CHAPTER 2

ATOMIC CHANGES AND CLASS EVOLUTION

2.1 HOW DOES A CLASS EVOLVE?

When software evolves due to one or more of the reasons cited in Chapter 1, it impacts some of the classes in the modified version. Three types of code changes can be broadly identified.

a) **Functional:** These changes typically involve class interface. New behavior may be added, existing behaviors may be removed, or method interfaces may undergo changes. An example would be adding a method `getTop()` to a Stack class to return the top element (without popping).

b) **Implementation:** The class interface essentially remains the same, but the underlying implementation may undergo changes for reasons such as reusing existing code, and enhancing performance by choosing a different algorithm. For instance, a sorting function, originally implemented using bubble sort algorithm, could be reimplemented to use the quick sort algorithm.

c) **Structural:** In this category, the developer adjusts the overall structure of the class without affecting the functionality. Some examples include changing the order of member declarations and modifying accessibility attributes.

This chapter shows that class evolution is governed by a finite set of atomic changes that are language-specific. A study of the properties of these atomic changes reveals that some of them do not induce retest on the modified version (Rangarajan et al 1996).
Section 2.2 outlines the notion of atomic change while section 2.3 describes their properties. In Section 2.4, the notion of Lexical Function Context (LFC) that provides clues to retesting an evolved class is proposed. Section 2.5 suggests how LFC may be used in retesting. The basic theory of atomic changes is explained in Section 2.6, and in section 2.7, the language-based retest approach is discussed. In Section 2.8, a macro process for identifying the atomic changes for an object-oriented programming language is put forth.

### 2.2 WHAT IS AN ATOMIC CHANGE?

An atomic change is defined as a change applied to a class definition (i.e., its source code) such that

a) it is minimal (cannot be decomposed into other atomic changes) and

b) the program compiles and builds after the change.

Requirement (a) ensures atomicity and (b) ensures syntactic validity.

The proposed notion of atomic change operates at the level of code, and not at a more abstract level of design or architecture. This guarantees that no matter why or how a piece of software changes, the changes can be described solely in terms of atomic changes. To make this point clear, consider an existing Java-based application that supports a 2-tier client server architecture. If this application is changed to support a 3-tier architecture using remote method invocation (RMI), this architectural change will certainly involve changing some of the classes in the existing code base. Atomic changes refer to these code-level changes.

**Consider the C++ class**

```cpp
class Example1 {
public:
    Example1() : val(0) { }
    void Reset() { val = -1; }
    int val;
};
```
If this class is changed to

```cpp
class Example1 {
public:
    Example1() : val(0) { }
    void Reset() { val = -1; }
private:
    int val;
};
```

the change introduced (changing the member access level) is atomic, since it cannot be decomposed into two or more changes and the new definition compiles and builds. On the other hand, consider

```cpp
class Example2 {
public:
    Example2() : val (0) { }
    void Reset() { val = -1; }
private:
    int val;
};
```

If this class is changed to

```cpp
class Example2 {
public:
    Example2() ; val (0) { }
    void Reset() { val = -1; }
private:
    const int val;
};
```

this change (making a data member `const`) is nonatomic in this class, since the new definition will not compile. However, the same change of making a data member `const` is atomic in the following class:

```cpp
// Before change
class Example3 {
public:
    Example3() : val (0) { }
    int GetVal() const { return val; }
};
```
private:
    int val;
);

// After change
class Example3 {
public:
    Example3() : val (0) { }
    int GetVal() const { return val; }
private:
    const int val;
};

As another example of what is not an atomic change, consider a trivial Java class

class X { }
If this is changed to

class X {
    private int
}

this is not a valid atomic change since the new (incomplete) definition will not compile. Hence part two of the definition of atomic change suggesting that the code must compile and build after the change is applied is important.

Atomic change is deliberately defined with respect to a class and not as an arbitrary change to a program. To elucidate this point, consider a programmer changing the body of a method belonging to a class. This is considered as an atomic change to the class, to be precise, changing the method of a class.

