Building Tomorrow's Enterprise

Infosys Global Agile Developer Certification

Refactoring

Study Material

www.infosys.com

Revision History

Version	Date	Remarks
1.0	Dec’2014	Initial Version

Table of Contents

List of Figures and Tables

1 Introduction

Refactoring is the practice of continuously improving the design of existing code, without changing the fundamental behavior. In Agile, teams maintain and enhance their code in an incremental basis from Sprint to Sprint. In case the code is not refactored in Agile project, it will lead to poor code quality i.e. unhealthy dependencies between classes or packages, bad allocation of class responsibilities, too many responsibilities per method or class, duplicate code, and many other varieties of confusion and clutter. Refactoring helps to remove these chaos and simplifies the unclear and complex code. Following are the definitions quoted by various experts in Agile on Refactoring concepts:

- *A refactoring is a "behavior-preserving transformation"* - *Joshua Kerievsky*
- *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 behavior" - Martin Fowler*

It is best to refactor continuously, rather than in phases. Refactoring continuously prevents the code from getting complicated, and helps to keep the code clean and easy to maintain. In Agile development, there can be short and separate sprint to accommodate refactoring. This module provides an overview of the technical/engineering practice of Refactoring which would be implemented in Agile projects.

2 Challenges

Though refactoring brings lot of benefits to the code quality of the software, there are multiple challenges which dissuades developer in Agile projects to continuously refactor the code. Following are few challenges which are mostly seen in Agile projects

- Time Constraint: Time is the biggest challenge for doing refactoring in Agile projects as the Sprints are time boxed with a defined set of deliverables.
- Reluctance: If the code is working fine without any refactoring done, there will be inclination towards not revisiting the code. This is primarily because of the mind-set that there is no error and hence no need to do additional activities i.e. refactoring.
- Integration with branches: To integrate the code across different branches post refactoring is considered as a challenge
- Fear factor: Developer often fears that refactoring will introduce bugs and break the existing functionality which is working fine
- Re-Testing: In case automated test suites are not available , developer is discouraged to do refactoring with the additional effort of manual testing to check the functionality
- Backward Compatibility: Maintaining backward compatibility often discourages developer from initiating refactoring effort

3 Concepts

3.1 Motivation for developers to do refactoring

The following points for motivation are more common among the developers while they do refactoring

- Makes it easier to add new code
- Improves the design of existing code
- Helps to gain a better understanding of code
- Makes coding less annoying

3.2 Key benefits from refactoring

Refactoring in small steps helps prevent the introduction of defects. Following points can be further perceived from the benefits of implementing Refactoring.

- Improves software extendibility
- Reduce code maintenance cost
- Provides standardized code
- Architecture improvement without impacting software behavior
- Provides more readable and modular code
- Refactored modular component – increase potential reusability

Design Guidelines for Agile Projects:

In traditional software development project, requirements and plans are set before development begins, which enables the team to know the road ahead with a reasonable confidence that requirements or design will not change drastically in the middle whereas Agile method is to enable and embrace change. On an Agile project, requirements can change at any point in the project cycle. Therefore it is imperative for Agile teams to have a code in a position where they can conveniently accept a new requirement or change.

One of the important Agile manifesto principle stresses on good design in Agile projects “Continuous attention to technical excellence and good design enhances agility”.

Following guidelines can be used for design in Agile projects:

1) Keep It Simple
2) Apply SOLID Principles
3) Continuous Code Refactoring
4) Test Driven Development (TDD)
5) Implement Design Patterns

Note: Please refer to Appendix for ‘Design principles used across key Agile methods’

3.3 Precautions from doing refactoring

- Too much of pattern focus takes away the focus from writing small, simple and understandable code
- Look at patterns from refactoring perspective and not just as reusable elements
- Do not under or over engineer the code

3.4 How to smell the bad code

'Code Smell' is the phrase coined by Kent Back and Martin Fowler. Some of the important "smells in code" are described for reference in the table below:

