

# Reengineering the Class - An Object Oriented Maintenance Activity

Gokul V. Subramaniam  
NORTEL - Northern Telecom  
P.O. Box 833871,  
Richardson, TX-75082.  
Phone: 972-685-8214  
Email: gokuls@nortel.ca

Eric J. Byrne  
Global Consultants, Inc  
601 Jefferson Rd,  
Parsippany, NJ 07054.  
Phone: 732-695-9180  
Email: ejbyrne@att.com

## Abstract

*When an Incremental Approach is used to develop an object-oriented system, there is a risk that the class design will deteriorate in quality with each increment. This paper presents a technique for detecting classes that may be prone to deteriorate, or if deterioration has occurred assists with reengineering those classes. Experience with applying this technique to an industrial software development project is also discussed.*

**Keywords:** *maintenance, object-oriented systems, object-oriented metrics, class reengineering.*

## 1.0 Introduction

A recognized risk of the Incremental Approach for OO software development is that it may degrade into a Code and Fix Approach[7]. This process degradation includes a failure to monitor class quality, which can result in class deterioration. Deteriorated Classes once satisfied good quality object-oriented class design properties (i.e., encapsulation, data abstraction, inheritance, information hiding etc.). Over time such classes may become difficult to maintain and prone to errors.

This paper is focused on the twin problems of identifying and reviving deteriorated classes. A technique called Class Deterioration Detection and Resurrection (CDDR) is presented. CDDR focuses on potential causes for class deterioration [6][9], and describes a technique to identify and revive "Deteriorated Classes" through the use of OO metrics [1][2][3][4] and reengineering techniques. CDDR primarily focuses on those deteriorated classes that have incurred high coupling and low cohesion and the resultant loss of abstraction and encapsulation during the course of system evolution. An important element of the CDDR activity is the application of OO design metrics to

prevent or detect class deterioration. Ebert and Morschel [3] used a similar goal-oriented metrics selection approach to evaluate object-oriented programs written in Smalltalk. For CDDR, a small set of OO metrics was selected from the literature. Chidamber and Kemerer [2] presented metrics for measuring class coupling and cohesion. Li and Henry [1] used coupling through abstract data types to evaluate violation of encapsulation within a class. Briand [4] and others have proposed coupling measures based on class-attribute, class-method and method-method interaction. The use of import and export coupling concepts suggested by Briand [4] helps to identify highly coupled classes and their impact to changes in large OO systems.

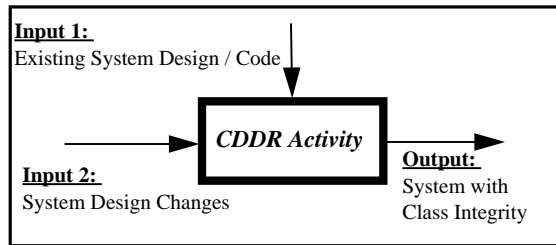
## 2.0 Class deterioration

An object model has several levels of representation, including (a) Application level, (b) Subsystem level, (c) Class level, and (d) Function level. While deterioration can occur at any level, the focus here is on class deterioration. This is the most fundamental level that constitutes a system. Improving deteriorated classes should help ameliorate subsystem and application deterioration.

When an Incremental Approach is used to develop OO software, the initial design model created in the first increment may have good design properties. However, over subsequent increments, this quality may be lost. Common causes for an Incremental Approach to produce a system with maintainability problems include (1) Schedule drives the process and unrealistic planning results in impossible delivery dates. (2) Lack of verification that the process is being followed and that activity deliverables satisfy intended quality goals.

These process difficulties can result in several problems. (1) Process activities lose focus with successive increments or entire process activities may be completely skipped. (2) Activities that are performed may not achieve their intended goals. For example, reviews are held but

**FIGURE 1. CDDR Activity**



reviewers are not focused on defect detection, but rather on obtaining “concurrency”.

One solution to prevent these problems is to use a process activity that periodically monitors the evolving system’s maintainability. This paper proposes such an activity: CDDR. This activity can be used to prevent the loss of maintainability, or restore it through reengineering.

### 3.0 Class Deterioration Detection and Resurrection - CDDR

The CDDR Activity can be used during the development phase of an Object-oriented project. It provides guidelines to ensure and restore maintainability.

