CHAPTER 3
CLASS EQUIVALENCE CATEGORIES

3.1 ATOMIC CHANGES AND CLASS EQUIVALENCE

Chapter 2 introduced the notion of atomic changes that govern the way a class evolves from one version to another. In this chapter these atomic changes are viewed as operations that affect the equivalence between different class versions. It is shown that between two classes there may be four types of equivalence.

Section 3.2 describes the equivalence classes with examples from C++ and Java and Section 3.3 compares these categories. Section 3.4 shows the impact of these atomic changes on class equivalence. Section 3.5 describes the S-equivalence determination algorithm.

3.2 CLASS EQUIVALENCE CATEGORIES

If a class C undergoes source modifications and evolves into a class C', C' may or may not be equivalent to C. This thesis suggests four types of equivalence that may exist between C and C'. The equivalence determination is based on examining the source code corresponding to C and C'.

3.2.1 L-Equivalence

The two classes C and C' are L-equivalent (L-Eq) if they are lexically equivalent. This essentially means that they are source equivalent. In other words\textsuperscript{1}, running a file differencing utility on the source corresponding to these classes will reveal no differences, implying C and C' are one and the same. This is the strongest kind of equivalence since it establishes that C' is the same as C. This is also the most trivial of class evolution possibilities!

\textsuperscript{1}Here we assume that each class is defined in a single source file.
3.2.2 S-Equivalence

The two classes C and C' are S-equivalent (S-Eq) if they are not lexically equivalent, but are structurally equivalent. Structural equivalence may be determined by building an abstract syntax tree (AST) for each of the two classes C and C', reducing them to a canonical form, and comparing the trees (see section 5). Under this interpretation, the following two Java classes are S-Equivalent:

\textbf{Class C:}

class SomeClass {
    private int i, j;
    public SomeClass() {
        i = 22; j = 33;
    }
}

\textbf{Class C':}

// This class does something useful.
class SomeClass {
    public SomeClass() {
        i = 22;
        j = 33;
    }
    private int i, j;
}

C' differs from C in the following ways:

a) A comment line has been added
b) Order of class members has changed
c) Source re-formatting in terms of indentation and braces has been performed

What are some of the reasons for a class to evolve this way? Programmers are usually required to follow certain coding practices that may require a minimum level
of code commenting, adopting a standard indentation style and so on (Henricson et al 1997, Taligent Press 1994).

3.2.3 T-Equivalence

Let us suppose that T(C) is some well-designed test suite produced originally as a test set for class C. Two questions are relevant here:

a) Will C' behave correctly with respect to T(C)? and
b) Does C' require additional test cases to distinguish from C?

C' is **correct** with respect to T(C) if C' exhibits the same behavior as does C with respect to the test suite. T(C) is **r-adequate** for C' if C' is **correct** with respect to T(C) and C' needs no additional tests to distinguish itself from C.

Two classes C and C' are T-Equivalent (T-Eq) if

a) they are not S-Equivalent
b) T(C) is r-adequate for C'
c) it is possible to establish (b) without actually running T(C) on C'.

Condition (c) is quite essential to the understanding of T-Equivalence. We are, in a sense, assuring that even though C' is different from C, it need not be retested. Is it possible to establish the adequacy of a test suite for an evolved class without the need to actually run the tests? This is indeed possible when the evolution is governed by certain source transformations (Rangarajan et al 1997 a, Rangarajan et al 1997 b). As an example, the following two C++ classes are T-Equivalent:

```cpp
Class C:
class SomeClass {
public:
    SomeClass(int a, int b) : i(a), j(b) {} 
    int getValue() { return j+i; }
}
```
private:
    int i;
    int j;
};

Class C':
class SomeClass {
public:
    SomeClass(int a, int b) : i(a), j(b) {}
    int getValue() const { return j+i; }
protected:
    const int i;
    const int j;
};

Here, C' differs from C as follows:

1) Data members are protected
2) Data members are const
3) Method getValue() is const

Why would a class evolve in this manner?

