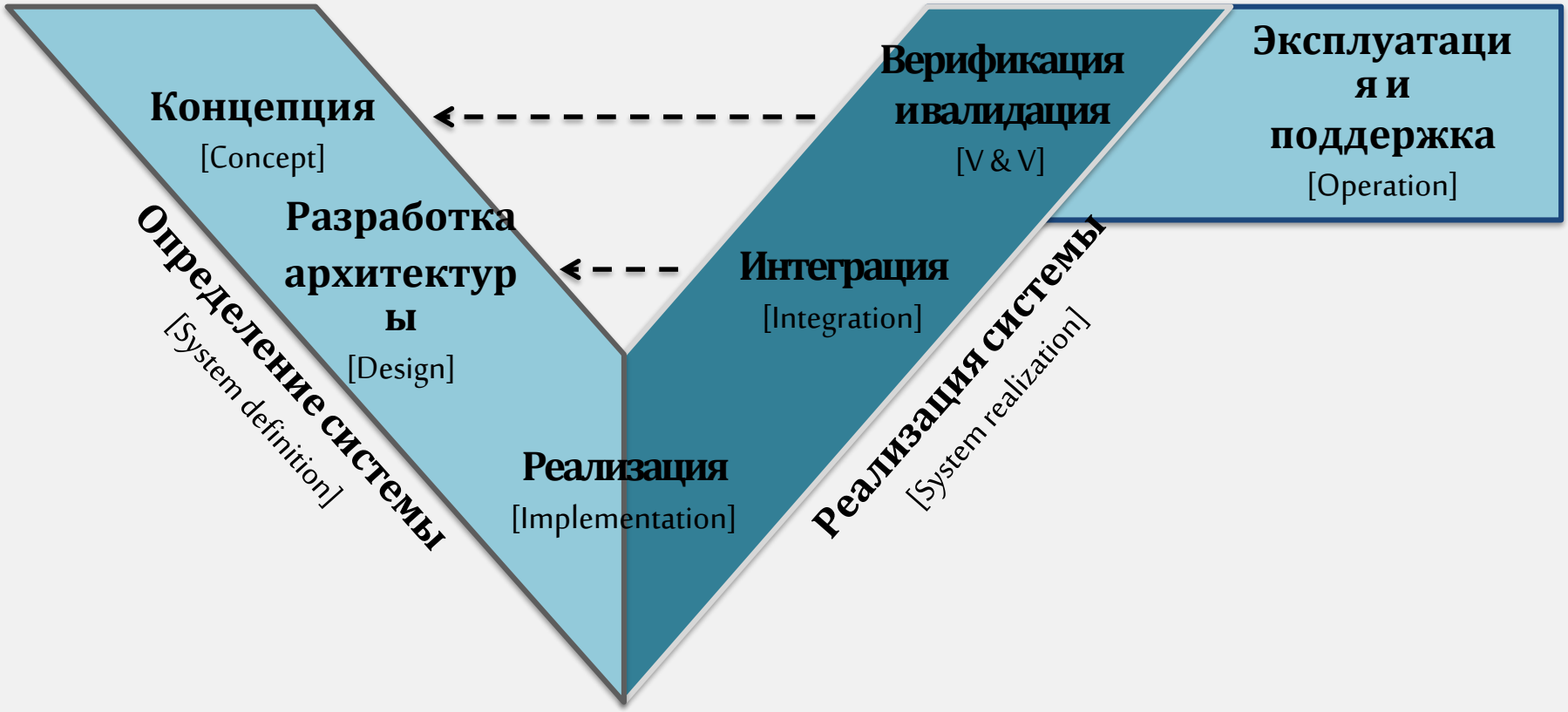
В результате вычислительноемкие системы следуют моделям ЖЦ, которые в значительной мере итеративны и делают выраженный акцент на прототипировании.