#	Smell	Description
1	Duplicated code	Identical or very similar code exists in more than one location
2	Long Method	A method/function/procedure that has grown too large
3	Long Class	A class that has grown too large
4	Too Many Parameters	A long list of parameters is hard to read, and makes calling and testing the function complicated
5	Feature envy	A class that uses methods of another class excessively
6	Lazy Class	A class that does too little
7	Contrived complexity	Forced usage of overly complicated design patterns where simpler design would suffice.
8	Excessively long identifiers	The use of naming conventions provide non ambiguity that should be implicit in the software architecture
9	Excessively short identifiers	The name of a variable should reflect its function unless the function is obvious.
10	Cyclomatic complexity	Too many branches or loops. This may indicate a function which needs to be broken up into smaller functions, or it has potential for simplification
11	Complex to debug	Code has become complex enough to debug
12	Complex to tune the performance	Performance tuning becoming complex
13	Scalability issue	Taking huge time in making basic change or adding new functionality

Table 1: Code Smells

Source: http://en.wikipedia.org/wiki/Code_smells

3.5 Guidelines for Refactoring

1. Make sure the code is working before you start.
2. Ensure that automated test suite is available and provides good coverage.
3. Run the tests frequently before, during and after each refactoring.
4. Use a version control tool and save a checkpoint before each refactoring. Not only does this mean recover can be done quickly from disasters, it also means refactoring can be tried out and then back it out if unsatisfied on the refactored code.
5. Break each refactoring down into smaller units.
6. Finally, use a refactoring tool if there is one available for your environment

4 Implementation of Refactoring Techniques

There are multiple refactoring techniques available which will make the existing code better in terms of performance and maintainability. The basic purpose of refactoring in Agile or any other methodology is *"Leave a module in a better state than you found it"*

The Refactoring techniques are categorized by Martin Fowler as follows:

- Composing methods
- Moving features between Objects
- Organizing data Simplifying Conditional Expression
- Making method calls simpler

Every category has certain refactoring techniques. Few techniques are explained below.

4.1 Composing Methods

It deals with proper packaging of method or function code. Large methods/function having very long complex logic implementation can reduce the maintainability and readability of the code. Applying refactoring techniques like extract methods will help making the code more maintainable and readable. Few of the refactoring techniques in composing method/function are detailed below:

- Extract Method
- Inline Method
- Replace method with method object

4.1.1 Extract Method

This is one of the popularly used Refactoring techniques. The implementation behind this technique is very simple. It consists of breaking up long methods by shifting complex chunks of code into new methods which have very descriptive identifiers. This method is used when:

- The code consists of long methods covering multiple logical flows that can be broken up in smaller methods
- If same code is duplicated in more than one flow, move this to single method and call from all other flows

Example written using Java:

Before Refactoring	After Refactoring
void printStudentRecord() {	void printStudentRecord() {
printSchoolName();	printSchoolName();
	printStudentDetails();
//print details	}
System.out.println ("Id: " + _id);	
System.out.println ("name " + _name);	void printStudentDetails(){
}	//print details
	System.out.println ("Id: " + _id);
	System.out.println ("name " + _name);
	}

The advantages of this method are:

- Proper code reorganization by grouping a set of statements into single smaller method
- Reduces code duplication
- Increases the readability of the code

Precaution- If method has too many exit points then care should be taken while applying this refactoring technique.

4.1.2 Inline method

In contrast to Extract Method technique of refactoring, Inline method techniques suggests to replace a method/function call with the body of method/function. It is suggested when source code of the method/function is very small. This method is used when:

- When function call is bottleneck in the performance
- When inline code increases the readability of the code since body of the function is same as function name
- When too many delegations are done in the code

Example written using Java:

Before Refactoring	After Refactoring

int getRating() {
  return (moreThanFiveLateDeliveries()) ? 2 : 1;
}
boolean moreThanFiveLateDeliveries() {
  return _numberOfLateDeliveries > 5;
}
	int getRating() {
	  return (_numberOfLateDeliveries > 5) ? 2 : 1;
	}

The advantages of this method are:

- Unnecessary complexity reduced, reduction in method calling overhead
- Increases the readability of the code

Precaution- One should not use this technique if the method code is used by other classes/methods.

