CHAPTER 4

CATALOG OF ATOMIC CHANGES IN C++

This chapter describes each of the atomic changes possible in a C++ program. In this chapter and the next, the following format is adhered to for documenting atomic changes:

a) **Description:** This describes the meaning and context of the atomic change under discussion.

b) **Examples:** One or more examples illustrating how the source gets affected by the atomic change.

c) **Equivalence:** Since some atomic changes may preserve class equivalence, this section mentions the impact of the atomic change on class equivalence.

d) **Reason:** This section explains the significance of the atomic change from programming perspective. Observations on how common it is and when it appears sensible are documented.

A summary of the atomic changes detailed in this chapter may be found in Appendix 3.

4.1 ATOMIC CHANGES

Atomic changes, explained in Chapter 2 are the fundamental, atomic, syntactically correct, source changes in a given program as it evolves. These are language dependent. A study of C++ reveals 61 of these atomic changes. Each of the changes is described here.
4.1.1 Add comments

4.1.1.1 Description
This corresponds to adding comments within a class. The assumption is that the new comment does not invalidate code.

4.1.1.2 Examples
Before change:
class Test {
    int a;
    public: void print() {
        cout << "Test::a = " << a;
    }
};

After change:
class Test {
    int a;
    // Print the instance variable for debug purposes
    public: void print() {
        cout << "Test::a = " << a;
    }
};

4.1.1.3 Equivalence
This atomic change preserves structural equivalence. Hence there is no need to retest the changed program.

4.1.1.4 Reason
Programmers are expected to liberally comment their source code. Inadequate level of commenting may be pointed out by code reviewers, forcing the programmer to add more comments at a later point of time.

4.1.2 Delete comments

4.1.2.1 Description
This corresponds to deleting comments from a class. The assumption is that this change does not activate new code.
4.1.2.2 Examples

Before change:

```cpp
class Test {
  int a;
  // Print the instance variable for debug purposes
  public: void print() {
    cout << "Test::a = " << a;
  }
};
```

After change:

```cpp
class Test {
  int a;
  public: void print() {
    cout << "Test::a = " << a;
  }
};
```

4.1.2.3 Equivalence

This atomic change preserves structural equivalence. Hence there is no need to retest the changed program.

4.1.2.4 Reason

Trivial comments that repeat the actual code statement are rarely useful. Reviewers might recommend deleting such comments.

4.1.3 Introduce a new instance variable

4.1.3.1 Description

Introduce a new instance variable in the class.

4.1.3.2 Example

Before change:

```cpp
class AddInstVar {
  private: int a;
  public: AddInstVar() { a = 0; }
  public: void f() { a++; }
};
```
After change:
```cpp
class AddInstVar {
  private: int a;
  protected: int val;
  public: AddInstVar() { a = 0; }
  public: void f() { a++; }
};
```

4.1.3.3 Equivalence

This change may or may not preserve T-equivalence. If defining the instance variable alters the LFC of any of the existing methods, then T-equivalence is not preserved. Otherwise, T-equivalence is preserved. Both cases are illustrated below.

Case 1 (LFC Unaltered):

Before change:
```cpp
class AddInstVar {
  private: int a;
  public: AddInstVar() { a = 0; }
  public: void f() { a++; }
};
```

After change:
```cpp
class AddInstVar {
  private: int a;
  protected: int val;
  public: AddInstVar() { a = 0; }
  public: void f() { a++; }
};
```

In this case, the newly introduced instance variable is not used by any method, so the LFC of each of the methods remains the same after change. Therefore, the class after change is T-equivalent to the original one.
Case 2 (LFC altered):

**Before change:**

class Base {
    protected: int val;
    protected: Base() { val = 0; }
};
class Derived : public Base {
    private: int a;
    public: Derived() { a = 8; }
    public: void f() {
        a++;
        val--;
    }
};

**After change:**

class Base {
    protected: int val;
    protected: Base() { val = 0; }
};
class Derived : public Base {
    private: int a;
    protected: int val;
    public: Derived() { a = 8; }
    public: void f() {
        a++;
        val--;
    }
};

Clearly, the behavior of method $f()$ in Derived is changed since it binds to the new instance variable and not to the base element!

### 4.1.3.4 Reason for change

This change is typically not complete by itself. The class designer usually follows this change with other changes such as changing method body. Since instance variables of a class preserve object
state across time, the primary reason for defining a new instance variable in a class across versions is to model the enhanced state.

4.1.4 Delete an instance variable

4.1.4.1 Description
An instance variable of the class is deleted in the modified version.

4.1.4.2 Example
**Before change:**
```cpp
class DelInstVar {
  private: int a;
  protected: int val;
  public: void f() { a++; }  
};
```

**After change:**
```cpp
class DelInstVar {
  private: int a;
  public: void f() { a++; }  
};
```

4.1.4.3 Equivalence
This change may or may not preserve T-equivalence. If deleting the instance variable alters the LFC of any of the existing methods, then T-equivalence is not preserved. Otherwise, T-equivalence is preserved. Both cases are illustrated below.

Case 1 (LFC Unaltered):

**Before change:**
```cpp
class DelInstVar {
  private: int a;
  protected: int val;
  public: DelInstVar() { a = 0; }
  public: void f() { a++; }  
};
```
After change:
class DellInstVar {
    private: int a;
    public: DellInstVar() { a = 0; }
    public: void f() { a++; }
};

In this case, the deleted instance variable is not used by any method, so the LFC of each of the methods remains the same after change. Therefore, the class after change is T-equivalent to the original one.

Case 2 (LFC altered):

Before change:
class Base {
    protected: int val;
    protected: Base() { val = 0; }
};
class Derived : public Base {
    private: int a;
    protected: int val;
    public: Derived() { a = 9; val = 6; }
    public void f() {
        a++;
        val--;
    }
};

After change:
class Base {
    protected: int val;
    protected: Base() { val = 0; }
};
class Derived extends Base {
    private: int a;
    public: Derived() { a = 9; val = 6; }
}
public void f() {
    a++;
    val--;
}

Clearly, the behavior of method f() in Derived is changed since it binds to the base element.