As a more meaningful example of a class evolving through atomic changes, consider:

class X {
    private:
        int a;
        int f();
        void g() { }
    }
public:
X(int val) : a(val) { }
void h() {
    g(f());
}
int X::f() {
    return a;
}

Assume that in the next version this class evolves to

class X {
    public:
        const int a;
    X(int val) : a(val) { }
    void h() const {
        g(f());
    }
    private:
        float fl;
    int f() const;
        void g() {
    };
    inline int X::f() const {
        return a;
    }
}

This evolution can be described as a combination of the following atomic changes:

a) introduce a new data member float fl in the private section
b) change the access level of data member a from private to public
c) make the data member a const
d) reorder public and private sections
e) make the member function f inline
f) make the member function f const
g) make the member function h const
In some situations, the determination of atomic changes corresponding to the modified version may not be straightforward. To understand this point, consider:

```cpp
class A {
    private:
        int i;
    public:
        void ff() {
            i = 9;
        }
};
```

If the class is modified to

```cpp
class A {
    private:
        long j;
    public:
        void ff() {
            j = 9;
        }
};
```

the atomic changes are

a) delete the data member i
b) add a data member j
c) change method body to use j instead of i

Another possibility comprises

a) rename data member i to j
b) change the type of j to long
c) change method body to use j instead of i

However, the former is preferred in our interpretation. The rule being followed is if the same element undergoes two or more atomic changes, the element is considered deleted and a new element added.
2.3 PROPERTIES OF ATOMIC CHANGES

Since atomic changes are the means by which one class evolves to another, it is useful to study their properties. This thesis identifies three properties of interest.

2.3.1 Atomic changes are language-dependent

Not surprisingly, the atomic changes are a function of the programming language. There may, however, be some atomic changes that are common across a group of object-oriented languages. A few of the atomic changes corresponding to C++ are

* Reordering class elements
* Changing the access level of a class member
* Defining a new instance variable
* Removing an instance method
* Making a member function const
* Inlining a member function
* Templating a class
* Changing the class derivation mode

Some of the atomic changes corresponding to Java are

* Reordering class elements
* Changing the access level of a class member
* Defining a new instance variable
* Removing an instance method
* Implementing an interface
* Making a method final
* Making a class abstract
* Making method synchronized
As is evident from this partial enumeration, first four atomic changes are common to both C++ and Java. Chapters 4 and 5 respectively describe the atomic changes possible in C++ and Java.

2.3.2 Atomic changes are finite

Though atomic changes are language dependent, they are finite in number. This can be seen from the fact that the atomic changes are closely tied to language features. For instance, Java supports the notion of synchronized methods and this feature gives rise to two atomic changes making a nonsynchronized method synchronized and making a synchronized method nonsynchronized. Since every language supports a finite set of features, the atomic changes that the language admits are also finite.

2.3.3 Atomic changes may or may not preserve class equivalence

When an atomic change is applied to a class, it mutates the class into a new version that may or may not be equivalent to the original one. This notion of class equivalence is useful in regression testing. Four types of equivalence have been identified between classes (Chapter 3 goes into the details of equivalence categories). They are,

1) Lexical Equivalence (L-Equivalence)
2) Structural Equivalence (S-Equivalence)
3) Test Equivalence (T-Equivalence)
4) Functional Equivalence (F-Equivalence)

If a class C' can be shown to be L, S or T-equivalent to its original version C, then C' need not be retested. Here are the equivalence properties of some of the atomic changes in C++ and Java (elaborated in Chapters 4 and 5 respectively):
Language | Atomic Change | Ensures
---|---|---
C++ | Introducing comments | S-Eq
C++ | Reordering class members | S-Eq
C++ | Reordering base classes | S-Eq
C++ | Inlining a member function | T-Eq
C++ | Changing member access level | T-Eq
C++ | Making a member function const | T-Eq
Java | Introducing comments | S-Eq
Java | Reordering class members | S-Eq
Java | Reordering interfaces | S-Eq
Java | Making a method final | T-Eq
Java | Making a class final | T-Eq
Java | Making a class public | T-Eq

Table 2.1 Atomic Change and Equivalences

What appears interesting is that the same atomic change may preserve equivalence in one language, but not in another. For example, making a method of a class `private` (changing member access level) guarantees T-Equivalence in C++, but not in Java. This means that the behavior of the evolved class (with respect to this atomic change) in Java may not be the same as the original version and hence retesting is called for. For a C++ class, since T-Equivalence is ensured, no retesting is necessary!

