Reverse Engineering and Refactoring Related Concept in Software Engineering

Ajit kumar and Navin kumar.
Research scholars.
B. R. Ambedkar Bihar University.
Muzaffarpur, Bihar (India).
Guide:-Dr.D.L.Das.

Abstract:

Reverse Engineering is the process of analyzing a subject system to identify the system's components and their interrelationships and create representations of the system in another form or at a higher level of abstraction. Refactoring is about improving the design of existing code. It is the process of changing a software system in such a way that it does not alter the external behaviour of the code, yet improves its internal structure. With refactoring you can even take a bad design and rework it into a good one. By comparing the two definitions we would find that refactoring and reverse engineering is closely related subject. To develop a good refactoring tool, it is wise idea to introducing reverse engineering concept and method. Refactoring is a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour.

ntroduction:

The reverse engineering process begins by extracted detailed design information, and from that extracted a high-level design abstraction. Detailed (low-level) design information is extracted from the source code and existing design documents. This
Information includes structure charts, data descriptions and PDL to describe processing details. A similar approach, but automated, is described elsewhere to recover Jackson and Warnier/Orr documents from code. The high-level design representation
Is extracted from the recovered detailed design and expressed using data-flow and control-flow diagrams. Throughout this paper the term 'recovered design' will be used to denote the extracted design.

Basis for Reverse Engineering

The procedure steps are discussed below.Figure1 summarizes the procedure.

1. Collect information. Collect all possible information about the program. Sources of information include source code, design documents and documentation for system calls and external rouwes. Personnel experienced with the software should also be identified.

2. Examine information. Review the collected information. This step allows the person doing the recovery to become familiar with the system and its components. A plan for dissembler the program and recording the recovered information can be formulated during this stage.

3. Extract the structure. Identify the structure of the program and use this to create a set of structure charts. Each node in the structure chart corresponds to a row called in the program. Thus the chart records the calling hierarchy of the program. For each edge in the chart, the data passed to a node and returned by that node must be recorded.

4. Record functionality. For each node in the structure chart, record the processing done in the program rouwe corresponding to that node. A PDL can be used to express the functionality of program rouwes. For system and library rows the functionality can be described in English or in a more formal notation.

5. Record data-flow. The recovered program structure and PDL can be analysed to identify data transformations in the software. These transformation steps show the data processing done in the program. This information is used to develop a set of hierarchical data flow diagrams that model the software.

6. Record control-flow. Identify the high-level control structure of the program and record it using control-flow diagrams. This refers to high-level control that affects the overall operation of the software, not to low-level processing control.

7. Review recovered design. Review the recovered design for consistency with available information and correctness. Identify any missing items of information and attempt to locate them. Review the design to verify that it correctly represents the program.

8. Generate documentation. The final step is to generate design documentation. Information explaining the purpose of the program, program-overview, history , etc, will need to be recorded. This information will most probably not be contained in the source code and must be recovered from other sources.