Каскадная и итеративная модели жизненного цикла ПО

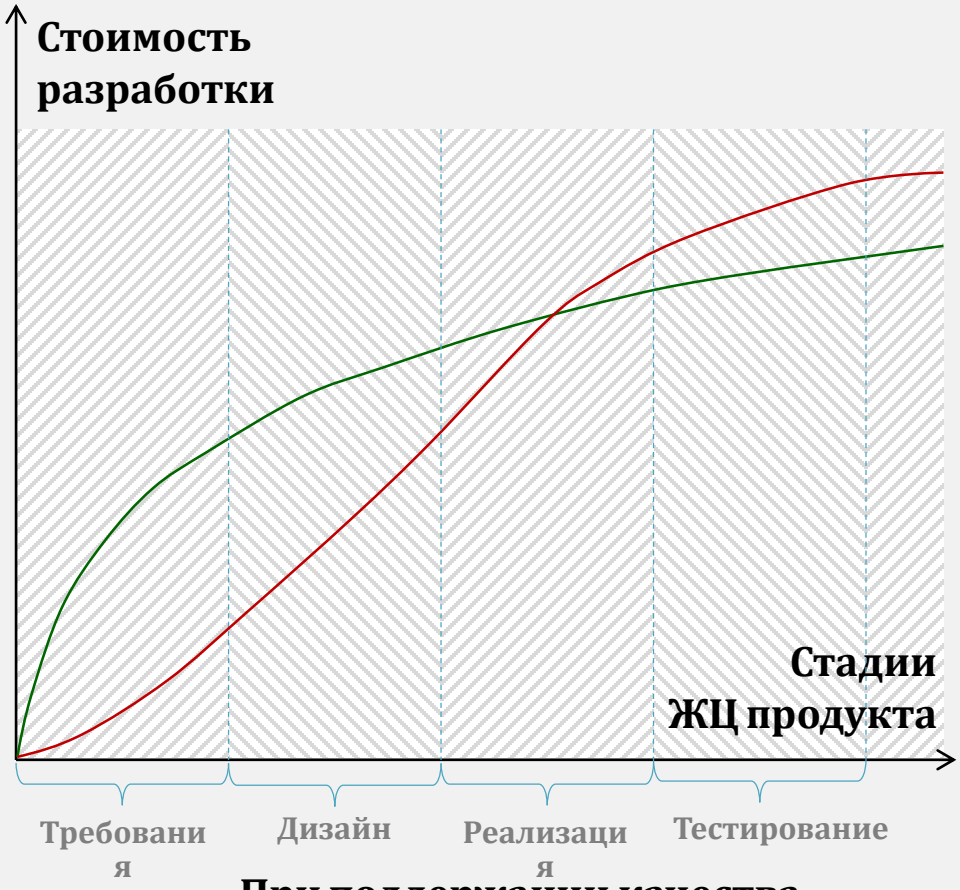
Классическая **каскадная**, или «**водопадная**», **модель** (англ. waterfall model) предполагает последовательное (во времени) однократное прохождение этапов жизненного цикла разработки ПО (фаз проекта) с жестким предварительным планированием в контексте однажды и целиком определенных требований к ПО.

Итеративная модель предполагает разбиение жизненного цикла на последовательность итераций («мини-проекты»), включающие все фазы проекта в применении к меньшим фрагментам функциональности. С завершением каждой итерации продукт развивается **инкрементально**.

V–модель жизненного цикла разработки программных систем



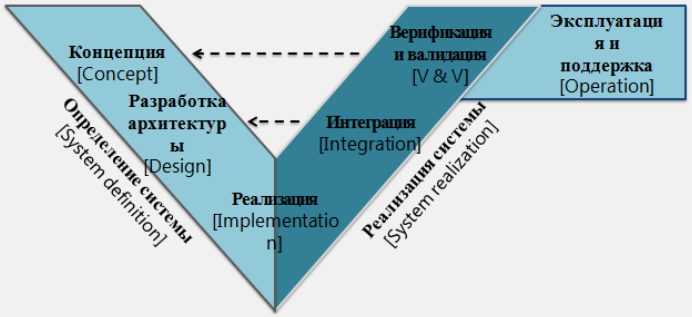
V–модель и экономическая эффективность поддержания качества



1

Хорошая новость

жертвуя качеством, можно быстрее «продвинуть» продукт по ранним стадиям разработки



2

Плохая новость

по окончании стадии реализации за принесенное в жертву качество придется платить

«Гибкие» методологии разработки

«Гибкие» (англ. agile) **методологии разработки** основаны на использовании итеративной модели жизненного цикла разработки ПО, динамическом формировании требований и постоянном взаимодействии внутри самоорганизующихся рабочих групп.