4.1.3 Replace method with method object

The Thumb rule of good code is that, it should be readable and easily manageable. At times we might come across a very long method having complicated logic implementation done inside it with many local variables. Such method cannot be simplified by applying extract method technique because of the usage of too many local variables, because passing around that many variables would be just as messy as the long method itself. In such cases, a class can be created for such a method. This class will have all local variables of that long method as its data members. One of the methods will be the long method. The complicated logic of this method can easily be simplified by applying extract methods technique.

Example written using Java

Before Refactoring	After Refactoring

class Employee{
        private:
        // some declarations
 	
  double CalSalary(float fVCPI, float fCPI, float fPPF)
        {
          double dIncomeTax;
          double dHRA;
          double dLTA;
        // complicated salary calculations
        }
};
	class SalaryCalculator{
	   private:
	          float m_fVCPI;
	          float m_fCPI;
	          float m_fPPF;
	          double m_dIncomeTax;
	          double m_dHRA;
	          double m_dLTA;
	   public:
	
	   SalaryCalculator(float fVCPIpercent, float fCPIpercent, float fPF, double dIncomeTaxPercent, double dHRA, double

	dLTA){
	
	        // intialize private data
	members
	}
	
	    double CalculateSal(){
	                // complicated salary calculations
	     }
	};
	
	class Employee{
	    private:
	                // some declarations
	    public:
	            double CalSalary()
	              {
	                  SalaryCalculator obj(14, 13, 200.43,
	dITPercent, dHRA,dLTA);
	              return obj.CalculateSal();
	               }
	};

4.2 Moving features within objects

One of the most important and critical part of object design is deciding on where to assign the responsibilities. There are many options. Refactoring techniques can help us deciding the same. Following are few of the refactoring techniques which can be used to decide the proper responsibility allocation:

- Move method
- Move field
- Hide delegate

4.2.1 Move Method

Consider a scenario for performing a functionality. A method is using members (data or functions) of another class i.e. Class-2 multiple times, as compared to the one in which it is written i.e. Class-1. Then such a code can be refactored using Move Method technique. According to this technique in such cases, that method can be moved to Class-2. Class-1 can call this method in order to complete the needful functionality and the method can be removed from Class-1 altogether.

Move method technique can also be applied when a method is used maximum number of times by one of the class in which it is declared and defined.

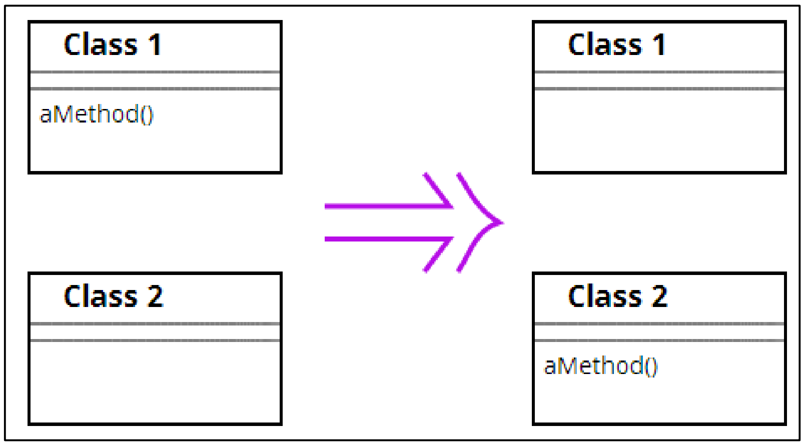

Figure 1: Move Method Refactoring Technique Explained

4.2.2 Move field

If a member of one class is used maximum number of times by access methods from some other class, then such field can be moved to another class itself. This needs necessary modifications in the earlier class which was having that data member in it. Move method techniques will also be applied when new classes are identified during the development phase.