#### 3.1 CDDR Process Activity

The descriptive model of the CDDR Activity is shown in Figure 1. A descriptive model specifies only the purpose of the process activity. CDDR as a stand alone activity can be performed in those projects that have suffered from development process degradation over time. If the activity is performed as stand-alone instantiation to resurrect an existing OO system from class deterioration, then Input 2 (in Figure 1) is not needed.

On the other hand, Input 2 (in Figure 1) is relevant if the activity is made an integral part of the development process. Figure 1 shows how the activity can fit into an

incremental development life-cycle. The design/code of a previous increment and the design changes of the current increment can be fed into the activity. CDDR as an integral part of the development phase not only helps retain class integrity but also continuously monitors the system for class deterioration.

Using CDDR activity as an integral part of the development process ensures class integrity and its stand-alone counterpart restores class integrity.

#### 3.2 Guidelines for detection and correction of Deteriorated Classes

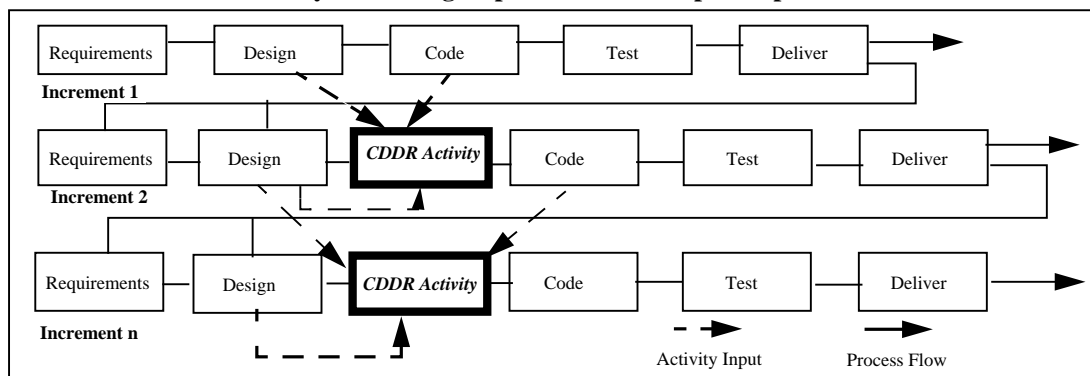
Steps instantiated within the CDDR activity provides developers with insight on where to focus when conducting design/code inspections. A similar approach is suggested by Briand et.al,[4] in their investigation of quality impact of the different design mechanisms in C++. However CDDR not only provides focal points for design/code inspection but also activities to rectify problems found. The explicit nature of the activity will help project managers to take into account the time and effort involved to develop a good quality OO system. The activity will aid managers to estimate realistic cost and time required during the development phase to build such a system.

The process of identifying deteriorated classes and resurrecting them is divided into four related sub-activities within the CDDR Activity. **Step 1** identifies parameters used to detect class deterioration. **Step 2** identifies candidate class categories that are prone to deteriorate. **Step 3** selects and applies OO metrics to detect deteriorated classes. **Step 4** reengineers the deteriorated classes.

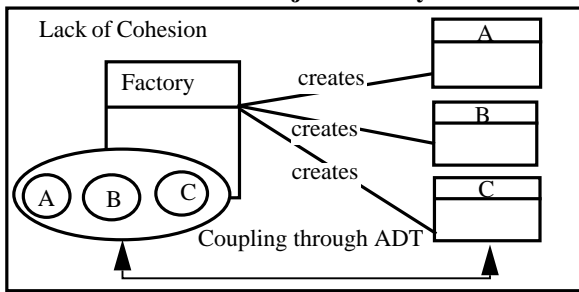
##### 3.2.1 Parameters to identify deteriorated classes

While there are several reasons for a class to lose quality over time, here the focus is on classes that have

**FIGURE 2. CDDR Activity as an integral part of the development process**



**FIGURE 3. Class / Object Factory Deterioration**



high coupling and low cohesion. These characteristics often result in loss of abstraction and encapsulation. It is those highly coupled classes that often lose cohesion during the course of development. Rumbaugh[5] states that the main warning sign towards identifying deteriorated classes is the lack of cohesion about the class. Based on designer intuition and viewpoints which are used in most empirical studies[2], we proceed with the following simple definitions for lack of cohesion and high coupling.