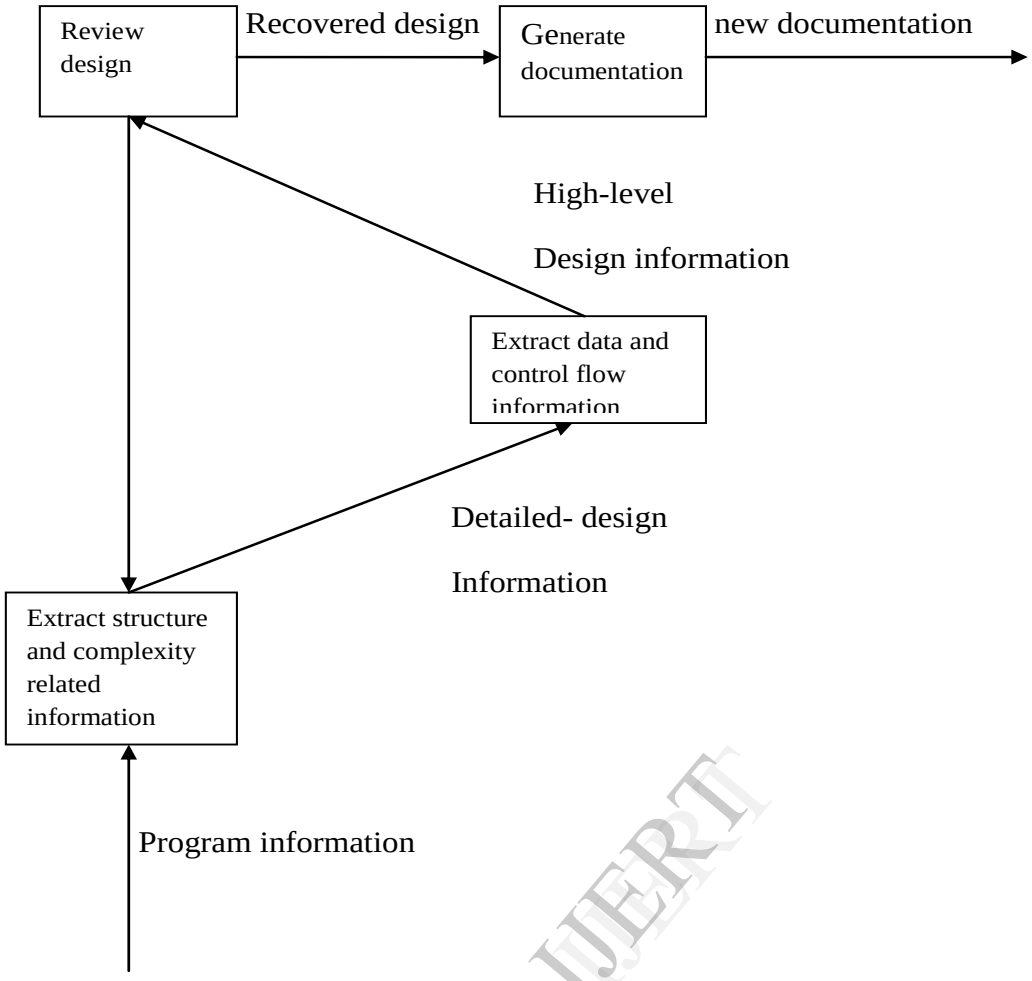


Figure 1: Reverse engineering procedure

Basis for Refactoring

"Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behaviour. Its heart is a series of small behaviour preserving transformations. Each transformation (called a 'refactoring') does little, but a sequence of transformations can produce a significant restructuring. Since each refactoring is small, it's less likely to go wrong. The system is also kept fully working after each small refactoring, reducing the chances that a system can get seriously broken during the restructuring." [Marw Fowler]

Refactoring is used to improve code quality, reliability, and maintainability throughout the software lifecycle. Code design and code quality are enhanced with refactoring. Refactoring also increases developer productivity and increases code reuse. For example, if two methods use a similar piece of code, the common code can be refactored into another method that the two parent methods can then call.

Software refactoring = "Restructuring existing code by altering its internal structure without changing its external behaviour"

Steps in Refactoring

The refactoring involves

• Development = (Adding features, Refactoring)

• Refactoring = (Testing, Small Steps)

• Small Steps = one of the refactoring types "Put things together when changes are together"

• Extract Methods

• Move Methods

• Rename Methods

• Replace Temp with Query

• Replace conditionals with polymorphism

• Replace Type code with State/Strategy

• Self Encapsulate Field

Applications of Refactoring

• We use three examples to explain some basic refactoring

(a)Extract method:

• signalled by comments

• Single-entry, single-exit

• increase the level of indirection

• reduce the length of a method

• increase the chance of reuse

(b) Move method:

• Place method together with the object; put things together when changes are together

(c)Replace conditions with polymorphism

• Switches are “hard code”, polymorphism is better for Applications.

