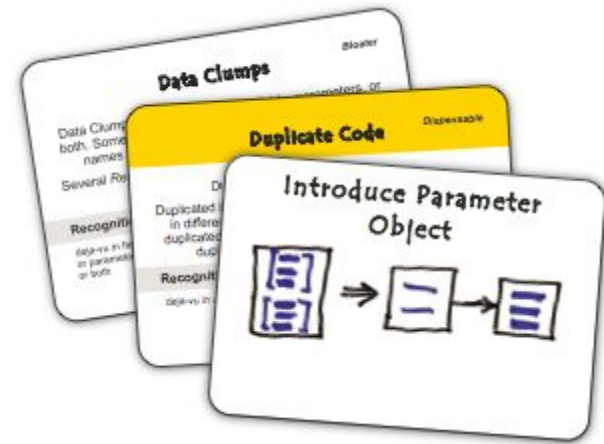


Refactoring Lecture Outline



- 1) Рефакторинг. Зачем и когда?
- 2) Рефакторинг **vs.**
Оптимизация
- 3) Признаки кода «с душком»
- 4) Приемы рефакторинга

Refactoring



**Martin
Fowler**

Refactoring: Improving the
Design of Existing Code (1999)

Рефакторинг

(сущ.)

Изменение во внутренней структуре программного обеспечения, имеющее целью облегчить понимание его работы и упростить модификацию, не затрагивая наблюдаемого поведения

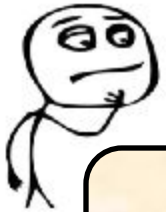
Рефакторинг

(глагол.)

Процесс изменения структуры программного обеспечения путем применения рефакторингов

“Improving the design after it has
been written”

Example



Каким код был

```
double PaymentAmount()  
{  
    if ( quantity < 5 ) return 0;  
    if ( months > 12 ) return 0;  
  
    // compute amount  
}
```



Каким код стал

```
double PaymentAmount()  
{  
    if ( NoPaymentNeeded() ) return 0;  
  
    // compute amount  
}
```

Why Refactor?

- Рефакторинг улучшает результаты проектирования ПО
 - без рефакторинга структура проекта будет ухудшаться, т.к. разработчики часто вносят изменения в сам проект
- Рефакторинг облегчает понимание структуры ПО
 - вследствие улучшения структуры проекта
- Рефакторинг помогает найти ошибки
 - рефакторинг способствует более глубокому вниканию в код
- Рефакторинг позволяет быстрее писать программы
 - вследствие всех вышеуказанных преимуществ

When Refactor?

ПРИМЕНЯТЬ

- Правило «Трех страйков»
 - если, минимум, в 3 местах дублируется код, применяйте рефакторинг
- При добавлении новой функции
- Когда необходимо исправить ошибку
- Во время Code Review (анализ кода в команде)

НЕ

- Когда код *слишком* запутан (легче написать заново)
- Когда код неработоспособен
- Когда близится дата сдачи проекта

Principles in Refactoring

**Рефакторинг должен
быть:**

- ✓ Систематичный
- ✓ Поэтапный
- ✓ Безопасный

Польза рефакторинга:

- ✓ В большинстве случаев объем кода уменьшается
- ✓ Запутанные структуры преобразуются в более простые (которые легче понимать и сопровождать)

Как надо проводить рефакторинг:

- ✓ Не гнушаться паттернами рефакторинга (см. книги Фаулера)
- ✓ Постоянное тестирование (в рамках концепции TestDrivenDevelopment - TDD)

Problems with Refactoring

- ✓ Рефакторить аспекты, связанные с БД, гораздо сложнее
- ✓ Некоторые рефакторинги требуют серьезных изменений интерфейсов
- ✓ Некоторые изменения в проектировании сложно поддаются рефакторингу

Refactoring Vs. Optimization

ОБЩЕЕ:

- Код меняется, а функциональность не меняется

ОТЛИЧИЯ:

- При рефакторинге код становится понятнее; при оптимизации производительности – , в основном, гораздо сложнее для восприятия
- При рефакторинге, в основном, код становится менее эффективным по памяти и по времени; при оптимизации – наоборот

“Bad Smells” in Code

❑ Дублирование кода (**Duplicated Code**)

Ситуации:

- один и тот же участок кода присутствует в 2 методах одного класса
- один и тот же участок кода встречается в 2 подклассах одного уровня
- дублирующийся код содержится в 2 разных классах

❑ Длинный метод (**Long Method**)

Длинные методы затрудняют понимание кода.

