Refactoring Software Projects Using Object Oriented Concepts

Ankit Desai[1], Jaimin Chavda[2,] Amit Ganatra[3], Amit Thakkar[4], Yogesh Kosta[5]
Charotar Institute of Technology, Charotar University of Science and Technology – Changa
[1]*ankitdesai.it@ecchanga.ac.in*
[2]*jaiminchavda.it@ecchanga.ac.in*
[3]*amitganatra.ce@ecchanga.ac.in*
[4]*amitthakkar.it@ecchanga.ac.in*
[5]*ypkosta.rec@ecchanga.ac.in*

Abstract

Successful software systems must be prepared to evolve or they will die. Although object-oriented software systems are built to last, over time they degrade as much as any legacy software system. As a consequence, one may identify various reengineering patterns that capture best practice in reverse- and re-engineering object-oriented legacy systems. Software re-engineering is concerned with re-implementing older systems to improve them or make them more maintainable, while Refactoring is re-engineering with-in an Object-Oriented context. In this paper, given an object-oriented program the opportunities present where refactoring can be applied are examined by using JDK 1.5 (java compiler). The opportunities are class misuse, violation of the principle of encapsulation, lack of use of inheritance concept, misuse of inheritance, misplaced polymorphism. Apart from these refactoring opportunities this paper also focus on one Java Runtime Error and its uncommon cause and solution of it.

Keywords*: Refactoring, JDK 1.5, class, encapsulation, inheritance, polymorphism, Runtime Error.*

1. Background

Software re-engineering is the transformation from one representation from to another at the relative abstraction level, while preserving the systems' external behavior. Reengineering a software system has two key advantages over more radical approaches to system evolution:

i. Reduced risk: There is a high risk in redeveloping software, which is presently an essential backbone of the organization. Errors may be made in system specification; development problems; financial risk may be high; etc.

ii. Reduced cost: The cost of re-engineering is significantly less than the cost of developing new software. Refactoring is reengineering with in the object oriented context. Software refactoring can be defined as "the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure" [3].

2. Details

"Improvements to its internal structure" means to improve the quality of software system through proper modeling. Example of such improvements are "making the code easier to understand and cheaper to modify". There are many opportunities in an object-oriented program where refactoring can be applied. They are:

A. class misuse: Prior to the invention of the Object-Oriented Programming (OOP), many projects were nearing (or exceeding) the point where the structured approach no longer worked. Object oriented methods were created to help programmers break through these barriers [1]. In an object-oriented language one defines the data and the routines that are permitted to act on data. Thus a data defines precisely what sort of operations can be applied on that data. The most fundamental mistake done in developing a program in OOP context is not designing a class properly. Even if one class is not designed properly the entire program is spoiled. One bad class design has a cumulative effect on all other classes in the entire software. So the first step in developing good software is to design the basic construct, i.e. the class, correctly. The class has to be designed taking into consideration of UML (Unified Modeling Language) concepts. Moreover its members, i.e. data and behaviors to be defined adequately, the interfaces properly laid down and the

relationships between different classes correctly defined.

The question arises is what happens if a single class in not defined correctly. Consider example 1 of a program relating to a company. The class employee contains the salary and office details, the class EmpAddress contain the address details and the class EmpName contains the name details. In order to distinguish objects of the same class the attribute employee_id has been used as a differentiator in all the classes.

The basic mistake in the above design of classes is that all the information pertains to a single employee. Hence all the information should be stored in a single class. If an employee leaves the organization and his records have to be destroyed, in the above design three different objects pertaining to three different classes have to be destroyed instead of one single class. Also, if information about a particular employee has to be obtained, then one has to access three different objects instead of one single object. Class misuse instances are wonderful opportunities for refactoring. Inheriting all the classes into a single class often does not solve all the problems as in the example 1 shown since some of the variables might repeated in more than one class and this leads to compile time errors during implementation. The variable employee_id is repeated in each of the three classes, so inheriting it leads to three sets of employee_id. Also, if the function for printing the class values is defined three times and one has to print the values in the inherited class then the three function have to be called one after the other, i.e. print(), printk(), printl(). This leads to poor functionality. Hence a new function has to be defined to print all the inherited class attributes. This leads to repetition of coding. The only option in to redesign the classes and make a single class with all the functionality in it, instead of many classes with the functionality spread out.

Example 1:

```
class Employee{
private:
    int employee_id;
    String designation, dept_name;
    float da, basic, gross;
public:
    print();
…….
}
class EmpAddress{
private:
   int employee_id;
   String  apartment_no, flat_name, street_name;

public:
    printk();
…….
}

class EmpName
{
private:
    int employee_id;
    String first_name, middle_name, last_name;
public:
    printl();
…….
}
```

B. Violation of the principal of encapsulation: Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interface and misuse. When code and data are linked together in this fashion an object is created. In other words, an object is the device that supports encapsulation. The violation of encapsulation arises mainly from class misuse.