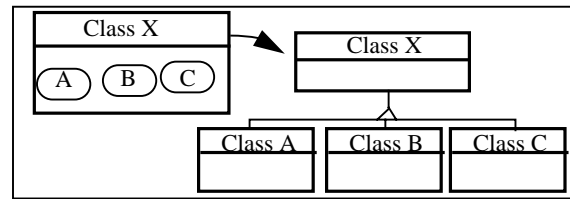
- Lack of cohesion within a class results in a tendency to break up into two or more parts due to the perceived sense of discord among the different attributes and operations.
- High coupling about a class results in a tendency to propagate changes across two or more classes.

Chidamber and Kemerer [2] provide definitions for these parameters in ontological terms. For a comprehensive detail of the definitions the reader is referred to [2] and [8]. For simplicity, the paper will use the above definitions and refer to those provided by [2].

### 3.2.2 Candidate Class Categories

A class could deteriorate for several reasons. A class may have conformed to good quality object-oriented design (i.e., encapsulation, information hiding, data abstraction etc.,) in the initial increments of development and lost integrity with increasing increments due to: (1) Addition of methods / data members. (2) A base class trying to accomplish too-much for its derived classes. (3)

**FIGURE 5. Multiple Personalities Class**



A class attempting to handle too many different situations, grouping what should be several different derived classes into a single class[6]. (4) A class with increasing number of relationships and associations with other classes and abstract data types.

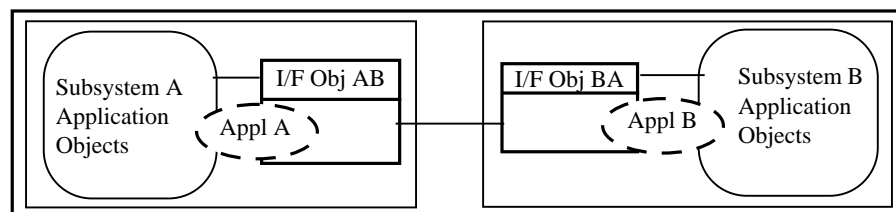
Based on these notions, certain classes can be categorized as potential classes that are more prone to deteriorate with development time. Categorization also provides early feedback to developers to consider design alternatives for such classes and inspection focal points.

Class Categories that are prone to deteriorate are Factory Class/Object (FC), Subsystem Interface Class/Object (SIC) and Multiple Personality Class (MPC). These class categories develop high coupling and become less cohesive over time.

#### Factory Class/Object (FC)

Class/Object Factories are those abstractions that create other classes or object instantiations and hold ownership as shown in Figure 1. It is a common practice in large systems that has several object instantiations to use Factory Classes as a design pattern. Class factories due to their ownership of other classes tend to acquire non-cohesive entities during the course of the development life-cycle. Figure 1 shows Class Factory acquiring non-cohesive entities (A, B and C) that should actually belong to Class A, Class B, Class C. Factory classes should be viewed as a repository of class access information for other entities that interact with those classes created by the Factory Class. Instead, Factory Classes tend to acquire methods (non-cohesive) that directly implement functionalities of those classes created by it.

**FIGURE 4. Subsystem Interface Class / Object Deterioration**



**Subsystem Interface Class/Object (SIC)**

Subsystem Interface Classes/Objects are those classes that act as an interface between the different subsystems that make up the entire software. With system evolution, these classes tend to gain non-cohesive entities that may belong to the actual subsystem application objects. As shown in Figure 4 the interface object - I/F Obj AB that acts as an interface between subsystems A and B gains non-cohesive entities that should actually belong to the application objects of subsystem A - Appl A. Similarly interface object - I/F Obj BA gains non-cohesive entities that belongs to subsystem B - Appl B.

**Multiple Personality Class (MPC)**

Classes that gain different personalities with system evolution. It should be noted that a multiple personality class is different from a class having several object instantiations.