Соображениями о меньшей эффективности большого числа малых методов можно пренебречь

“Bad Smells” in Code

- ❑ **Большой класс (Large Class)**
Часто из-за больших классов увеличивается сцепление и уменьшается связность
- ❑ **Длинный список параметров (Long Parameter List)**
Сложен для понимания
- ❑ **Расходящиеся модификации (Divergent Change)**
Когда один тип изменений требует изменения одного подмножества частей класса, другой тип изменений – другого подмножества

“Bad Smells” in Code

- ❑ **Стрельба дробью (Shotgun Surgery)**
При выполнении любых модификаций приходится вносить много мелких изменений во многих классах
- ❑ **Завистливые функции (Feature Envy)**
Метод, который больше обрабатывает данные и вызывает функции чужого класса, чем родного
- ❑ **Группы данных (Data Clumps)**
Часто встречающиеся и используемые связки данных, не являющиеся частью одного класса

“Bad Smells” in Code

- ❑ Одержимость элементарными типами (**Primitive Obsession**)
Избегание методики обертки данных в классы
- ❑ Операторы switch (**Switch Statements**)
Часто они дублируются в коде и часто могут быть заменены полиморфизмом
- ❑ Параллельные иерархии наследования (**Parallel Inheritance Hierarchies**)
Случай «стрельбы дробью» для классов, связанных отношением наследования. При внесении изменений в один подкласс, приходится вносить изменения во все подклассы параллельных иерархий

“Bad Smells” in Code

❑ Ленивый класс (**Lazy Class**)

Класс, существование которого уже не целесообразно (например, он в свое время нужен был, но после рефакторингов в нем отпала необходимость, или это класс, добавленный для планируемой модификации, которая не была произведена)

❑ Теоретическая общность (**Speculative Generality**)

Возникает тогда, когда хотят обеспечить набор механизмов для работы с вещами, которые, **возможно**, будут нужны в будущем. Теоретическая общность может быть обнаружена, когда единственными пользователями метода или класса являются контрольные примеры (тесты)

“Bad Smells” in Code

❑ Временное поле (**Temporary Field**)

В некотором объекте свойство устанавливается / меняется только при некоторых обстоятельствах (типичный пример – вспомогательные переменные помещаются в свойства класса)

❑ Цепочки сообщений (**Message Chains**)

Объект, делающий запрос, входящий в цепочку запросов к другим объектам, зависит от структуры навигации

❑ Посредник (**Middle Man**)

Плохой признак, если класс делегирует слишком много своих действий другим классам (нужен ли он тогда сам вообще?)

“Bad Smells” in Code

- ❑ **Неуместная близость (Inappropriate Intimacy)**
Пара классов, которые слишком много знают и позволяют друг другу
- ❑ **Альтернативные классы с разными интерфейсами (Alternative Classes with Different Interfaces)**
Два или более метода классов делают практически одно и то же, но имеют разные сигнатуры
- ❑ **Неполнота библиотечного класса (Incomplete Library Class)**
Библиотечный класс не делает всего того, что Вам нужно

“Bad Smells” in Code

❑ Классы данных (**Data Class**)

Классы, которые содержат только свойства, геттеры и сеттеры для этих свойств, и ничего более (dumb data holders). Объекты должны отражать **данные и обработку данных**

❑ Отказ от наследства (**Refused Bequest**)