4.1.4.4 Reason for change
This change is typically not complete by itself. The class designer usually follows this change with other changes such as changing method body.

4.1.5 Change the access level of a member

4.1.5.1 Description
Change the access level of an element of a class. C++ supports three levels of access control: private, protected, and public. Private elements are accessible only to methods of the class; public elements are accessible anywhere the class itself is visible; protected members are visible to methods of the class, methods of derived classes, and to friends of the class.

4.1.5.2 Example
Before change:

class A {
    public: int a;
    private: float f;
    public: void ff() { f += a; }
};

After change:

class A {
    protected: int a;
    private: float f;
    public: void ff() { f += a; }
};
4.1.5.3 Equivalence

This atomic change preserves T-equivalence. Hence the modified class need not be retested.

4.1.5.4 Reason for change

It has been observed that during initial code development, programmers are generally lax when it comes to choosing the most appropriate visibility qualifier. Many coding guidelines recommend, for example, making data members \textit{private}. A programmer who follows this rule automatically makes data members private, but may realize that some derived class requires to have access to this element at a later time, and changes the visibility to \textit{protected}. Similarly, most of the methods are qualified as public (to start with), but some are eventually changed to private if they are used for internal code factoring.

4.1.6 \textbf{Add class variable}

4.1.6.1 Description

Add a new class (static) variable to the class.

4.1.6.2 Example

\textbf{Before change:}

class AddClassVar {
    private: int a;
    public: AddClassVar() {
        a = 6;
    }
    public: void f() { a++; }
};

\textbf{After change:}

class AddClassVar {
    private: int a;
    protected: static int val;
    public: AddClassVar() { a = 6; }
}
public: void f() { a++; }
}

4.1.6.3 Equivalence

This change may or may not preserve T-equivalence. If defining the class variable alters the LFC of any of the existing methods, then T-equivalence is not preserved. Otherwise, T-equivalence is preserved. Both cases are illustrated below.

Case 1 (LFC Unaltered):

Before change:

class AddClassVar {
    private: int a;
    public: AddClassVar() { a = 6; }
    public: void f() { a++; }
};

After change:

class AddClassVar {
    private: int a;
    protected: static int val;
    public: AddClassVar() { a = 6; }
    public: void f() { a++; }
};

In this case, the newly introduced instance variable is not used by any method, so the LFC of each of the methods remains the same after change. Therefore, the class after change is T-equivalent to the original one.

Case 2 (LFC altered):

Before change:

class Base {
    protected: int val;
    protected: Base() { val = 9; }
};

Before change:

class AddClassVar {
    private: int a;
    protected: static int val;
    public: AddClassVar() { a = 6; }
    public: void f() { a++; }
};
class Derived : public Base {
    private: int a;
    public: Derived() { a = 6; }
    public: void f() {
        a++;
        val--;
    }
};

**After change:**

class Base {
    protected: int val;
    protected: Base() { val = 9; }
};
class Derived : public Base {
    private: int a;
    public: Derived() { a = 6; }
    protected: static int val;
    public void f() {
        a++;
        val--;
    }
};

Clearly, the behavior of method \( f() \) in Derived is changed since it binds to the new class variable and not to the base element!

4.1.6.4 **Reason for change**

A class variable implements state that is shared by all instances of a class, and hence is global to those instances. This change is appropriate when such a state sharing is needed.

4.1.7 **Rearrange class elements**

4.1.7.1 **Description**

Members of a class are physically rearranged within the class.
4.1.7.2 Example

**Before change:**

class A {
    private: int i;
    public: A() { i = 0; }
    int getValue() const { return i; }
};

**After change:**

class A {
    public: A() { i = 0; }
    int getValue() const { return i; }
    private: int i;
};

4.1.7.3 Equivalence

This change preserves S-equivalence. Retesting is not needed.

4.1.7.4 Reason for change

Coding guidelines adopted within the team may require ordering class elements in a specific way. For instance, one of the widely used conventions is to define public elements first and private elements last.

4.1.8 Define a new instance method

4.1.8.1 Description

An instance method is defined in the class.

4.1.8.2 Example

**Before change:**

class A {
    public: void ff() { }
};

**After change:**

class A {
    public: void ff() { }
}
void gg() {} 

4.1.8.3 Equivalence
This atomic change does not preserve T-equivalence. Retesting is needed.

4.1.8.4 Reason for change
Adding new methods in a class may occur as part of class functionality enhancement.

4.1.9 Delete an instance method

4.1.9.1 Description
An instance method is deleted from the class.

4.1.9.2 Example

**Before change:**
```cpp
class A {
public: void ff() {
    void gg() {

};
```

**After change:**
```cpp
class A {
public: void ff() {
);
```

4.1.9.3 Equivalence
This atomic change does not preserve T-equivalence. Retesting is needed.

4.1.9.4 Reason for change
Deleting one or more instance methods is a major code change and may occur as part of code restructuring.

4.1.10 Define a new static method

4.1.10.1 Description
A new static method is introduced in the class.
4.1.10.2 Example

Before change:

class A {
public: void ff() {
}
};

After change:

class A {
public: void ff() {
static void gg() {
}
};

4.1.10.3 Equivalence
This atomic change does not preserve T-equivalence. Retest is required.

4.1.10.4 Reason for change
Adding new methods in a class may occur as part of class functionality enhancement.

4.1.11 Delete a static method

4.1.11.1 Description
An existing static method is deleted from the class.

4.1.11.2 Example

Before change:

class A {
public: void ff() {
static void gg() {
}
};

After change:

class A {
public: void ff() {
}
};

4.1.11.3 Equivalence
This atomic change does not preserve T-equivalence. Retest is required.
4.1.11.4 **Reason for change**
Deleting one or more static methods is a major code change and may occur as part of code restructuring.

4.1.12 **Make a function inline**

4.1.12.1 **Description**
A function may be qualified inline to suggest inline expansion at the place of call. This is applicable to member as well as nonmember functions.