Replacing an existing stand-alone class by an inheritance relationship in an evolving system having several thousand lines of code is suspect to overhead. Due to this overhead, designers often cause a class to become an MPC instead of introducing an inheritance relationship whenever necessary. The use of Export and Import Coupling measures[4] can be used to resolve such classes. As shown in Figure 5, Class X gains personalities A, B and C with system evolution which can be actually represented as base class, Class X and derived classes, Class A, Class B and Class C.

**3.2.3 OO Metrics and Class Reengineering Strategy**

Following the detection of deteriorated classes based on Section 3.2.1 and Section 3.0, changes are applied to reengineer the classes to restore maintainability. In order to reengineer a deteriorated class, one or more changes have to be performed. **Change 1:** Removal of methods or attributes from an existing class declaration. **Change 2:** Addition of methods or attributes

into an existing class declaration. **Change 3:** Addition of a new class within the system. **Change 4:** Conversion of an existing class into an abstract base class through modification (using Change 1 and Change 2).

Lack of cohesion will be an important parameter that will be used to reengineer a deteriorated class. High coupling is one of the reasons for gaining non-cohesive entities within a class. The subsequent discussion will make use of metrics for lack of cohesion suggested by [2] and coupling metrics suggested by [1][2][4].

Figure 6 shows the proposed class reengineering strategy using OO metrics. The strategy makes use of the 3 steps. **Step 1:** Apply and record OO metrics for coupling and cohesion evaluation to candidate class categories. **Step 2:** Reengineer detected deteriorated classes using the changes mentioned above. **Step 3:** Re-apply and record the OO metrics to the reengineered classes. Compare the recorded metrics results to evaluate class design improvement.

An object or a class, *i*, is a representation of the application domain that encapsulates (E(*i*)) a collection of its attributes (I(*i*)) and operations (M(*i*)),

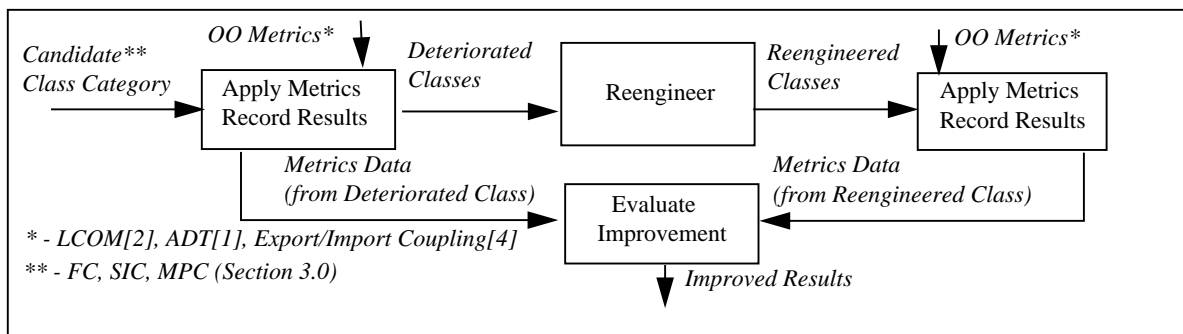
$$E(i) = M(i) \cup I(i)$$

where E(*i*) is the encapsulation of the elements of class *i*, M(*i*) is the set of methods of class *i* and I(*i*) is the set of attributes of class *i*. This **Class definition** will be used in the subsequent discussion. The reengineering strategy using OO metrics will be discussed with reference to the example in Figure 1. This example is chosen due its clear representation of a class deterioration due lack of cohesion induced by high coupling. It should be noted that this example contains a FC class with its associated ownership of other classes (Refer Section 3.0).

**Reengineering of FC Class Category**

Applying class definition formalization for the classes shown in Figure 1,

**FIGURE 6. Class Reengineering Strategy**



$$\begin{aligned}
E(Factory) &= M(Factory) \cup I(Factory) \\
E(A) &= M(A) \cup I(A) \\
E(B) &= M(B) \cup I(B) \\
E(C) &= M(C) \cup I(C)
\end{aligned}$$

Referring to Figure 1, class Factory falls under the FC class category and holds ownership of classes A, B and C. Due to this relationship, there exists a coupling through abstract data type (ADT)[1] between Factory and each one of the created classes A, B and C. Since Factory belongs to the FC class category, the ownership of the created objects as an ADT will be part of its class attributes. Such attributes are called Ownership Attributes.