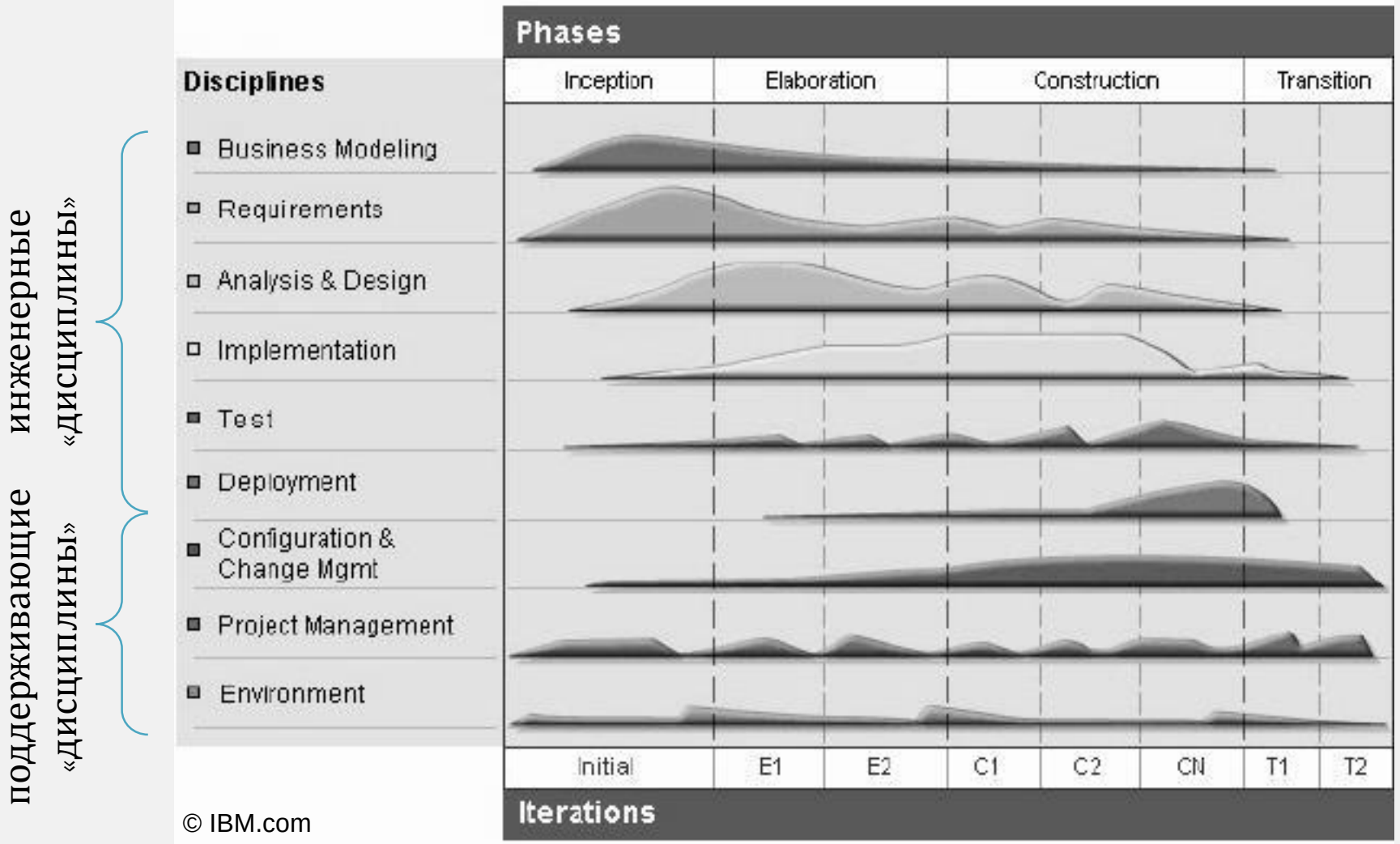
Отличительные черты:

- короткий цикл разработки;
- непосредственное общение (в том числе с заказчиком);

Основные идеи:

- личности и их взаимодействия важнее процессов и инструментов;
- работающий продукт важнее полной документации;
- сотрудничество с заказчиком важнее договорных обязательств;
- реакция на изменения важнее следования плану.