4.1.12.2 **Example**

**Before change:**
```c
int max(int a, int b)
{
    return a > b ? a : b;
}
```

**After change:**
```c
inline int max(int a, int b)
{
    return a > b ? a : b;
}
```

4.1.12.3 **Equivalence**
This atomic change preserves T-equivalence. Hence retesting not required after modification.

4.1.12.4 **Reason for change**
Inlining a function is done to reduce function call overhead, thereby increasing speed of execution. Methods that are small and are executed in a tight loop, are frequently made inline.

4.1.13 **Make a function noninline**

4.1.13.1 **Description**
A function originally tagged as inline is made noninline.
4.1.13.2 Example

**Before change:**
inline int max(int a, int b)
{
    return a > b ? a : b;
}

**After change:**
int max(int a, int b)
{
    return a > b ? a : b;
}

4.1.13.3 Equivalence
This atomic change preserves T-equivalence. No need to retest after modification.

4.1.13.4 Reason for change
Inlining of methods is usually done to reduce method call overhead. This makes sense only for functions whose call overhead is comparable to time spent in executing method body. If the function is large, for example, inlining is not advantageous, and may actually add to space overhead since the compiler might ignore the inline recommendation. In such cases, programmers are advised to remove the inline keyword.

4.1.14 Change the body of a member function

4.1.14.1 Description
The body of a member function is modified.

4.1.14.2 Example

**Before change:**
class A {
private:
    int i;
public:
    A() { i = 0; }
}
After change:

class A {
private: int i;
public: A() { i = 40; }
   // ...
};

4.1.14.3 Equivalence
This atomic change, in general, does not preserve equivalence. Hence retesting is called for.

4.1.14.4 Reason for change
A method’s implementation may be changed for instance, to employ a different algorithm, for performance enhancement. Another reason is to fix a bug in the method logic.

4.1.15 Make a member function virtual

4.1.15.1 Description
A nonvirtual member function is made virtual.

4.1.15.2 Example

Before change:

class Base {
public:
   void gg() { ff(); }
   void ff() { /* ... */ }
};

After change:

class Base {
public:
   void gg() { ff(); }
   virtual void ff() { /* ... */ }
};
4.1.15.3 Equivalence

This atomic change, in general, may not preserve T-equivalence. However, in the example given above, T-equivalence is preserved. If the LFC remains the same, retesting is unnecessary, otherwise, retesting has to be done. The following example is one that requires retest.

Before change:

```cpp
class Base {
public:
    void gg() { ff(); }
    void ff() { /* ... */ }
};
class Derived : public Base {
public: void ff() { /* ... */ }
};
```

After change:

```cpp
class Base {
public:
    void gg() { ff(); }
    virtual void ff() { /* ... */ }
};
class Derived : public Base {
    public: void ff() { /* ... */ }
};
```

As can be seen, LFC(void_Base::ff_void) has changed after making Base::ff() virtual.

4.1.15.4 Reason for change

A function is made virtual so that it can be overridden in a derived class. In the absence of virtual qualifier, a derived method with identical signature merely hides and not overrides the base method.
4.1.16 Change the derivation mode

4.1.16.1 Description
C++ permits three types of derivations: private, protected and public. This atomic change is about changing from one type to another.

4.1.16.2 Example

Before change:
```cpp
class Base {
    // ...
};
class Derived : private Base {
    // ...
};
```

After change:
```cpp
class Base {
    // ...
};
class Derived : protected Base {
    // ...
};
```

4.1.16.3 Equivalence
This atomic change preserves T-equivalence. Hence retesting is not needed.

4.1.16.4 Reason for change
This is an uncommon change. Altering the derivation mode has control over access to base members inherited into the derived class. In the example given above, changing from private to protected derivation allows access to Base members within classes that may be derived in future from Derived.
4.1.17 Make the derivation from base virtual

4.1.17.1 Description
It possible to qualify a derivation relationship between two classes as virtual. This atomic change introduces the virtual qualifier into the derivation.

4.1.17.2 Example

**Before change:**
```cpp
class Base {
   // ...
};
class Derived : public Base {
   // ...
};
```

**After change:**
```cpp
class Base {
   // ...
};
class Derived : virtual public Base {
   // ...
};
```

4.1.17.3 Equivalence
Introducing the virtual qualifier does not alter the behavior of a class (provided it compiles correctly). Hence this preserves T-equivalence and retesting is not necessary.

4.1.17.4 Reason for change
The only reason to derive a class virtually from another is to ensure one physical copy of the base object in a derived class even though the base may be inherited multiply, via several paths. This is a special case of multiple inheritance and can cause ambiguities in the code. Deriving virtually solves this problem.
4.1.18 Make the derivation from base nonvirtual

4.1.18.1 Description
A virtually derived class is now made to derive nonvirtually from its base.

4.1.18.2 Example

Before change:
```cpp
class Base {
  // ...
};
class Derived : virtual public Base {
  // ...
};
```

After change:
```cpp
class Base {
  // ...
};
class Derived : public Base {
  // ...
};
```

4.1.18.3 Equivalence
This change preserves T-equivalence. Retesting is not needed.

4.1.18.4 Reason for change
This is a rare change. There are not many examples suggesting this atomic change.

4.1.19 Change a class into a struct and vice versa

4.1.19.1 Description
In C++, both class and struct can be used to define user types. The access qualifiers `private`, `public` and `protected` can be used inside as struct as they are used in a class, with identical behavior. This atomic change is about replacing the `class` keyword with `struct` keyword, or vice versa.
4.1.19.2 Example

Before change:

```cpp
class A {
private: int i;
protected: float f;
public: A() { i = 0; f = 1.23; }
   void ff() { /* ... */ }
};
```

After change:

```cpp
struct A {
private: int i;
protected: float f;
public: A() { i = 0; f = 1.23; }
   void ff() { /* ... */ }
};
```

4.1.19.3 Equivalence

This change preserves T-equivalence. No need to retest.

4.1.19.4 Reason for change

This is not a very common change. One convention generally followed by programmers is to use a `struct` to model an aggregate of data elements without behavior. A `class` is used to model an abstract data type.