We use three examples to explain some basic Refactoring

(a) Extract method:

• signalled by comments

• Single-entry, single-exit

• increase the level of indirection

• reduce the length of a method

• increase the chance of reuse

(b) Move method:

• Place method together with the object; put things together when changes are together

(c) Replace conditions with polymorphism

• Switches are “hard code”, polymorphism is better for extensibility in object oriented Applications

Examples on Refactoring using three main methods

Example (a) – Extract method

```
Void f1 () {
...
Compute score1
score1 = a * b + c;
```

```
score1 -= discount;
}
Void f() {
...
computeScore1 ();
}
Void computeScore1 () {
score1 = a * b + c;
score1 -= discount;
}
```

Example (b) – Move method

```
class X {
...
}
class R {
private bool isAccurate(X a1) {
switch(a1.b1) {
case Y: return true;
case Z: return true;
case W: return false;
}
}
}
class  X {
bool  isAccurate() {
```

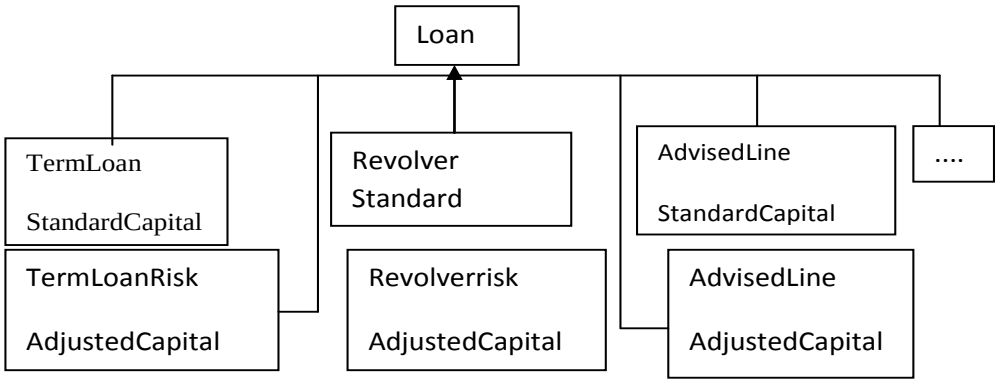
```
switch(b1) {
case Y: return true;
case Z: return true;
case W: return false;
} } }
class R {
private bool isAccurate(X a1) {
return a1.isAccurate();
}
```

Example (c) - Replace conditions with polymorphism

The example in the code sketch presented in the introduction to this refactoring deal with calculating capital for three different kinds of bank loans: a term loan, a revolver, and an advised line. It contains a fair amount of conditional logic used in performing the capital calculation, though it's less complicated and contains less conditional logic than was present in the original code, which handled capital calculations for seven distinct loan types.

In this example, we'll see how Loan's method for calculating capital can be strategized (i.e., delegated to a Strategy object). As you study the example, you may wonder why Loan wasn't simply subclasses to support the different styles of capital calculations. That would not have been a good design choice because the application that used loan needed to accomplish the following.

Calculate capital for loans in a variety of ways. Had there been one Loan subclass for each type of capital calculation, the Loan hierarchy would have been overburdened with subclasses, as shown in the diagram on the following page.



- Change a loan's capital calculation at runtime, without changing the class type of the Loan instance. This is easier to do when it involves exchanging a Loan object's Strategy instance for another Strategy instance, rather than changing the whole Loan object from one subclass of Loan into another.

Now let's look at some code. The Loan class, which plays the role of the context, contains a calculation method called capital ():

```
public class Loan...
  public double capital () {
    if (expiry == null && maturity! = null)
      return commitment * duration () * riskFactor ();
    if (expiry != null && maturity == null) {
      if (getUnusedPercentage ()! = 1.0)
        return commitment * getUnusedPercentage () * duration () * riskFactor ();
      else
        return (outstandingRiskAmount () * duration () * riskFactor())
          + (unusedRiskAmount () * duration ()* unusedRiskFactor ());
    }
    return 0.0;
  }
```

References:-

[1.] http://openseminar.org

[2.] http://www.cs.toronto.edu

[3.] http://www.informit.com