«Горбатая диаграмма» в Унифицированном процессе



Практикум №7

Постановка задачи

Произвести статический анализ исходного кода приложения группы-партнера, в том числе на наличие анти-шаблонов, установить точки накопления «технического долга».

Осуществить рефакторинг собственного приложения в точках с наивысшим приоритетом, добиться самодокументируемости исходного кода системы.

Цель — повысить статическое качество исходного кода и архитектуры системы, сократить «технический долг» и заложить основы дальнейшего развития проекта.

ТЕХНОСФЕРА

Спасибо за внимание!

Алексей Петров

Приложения

Идиома №2а. «Автоматическая» сборка мусора. Подсчет указателей (1 / 3)

```
class StringRep {      // класс-представление
friend class String;
public:
    StringRep() { *(_rep = new char[1]) = '\0'; }

    StringRep(const StringRep& s) {
        strcpy(_rep = new char[strlen(s._rep) + 1], s._rep);
    }

    StringRep(const char *s) {
        strcpy(_rep = new char[strlen(s) + 1], s);
    }
```

Идиома №2а. «Автоматическая» сборка мусора. Подсчет указателей (2 / 3)

```
// class StringRep
    ~StringRep()        { delete [] _rep; }
    int length() const { return strlen(_rep); }
private:
    char*       _rep;   // общее представление
    int         _count; // количество указателей
};

class String {          // класс-строка
public:
    String() { (p = new StringRep())->_count = 1;  }
    String(const String& s) { (p = s.p)->_count++; }
```

Идиома №2а. «Автоматическая» сборка мусора. Подсчет указателей (3 / 3)

```
// class String
    String(const char *s) {
        (p = new StringRep(s))->_count = 1;
    }
    StringRep* operator->() const { return p; }
    String& operator=(const String& q) {
        if(--p->_count <= 0 && p != q.p) delete p;
        (p = q.p)->_count++; return *this;
    }
    ~String() { if(--p->_count <= 0) delete p; }
private:
    StringRep *p;               // общее представление
};
```

Идиома №5. Модификация типа и размера класса на месте

Модификация типа и размера класса на месте — «нефабричный виртуальный конструктор», который **используется, когда**:

- тип и размер объекта неизвестны заранее;
- в отличие от идиомы «конверт/письмо», неприемлем дополнительный уровень косвенных обращений;
- задача связана с управлением или использованием системных ресурсов.

Решение:

- выборочная перегрузка операции new с возможностью замещения объекта базового класса объектом производного класса;
- построение объектов в заранее выделенном буфере со смещением базового адреса свободного пула за границу размещенных данных.

Недостатки:

- фрагментация памяти.

Модификация типа и размера класса (1 / 2)

```
class Base;
inline void *operator new(std::size_t, Base *b){ return b; }
unsigned char buf[BUF_SIZE], *pbuf = &buf;

class Base {   // класс, «создаваемый» пользователем
public:
    void *operator new(std::size_t) { return (void*) pbuf; }
    Base() { /* ПУСТО */ }
    Base(T t) {
        // ...
        (void) new(this) Derived(t);
        // ...
    }
};
```

Модификация типа и размера класса (2 / 2)

```
// выравнивание адреса a на границу в b байт
inline int round(int a, int b) {
    return ((a + b - 1) / b) * b;
}

class Derived : public Base {
private:
    Derived(T t) : Base() {
        // ...
        // выравнивание на границу DWORD
        pbuf += round(sizeof(Derived), 0x04);
    }
};
```

Рефакторинг при составлении методов

К методам рефакторинга объектно-ориентированного кода при составлении методов следует отнести:

- **выделение метода** (англ. Extract Method);
- **встраивание метода** (англ. Inline Method);
- **встраивание временной переменной** (англ. Inline Temp);
- **замена временной переменной вызовом метода** (англ. Replace Temp with Query);
- **введение поясняющей переменной** (англ. Introduce Explaining Variable);
- **расщепление временной переменной** (англ. Split Temporary Variable);
- **удаление присваиваний параметрам** (англ. Remove Assignments to Parameters);
- **замена метода объектом методов** (англ. Replace Method with Method Object);
- **замещение алгоритма** (англ. Substitute Algorithm).