4.1.20 Change a class into a union and vice versa

4.1.20.1 Description

A union type, in C++, facilitates multiple logical views of a single physical entity. It is possible to define member functions in a union. This atomic change concerns replacing `class` keyword with `union` and vice versa.

4.1.20.2 Example

Before change:

```cpp
class A {
private: int a; float f;
```
public: int getInt() { return a; }
float getFloat() { return f; }

After change:
union A {
private: int a; float f;
public: int getInt() { return a; }
float getFloat() { return f; }
};

4.1.20.3 Equivalence
This atomic change does not preserve equivalence. Retesting is needed in the modified version.

4.1.20.4 Reason for change
This is a very unusual change. No practical application of this change has been encountered.

4.1.21 Make a data member const

4.1.21.1 Description
A data member of a class is qualified as const.

4.1.21.2 Example

Before change:
class A {
    private: int value;
    public: A(int v) : value(v) { }
};

After change:
class A {
    private: const int value;
    public: A(int v) : value(v) { }
};

4.1.21.3 Equivalence
This atomic change preserves T-equivalence. Hence the modified class need not be retested.
4.1.21.4 **Reason for change**

If a data member of a class is not to be changed during the lifetime of an object, the member is made `const` to reflect this insight.

4.1.22 **Make a data member nonconst**

4.1.22.1 **Description**

A data member which is qualified `const` is made nonconst.

4.1.22.2 **Example**

**Before change:**
```cpp
class A {
    private: const int value;
    public: A(int v) : value(v) { }
};
```

**After change:**
```cpp
class A {
    private: int value;
    public: A(int v) : value(v) { }
};
```

4.1.22.3 **Equivalence**

This atomic change does not change the class behavior and preserves T-equivalence. Retesting is not needed.

4.1.22.4 **Reason for change**

It is not common to find this as the only change in the class as part of evolution. It is more likely that at least one member function in the evolved version will alter the data member.

4.1.23 **Make a data member volatile**

4.1.23.1 **Description**

A data member of the class is qualified `volatile`. 
4.1.23.2 Example

**Before change:**

```cpp
class A {
private: int value;
public: A(int val) : value(val) { }
};
```

**After change:**

```cpp
class A {
private: volatile int value;
public: A(int val) : value(val) { }
};
```

4.1.23.3 Equivalence

The `volatile` keyword has implementation-defined semantics relating to optimization. In general, this atomic change does not guarantee equivalence. Retesting is required.

4.1.23.4 Reason for change

Programmers tag a data member as `volatile` to prevent the compiler from aggressively optimizing read access to the variable, when the variable is updated in some other part of code, perhaps by a different thread.

4.1.24 Make a data member nonvolatile

4.1.24.1 Description

A `volatile` qualified data member of a class is made nonvolatile.

4.1.24.2 Example

**Before change:**

```cpp
class A {
private: volatile int value;
public: A(int val) : value(val) { }
};
```

**After change:**

```cpp
class A {
private: int value;
```
public: A(int val) : value(val) {};

4.1.24.3 Equivalence
The volatile keyword has implementation-defined semantics relating to optimization. In general, this atomic change does not guarantee equivalence. Retesting is required.

4.1.24.4 Reason for change
This atomic change is rare. Practical applications of this atomic change have not been encountered.

4.1.25 Declare a friend function or class

4.1.25.1 Description
A function or class is declared as a friend of the evolving class.

4.1.25.2 Example

Before change:
class A {
private: int i;
public: A() : i(0) {} 
  void setValue(int a) { i = a; }
  int getValue() const { return i; }
};

After change:
class A {
friend void ff(A & ar);
private: int i;
public: A() : i(0) {} 
  void setValue(int a) { i = a; }
  int getValue() const { return i; }
};

It is assumed that the nonmember (global) function ff() is defined in some module.
4.1.25.3 Equivalence
Merely declaring a function or class as a friend in the class does not affect the behavior of the modified class. This atomic change preserves T-equivalence and hence retesting is not needed.

4.1.25.4 Reason for change
This atomic change is usually followed by other atomic changes such as defining a new function or changing the body of a function to use private elements of the friendship-granting class. One common reason for declaring friendship is to bypass the normal encapsulation layer for performance reasons.

4.1.26 Delete a friend declaration

4.1.26.1 Description
A friend declaration (of a function or class) is deleted from the evolving class.

4.1.26.2 Example
Before change:
```cpp
class A {
friend void ff(A & ar);
private: int i;
public: A() : i(0) {}
    void setValue(int a) { i = a; }
    int getValue() const { return i; }
};
```

After change:
```cpp
class A {
private: int i;
public: A() : i(0) {}
    void setValue(int a) { i = a; }
    int getValue() const { return i; }
};
```

It is assumed that the nonmember (global) function `ff()` is defined in some module.
4.1.26.3 Equivalence
Merely deleting friend declaration does not affect the behavior of the modified class, provided the code builds after the change as per the requirements of atomic change. This preserves T-equivalence and hence retesting is not necessary.

4.1.26.4 Reason for change
The function or class that required the friendship of a class earlier, might be rewritten to use its public interface in the next version, thus obviating the need for friendship. This has the beneficial effect of reducing coupling between software elements.

4.1.27 Make a data member mutable

4.1.27.1 Description
A datamember of a class is qualified mutable.

4.1.27.2 Example
Before change:
class A {
    private: int count;
    public: A() : count(0) {}
    // ...
};

After change:
class A {
    private: mutable int count;
    public: A() : count(0) {}
    // ...
};

4.1.27.3 Equivalence
The mutable keyword does not alter the behavior of the class. Hence it preserves T-equivalence and retesting is not necessary.
4.1.27.4 **Reason for change**
The `mutable` keyword allows a `const` member function to modify an instance variable of the class selectively. A typical use is for caching the state of an object.

4.1.28 **Delete mutable property of a data member**

4.1.28.1 **Description**
The `mutable` qualifier applied to an instance variable is removed.