Подкласс игнорирует большинство методов и данных родительского класса

❑ Комментарии (**Comments (!)**)

Излишние и некачественные комментарии. Комментарии иногда используются для сокрытия некачественного кода

Refactorings

Реорганизация функций и данных

- Составление методов
- Перемещение функций между объектами
- Реорганизация данных
- Упрощение вызовов методов

Реорганизация условных выражений

Реорганизация обобщений

Refactorings

Составление методов

- Выделение метода
- Встраивание метода
- Встраивание временной переменной
- Замена временной переменной вызовом метода
- Введение поясняющей переменной
- Расщепление временной переменной
- Удаление присваиваний параметрам
- Замена метода объектом методов
- Замещение алгоритма

Extract Method

Прием «Выделение метода»

Описание: Есть участок кода, который можно сгруппировать.

Действие: Поместить участок кода в метод, название которого отвечает назначению.

Extract Method (Example)

```
class DeanOffice
{
public:
    // фрагмент кода, печатающий заголовок отчета, можно сгруппировать
    void PrintSalaryReport(void)
    {
        cout<<"Dean Office"<<endl
            <<"Donetsk National University"<<endl;

        cout<<"Salary report";
        for_each(teachers.begin(), teachers.end(), [](Teacher *t)
        {

        });
    }

    // фрагмент кода, печатающий заголовок отчета, можно сгруппировать
    void PrintStudentReport(void)
    {
        cout<<"Dean Office"<<endl
            <<"Donetsk National University"<<endl;

        if(NumberOfStudentsIsMoreThanHundred())
            cout<<"We have a lot of students here";
        else

        });
    }

    // Тело метода столь же понятно, как и его название.
    // Временная переменная numberOfStudents мешает проведению рефакто.
    bool NumberOfStudentsIsMoreThanHundred(void)
    {
        int numberOfStudents = students.size();
    }
};
```

```
class DeanOffice
{
public:
    // Проведено выделение метода.
    void PrintSalaryReport(void)
    {
        PrintReportTitle();
        cout<<"Salary report";
        for_each(teachers.begin(), teachers.end(), [](Teacher *t)
        {
            t -> PrintName();
            cout<<"Full salary: " << t -> GetFullSalary();

        });
    }

    // Проведено выделение метода.
    void PrintStudentReport(void)
    {
        PrintReportTitle();
        if(NumberOfStudentsIsMoreThanHundred())
            cout<<"We have a lot of students here";
        else
            cout<<"Where are the students??!!11";

        });
    }

    void PrintReportTitle(void)
    {
        cout<<"Dean Office"<<endl
            <<"Donetsk National University"<<endl;
    }
};
```

Refactorings

Прием «Встраивание метода»

Описание: Тело метода столь же понятно, как и его название.

Действие: Поместить тело метода в код, к-ый его вызывает, и удалить метод

```
// Проведено выделение метода.  
void PrintStudentReport(void)  
{  
    PrintReportTitle();  
    if(NumberOfStudentsIsMoreThanHundred())  
        cout<<"We have a lot of students here";  
    else  
        cout<<"Where are the students??!!11";  
  
    <<"Donetsk National University"<<endl;  
}
```

```
// Тело метода столь же понятно, как и его название.  
// Проведено встраивание временной переменной.  
bool NumberOfStudentsIsMoreThanHundred(void)  
{  
    return students.size() > 100;  
}
```

```
// Проведено выделение метода.  
// Проведено встраивание метода.  
void PrintStudentReport(void)  
{  
    PrintReportTitle();  
    if(students.size() > 100)  
        cout<<"We have a lot of students here";  
    else  
        cout<<"Where are the students??!!11";  
  
    <<"Donetsk National University"<<endl;  
}
```

```
// Возвращается сложное выражение.  
// Временная переменная TeachersPercent (после :  
double GetRating(void)  
{  
    double professorsSize = 0;  
    for_each(teachers.begin(), teachers.end(),
```

Refactorings

Прием «Встраивание временной переменной»

Описание: Есть временная переменная, к-ой один раз присваивается простое выражение, и она мешает проведению «Выделения метода».