Рефакторинг при перераспределении обязанностей

К методам рефакторинга объектно-ориентированного кода при перераспределении обязанностей относятся:

- **перемещение метода** (англ. Move Method);
- **перемещение атрибута** (англ. Move Attribute);
- **выделение класса** (англ. Extract Class);
- **встраивание класса** (англ. Inline Class);
- **сокрытие делегирования** (англ. Hide Delegate);
- **удаление посредника** (англ. Remove Middle Man);
- **введение внешнего метода** (англ. Introduce Foreign Method);
- **введение локального расширения** (англ. Introduce Local Extension).

Рефакторинг при организации данных (1 / 2)

К методам рефакторинга объектно-ориентированного кода при организации данных относятся (начало):

- **самоинкапсуляция атрибута** (англ. Self Encapsulate Attribute);
- **замена значения данных объектом** (англ. Replace Data Value with Object);
- **замена значения ссылкой** (англ. Change Value to Reference);
- **замена ссылки значением** (англ. Change Reference to Value);
- **замена массива объектом** (англ. Replace Array with Object);
- **дублирование видимых данных** (англ. Duplicate Observed Data);
- **замена магического числа символической константой** (англ. Replace Magic Number with Symbolic Constant).

Рефакторинг при организации данных (2 / 2)

К методам рефакторинга объектно-ориентированного кода при организации данных относятся (окончание):

- **инкапсуляция атрибута** (англ. Encapsulate Attribute);
- **инкапсуляция коллекции** (англ. Encapsulate Collection);
- **замена записи классом данных** (англ. Replace Record with Data Class);
- **замена кода типа классом** (англ. Replace Type Code with Class);
- **замена кода типа подклассами** (англ. Replace Type Code with Subclasses);
- **замена кода типа состоянием / стратегией** (англ. Replace Type Code with State / Strategy);
- **замена подкласса атрибутами** (англ. Replace Subclass with Fields).

Рефакторинг при упрощении ветвлений

К методам рефакторинга объектно-ориентированного кода при упрощении ветвлений относятся:

- **декомпозиция ветвления** (англ. Decompose Conditional);
- **консолидация условного выражения** (англ. Consolidate Conditional Expression);
- **консолидация повторяющихся фрагментов ветвлений** (англ. Consolidate Duplicate Conditional Fragments);
- **удаление управляющего флага** (англ. Remove Control Flag);
- **замена вложенных ветвлений граничным оператором** (англ. Replace Nested Conditional with Guard Clauses);
- **замена ветвления полиморфизмом** (англ. Replace Conditional with Polymorphism);
- **введение объекта Null** (англ. Introduce Null Object);
- **введение утверждения** (англ. Introduce Assertion).

Рефакторинг при упрощении вызовов методов (1 / 2)

К методам рефакторинга объектно-ориентированного кода при упрощении вызовов методов относятся (начало):

- **переименование метода** (англ. Rename Method);
- **добавление параметра** (англ. Add Parameter);
- **удаление параметра** (англ. Remove Parameter);
- **разделение запроса и модификатора** (англ. Separate Query from Modifier);
- **параметризация метода** (англ. Parameterize Method);
- **замена параметра явными методами** (англ. Replace Parameter with Explicit Methods);
- **сохранение всего объекта** (англ. Preserve Whole Object);
- **замена параметра вызовом метода** (англ. Replace Parameter with Method).

Рефакторинг при упрощении вызовов методов (2 / 2)

К методам рефакторинга объектно-ориентированного кода при упрощении вызовов методов относятся (окончание):

- **введение граничного объекта** (англ. Introduce Parameter Object);
- **удаление метода установки значения** (англ. Remove Setting Method);
- **сокрытие метода** (англ. Hide Method);
- **замена конструктора фабричным методом** (англ. Replace Constructor with Factory Method);
- **инкапсуляция нисходящего преобразования типа** (англ. Encapsulate Downcast);
- **замена кода ошибки на исключение** (англ. Replace Error Code with Exception);
- **замена исключения на проверку** (англ. Replace Exception with Test).