4.1.28.2 **Example**

**Before change:**
```cpp
class A {
  private: mutable int count;
  public: A() : count(0) {} // ...
};
```

**After change:**
```cpp
class A {
  private: int count;
  public: A() : count(0) {} // ...
};
```

4.1.28.3 **Equivalence**
Deleting the `mutable` qualifier does not affect class behavior in any way. Hence T-equivalence is preserved and retesting is not necessary.

4.1.28.4 **Reason for change**
This is not very common. However, this atomic change may be part of several changes introduced in the code as part of restructuring.

4.1.29 **Make a virtual function pure**

4.1.29.1 **Description**
A virtual function defined in the class is made `pure`. 
4.1.29.2 Example

**Before change:**
```cpp
class A {
private: int i;
public: A() { i = 1; }
    virtual void ff();
};
void A::ff() {
    // Body
}
```

**After change:**
```cpp
class A {
private: int i;
public: A() { i = 1; }
    virtual void ff() = 0;
};
void A::ff() {
    // Body
}
```

4.1.29.3 Equivalence

Making a virtual function pure does not have any behavior change assuming the code builds correctly. Preserves T-equivalence, so no retest is needed.

4.1.29.4 Reason for change

Defining a pure virtual function in a class makes the class abstract, that is, objects of that class cannot be instantiated. This is sometimes done to simulate an interface.

4.1.30 Delete pure specifier from a virtual function

4.1.30.1 Description

An existing pure virtual specification is removed from a method.
4.1.30.2 Example

**Before change:**
```cpp
class A {
private: int i;
public: A() { i = 1; }
  virtual void f£() = 0;
};
void A::££() {
  // Body
}
```

**After change:**
```cpp
class A {
private: int i;
public: A() { i = 1; }
  virtual void ff() = 0;
};
void A::ff() {
  // Body
}
```

4.1.30.3 Equivalence
This atomic change preserves T-equivalence. Retesting is not needed.

4.1.30.4 Reason for change
An abstract class may need to become concrete as part of program evolution. In such cases, virtual functions need to be **nonpure**.

4.1.31 Change a nontemplate function into a template function

4.1.31.1 Description
A nontemplate function is converted into a template function.

4.1.31.2 Example

**Before change:**
```cpp
int min(int a, int b) {
  return a < b ? a : b;
}
```
After change:

```cpp
template <typename T>
T min(T a, T b) {
    return a < b ? a : b;
}
```

4.1.31.3 Equivalence
This atomic change does not preserve T-equivalence. Retesting is necessary.

4.1.31.4 Reason for change
Type-independent utility functions such as the one above are usually turned into templates due to their wider applicability.

4.1.32 Change a template function into a nontemplate function

4.1.32.1 Description
A template function is converted into a nontemplate, type-specific implementation.

4.1.32.2 Example

Before change:

```cpp
template <typename T>
T min(T a, T b) {
    return a < b ? a : b;
}
```

After change:

```cpp
int min(int a, int b) {
    return a < b ? a : b;
}
```

4.1.32.3 Equivalence
This atomic change does not preserve T-equivalence. Retesting is therefore necessary.

4.1.32.4 Reason for change
This atomic change is rare. It is not obvious where it is useful.
4.1.33 Change a nontemplate class into a template

4.1.33.1 Description
A nontemplate class is modified into a template class.

4.1.33.2 Example

Before change:
```cpp
class Stack {
public:
    Stack(int sz);
    void push(int val);
    int pop();
    // ...
};
```

After change:
```cpp
template <typename T>
class Stack {
public:
    Stack(int sz);
    void push(T val);
    T pop();
    // ...
};
```

4.1.33.3 Equivalence
This does not preserve T-equivalence. Retesting is necessary after change.

4.1.33.4 Reason for change
Widely applicable classes such as Stack, List, Vector, and List are good candidates for templating.

4.1.34 Change a template class into a nontemplate class

4.1.34.1 Description
An existing template class is made type-specific.
4.1.34.2 Example

**Before change:**

```cpp
template <typename T>
class Stack {
public: Stack(int sz);
    void push(T val);
    T pop();
    // ...
};
```

**After change:**

```cpp
class Stack {
public: Stack(int sz);
    void push(int val);
    int pop();
    // ...
};
```

4.1.34.3 Equivalence

Atomic change does not preserve T-equivalence. Retesting is called for.

4.1.34.4 Reason for change

This change is rare. Practical applicability is not clear.

4.1.35 Define a member template

4.1.35.1 Description

Define a member template function (or class) in a class.

4.1.35.2 Example

**Before change:**

```cpp
class A {
private: int i;
public: A() : i(0) {} // ...
};
```
After change:

class A {
private: int i;
public: A() : i(0) {}

template <class T> operator T() { /* ... */
  //...
};

4.1.35.3 Equivalence
This change does not preserve T-equivalence. Retesting is needed after the change.

4.1.35.4 Reason for change
Member templates are a very recent addition to C++. Not much experience is available regarding their use.

4.1.36 Remove a member template

4.1.36.1 Description
An existing member template is deleted.

4.1.36.2 Example

Before change:

class A {
private: int i;
public: A() : i(0) {}
  //...
};

After change:

class A {
private: int i;
public: A() : i(0) {}

template <class T> operator T() { /* ... */
  //...
};

4.1.36.3 Equivalence
This change does not preserve T-equivalence. Retesting is needed after the change.
4.1.36.4  Reason for change
Member templates are a very recent addition to C++. Not much experience is available regarding their use.

4.1.37  Make a member function const

4.1.37.1  Description
An existing method of a class is made const.

4.1.37.2  Example

Before change:
```cpp
class A {
private: int i;
public: A(int v) : i(v) {} 
  int getValue() { return i; }
};
```

After change:
```cpp
class A {
private: int i;
public: A(int v) : i(v) {} 
  int getValue() const { return i; }
};
```

4.1.37.3  Equivalence
This change preserves T-equivalence. Retesting is not needed.

4.1.37.4  Reason for change
Methods that do not alter the state of an object are usually made const as per a widely followed recommendation.