1) The designer realizes that a future derived class may need to access these data members and so makes them protected
2) Since the data members are not changed (and are not to be changed) by any method of the class, the designer captures this by making them const
3) Since the getValue() method does not change the data members, it is sensible to assert this by making the method const

A study of C and C' reveals that C' is not functionally different from C and hence a test suite that was designed for C would be adequate for C'. In fact, we can make a stronger claim: There is no need to retest C'!
3.2.4 F-Equivalence

The last of the equivalence categories is F-Equivalence. C and C' are F-Equivalent (F-Eq) if they are not S-Equivalent, but are functionally equivalent. Functional equivalence implies that the two classes provide the same functionality even though the implementation may be different. The following two classes are F-Equivalent:

**Class C:**
```cpp
class SomeClass {
public:
    void ff(int val) {
        if( val < 45 ){
            cout << "Less than 45";
            return;
        }
        cout << "Greater than or equal to 45";
    }
}
```

**Class C':**
```cpp
class SomeClass {
public:
    void ff(int val) {
        if( val >= 45 )
        
            cout << "Greater than or equal to 45";
        else
            cout << "Less than 45";
    }
}
```

Changes of this nature happen frequently. Developers modify their source code in order to achieve better performance, reduced code size, and so on. It is essential to remember that the evolved class must provide the same functionality to its consumers for it to be considered in this category. In this example, C' could also have been implemented as suggested below for it to be F-Equivalent to C.
Class C:

```cpp
class SomeClass {
public:
    void f(int val) {
        checkValue(val);
    }
private:
    void checkValue(int val) {
        if (val >= 45)
            cout << "Greater than or equal to 45";
        else
            cout << "Less than 45";
    }
};
```

If the `checkValue()` function is made public, then F-Equivalence is not preserved since the public interface itself changes (thereby requiring an enhanced test suite). These arguments imply that if \( C' \) F-Eq \( C \), \( T(C) \) is \( r \)-adequate for \( C' \).

How is F-Eq different from T-Eq since both depend on test suite adequacy? If the \( r \)-adequacy of \( T(C) \) for \( C' \) can be established \textit{without} running the tests, then \( C' \) T-Eq \( C \). Otherwise, \( C' \) F-Eq \( C \).

### 3.3 RELATIONSHIP BETWEEN EQUIVALENCE CLASSES

The equivalence categories presented in Section 2 reveal a monotonic ordering among themselves. This is shown in Figure 3.1.

![Figure 3.1 Equivalence](image_url)
As expressed in the figure,

\[ C' \text{ L-Eq } C \implies (C' \text{ S-Eq } C) \text{ and } (C' \text{ T-Eq } C) \text{ and } (C' \text{ F-Eq } C) \]

In other words, L-Eq is the strongest form of class equivalence and F-Eq is the weakest. Given this relationship, it is easy to understand that a class \( C' \) is not equivalent to another class \( C \) iff \( C' \text{ F-Eq } C \) is false.

### 3.4 EQUIVALENCE CATEGORIES AND REGRESSION TESTING

How does this classification help us in practice? Suppose it is known that a class \( C \) undergoes changes and evolves to \( C' \) such that \( C' \) and \( C \) are equivalent as per the suggested classification. Does the modified class need to be regression tested? As a study of the equivalent categories shows, regression testing is not always necessary. Table 3-1 depicts this dependency.

<table>
<thead>
<tr>
<th>Equivalence</th>
<th>Regression Testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>( C' \text{ L-Eq } C )</td>
<td>Not Needed</td>
</tr>
<tr>
<td>( C' \text{ S-Eq } C )</td>
<td>Not Needed</td>
</tr>
<tr>
<td>( C' \text{ T-Eq } C )</td>
<td>Not Needed</td>
</tr>
<tr>
<td>( C' \text{ F-Eq } C )</td>
<td>Needed</td>
</tr>
</tbody>
</table>

Table 3.1 Equivalence and Retesting

### 3.5 DETERMINING S-EQUIVALENCE

Structural Equivalence of two classes can be determined by building an abstract syntax tree for each of the two classes, reducing them to a canonical form, and comparing the trees.
3.5.1 Abstract syntax tree (AST)

Abstract syntax tree is an intermediate datastructure built during the semantic analysis phase and is used by subsequent stages. It typically captures symbol bindings and type mappings and makes use of auxiliary datastructures such as symbol tables. For example, the syntax tree for the expression \( a = b + 2 \) might look like

\[
\begin{align*}
= & \quad \text{(AssignmentNode)} \\
\text{a} & \quad + \quad \text{(BinaryOperatorNode)} \\
\text{b} & \quad 2
\end{align*}
\]

3.5.2 Canonical form

To ascertain the structural equivalence of two classes, the corresponding abstract syntax trees must be compared. Before comparing, however, the trees must be reduced to their canonical form. This is necessary since class elements may have been reordered from one version to another.