Действие: Заменить этим выражением все ссылки на данную переменную

```
// Тело метода столь же понятно, как и его название.  
// Временная переменная numberOfStudents мешает проведению рефакторинга.  
bool NumberOfStudentsIsMoreThanHundred(void)  
{  
    int numberOfStudents = students.size();  
    return numberOfStudents > 100;  
}  
  
// Тело метода столь же понятно, как и его название.  
// Проведено встраивание временной переменной.  
bool NumberOfStudentsIsMoreThanHundred(void)  
{  
    return students.size() > 100;  
}
```

Refactorings

Прием «Замена временной переменной вызовом метода»

Описание: временная переменная используется для хранения значения выражения

Действие: преобразовать выражение в метод. Заменить все ссылки на временную переменную вызовом метода. Новый метод может быть использован в других методах

```
// Проведено введение поясняющих переменных.
// Временная переменная TeachersPercent (после 1 рефакторинга) может быть использована в другом методе.
double GetRating(void)
{
    double professorsSize = 0;
    for_each(teachers.begin(), teachers.end())
    {
        if(t -> IsProfessor())
            ++professorsSize;
    };
    ++professorsSize;
};

double teachersPercent = teachers.size() / students.size();
double professorsPercent = professorsSize / teachers.size();

return teachersPercent + professorsPercent;
}

double TeachersPercent(void)
{
    return teachers.size() / students.size();
}

// Проведено введение поясняющих переменных.
// Проведена замена переменной вызовом метода.
double GetRating(void)
{
    double professorsSize = 0;
    for_each(teachers.begin(), teachers.end())
    {
        if(t -> IsProfessor())
            ++professorsSize;
    };
    ++professorsSize;

    double professorsPercent = professorsSize / teachers.size();

    return TeachersPercent() + professorsPercent;
}
```

Refactorings

Прием «Введение поясняющей переменной»

Описание: имеется сложное выражение

Действие: поместить результат выражения или его части во временную переменную, имя которой поясняет его назначение

```
// Возвращается сложное выражение.  
// Временная переменная TeachersPercent (после 1 рефакторинга) может быть использована  
double GetRating(void)  
{  
  
    if(t -> IsProfessor())  
        ++professorsSize;  
    });  
    return teachers.size() / students.size() + professorsSize / teachers.size();  
}
```

```
rate:  
vector<Teacher *> teachers;
```

```
// Проведено введение поясняющих переменных.  
// Временная переменная TeachersPercent (после 1 рефакторинга) мож  
double GetRating(void)  
{  
  
    if(t -> IsProfessor())  
        ++professorsSize;  
    });  
  
    double teachersPercent = teachers.size() / students.size();  
    double professorsPercent = professorsSize / teachers.size();  
  
    return teachersPercent + professorsPercent;
```


Refactorings

Прием «Расщепление временной переменной»

Описание: имеется временная переменная, которой неоднократно присваивается значение, но это не переменная цикла и не для накопления результата

Действие: создать для каждого присваивания отдельную временную переменную

До

```
double temp = _width * _height;  
cout << "Square: " << temp;  
  
temp = 2 * (_width + _height);  
cout << "Perimeter: " << temp;
```

После

```
double square = _width * _height;  
cout << "Square: " << square;  
  
double perimeter = 2 * (_width + _height);  
cout << "Perimeter: " << perimeter;
```

Refactorings

Прием «Удаление присваиваний параметрам»

Описание: выполняется присваивание параметру

Действие: заменить это временной переменной

До

```
double discount( double price, int quantity ) {  
    if ( price > 1000.0 ) {  
        price *= 0.9;  
    }  
    if ( quantity >= 5 ) {  
        price *= 0.8;  
    }  
    return price;  
}
```