4.1.38  Make a member function nonconst

4.1.38.1  Description
A const member function is made nonconst.

4.1.38.2  Example

Before change:
```cpp
class A {
private: int i;
```
public: A(int v) : i(v) {}
    int getValue() const { return i; }
};

After change:

class A {
    private: int i;
    public: A(int v) : i(v) {}
        int getValue() { return i; }
    };

4.1.38.3 Equivalence
This preserves T-equivalence. No need to retest.

4.1.38.4 Reason for change
This change is usually uncommon by itself. It is typically followed by changes to the method body.

4.1.39 Make a member function volatile

4.1.39.1 Description
An existing member function is qualified as volatile.

4.1.39.2 Example

Before change:

class A {
    private: const int loc;
    public: A(): loc(0) {}
        void ff() { /* ... */ }
    };

After change:

class A {
    private: const int loc;
    public: A(): loc(0) {}
        void ff() volatile { /* ... */ }
    };

4.1.39.3 Equivalence
This atomic change may not preserve T-equivalence. Hence retesting is called for.
4.1.39.4 **Reason for change**
This has optimization-related semantics. Typically useful in writing device-drivers and other low-level software.

4.1.40 **Make a member function nonvolatile**

4.1.40.1 **Description**
An existing `volatile` member function is made nonvolatile.

4.1.40.2 **Example**

**Before change:**
```cpp
class A {
  private: const int loc;
  public: A(): loc(0) {} 
    void ff() volatile { /* ... */ }
};
```

**After change:**
```cpp
class A {
  private: const int loc;
  public: A(): loc(0), {} 
    void ff() { /* ... */ }
};
```

4.1.40.3 **Equivalence**
This change may not preserve T-equivalence. Retesting is needed.

4.1.40.4 **Reason for change**
This change is rare. Typically used in low-level software.

4.1.41 **Rename a method**

4.1.41.1 **Description**
Change the name of a method to something else. This is a valid atomic change only under one the following conditions:

- the method is not used anywhere
- renaming the method causes existing usage points to bind to another method with same name
4.1.41.2 Example

**Before change:**

class Base {
    public: void ff() { /* ... */ }
};
class Derived : public Base {
    public: void ff() { /* ... */ }
    public: void gg() {
        ff(); // Binds to Derived::ff()
    }
};

**After change:**

class Base {
    public: void ff() { /* ... */ }
};
class Derived : public Base {
    public: void hhh() { /* ... */ }
    public: void gg() {
        ff(); // Binds to Base::ff()
    }
};

4.1.41.3 Equivalence

This change may not preserve equivalence. In the above example, the behavior of Derived::gg() has changed due to the renaming of Derived::ff() to Derived::hhh(). The only time this atomic change preserves equivalence is when the method is not used anywhere.

4.1.41.4 Reason for change

Not a very common change. However, sometimes, as part of code restructuring, this might occur.

4.1.42 Change order of elements in throws list

**Description**

Order of exception types declared in the method signature is changed.
Example

**Before change:**

```cpp
class A {
    private: int i;
    public: A() : i(0) {} 
    void ff() throw (A, int, E) { /* ... */}
};
```

**After change:**

```cpp
class A {
    private: int i;
    public: A() : i(0) {
    void ff() throw (int, A, E) { /* ... */}
};
```

**Equivalence**

This change preserves T-equivalence. No need to retest.

**Reason for change**

This may occur when some coding guideline that requires listing the exceptions with primitive types first, is followed.

**Change method throws list**

**Description**

The list of exceptions shown as part of method signature is changed.

**Example**

**Before change:**

```cpp
class A {
    private: int i;
    public: A() : i(0) {
    void ff() throw (A, int, E) { /* ... */}
};
```

**After change:**

```cpp
class A {
    private: int i;
```
public: A() : i(0) {} 
void ff() throw (A, E) { /* ... */ }

4.1.43.3 Equivalence
This atomic change may alter the behavior of the method. T-equivalence is not guaranteed. Requires retesting.

4.1.43.4 Reason for change
Typically, this change occurs when the method body is changed. It is rare to find this happening by itself.

4.1.44 Change the type of a data member

4.1.44.1 Description
The type of an instance or class variable is changed.

4.1.44.2 Example

Before change:
```
class A {
private: long size;
public: A() : size(0) {} 
    void setSize(long l) { size = l; }
};
```

After change:
```
class A {
private: short size;
public: A() : size(0) {} 
    void setSize(long l) { size = l; }
};
```

4.1.44.3 Equivalence
In general, this atomic change may not preserve T-equivalence. Retesting is necessary.

4.1.44.4 Reason for change
Data type changes may occur as part of code fine-tuning.
4.1.45 Introduce a new type into a class

4.1.45.1 Description
C++ permits definition of other types within a class. These may be classes, structs, unions, typedefs, and enums. This atomic change is concerned with such a nested type definition.

4.1.45.2 Example

Before change:
```cpp
struct E {
    double d;
    void ff() { /* ... */ }
};
class A {
    private: int val;
    E el;
    public: A() : val(0) {
    void gg() { el.ff(); }
};
}
```

After change:
```cpp
struct E {
    double d;
    void ff() { /* ... */ }
};
class A {
    class E {
        public: void ff() { /* ... */ }
        void hh() { /* ... */ }
    }
    private: int val;
    E el;
    public: A() : val(0) {
    void gg() { el.ff(); }
};
```
4.1.45.3 Equivalence
This atomic change may not preserve T-equivalence. Hence retesting is required.

4.1.45.4 Reason for change
This is not a very common change.

4.1.46 Delete a type from a class

4.1.46.1 Description
Definition of a nested type such as typedef, class, struct, union, and enum is deleted from a class.