When classes are not designed correctly, reflection has to be used. Reflection is a feature in the Java programming language. It allows an executing Java program to examine or "introspect" upon itself, and manipulate internal properties of the program. For example, it is possible for a Java class to obtain the names of all its members and display them. As shown in the previous example, instead of defining a single class, when there are many classes and one has to access private members of these different defined classes (which otherwise would have been in a single class), then one has to use reflection concept.

Once a class has been designed, executed and tested it becomes a black box where one knows the inputs and outputs. The use of reflection [5], leads to ineffective design as usage of such reflection classes should be avoided and if present, such classes should be removed by refactoring. Manipulation of the existing classes by adding reflection functionality often results in unwanted side effects, which are not known initially and are discovered at a much later stage after spending considerable time and energy. Thus the idea of encapsulation is lost such programs are constant source of bugs and the more one tries to fix it the more the errors arise. These problems pose a great scope for refactoring.

In order to refactoring programs containing such problems new classes have to be designed and the reflection classes have to be replaced by some other traditional classes. Moreover the developers of the reflection classes say “This is a relatively advanced

feature and should be used only by developers who have a strong grasp of the fundamentals of the language. With that caveat in mind, reflection is a powerful technique and can enable applications to perform operations which would otherwise be impossible."

C. Lack of use of Inheritance concept: In JAVA, inheritance is supported by allowing one class to incorporate another class into its declaration the process involves first defining a base class, which defines those qualities common to all objects to be derived from the base [2]. The base class represents the most general description. The class derived form the base is referred to as derived class. A derived class includes all features of the generic base class and adds qualities specific to the derived class.

Assuming a class X defined as follows:

Example 2:
```
class X{
private:
     int i, j;
public:
    void set();
    void show();
}
```

Later on another class Y is defined which contains the variable k and method showk() in addition to the methods and variables defined in X. Now instead of defining it right from scratch, the method followed is to build up the derived class on the base class. So class Y is defined as follows:

Example 3:
```
class  Y extends X{
private:
      int k;
public:
      void showk();
}
```
Example 4:
```
class Y{
private:
       int i,j,k;
public:
       void set();
       void show();
       void showk();
}
```

The advantage of using inheritance is that once that base class is tested and debugged correctly then only the elements defined in the derived class have to be tested and debugged. So the testing time and size of the source code becomes very less. This concept can be extended to the concepts of class libraries wherein class are defined and tested correctly from which future classes can be created with far lesser effort.

However, inheritance may not be properly used. Programmers tend to define their classes right form the scratch. So the same code gets duplicated in more than one class. If any change has to be made to any one method, then the changes have to make in all the classes which contains the code, thus duplicating effort. If inheritance has been implemented, then the changes will have to be made in only one of the classes in which it is defined. As the concept of inheritance can be extended to many generations, this code replication can be avoided. This provides a wonderful opportunity for refactoring. Wherever the inheritance is implemented and code of a particular member function should be implemented in only one class. This makes the software more maintainable and will solve problems in future.

D. Misuse of Inheritance: Inheritance is a powerful concept (if use correctly) but is often misused. Let a class be defined initially as:

Example 5:
```
class A{
private:
      int i, j, k;
public:
      void show();
void setall();
}
```

Now there is another class B that has to be defined as follows:

Example 6:
```
class B{
private:
      int j, k, l;
public:
      void show();
      void setall();
      void showk();
}
```

Programmer tend to use the concept of inheritance in the example shown above although the variable 'i' is not defined in class B. if class B is inherited from class A then there will be an extra variable 'i' which is not necessary for class B. such dangling variables are dangerous. This poses a problem for future maintenance of the software. This extra variable will not be present in the specification and has to be initialized correctly to some initial value; else it might lead to bizarre error [4]. Programmers often in the haste of implementing inheritance do such mistakes.

A base class should be inherited by the derived class when all the contents of the base class are used fully and at the same time distinct in the derived class. Often inheritance is implemented for code reuse rather than polymorphism. For Example:

Example 7:
```
class C{
private:
      int  i, j;
public:
      void set();
      void showall();
}
class D{
private:
      int i, j, k;
public:
      void set();
      void showall();
      void setk();
}
```

Now class C should be inherited by class D if and only if when the functions set() and showall() along with the variables are implemented in totality by class D. If set() is defined in class C as

Example 8:
```
void set(){
            i=10;
            j=15;
}
```

If set() and setk() are defined in class D as

Example 9:
```
void set(){
            i=10;
            j=15;
}
void setk(){
            k=20;
}
```

In such a case in order to implement inheritance the function set() in class D should not be broken in two parts as two different functions in order to implement inheritance. Inheritance should not be implemented to make the program easier to develop, implement and understand. Inheritance is more to achieve Polymorphism rather to code reuse.