2.4 LEXICAL FUNCTION CONTEXT

In order to determine which atomic change may require retesting a class, this thesis proposes the concept of Lexical Function Context (LFC) for class member functions.
The LFC of a class member function is the set of all symbols this function binds to *statically*. For example, consider the C++ code fragment shown below:

```cpp
text
extern void g(int);
class X {
    private:
    int a;
    int f();
public:
    X(int val) : a(val) {
    void h() {
        g(f());
    }
    int X::f() {
        return a;
    }
};
```

The LFCs of member functions of X are

- \( \text{LFC}(X::\text{void}) = \{ X::\text{int}_a \} \)
- \( \text{LFC}(X::\text{int}_f_\text{void}) = \{ X::\text{int}_a \} \)
- \( \text{LFC}(X::\text{void}_h) = \{ X::\text{int}_f_\text{void}, ::\text{void}_g_\text{int} \} \)

As the above set shows, the LFC includes for each symbol its signature as well. This thesis uses a name mangling scheme similar to the one used by C++ compilers to derive a unique name from its complete signature (see Appendices 1 and 2).

## 2.5 LFC AND CLASS RETESTING

How does one know whether a class needs to be retested after a sequence of atomic changes? Here are the criteria:

1) If a new member function is introduced as part of the change, then this function needs to be retested.
2) If the body of a member function changes, that member function needs to be retested.

3) If the signature of a member function changes, the changed member function needs to be retested.

4) In all other cases, the LFC of each member function prior to change and after change is computed. Assuming S represents the sequence of atomic changes applied to class C, it can be concluded that a member function C::mi requires retest only if

\[ \text{LFC}(C::mi) \neq \text{LFC}(C^S::mi) \]

This rule is stronger (hence, safer) than necessary. In some cases, this rule will suggest retesting though it may not be necessary. In such cases, properties of the actual atomic change that causes the LFC to change needs to be looked up. If the atomic change preserves S or T equivalence, retest will not be required.

An example of this type of atomic change is a method in C++ becoming \textit{const} in the modified class.

5) Let F be the set of member functions of the class C that require retest. Each member function of the class that binds to a function in the set F will also require to be retested.

For instance, if one applies the following sequence of atomic changes to the class X shown above:

a) introduce a new data member \texttt{float ff} in the \texttt{private} section
b) change the access level of data member \texttt{a} from \texttt{private} to \texttt{public}

c) make the data member \texttt{a const}
d) rearrange \texttt{public} and \texttt{private} sections
e) make the member function \texttt{f inline}
f) make the member function \texttt{f const}
g) make the member function \texttt{h} \texttt{const}

The changed class definition is

\begin{verbatim}
extern void g(int);
class X {
  public:
    const int a;
  X(int val) : a(val) {} 
  void h() const {
    g( f() );
  }
  private:
    float ff;
    int f() const ;
  }
  inline int X::f() const {
    return a;
  }
}
\end{verbatim}

The corresponding LFCs are,

LFC(X::\texttt{void} X) = \{ X::int_a \}

LFC(X::int_f_void) = \{ X::int_a \}

LFC(X::\texttt{void} h) = \{ X::int_f_void@const, ::void_g_int \}

Though LFC of X::\texttt{h} has changed, since making a member function \texttt{const} preserves T-equivalence, the modified class does not require retesting.

As an example of when a class needs retesting, consider the class X evolving to

\begin{verbatim}
extern void g(int);
class X {
  private:
    int a;
    int f();
  void g(int i) { /* body */ }
  public:
    X(int val) : a(val) {
}
\end{verbatim}
void h() {
    g( f() );
}
}
int X::f() {
    return a;
}
The LFCs of member functions of X are
LFC(X::void_X) = { X::int_a }
LFC(X::int_f_void) = { X::int_a }
LFC(X::void_h) = { X::int_f_void, X::void_g_int }

Since X::h binds to a different function in the modified version, that method (and hence the class) needs to be retested.

2.6 THE THEORY

Let $L$ be any object-oriented programming language. Let the finite set $A^L$ represent the atomic changes corresponding to $L$.

Then $A^L = A^L_N \cup A^L_M \cup A^L_R$

Where $A^L_N$ = set of Atomic changes of $L$ not requiring retest.