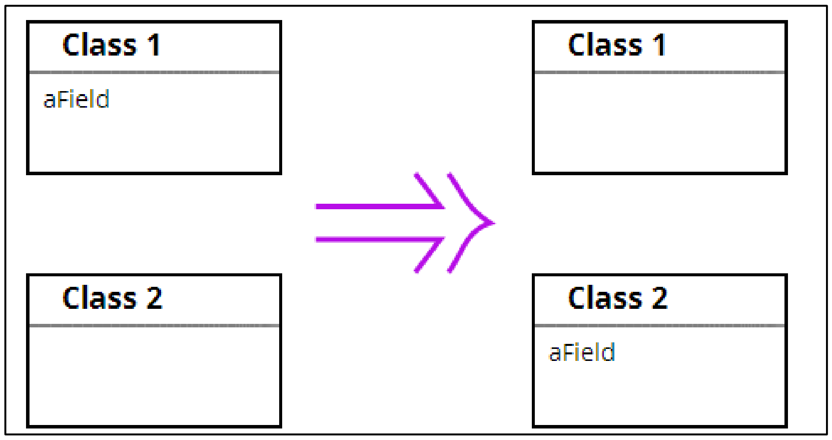

Figure 2: Move Field Refactoring Technique Explained

4.2.3 Hide delegate

One of the important features of object oriented design is if changes are made in one of the class, then no other classes or very few classes should have an impact on its code because of this change. If one class is sending its requests to another class then such a class can be referred as client class. The class which serves the sent requests can be referred as server class.

Consider a function has to get an object of class B by calling one of the methods of class A. This function has to also call a method of class B to get the required data. This means that function has to interact with class A first and then with class B. In this case, the internal structure of class B is partially revealed to that function. The same can be avoided if class A has a method which itself will get the necessary things from class B and send the same to the client function. This delegation part can be made hidden from the client function. Below diagram shows pictorial presentation where ‘Department’ Class is hidden from ‘Client’, as the call to ‘Department’ Class is moved to ‘Person’ class method.

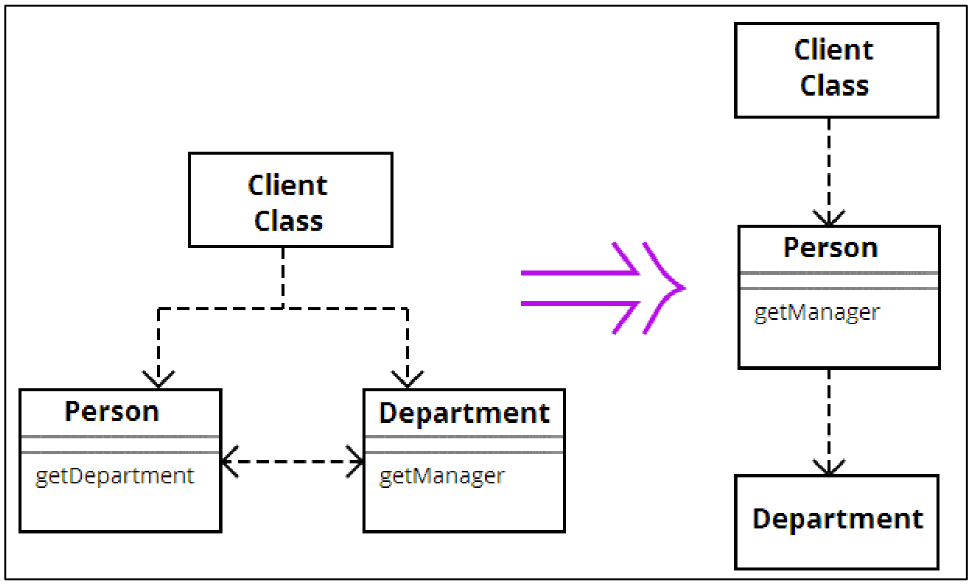

Figure 3: Hide Delegate Refactoring Technique Explained

4.3 Organizing Data

Following section illustrates few techniques which would make data organization better. This also makes working with data easier.

4.3.1 Replace array with Object

One of the most common data structure used for data organization is array. Array is a collection of homogeneous data in contiguous locations of memory having the same name. Although homogenous, the same array can contain data with a different meaning causing confusion while accessing this data structure. In such cases, it is good to replace the array with an object as demonstrated in the example below:

Before Refactoring	After Refactoring

String[] record = new String[3];
record [0] = "Ball";
record [1] = "Red";
record [2] = “12 inch”;
	ObjectDetails record = new ObjectDetails();
	record.setObjectType("Ball");
	record.setColour(“Red”);
	record.setSize(“12 inch”);

4.3.2 Replace magic numbers with symbolic constraints

Consider the code has a constant number assigned with a particular meaning i.e. Tax percentage, VAT (Value Added Tax) percentage, height, weight or any other parameter. Defining this magic number with a meaningful variable name improves code readability as shown in the example below.