O(i, j) represents an Ownership Attribute of class j and the data type of this attribute is an object instantiation of class i. From the above example, O(A, Factory), O(B, Factory) and O(C, Factory) are Ownership Attributes of Factory and their data types are object instantiations of classes A, B and C respectively. The metric that measures the coupling complexity caused by ADTs is data abstraction coupling (DAC = number of ADTs defined in a class[1]). For the above example DAC = 3. We can also use the Coupling between object classes (CBO) suggested by Chidamber and Kemerer[2]. CBO = number of other classes to which a class is coupled[2].

Using the notation for such a relationship, the union of the set of Ownership Attributes is an element of attribute set I(Factory) as shown below.

$$O(A, Factory) \cup O(B, Factory) \cup O(C, Factory) \in I(Factory)$$

If any given method, m(n) from the set of methods of Factory (M(Factory)) uses only a specific ADT and the attributes and methods of that ADT, then the method m(n) is a non-cohesive entity. Formalizing,

$$\begin{aligned}
m(1) &\Leftrightarrow O(A, Factory) \cup I(A) \cup M(A) \\
m(2) &\Leftrightarrow O(B, Factory) \cup I(B) \cup M(B) \\
m(3) &\Leftrightarrow O(C, Factory) \cup I(C) \cup M(C) \\
m(1) \cup m(2) \cup m(3) &\in m(nce) \\
m(nce) &\in M(Factory)
\end{aligned}$$

In the above formalization, m(nce) is the set of all non cohesive entities - m(1), m(2) and m(3). The set of all such non-cohesive methods should be distributed across those objects which is being used as ADTs in the deteriorated class. After moving the non-cohesive entities to the appropriate classes they can still be accessed by the reengineered class using the ownership attributes (O(A, Factory), O(B, Factory) or O(C, Factory)). This is more appropriate if such an access is association-driven. The metrics to measure cohesion that is used in this paper is Chidamber and Kemerer's [2] Lack of Cohesion in

Methods (LCOM) within a class (LCOM[2] = Number of similar method pairs - Number of dissimilar method pairs).

Applying LCOM not only helps to find methods that contribute to the lack of cohesion but also those attributes that may potentially belong to an existing class or a new class. Lack of cohesion implies class parts that should be moved to an existing class (as shown in the FC Class example) applying Change 1 and Change 2.

Lack of cohesion can also result in parts of a class giving rise to a new class applying Change 3. It is suggested to subject those new classes created from non-cohesive entities of existing classes to the class definition criterion given at the beginning of this section.

### Reengineering of SIC Class Category

The SIC class category develops similar non-cohesive entities due to its high coupling with the subsystem application objects. With time and feature enhancements to the system, soon SIC objects leak into the application and start acquiring functionalities of the application objects. Similar to FC class categories, the SIC can be detected using the LCOM [2] and DAC [1] metrics. They can reengineered by applying Changes 1 and 2. The key difference in the case of reengineering SIC classes is that Change 3 is also likely to be applied, wherein a new application object can be created from the existing non-cohesive entities of the SIC object. It should be noted that the FC and SIC class categories suffer from LCOM results accompanied by a high value of DAC results.

### Reengineering of MPC Class Category

The MPC class categories arise due to the conglomeration of a good inheritance relationship into a single class. Using LCOM[2], we can detect a potential deteriorated class. Any deteriorated class detected as a result of a lack of cohesion, but does not suffer from a high level of coupling (LCOM present with a low DAC value, unlike FC and SIC class categories) belongs to this class category. In other words, MPC class category does not experience low cohesion due to high coupling with other entities, but due to multiple abstractions hidden within the same class. Applying DAC[1] will reveal the above concept. Change 4 can be implemented when splitting a deteriorated class belonging to a MPC class category into one more classes with an inheritance design as shown in Figure 5. Use of Export/Import Coupling[4] can help guide the re-design in order to avoid high coupling due to the resulting inheritance. The metrics used to evaluate the impact of the introducing inheritance relationship are: DIT[1] (Depth of Inheritance Tree = number of super-