E. Misplaced Polymorphism: Polymorphism is the attribute that allows one interface to control access to a general class of actions. The specific action selected is determined by the exact nature of the situation. Function overloading is the process of using the same name for two or more functions. The secret to overloading is that each redefinition of the function must use either different types of parameters or a different number of parameters. For example, there may be a program that defines three different types of stacks. One stack is used for integer values, one for character values and one for floating point values. Because of the polymorphism, only one set of names push() and pop() can be defined which are used for all three specific versions of these functions are created, one for each type of stack, but names of the functions will be the same. The compiler will automatically select the right function based upon the data being used. The individual version of these functions defines the specific implementations for each type of data.

However, often polymorphism is not implemented properly as classes are constructed ad-hoc and not step by step. Consider an example the one explained in the previous paragraph. If the class is developed initially for integer values and much later the class is modified to include for character values then for the same push() operation the function for character values is given a different name. These results in similar code (not the same code) in different names provide same interface for different types. Such cases provide a wonderful chance for refactoring. Implementation of polymorphism, while refactoring is a slow task. The functions have to scanned and the actions performed by the functions to be defined. Functions performing the same action on different data types should be given the same name. Polymorphism helps reduce complexity allowing the same interface to be used to access a general class of actions.

3. Compilation Problem

Many times when using java compiler many programmers have came across the following error.

i.e. “java.lang.UnsupportedClassVersionError”.

There are many possible causes of this error i.e. Java file is being compiled by latest version of the JDK 1.5 and being run by older version of the JDK1.4 or older, Environment variable path is not set, class file name is not properly given, so on and so forth.

For all above causes the solution is so simple and very much familiar.

That uncommon cause of this error is: When we have JRE (Java Runtime Environment) and ORACLE installed on the same machine, this error may create problem while running the java program. Even though the class path is set the JDK prompts error because when you will analyze the path value, the path of the ORACLE machine comes before the path of jdk/bin; and while you are compiling the program, value of the path of ORACLE machine (which is able to compile the java program) comes

before the path of javac, it gets compiled by ORACLE machine and creates a class-file which can run neither by ORACLE nor by java. So, while running that class-file it gives error of: "UnsupportedClassVersionError".

Table 1. Effect of various opportunities of Refactoring on Complexity Measures

	Complexity Measures Before Applying Refactoring			Refactoring Cost
Refactoring Opportunities	**Space**	**Time**	**Design**	
Class Misuse	Very High	High	High	Low
Violation of the principle of encapsulation	Not Applicable	Very High	Very High	Very High
Lack of usage of inheritance concept	Very High	Medium	Very High	High
Misuse of inheritance	Not Applicable	High	High	Medium
Misplaced Polymorphism	High	Very Low	High	Very Low

4. Conclusion

The issues discussed above if used while refactoring an object oriented program leads to much efficient software. The objective of refactoring is to improve the system structure and make it easier to understand. Refactoring increases the time span of the usab8lity of a program. Refactoring, if not applied properly might lead to further deterioration of the software. There are, however, practical limits to the extent that a system can be improved by refactoring. Major architectural changes or radical reorganizing of the system data management cannot be carried out automatically, and would involve high additional costs. The costs of refactoring obviously depend on the extent of the work that is carried out.

5. References

[1] U. Zdun: "Using Split Objects for Maintenance and Reengineering Tasks", in: 8th European Conference on Software Maintenance and Reengineering (CSMR), Tampere, Finland, March, 2004.

[2] Netmation, object oriented development, web article: http://netmation.com/exp0028.htm

[3] Serge Demeyer, Stéphane Ducasse, Kim Mens, Adrian Trifu, Rajesh Vasa and Filip Van Rysselberghe: "Object Oriented Reengineering", ISBN 978-3-540-22405-1, Pages72-85, June, 2004.

[4] Oscar Nierstrasz, Stéphane Ducasse and Serge Demeyer: "Object Oriented Reengineering Patterns an Overview" ", ISBN 978-3-540-29138-1, Pages1-9, October, 2005.

[5] Sun Microsystem, web article on Reflection, web link1: http://java.sun.com/docs/books/tutorial/reflect/index.html
link2:
http://java.sun.com/developer/technicalArticles/ALT/Reflection/