4.1.46.2 Example

Before change:
```c++
struct E {
    double d;
    void ff() { /* ... */ }
};

class A {
    class E {
        public: void ff() { /* ... */ }
        void hh() { /* ... */ }
    };
    private: int val; E el;
    public: A() : val(0) {} 
        void gg() { el.ff(); } 
    };
```

After change:
```c++
struct E {
    double d;
    void ff() { /* ... */ }
};

class A {
    private: int val;
    E el;
    public: A() : val(0) {}
```
void gg() { el.ff(); }

4.1.46.3 Equivalence
This may not preserve T-equivalence. Retesting is required.

4.1.46.4 Reason for change
This is not a common atomic change.

4.1.47 Rename method argument

4.1.47.1 Description
Change the name of a formal parameter of method. This is a valid atomic change only under any of the following conditions:

- the parameter is not used within the method
- all occurrences of the old name within the method are correctly changed to the new name
- renaming the argument without changing the method body causes all occurrences of the old name within the method to bind to another object

4.1.47.2 Example
Case 1: Argument not used.

**Before change:**
```cpp
class Sample {
    private: int i;
    public: String ff(String s, int a) {
        s = s + "hello";
        return s;
    }
};
```

**After change:**
```cpp
class Sample {
    private: int i;
```
public: String ff(String s, int b) {
    s = s + "hello";
    return s;
}

Case 2: Argument and method body are changed.

**Before change:**

class Sample {
    private: int i;
    public: String ff(String s, int a) {
        s = s + a;
        return s;
    }
};

**After change:**

class Sample {
    private: int i;
    public: String ff(String s, int b) {
        s = s + b;
        return s;
    }
};

Case 3: Renaming changes symbol bindings

**Before change:**

class Sample {
    private: int i;
    public: String ff(String s, int i) {
        s = s + i;
        return s;
    }
};

**After change:**

class Sample {
    private: int i;
}
public: String $(String s, int b)$ {
    $s = s + i$;
    return $s$;
}

4.1.47.3 Equivalence
As can be seen from the above three cases, first two cases preserve T-equivalence, whereas the last does not.

4.1.47.4 Reason for change
This is a rare atomic change.

4.1.48 Change method signature

4.1.48.1 Description
Change the signature of a method. Return type is included as part of this atomic change.

4.1.48.2 Example

Before change:
```java
class Sample {
    public: void $f(int a)$ { /* ... */ }
};
```

After change:
```java
class Sample {
    public: void $f(long a)$ { /* ... */ }
};
```

4.1.48.3 Equivalence
This atomic change does not preserve equivalence. Retesting is needed.

4.1.48.4 Reason for change
Argument type promotions as shown in the example, and other signature changes, are sometimes done as part of code review.
4.1.49 Add a new base class

4.1.49.1 Description
Derive the class from another class. There are two possibilities:

- the class does not have any explicit base class
- the class is already derived from another base class

4.1.49.2 Example
Case 1: There is no explicit base class

Before change:
```cpp
class Sample {
    public: void ff() {
        // ...
    }
};
```

After change:
```cpp
class Base {
    // ...
};
class Sample: public Base {
    public: void ff() {
        // ...
    }
};
```

Case 2: There is already a base class

Before change:
```cpp
class Base {
    protected: void gg() {
        // ...
    }
};
```
class Sample : public Base {
    public: void ff() {
        gg();
    }
};

After change:
class Base {
    protected: void gg() {
        // ...
    }
};
class NewBase : public Base {
    protected: void gg() {
        // ...
    }
};
class Sample : public NewBase {
    public: void ff() {
        gg();
    }
};

4.1.49.3 Equivalence
This change may or may not preserve T-equivalence. In the first case above, the behavior of Sample does not change after adding a base. However, in the second case, the behavior has changed. In general, therefore, retesting is necessary.

4.1.49.4 Reason for change
This atomic change is likely when the designer performs code factoring resulting in changes in the class hierarchy. The primary reason to add a base class is to inherit behavior provided by an existing class.
4.1.50  **Delete a base class**

4.1.50.1  **Description**

A base class of the class under consideration is removed from derivation list. For this to be an atomic change, no element of the base class must be used in the context of the derived class.

4.1.50.2  **Example**

**Before change:**

```cpp
class Base {
    // ...
};
class Derived : public Base {
    // ...
};
```

**After change:**

```cpp
class Base {
    // ...
};
class Derived {
    // ...
};
```

4.1.50.3  **Equivalence**

Assuming the code builds after this change, T-equivalence is preserved. No need to retest.

4.1.50.4  **Reason for change**

This atomic change is likely when the designer performs code factoring resulting in changes in the class hierarchy.

4.1.51  **Rename an instance variable**

4.1.51.1  **Description**

Change the name of an instance variable. For this to be a feasible atomic change, any of the following conditions must apply:

- the instance variable is not used anywhere
• renaming the variable causes current symbol bindings to change

4.1.51.2 Example
Case 1: The instance variable is unused

**Before change:**

```cpp
class A {
    private: int i;
    private: double d;
    public: void f() { d += 3.4; }
};
```

**After change:**

```cpp
class A {
    private: int ii;
    private: double d;
    public: void f() { d += 3.4; }
};
```

Case 2: Symbol bindings change due to renaming

**Before change:**

```cpp
class Base {
    protected: int i;
    // ...
};
class Derived : public Base {
    private: int i;
    private: double d;
    public: void f() {
        d += 3.4;
        i++;
    }
};
```

**After change:**

```cpp
class Base {
    protected: int i;
```
class Derived : public Base {
    private: int ii;
    private: double d;
    public: void f() {
        d += 3.4;
        i++;
    }
};

4.1.51.3 Equivalence
This atomic change preserves equivalence under Case 1, but not under Case 2 as shown above. Hence retesting may be required in these cases because it fails to preserve T-equivalence.

4.1.51.4 Reason for change
Renaming an instance variable can be required when the developer follows coding guidelines.

4.1.52 Rename a class variable

4.1.52.1 Description
Change the name of a class variable. For this to be a feasible atomic change, any of the following conditions must apply:

   • the class variable is not used anywhere
   • renaming the variable causes current symbol bindings to change

4.1.52.2 Example
Case 1: The class variable is unused

Before change:
class A {
    private: static int i;
    private: double d;
    public: void f() { d += 3.4; }
};
After change:
class A {
    private: static int ii;
    private: double d;
    public: void f() { d += 3.4; }
};

Case 2: Symbol bindings change due to renaming

Before change:
class Base {
    protected: static int i;
    // ...
};
class Derived : public Base {
    private: static int i;
    private: double d;
    public: void f() {
        d += 3.4;
        i++;
    }
};

After change:
class Base {
    protected: static int i;
    // ...
};
class Derived : public Base {
    private: static int ii;
    private: double d;
    public: void f() {
        d += 3.4;
        i++;
    }
};
4.1.52.3 Equivalence
This atomic change preserves equivalence under Case 1, but not under Case 2 as shown above. Hence retesting may be required in these cases because it fails to preserve T-equivalence.