$A^L_M$ = set of Atomic changes of $L$ may require retest.

$A^L_R$ = set of Atomic changes of $L$ require retest.

The three sets $A^L_N$, $A^L_M$ and $A^L_R$ are mutually exclusive for a given programming language. This is depicted in figure 2-1.
Figure 2.1 The set of Atomic Changes

The following description uses terms defined in Z (Jocky 1997).

Let $S$ be a sequence over $A$. A sequence is an ordered collection of elements.

$S : \text{seq } A$

Let $C^S$ denote a class $C$ that has undergone a sequence of atomic changes.

If $P$ and $Q$ are sequences over $A$ such that $S$ is a concatenation of the two, i.e.,

$P, Q : \text{seq } A$

$S : P \cap Q$

one can infer

$C^S = (C^P)^Q$

In the special case where

$\text{ran } S \subseteq A_N$

$C^S$ does not require any retest. That is, when a class undergoes a series of atomic changes each of which does not induce retest, the changed class need not be retested. Similarly,
if \( ran \ S \subseteq A_R \) the class requires retest and

if \( ran \ S \subseteq A_M \) the class may require retest.

2.7 ATOMIC CHANGES AND RETESTING

The Language-Based approach to regression testing proposed in this thesis may be described in terms of the following steps:

For each object-oriented language \( L \), identify the set \( A^L \) of atomic changes. These are finite and static.

Group the atomic changes into sets \( A^N_L \), \( A^R_L \) and \( A^M_L \).

Given a program in language \( L \), \( p^L \) and its evolved version \( p'^L \), identify the set of atomic changes \( B^L \subseteq A^L \) such that \( p'^L \) is derivable from \( p^L \) through finite application of elements of \( B^L \).

If \( B^L \subseteq A^N_L \) retesting is not required.

If \( B^L \subseteq A^R_L \) retesting is required.

If \( B^L \subseteq A^M_L \) retesting may be required.

If retesting may be required, compute LFC for each class member function before and after evolution and confirm/reject the need for retesting.

2.8 IDENTIFYING ATOMIC CHANGES

Given that atomic changes are language-dependent, how does one go about identifying these for a specific object-oriented programming language? This thesis proposes a Language – Feature – Syntax framework for identifying atomic changes. In this framework, the first step is to select the language of interest. Then the language features are enumerated. Lastly, for each feature, the syntactic variations of that feature are listed. Assuming that a language has been chosen, here is the macro process:
Step 1: Identify language features centered around a class. This includes enumerating the intra-class element categories and inter-class relationships. Ignore statement categories as these are embedded inside a block (typically in a method) and therefore are not visible at the class level. Examples of such features in Java are

a) instance variable
b) class variable
c) instance method
d) inner class
e) static block
f) native method
g) final class
h) interface implementation
i) subclassing
j) method signature
k) method throw list

Step 2: For each feature so identified, enumerate the possible programmatic (syntactic) variations that may occur with respect to that feature. For the list identified above, these are

a) instance variable
   i) defining an instance variable
   ii) removing an instance variable
b) native method
   i) making a method native
   ii) making a method non-native
Step 3: Evaluate those language aspects that may not crisply belong to inter-class or intra-class category. If they have any bearing on class evolution, include them in the features list. For C++ and Java, two features have been identified in this group. They are

a) comments

b) class element ordering

These give rise to the following atomic changes:

a) comments
   i) adding a comment in a class scope
   ii) removing a comment from class scope

b) class element ordering
   i) changing the order of class members

While working on this research, the need for a formal specification of programming languages (see Rangarajan and Eswar 1995 for an attempt) was strongly felt. In the absence of any such formal document, there is a need to iterate over these three steps several times till a stable state of not being able to include any more to the list is achieved.
2.9 SUMMARY

Atomic changes are the fundamental language-specific units of change in an evolving object-oriented program. Every program that changes across versions can be described as a permutation of these changes. For a given object-oriented programming language, the atomic changes are finite in number. Some of these atomic changes have the interesting property that they do not alter the program behavior and hence do not induce retest on the changed code. This property may be understood in terms of the notion of class equivalence. When a class evolves, the evolved class may be equivalent to the original class in one of four ways: L, S, T, or F. The next chapter describes this notion of class equivalence in detail.