Before Refactoring	After Refactoring

return (price * 0.21);	double vatPercentage = 0.21;
	return (price * vatPercentage);

4.4 Simplifying conditional expressions

Understanding conditional logic implemented in a program can be one of the challenging tasks. Using refactoring techniques, the conditional logic can be simplified so that the code can be understood easily. This section has few refactoring techniques using which conditional logic can be simplified easily.

4.4.1 Decompose Conditional

The complicated condition present as part of conditional statements is one of the complex parts which might hinder the readability of the program. Such complicated conditions can be extracted in a method and used at the place of condition. This will increase the readability and maintainability.

Before Refactoring	After Refactoring

if (PhysicsMarks >= 65) && (ChemistryMarks >= 65) &&
(BiologyMarks >= 65){
System.out.println("Passed in Science");
}
else {
System.out.println("Failed in Science);
}
	boolean hasClearedScience(int Phy, int Chem, int Bio))
	{
	return((Phy >= 65) && (Chem >= 65) && (Bio >= 65));
	}
	
	if (hasClearedScience (PhysicsMarks, ChemistryMarks,
	BiologyMarks)){
	System.out.println("Passed in Science");
	}
	else {
	System.out.println("Failed in Science);
	}

4.4.2 Replace nested condition with guard clause

In nested conditional expressions if every conditional construct is equally important then the if-else-if ladder may be used. If conditional construct indicates some unusual and rare condition then, perform condition check and return if condition results to true. This type of checking is called as checking with guard clause

Before Refactoring	After Refactoring

double fnSalaryHike(){
    double dHike;
    if(JobBand == 'C')
    {
      dHike = CalForC();
    }
    else
    {
      if(Salary > = 50000)
      {
        dHike = CalForSalary();
      }
      else
      {
        if(LastCRR == 1)
        {
          dHike = CalForCRR();
        }
        else
        {
          dHike = Calc();
        }
      }
    }
    return dHike;
}
	double fnSalaryHike(){
	    if(JobBand == 'C') return CalForC();
	    if(Salary > = 50000) return CalForSalary();
	    if(LastCRR == 1) return CalForCRR();
	    return Calc();
	}

4.5 Making method calls simpler

Objects can interact with each other and outside world through public methods. These methods should be easy to understand then the entire object oriented design will become easy to understand. In this section, we will discuss few refactoring techniques which will make the methods calls easier.

4.5.1 Rename method

This is a simple and one of the most effective refactoring techniques, renaming a property/attribute, method or an object.
Renaming identifiers can reduce the need for code comments and nearly always helps to promote greater clarity. This is a fundamental part of other refactoring techniques to aid understanding of the code.

This technique relies on giving items a descriptive name to ensure the developer knows at a glance exactly what it does.

Before Refactoring	After Refactoring

int GetACode(){
        return m_AreaCode;
}
	int GetAreaCode(){
	        return m_AreaCode;
	}

4.5.2 Separate Query from modifier

Functionality which modifies the data should be separated from the one which does not modify the data. This can be done by splitting a single method into 2 methods.

Before Refactoring	After Refactoring

void saveCustomerAndDisplayAll(Customer oCust){
        // functionality
}
	void saveCustomer (Customer oCust){
	        // functionality
	}
	Void displayCustomers () {
	        // functionality
	}

5 Refactoring in Test Driven Development

Test-Driven Development (TDD) is a software engineering approach that consists of writing failing test case(s) first covering functionality. And then implementing the necessary code to pass the tests and finally refactor the code without changing external behavior.

TDD deals with 2 types of tests i.e.

- Unit tests – checking functionality of a single class isolated from its environment
- Acceptance tests – checking a single functionality

On similar lines, Refactoring can also be categorized in different type.

<u>Refactoring at local class level:</u>
The unit tests for this class are unchanged and the tests are still green. If the Unit tests are designed to check complete scenarios instead of single member call, the refactoring falls into this category.

<u>Refactoring impacting multiple classes:</u>
In this case, Acceptance tests are the only way to check complete functionality after refactoring. This is useful if new way of functionality is provided, and implemented by changing class after class and associated unit tests.

