9 Refactoring

• Refactoring changes the software structure but does not change the functionality of the program
  – important activity during evolution

• Refactoring consists of behaviour preserving transformations
  – Fowler: Refactoring (book, web site)
    http://www.refactoring.com/

• Restructuring, reengineering
Two roles of refactoring

- Prefactoring
  - before attempting the actualization
    - refactoring will make actualization easier
- Postfactoring
  - prepares software for future evolution
  - cleans up the code
Examples of refactoring

• Rename an entity

• Encapsulate part of the code as a function
  – opposite: expand a function in a place of call

• Move function into/out of class

• Merge and divide classes
  – factor out a base class, component class
Extract function

- During the software evolution some functions may grow to be too large
- Or we may need to separate two concepts the function currently deals with
- Extracting part of the function into another function will make it
  – easier to understand
  – reusable
void foo(char c, int& count) {
    int i, len;
    char str[MAX];
    cin >> str;
    len = strlen(str);
    count = 0;
    for (i = 0; i <= len; i++)
        if (str[i] == c)
            count++;
}

void foo(char c, int& count) {
    int len;
    char str[MAX];
    cin >> str;
    len = strlen(str);
    newfun(count, len, str, c);
}

void newfun(int& count, int len, char* str, char c) {
    int i;
    count = 0;
    for (i = 0; i <= len; i++)
        if (str[i] == c)
            count++;
}

© 2012 Václav Rajlich            Software Engineering: The Current Practice   Ch. 1
Extract function process

• Select a block of code for extraction
• Is the block syntactically complete?
• Create new function
• Extract the selected block as a function body
• Replace the code block with the function call
Variables during extract function

• Local variable
  – value assigned inside, used only inside

• Parameter passed by value
  – value assigned outside

• Parameter passed by reference
  – value assigned and used outside, changed inside

• Global variable
Extract a base class

• In code development, derived classes always come before the base classes
  – developers may miss some base classes
  – refactoring will correct these omissions
• Extract a base class prepares software for incorporation of new functionality through polymorphism
  – applicable when old and new functionality have a large overlap
Example

class Matrix{
protected:
    int elements [100][100], columns, rows;
public:
    Matrix();
    inverse();
    matrix multiply (Matrix&);
    int get (int, int);
    void put(int, int, int);
};                         // dense matrix
Extract class AbstractMatrix

- Change request: Add sparse matrix to the code.
  - sparse matrix uses the same algorithm for “multiply” and “inverse”.
  - only access to the elements are different (functions “get” and “put”)
- Extract abstract class AbstractMatrix
  - DenseMatrix and SparseMatrix will be derived from it
  - this will make the change (much) easier
  - it will allow to incorporate SparseMatrix through polymorphism
Step 1 of refactoring

• Rename class
  
  – class Matrix → class DenseMatrix

• There will be several classes dealing with matrix
  
  – name needs to be more specific
The next steps of refactoring and incorporation

2. Extracting base class

DenseMatrix

AbstractMatrix

3. Incorporating SparseMatrix

DenseMatrix

SparseMatrix

AbstractMatrix
Steps of refactoring

• Create a new class `AbstractMatrix`
• Make `DenseMatrix` derived from `AbstractMatrix`
• Replace all references to the elements by `get` and `put`
• Move variables `columns` and `rows` to `AbstractMatrix`
• Move functions `inverse` and `multiply` to `AbstractMatrix`
• Add virtual functions `get` and `set` into `AbstractMatrix`
Results

• After refactoring, it is easy to incorporate \textit{SparseMatrix}

• Refactoring preserves the behaviour
Component class extraction

• Motivation: Incorporation by replacement
  – primitive implementation of the class is replaced by a full functionality
• Concept sometimes does not have class of its own
  – must be extracted from another class
  – prefactoring for incorporation by replacement
Example: Price in PoS

Store
- balance : double
- inventory : Inventory
+ getBalance() : double
+ processSale() : double
+ resetStore() : void
+ Main() : void

Inventory
- inventory : Item

Item
- upc : long
- name : string
- inventory : int
- price : double
- tax : double
+ getPrice() : double
+ setPrice(double)
+ calcSubTotal() : double
+ calcTotal() : double
Refactoring
public double calcSubTotal(int numberToSell) {
    if (numberToSell < 1) return 0.0;
    else
        // return numberToSell * price;
    return
        numberToSell*price.getPrice();
}
Incorporation after prefactoring
Move function from composite to component

class A {  // A is composite
    public:
        B* b;    // B is component
        foo();
    
};
Class B {
}

Add a new parameter of the composite type to the function
Move the misplaced function from the composite into the component
In the function body, access all composite members through the new parameter
Move function - example

class A {
public:
    B* b;
    int a_data;
    void a_fun();
    void foo();
};

void A::a_fun() {
    foo();
}

void A::foo() {
// access members of A
this->...
// access members of B
b->...
}

class B {
};

class A {
public:
    B* b;
    int a_data;
    void a_fun();
};

void A::a_fun() {
    b->foo(this);
}

void B::foo(A* d) {
    // access members of A
d->...
    // access members of B
    this->...
}

class B {
public:
    void foo(A*);
};