A class can be reduced to a canonical form by sorting the class members in a predefined order. This ensures that ASTs of two classes, with only the order of class member declaration changed, will be the same.

class A:
class A {
int i = 2, p = 3;
int method(int p) {
    return p + i;
}
}
class A':
class A {
    int p = 3;
    int method(int p) {
        return p + i;
    }
    int i = 2;
}

Figure 3.2 Classes before canonical ordering

Classes A and A' are structurally equivalent, since only the class members have been reordered. However, direct comparison of their abstract syntax trees will fail. This problem can be solved by sorting the class members (in the AST's) in a predefined order. The exact order in which the members are sorted is not important as long as it is consistent and ensures a unique representation.

Figure 3.3 Classes after canonical sorting
3.5.3 Class element sorting strategy

The class members are sorted canonically by grouping the class members of same type together, and arranging them in a predefined order. The following table (applies to Java) depicts the order in which class members are grouped and the key used to sort each type of class member.

<table>
<thead>
<tr>
<th>Class Member</th>
<th>Sorted By</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Inner Class</td>
<td>Class name</td>
</tr>
<tr>
<td>2 Static Initializer</td>
<td>—</td>
</tr>
<tr>
<td>3 Method</td>
<td>Method signature</td>
</tr>
<tr>
<td>4 Instance Initializer</td>
<td>Original order is maintained</td>
</tr>
<tr>
<td>5 Field</td>
<td>Field name</td>
</tr>
</tbody>
</table>

Table 3.2 Element Sorting Strategy

3.5.4 Comparing abstract syntax trees

Once the class is reduced to a canonical form, the abstract syntax trees of the class members can be compared. This can be done by comparing the AST of each class member of one class with the corresponding class member of the other. Since the two classes are in a canonical form, the class members of the two classes will be in the same order, and can be compared directly.

The syntax trees of the class members are compared by matching each node of the first tree with the corresponding node of the second tree. We do this by first matching the root nodes and then descending the tree, comparing each node along the way.

\[
\text{algorithm CheckSEQ}(C, C') : \text{Result} \\
\text{input C: AST corresponding to class} \\
\text{C': AST corresponding to its modified version} \\
\text{output Result: bool} \\
\begin{aligned} 
\end{aligned}
\]
if root_of(C) != root_of(C')
    return false

let m = number of immediate subtrees of C
let n = number of immediate subtrees of C'
if m != n
    return false

for i = 0 to m
begin
    if CheckSEQ(Xi, Yi) == false
        return false
    /* Xi and Yi are the ith subtrees of X and Y respectively */
end
return true
end CheckSEQ

Let us apply this algorithm to the two classes given below:

**Before change:**

class A {
    int i = 2, q = 3;
    int method(int p) {
        return p + i;
    }
}

**After change:**

class A {
    int q = 3;
    int method(int p) {
        return p + i * q;
    }
    int i = 2;
}

When this algorithm is applied on the two trees shown in Figure 3-4, CheckSEQ will return false when it encounters the distinct nodes 'i' and the Binary Operator Node (q * j).
The false value propagates up the tree and CheckSEQ returns false for the entire statement.

3.6 SUMMARY

If two classes are equivalent to each other, then this equivalence will be one of four categories: L, S, T, and F. L is the strongest form of equivalence whereas F is the weakest. Each of the atomic changes supported by an object-oriented programming language such as C++ has predictable properties with respect to equivalence preservation. This chapter has described the four types of equivalence categories along with an algorithm to detect S-equivalence. T-equivalence is an important notion since it predicates that a test suite T(C) designed for class C need not be run for the modified class C' iff the new class is T-equivalent to the original one. Establishing T-equivalence between C and C' requires understanding the atomic changes between the two classes through static analysis, and is independent of T(C). Since running a test suite involves a constant set up time and execution time proportionate to the size of the suite, in general, the static analysis process will be more efficient than rerunning the entire test suite.