An important aspect to remember for refactoring in TDD is, when code is refactored impacting interface/API, all the callers of this interface/API along with Tests written for the interface/API need to be changed as part of refactoring. In congruence with the definition, refactoring is a "behavior-preserving" change.

6 Database Refactoring

Database refactoring is a functional or a structural change in a database schema in order to improve SQL database design while retaining its semantics without taking away or adding any functionality.

According to Scot W. Ambler and Pramod J. Sadalage, database refactoring can be described in six categories of database refactors

CATEGORY	DESCRIPTION
Structural	A change in the definition of tables, views, and columns. Some of these refactoring techniques include the replace one- to- many relationship with associative table, split table, rename table, and more
Data Quality	This category of refactoring techniques covers changes that improve the quality of the data stored in a database. Refactoring techniques in this category are add lookup table, drop standard type, move data, and more
Referential Integrity	Changes that ensures that any data that is referenced by one table exists, and also that unused data is removed
Architectural	Changes with a goal of improving the overall methodology in which external applications interact with a database
Method	Code changes like adding/removing parameters to a stored procedure, with a goal to improve overall quality
Transformations	Changes in a database schema. For example introducing a new table or a column, and inserting and updating data

Table 2: Categories of Database Refactoring

Source: http://solutioncenter.apexsql.com/sql-database-refactoring-techniques/

The following diagram depicts the process of complete Database Refactoring:

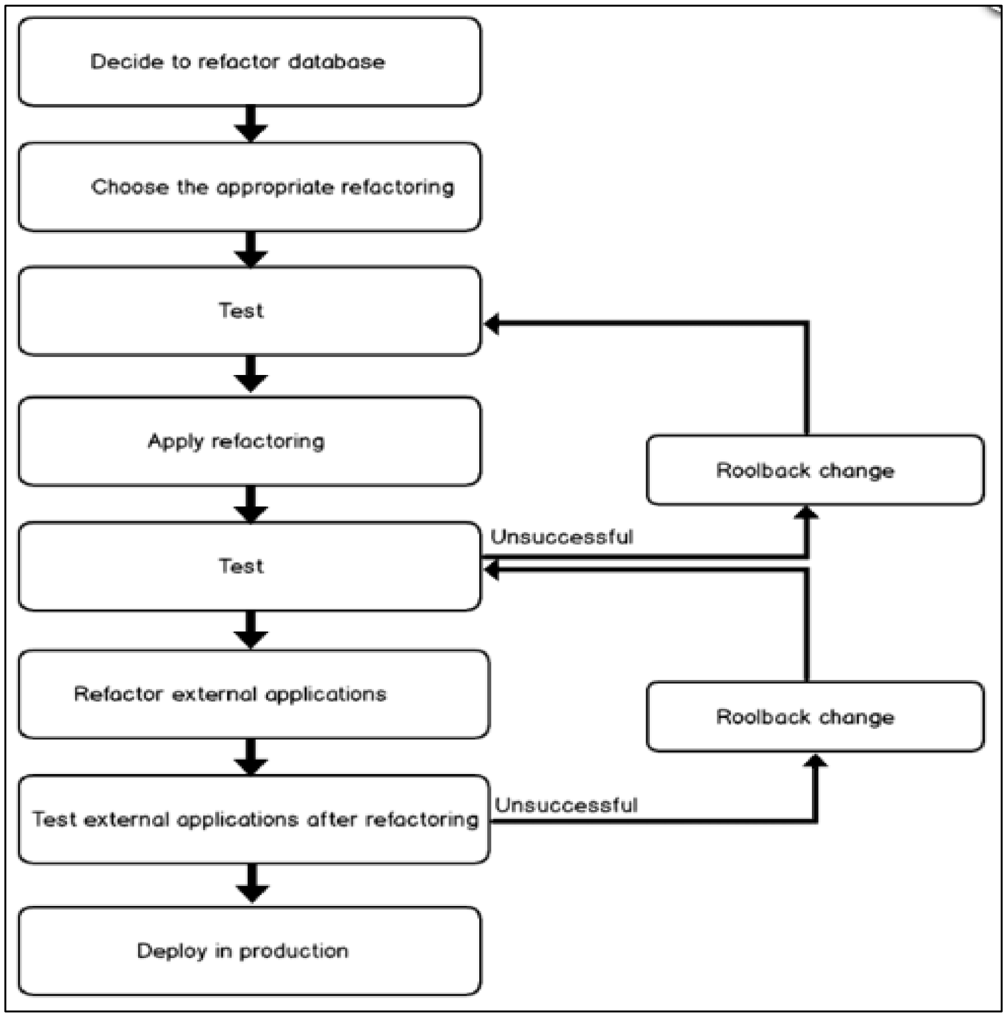
Decide to refactor database
Choose the appropriate refactoring
Test
Apply refactoring
Test
Refactor external applications
Test external applications after refactoring
Deploy in production
Roolback change
Unsuccessful
Roolback change
Unsuccessful