После

```
double discount( double price, int quantity ) {  
    double result = price;  
  
    if ( price > 1000.0 ) {  
        result *= 0.9;  
    }  
    if ( quantity >= 5 ) {  
        result *= 0.8;  
    }  
    return result;  
}
```

Refactorings

Перемещение функций между объектами

- Перемещение метода
- Перемещение поля
- Выделение класса
- Встраивание класса
- Соккрытие делегирования
- Удаление посредника
- Введение внешнего метода
- Введение локального расширения

Refactorings

Прием «Соккрытие делегирования»

Описание: объект-клиент обращается к делегируемому классу объекта

Действие: создать на объекте-сервере методы, скрывающие делегирование

До

```
class Person {
    Department m_Dep; ...
    Department* GetDepartment() { ... }
};
class Department {
    Person* m_Manager; ...
    Person* GetManager() { ... }
};

Person john; ...
Person* manager =
    john.GetDepartment().GetManager();
```

После

```
class Person {
    Department m_Dep;
    Person* GetManager() {
        return m_Dep.GetManager();
    }
};
class Department {
    Person* m_Manager; ...
};

Person john; ...
Person* manager = john.GetManager();
```

Refactorings

Реорганизация данных

- Самоинкапсуляция поля
- Замена значения объектом
- Замена значения ссылкой (и наоборот)
- Замена массива объектом
- Дублирование видимых данных
- Замена однонаправленной связи на двунаправленную (и наоборот)
- Замена магического числа символической константой
- Инкапсуляция поля
- Инкапсуляция коллекции

Refactorings

Упрощение вызовов методов

- Переименование метода
- Добавление / удаление параметра
- Разделение запроса и модификатора
- Параметризация метода
- Замена параметра явными методами
- Замена параметра вызовом метода
- Введение объекта параметров
- Удаление сеттера
- Соккрытие метода

Refactorings

Прием «Параметризация метода»

Описание: Несколько методов выполняют сходные действия, но с разными значениями, содержащимися в теле метода

Действие: Создать один метод, к-ый использует для задания разных значений параметр

До

```
void DiscountForMen()  
{...}  
  
void DiscountForWomen()  
{...}
```

После

```
void Discount( char cSex )  
{  
...  
}
```

Refactorings

Реорганизация условных выражений

- Декомпозиция условного оператора
- Консолидация условного выражения
- Консолидация дублирующихся условных фрагментов
- Удаление управляющего флага
- Замена вложенных условных операторов граничным оператором
- Замена условного оператора полиморфизмом
- Введение объекта NULL

Refactorings

Прием «Декомпозиция условного оператора»

Описание: имеется сложная условная цепочка проверок

Действие: выделить методы из условия, блоков THEN и ELSE

| | |
|--|---|
| <pre>// Сложная и неочевидная условная проверка. double GetMonthSalary(void) { int month = GetCurrentMonth(); if(month < JUNE month > AUGUST) return workMonthSalary; return workMonthSalary / 2; }</pre> | <pre>// Проведена декомпозиция условного оператора. double GetMonthSalary(void) { if(IsNotSummerMonth(GetCurrentMonth())) return workMonthSalary; return workMonthSalary / 2; }</pre> |
| <pre>// Один и тот же фрагмент кода присутствует во всех double GetFullSalary(void) { double salary = 0;</pre> | <pre>bool IsNotSummerMonth(int month) { return month < JUNE month > AUGUST; }</pre> |

Refactorings

Прием «Консолидация условного выражения»

Описание: имеется ряд проверок условия, дающих одинаковый результат

Действие: объединить их в одно условное выражение и выделить его

```
// Ряд проверок условия, дающих одинаковый результат.  
// Используется число 5, имеющее определенный смысл.  
bool CanBeMagister(void)  
{  
    if(ReceivesScholarship())  
        return true;  
    else if(bonusMarks > 5)  
        return true;  
    else if(participatedInCompetitions)  
        return true;  
    else  
        return false;  
}
```

```
// Проведена консолидация условного выражения.  
// Используется число 5, имеющее определенный смысл.  
bool CanBeMagister(void)  
{  
    if(ReceivesScholarship() || bonusMarks > 5 || participatedInCompetitions)  
        return true;  
    return false;  
}  
private:  
    // Проведен спуск поля.  
    int studentCardNumber;  
  
    int bonusMarks;
```