Рефакторинг при решении задач обобщения (1 / 2)

К методам рефакторинга объектно-ориентированного кода при его обобщении относятся (начало):

- **подъем атрибута** (англ. Pull Up Attribute);
- **подъем метода** (англ. Pull Up Method);
- **подъем тела конструктора** (англ. Pull Up Constructor Body);
- **спуск атрибута** (англ. Push Down Attribute);
- **спуск метода** (англ. Push Down Method);
- **выделение производного класса** (англ. Extract Subclass);
- **выделение базового класса** (англ. Extract Superclass).

Рефакторинг при решении задач обобщения (2 / 2)

К методам рефакторинга объектно-ориентированного кода при его обобщении относятся (окончание):

- **выделение интерфейса** (англ. Extract Interface);
- **свертывание иерархии** (англ. Collapse Hierarchy);
- **формирование шаблона** (англ. Form Template);
- **замена наследования делегированием** (англ. Replace Inheritance with Delegation);
- **замена делегирования наследованием** (англ. Replace Delegation with Inheritance).

Вопросы оптимизации

Оптимизация объектно-ориентированного программного кода может осуществляться:

- **на основе** статистики инструментальных **замеров** («профилирования» системы или ее частей);
- **исходя из экспертного опыта** в системной архитектуре и разработке.

При этом **предметом оптимизации** могут выступать:

- объем памяти данных;
- размер объектного кода;
- производительность приложения.

Оптимизация программ по производительности (1 / 2)

Оптимизация программного кода **по производительности** требует обращать внимание на следующие аспекты (начало):

- теоретические *O*-оценки производительности и обоснованность совместного применения алгоритмов и структур данных;
- использование раннего или позднего связывания;
- программное кэширование результатов «дорогостоящих» операций;
- эффективная работа с аппаратной кэш-памятью ЦП ЭВМ;
- использование отложенной инициализации больших или труднодоступных информационных объектов;
- специализация шаблонов глобальных функций, классов и методов.

Оптимизация программ по производительности (2 / 2)

Оптимизация программного кода **по производительности** требует обращать внимание на следующие аспекты (окончание):

- использование пакетного (блочного, потокового) режима выполнения системных и прикладных операций, в том числе по управлению памятью;
- наличие необязательных (устранимых) уровней косвенности;
- возможность полного отказа от выполнения операций или замены их более «щадящими» вариантами (например, тривиальный деструктор);
- возможность замены высокоуровневой реализации низкоуровневым аналогом (выполняемым путем поразрядных манипуляций с памятью);
- возможность «приближения» вычислений к пользователю (переноса вычислительного процесса на сторону клиента).

Оптимизация программ по размеру объектного кода и памяти данных

Оптимизация программного кода **по размеру объектного кода и объему памяти данных** требует обращать внимание на:

- обобществление данных, в том числе через использование «оптимизирующих» шаблонов проектирования (напр., «приспособленец»);
- возможность унификации алгоритмов;
- целесообразность специализации шаблонов глобальных функций, классов, а также их методов;
- целесообразность встраивания глобальных функций и методов;
- наличие необязательных (устранимых) уровней косвенности;
- возможность полного отказа от выполнения операций или замены их более «щадящими» вариантами (например, тривиальный деструктор).

Метрики качества по МакКолу (1 / 2)

- **Auditability** — удобство проверки.
- **Accuracy** — точность управления и вычислений.
- **Communication commonality** — степень стандартности интерфейсов.
- **Completeness** — функциональная полнота.
- **Consistency** — однородность (правил и пр.).
- **Data commonality** — степень стандартности форматов данных.
- **Error tolerance** — устойчивость к ошибкам.
- **Execution efficiency** — эффективность работы.
- **Expandability** — расширяемость.
- **Generality** — широта области использования.

Метрики качества по МакКолу (2 / 2)

- **Hardware independence** — независимость от аппаратной платформы.
- **Instrumentation** — полнота протоколирования ошибок (событий).
- **Modularity** — модульность.
- **Operability** — удобство использования.
- **Security** — защищенность.
- **Self-documentation** — самодокументированность.
- **Simplicity** — простота работы.
- **Software system independence** — независимость от программной платформы.
- **Traceability** — возможность соотнесения с требованиями.
- **Training** — удобство обучения.

Дополнительные атрибуты качества по Боэму

- Clarity — ясность.
- Documentation — документированность.
- Economy — экономическая эффективность.
- Functionality — функциональность.
- Modifiability — удобство внесения изменений.
- Resilience — устойчивость.
- Understandability — понятность.
- Validity — адекватность.