Figure 4: Database Refactoring Process

Source:* *http://solutioncenter.apexsql.com/sql-database-refactoring-techniques/

7 Tools for Refactoring

There are many refactoring tools available, in most cases they are built into, or are extensions of, IDEs (Integrated Development Environments). Below are few examples

#	Language	Tools
1	**C**	• Lint • Visual Studio
2	**C++**	• Lint • Visual Assist • Visual Studio
3	**C#, .NET**	• CodeRush • Resharper • Visual Studio
4	**Java**	• Eclipse • IntelliJ IDEA • Netbeans • Oracle JDeveloper
5	**Oracle**	• Toad • SQL Developer • Few utilities from Oracle like below ○ AWR report ○ ADDM report ○ TKProof ○ Set Auto Tuning ON Above utilities are useful in SQL diagnostic as well as identifying performance bottlenecks.

8 Practice Questions

Question 1

Key benefits of Refactoring in Agile include:

a) Reduce maintenance cost
b) Architecture improvement without impacting software behavior
c) Increase readability and modularity of code
d) All of the above

Question 2

Refactoring is:

a) A practice to change functional behavior of the code
b) A practice of continuously improving the design of existing code
c) A exercise to increase business revenue by meeting customer requirements
d) A exercise to modify application architecture drastically to improve performance

Question 3

What is the correct sequence of steps for Database refactoring? (Concept to be tested: Database Refactoring in Agile)

a) Write Test → Apply DB refactoring → Refactor external application → Run Test
b) Write Test → Refactor external application → Run Test→ Apply DB refactoring
c) Write Test → Run Test→ Apply DB refactoring→ Refactor external application
d) Write Test → Apply DB refactoring → Run Test→ Refactor external application

Question 4

"Replace array with Object" technique belongs to which of the following Refactoring technique?

a) Composing methods
b) Moving features between Objects
c) Organizing data
d) Simplifying Conditional Expression

Question 5

'Cyclometic Complexity' is an indicator to smell the bad code. TRUE or FALSE?

a) TRUE
b) FALSE

9 References

- Joshua Kerievsky, *'Refactoring to Patterns*'

- Martin Fowler, Kent Beck, John Brant, and William OpdykeM, *"Refactoring: Improving the Design of Existing Code"*, Addison-Wesley Pearson Education

- Team Wiki of ENCORE:
  - http://teamwiki/ENCOREWiki/MACollatoralsRefactoring
  - http://teamwiki/ENCOREWiki/MATrainingMaterialsRefactoring

- Websites:
  - http://www.infosys.com/engineering-services/white-papers/Documents/refactoring.pdf
  - http://sparsh/v1/myUnit/QD/CAST/ToolsGroup_Presentation_CAST_Delivery_V3.pdf
  - http://refcardz.dzone.com/refcardz/refactoring-patterns#refcard-download-social-buttons-display
  - http://databaserefactoring.com/
  - http://solutioncenter.apexsql.com/sql-database-refactoring-techniques/
  - http://martinfowler.com/articles/workflowsOfRefactoring/fallback.html
  - http://en.wikipedia.org/wiki/Code_smells
  - http://sourcemaking.com/refactoring/problems-with-refactoring
  - http://research.microsoft.com/pubs/172573/kim-fse-2012.pdf

Infosys® | Building Tomorrow's Enterprise

www.infosys.com