**TABLE 1. Reengineering deteriorated classes**

Category	Change	Metric	Remark
FC	Changes 1, 2	DAC[1], CBO[2], LCOM[2]	Redistribution of non-cohesive entities among existing classes.
SIC	Changes 1, 2, 3	DAC[1], CBO[2], LCOM[2]	Redistribution of non-cohesive entities among existing classes or into a new class.
MPC	Change 4	DAC[1], CBO[2], LCOM[2], DIT[1], NOC[1], Export/Import Coupling[4]	Evolution of an inheritance relationship.
Others	Change 3	DAC[1], CBO[2], LCOM[2]	Any given class that has increased in size with time.

classes of a class) and NOC[1] (Number of Children= number of children or sub-classes affected).

Table 1 provides a summary of the guidelines used for reengineering a deteriorated class. The technique does not impose on the choice of metrics to be used for detection of deteriorated classes.

#### 4.0 Case Study

The CDDR activity was applied to an industrial telecommunications network management system written in C++. The project lasted for a two-year period and involved 20 developers. A project post-mortem determined that the Incremental Approach had collapsed into code-and-fix work. The CDDR activity was evolved at this time to correct problems observed in the OO system.

Here, the CDDR was conducted as a stand-alone activity in an effort to improve the maintainability of one subsystem. The subsystem consisted of 30 classes, four of which had deteriorated. Applying metrics suggested by Li[1] and Chidamber[2] identified and validated the deteriorated classes. The classes belonged to the Factory Class/Object (FC) and Subsystem Interface Class/Object (SIC) categories. Changes 1, 2 and 3 were applied to these classes.

Applying the CDDR activity to revive the deteriorated classes improved sub-system maintenance. Developers felt that subsequent changes to the reengineered classes for later releases were easier to produce and reduced the effort and time to test the system. Efforts are under way to incorporate the CDDR activity an integral part of the development LifeCycle (Figure 1) for future projects.

#### 5.0 Conclusion

This paper focused on an important maintenance problem faced by industrial object-oriented. Class Deterioration is one of the factors that cause

maintainability problems in object-oriented systems. Webster[6] and Rumbaugh[5] have pointed out object bloating and fat objects as design issues. We have addressed a process activity that can avoid, detect and rectify this problem from a practical standpoint. The paper focused on class deterioration primarily due to lack of cohesion induced by high coupling, which is prevalent in large-scale systems. Future work will be directed towards other parameters that contribute to class deterioration and maintenance issues that can provide a comprehensive process model. The CDDR activity is a key activity that will become part of a comprehensive process model to assess and improve large scale OO system development and maintenance.

#### 6.0 References

- [1] Li, Wei; Henry, Sallie, "Object-Oriented Metrics that Predict Maintainability", *Journal of Systems Software*, Vol. 23, pp. 111 - 122.
- [2] Chidamber, Shyam R., Kemerer, Chris F., "A Metrics Suite for Object Oriented Design", *IEEE Transactions on Software Engineering*, Vol. 20, No. 6, June 1994, pp. 476 - 493.
- [3] Ebert, Christof; Morschel, Ivan, "Metrics for quality analysis and improvement of object-oriented software", *Information and Software Technology*, Vol. 39, 1997, pp. 497 - 509.
- [4] Briand Lionel, Devanbu Prem, Melo Walcelio, "An Investigation into Coupling Measures for C++", *International Conference on Software Engineering*, Boston, MA, 1997, pp. 412 - 421.
- [5] Rumbaugh James, "Trouble with twins: Warning signs of mixed-up classes", *Journal of Object Oriented Programming*, July-August 1994, pp. 16 - 21.
- [6] Webster Bruce F., "Pitfalls of Object-Oriented Development", M&T Books, New York, 1995.
- [7] Pressman Roger S., "Software Engineering - A practitioner's approach", McGraw Hill, Inc., (3rd Ed.), New York, 1992.
- [8] Tagaki K., Wand Y., "An object-oriented information systems model based on ontology", *Object Oriented Approach in Information Systems*, F. Van Assche and B. M. C. Rolland, Eds. New York: Elsevier Science Publishers B.V. (North Holland), 1991.
- [9] Pittman, Matthew, "Lessons learned in managing object-oriented development", *IEEE Software*, January 1993, pp. 43 - 53.