Refactorings

Прием «Консолидация дублирующихся условных фрагментов»

Описание: один и тот же фрагмент кода присутствует во всех ветвях условного выражения

Действие: переместить его за пределы условного выражения

До

```
...
if ( IsSpecialDeal() )
{
    total = price * 0.8;
    Send();
}
Else
{
    total = price * 0.9;
    Send();
}
```

После

```
...
if ( IsSpecialDeal() )
{
    total = price * 0.8;
}
Else
{
    total = price * 0.9;
}
Send();
```

Refactorings

Прием «Удаление управляющего флага»

Описание: имеется переменная-флаг

Действие: использовать вместо нее break или return

```
// Переменная finish используется в качестве управляющего флага в ряде булевых выражений.
```

```
// Название метода не отображает его назначение.
```

```
bool IsGoodStudent(void)
```

```
{
```

```
    bool passedAllExams = true;
```

```
    int index = 0;
```

```
    int sum = 0;
```

```
    while(index < marks.size() && passedAllExams)
```

```
    {
```

```
        sum += marks[index];
```

```
        if(marks[index] == 0)
```

```
            passedAllExams = false;
```

```
        ++index;
```

```
    }
```

```
    if(passedAllExams && sum / marks.size() > 3)
```

```
        return true;
```

```
    return false;
```

```
}
```

```
// Проведено удаление управляющего флага.
```

```
// Название метода не отображает его назначение.
```

```
bool IsGoodStudent(void)
```

```
{
```

```
    int index = 0;
```

```
    int sum = 0;
```

```
    int size = marks.size();
```

```
    while(index < size)
```

```
    {
```

```
        sum += marks[index];
```

```
        if(marks[index] == 0)
```

```
            break;
```

```
        ++index;
```

```
    }
```

```
    if(index == size && sum / size > 3)
```

```
        return true;
```

```
    return false;
```

```
}
```

Refactorings

Реорганизация обобщений

- Подъем поля / Подъем метода
- Спуск поля / Спуск метода
- Выделение подкласса
- Выделение родительского класса
- Выделение интерфейса
- Свертывание иерархии
- Формирование шаблона метода
- Замена наследования делегированием и наоборот

Refactorings

Прием «Подъем метода»

Описание: в подклассах есть методы с одинаковыми результатами

Действие: переместить их в родительский класс

| | | |
|---|--|--|
| Human.h Last modified: 09.06.2011 22:27:49, Status: modified, Kind: file | Student.h Last modified: 09.06.2011 22:27:49, Status: modified, Kind: file | |
| <pre>{ return studentCardNumber; } private: // Поле относится только к int studentCardNumber; };</pre> | Human.h Last modified: 09.06.2011 22:45:40, Status: modified, Kind: file <pre>{ return studentCardNumber; } // Проведен подъем метода. void PrintName(void) { cout<<"My name is "<<name; } private: // Поле относится только к подклассу "Student". int studentCardNumber; // Проведен подъем поля. char *name; };</pre> | <pre>Human " тоже есть этот метод. " <<name; используется в качестве управляющего е отображает его назначение. ; " тоже есть это поле.</pre> |
| Teacher.h Last modified: 09.06.2011 22:27:49, Status: modified, Kind: file <pre>class Teacher : public Human { public: // В классе "Student" тоже void PrintName(void) { cout<<"My name is "<<name; } };</pre> | <pre>int bonusmarks, bool participatedInCompetitions; vector<int> marks;</pre> | |

Refactorings

Прием «Спуск метода»

Описание: в родительском классе есть поведение, относящееся только к некоторым его подклассам

Действие: переместить поведение в соответствующие подклассы