Атрибуты эффективности (выше) и функциональности (ниже) в модели ISO 9126

- **Efficiency compliance** — соответствие стандартам эффективности.
- **Resource utilization** — использование ресурсов.
- **Time behavior** — временные характеристики.

- **Accuracy** — точность, правильность.
- **Compliance** — соответствие стандартам.
- **Interoperability** — способность к взаимодействию.
- **Security** — защищенность.
- **Suitability** — пригодность для конкретных задач.

Атрибуты удобства сопровождения (выше) и переносимости (ниже) в модели ISO 9126

- **Analyzability** — удобство анализа.
- **Changeability** — удобство внесения изменений.
- **Maintenance compliance** — соответствие стандартам удобства сопровождения.
- **Stability** — риск проявления побочных эффектов при изменении.
- **Testability** — удобство проверки.

- **Adaptability** — удобство адаптации.
- **Co-existence** — способность сосуществовать с другим ПО.
- **Installability** — удобство установки.
- **Portability compliance** — соответствие стандартам переносимости.
- **Replaceability** — удобство замены другого ПО данным.

Атрибуты надежности (выше) и удобства использования (ниже) в модели ISO 9126

- **Fault tolerance** — отказоустойчивость.
- **Maturity** — завершенность, зрелость (обратна к частоте отказов).
- **Recoverability** — способность к восстановлению после сбоев.
- **Reliability compliance** — соответствие стандартам надежности.

- **Attractiveness** — привлекательность.
- **Learnability** — удобство обучения.
- **Operability** — работоспособность.
- **Understandability** — понятность.
- **Usability compliance** — соответствие стандартам удобства использования.

Примеры нотаций (1 / 2)

Основные:

- «верблюжья нотация» (англ. CamelCase);
- стиль_через_подчеркивание (англ. under_score).

Примером смешанного варианта выступает **«венгерская нотация»** (Ч. Симони (C. Simonyi), Microsoft, 1999) — соглашение о префиксации идентификаторов в соответствии с их функциональной нагрузкой:

```
m_pHead;        // указатель-атрибут класса
g_nObjCount;    // глобальная целочисленная переменная
CWarrior;       // класс
szConnPar;      // строковый (char*) локальный объект
```

Примеры нотаций (2 / 2)

Google C++ Style Guide — определяет не только порядок оформления кода, но и допустимость использования отдельных языковых элементов.

Сложившийся на сегодняшний день стиль написания объектно-ориентированного кода и современные IDE позволяют не указывать тип переменной в имени, а сосредоточиться на **«правильном» форматировании** блоков, **расстановке разделителей**, **знаков операций**, **левых отступов** и т.д.

Методологии TDD, Scrum

Разработка через тестирование (англ. Test-Driven Development, TDD) — методология разработки, реализующая концепцию «сначала тест» и состоящая в последовательном расширении набора тестов, покрывающих требуемую функциональность, и написании кода, позволяющего пройти существовавшие и вновь введенные тесты.

Scrum — методология итеративной разработки с акцентом на контроль процесса разработки ПО, в которой:

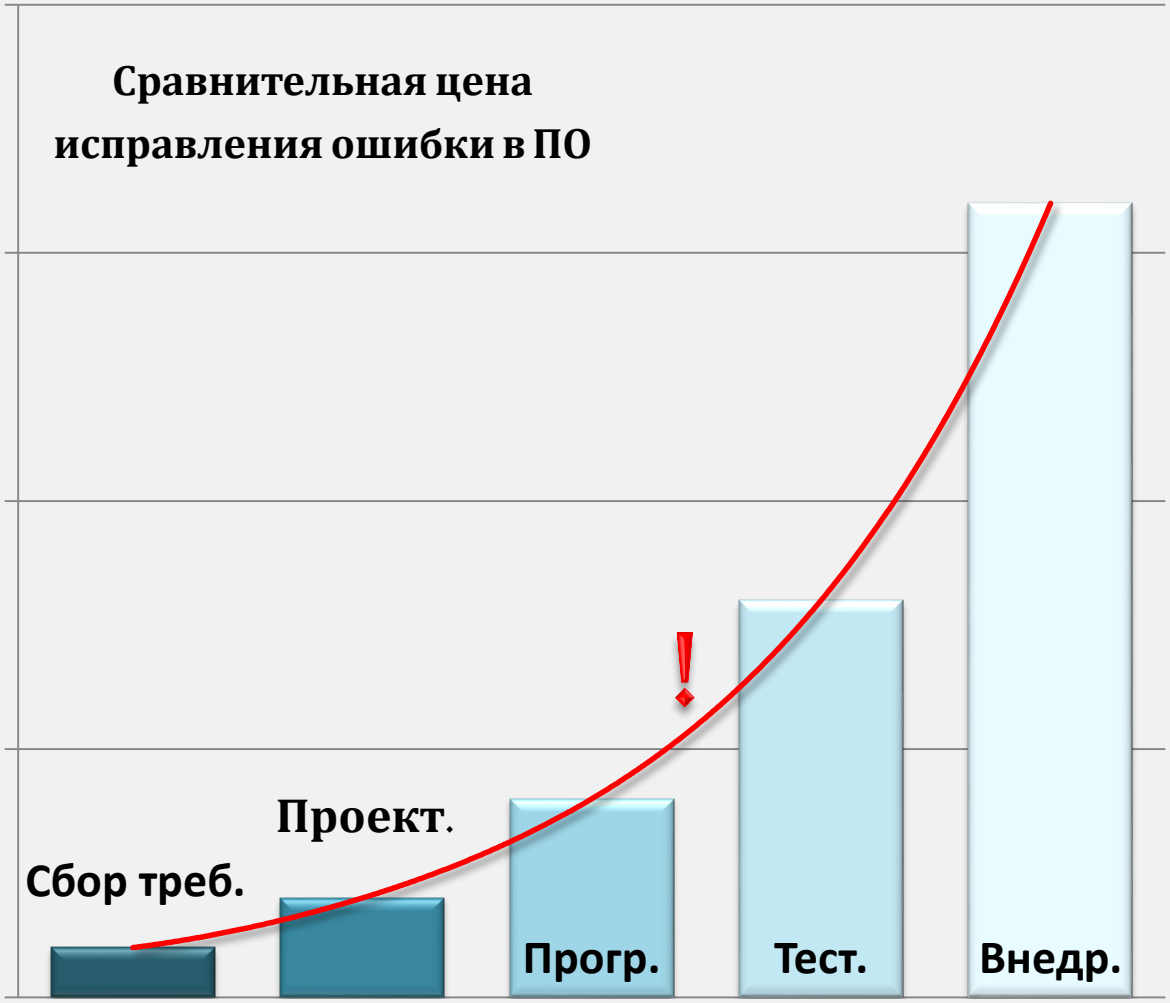
- каждая итерация («спринт») имеет фиксированную продолжительность;
- требования к продукту упорядочиваются по их важности, отражаются в документах Product Backlog и Sprint Backlog и фиксируются на время спринта.

Руководство SWEBoK

Software Engineering Body of Knowledge (SWEBoK) – «Свод знаний в области программной инженерии», подготовленный при участии IEEE и призванный определить набор профессиональных знаний и рекомендуемые практики в следующих областях (англ. knowledge area):

Требования к ПО	Управление конфигурациями ПО
Проектирование ПО	Управление программной инженерией
Конструирование ПО	Процесс программной инженерии
Тестирование ПО	Методы и инструменты программной инженерии
Сопровождение ПО	Качество ПО

Экономика и ответственность (1 / 2)



Программисты должны получать **качественные** проектные артефакты с **качественно** описанными требованиями к ПО. Сравнительная цена ошибки (стоимость ее исправления) на этапе программирования выше, чем на этапе проектирования и тем более — на этапе сбора требований к ПО (см. график)

Экономика и ответственность (2 / 2)

Структура совокупных затрат на выпуск и сопровождение ПО

Со ссылкой на кн.: Martin, J., McClure, C. Software Maintenance — The Problem and Its Solutions, Prentice Hall, Englewood Cliffs, New Jersey, 1983 — С. Канер и др. (2001) приводят следующую оценку распределения статей затрат на выпуск и сопровождение ПО в разрезе жизненных циклов производимой системы (см. график).