4.1.52.4 Reason for change
Renaming a class variable can be required when the developer follows coding guidelines.

4.1.53 Make a data member static

4.1.53.1 Description
An instance variable is changed into a class variable by using the qualifier static.

4.1.53.2 Example

**Before change:**
```cpp
class Sample {
    private: int count;
    public: Sample() { count = 0; } // Other methods
};
```

**After change:**
```cpp
class Sample {
    private: static int count;
    public: Sample() { count = 0; } // Other methods
};
int Sample::count = 0;
```

4.1.53.3 Equivalence
This change does not preserve equivalence. Retesting is needed.

4.1.53.4 Reason for change
This is a major change that can come about in the process of fixing existing bugs.
4.1.54 Make a data member nonstatic

4.1.54.1 Description
A static data member is made nonstatic.

4.1.54.2 Example

Before change:
class Sample {
    private: static int count;
    public: Sample() { count = 0; } /* Other methods */
    int Sample::count = 0;
};

After change:
class Sample {
    private: int count;
    public: Sample() { count = 0; } /* Other methods */
};

4.1.54.3 Equivalence
This change does not preserve equivalence. Retesting is needed.

4.1.54.4 Reason for change
This is a major change that can come about in the process of fixing existing bugs.

4.1.55 Make a member function static

4.1.55.1 Description
A nonstatic method is made static. This is an atomic change only when the method does not access any instance variable.

4.1.55.2 Example

Before change:
class A {
    private: static int count;
public: void setCount(int c) {
    count = c;
}
// other methods
);

After change:
class A {
private: static int count;
public: static void setCount(int c) {
    count = c;
}
// other methods
};

4.1.55.3 Equivalence
This atomic change preserves T-equivalence. Retesting is not needed.

4.1.55.4 Reason for change
This may come about as part of code review. Methods that do not access any instance variable of the class are often made static.

4.1.56 Make a member function nonstatic

4.1.56.1 Description
A static member function is made nonstatic.

4.1.56.2 Example
Before change:
class A {
private: static int count;
public: static void setCount(int c) {
    count = c;
}
// other methods
};
After change:

class A {
private: static int count;
public: void setCount(int c) { count = c; }
    // other methods
};

4.1.56.3 Equivalence
This change by itself preserves T-equivalence. Retesting is unnecessary.

4.1.56.4 Reason for change
This change is unlikely to happen as the only change in evolution.
Usually, the method is also made virtual or its body is changed.

4.1.57 Introduce access declaration

4.1.57.1 Description
Declare elements of the base class so as to restore their access level.

4.1.57.2 Example

Before change:

class Base {
    public: void ff() { /* ... */ }
    // Others
};
class Derived : private Base {
    public: void gg() { /* ... */ }
    // Others
};

After change:

class Base {
    public: void ff() { /* ... */ }
    // Others
};
class Derived : private Base {
    public: void gg() { /* ... */ }
}
4.1.57.3 Equivalence
This change preserves T-equivalence. Retesting is not needed.

4.1.57.4 Reason for change
This is primarily useful when private or protected derivation is employed. Not a very common change.

4.1.58 Delete access declaration

4.1.58.1 Description
An existing access declaration is removed.

4.1.58.2 Example

Before change:
```cpp
class Base {
    public: void ff() { /* ... */ }
    // Others
};
class Derived : private Base {
    public: void gg() { /* ... */ }
    Base::ff;
    // Others
};
```

After change:
```cpp
class Base {
    public: void ff() { /* ... */ }
    // Others
};
class Derived : private Base {
    public: void gg() { /* ... */ }
    // Others
};
```

4.1.58.3 Equivalence
This preserves T-equivalence. Retesting is not needed.
4.1.58.4 Reason for change
This is a rare atomic change. It may only occur in the context of other changes.

4.1.59 Define a namespace

4.1.59.1 Description
A namespace scope is defined to enclose one or more program elements. For this to be an atomic change, all existing uses of the elements brought into the namespace must be in the same namespace.

4.1.59.2 Example
Before change:
```cpp
class A {  
    // Members
};
```

After change:
```cpp
namespace MMS {  
    class A {  
        // Members
    };  
};
```

4.1.59.3 Equivalence
Assuming a proper program build, this change preserves T-equivalence. Hence there is no need to retest.

4.1.59.4 Reason for change
A large system is typically made up of several namespaces, each comprising a set of cohesive elements. As part of program evolution, this code reorganization is likely.
4.1.60  Remove a namespace

4.1.60.1  Description
An existing namespace scope is removed. For this to be an atomic change, all existing uses of the elements in the namespace must be in the same namespace.

4.1.60.2  Example

**Before change:**
```cpp
namespace MMS {
    class A {
        // Members
    };
};
```

**After change:**
```cpp
class A {
    // Members
};
```

4.1.60.3  Equivalence
Assuming a proper program build, this change preserves T-equivalence. Hence there is no need to retest.

4.1.60.4  Reason for change
This change is not common.

4.1.61  Change the namespace name

4.1.61.1  Description
An existing named namespace is given a different name.

4.1.61.2  Example

**Before change:**
```cpp
namespace MMS {
    class A {
        // Members
    };
};
```
After change:

```cpp
namespace MMS_UTILS {
    class A {
        // Members
    }
}
```

4.1.61.3 Equivalence
Assuming proper code build after this change, T-equivalence is preserved. Retest is not required.

4.1.61.4 Reason for change
This might occur as part of code restructuring.

4.2 SUMMARY

This chapter has described the atomic changes corresponding to C++. The next chapter documents the atomic changes possible in Java.