This document describes a short-term attempt to allow users of a Java class library to augment that library in a manner orthogonal to typical forms of subclassing. I will outline several reasons why all attempts to extend the functionality of the library with normal subclassing cannot succeed. To begin with, I will motivate this problem by showing a typical (and impractical) work-around for this problem. I will then describe several reasons why subclassing does not help the problem. Finally, I propose a solution.

0.0.1 Ideal Scenario

Ideally, the programmer would simply add a virtual method to the root of the class hierarchy. Suppose we would like to add a method named `ideal` to the `TreeNode`, `PlusNode`, and `ExprNode` classes to compute some metric. The body of each method is inserted into the code for each class:

```java
class TreeNode {
    // rest of TreeNode’s code
    // insert new method
    void ideal () { ... }
}

class ExprNode extends TreeNode {
    // rest of ExprNode’s code
    // insert new method
    void ideal () { ... }
}

class PlusNode extends ExprNode {
    // rest of PlusNode’s code
    // insert new method
    void ideal () { ... }
}
```

This example represents what a programmer would do given access and permission to modify the library’s source code. However, the programmer does not always have this permission. This example motivates the proposed solution here, which is now described in more detail.
0.0.2 Problem Motivation

The problem is that since the augmenter cannot directly modify the library, he cannot create and use virtual functions of the library's class hierarchy. He is left writing a code such as:

```java
class X {
    void x(TreeNode n)
    {
        if (n instanceof PlusNode)
        {
            // body for PlusNode
        }
        else if (n instanceof ExprNode)
        {
            // body for ExprNode
        }
    }
}
```

Code such as this must be written because the programmer does not have the ability (for whatever reason—security, lack of source code, etc) to modify the classes inside the library. Since he cannot modify these classes, he is forced to write code such as the above sample instead of virtual functions. A virtual function is a very convenient mechanism for implementing a recursive descent analysis of a tree data structure, so it will clearly benefit users of my JavaTime library.

This code is not only inconvenient to write, it goes against one's standard notion of how to program in the object oriented style. Further, it is difficult to simulate the superclass method accessors (e.g. `super.x` in the above example) and as a result, code is likely to be duplicated. The ability to call a method as the superclass is difficult to simulate because it requires the programmer to keep track of the effective-class of the object for each activation. There is no built-in mechanism for this in Java. It can be simulated by adding an additional argument which is the calling depth and replacing the `instanceof` operator with methods `derivedFromDistanceGreaterThan` and `derivedFromDistance`. I might rewrite the above as:

```java
class X {
    void x (TreeNode n)
    {
        x2 (n, 0);
    }

    void x2 (TreeNode n, int level)
    {
        if (derivedFromDistanceGreaterThan (n, PlusNode, level))
```
In the above example, I have to test whether the targetted class (to dispatch on) is the closest available if-branch to the current level. It also requires that the if clauses are ordered such that no super class appears before a derived class, or else the super logic will skip a method. This is extremely difficult to write correctly, even if you do want to implement the \texttt{derivedFromDistance} and \texttt{derivedFromDistanceGreaterThan} functions (they can’t be written as above, since the type names are not variables...).

\subsection*{0.0.3 Failed Solutions}

\subsubsection*{0.0.3.1 Interior Nodes}

Another reason standard sub-classing does cannot solve this problem is that sub-classing cannot be used on interior (non-leaf) nodes of the class hierarchy, since the ancestry relation is not preserved. To demonstrate this, the classes \texttt{Super} and \texttt{Derived} obey the relation:

\begin{verbatim}
Super > Derived
\end{verbatim}

But sub-classing each does not preserve this partial ordering:

\begin{verbatim}
subclass (Super) <> subclass (Derived)
\end{verbatim}

So, sub-classing could only let us add functionality to the leaf nodes, and is therefore already not going to prove helpful.
0.0.3.2 Constructors

Since the AST data structure is constructed by the library (which knows nothing of the subclasses you have created (to add functionality), the constructed data structure will still not be constructed of your subclasses without some sort of virtual constructor or factory scheme. A virtual constructor is the name given to a scheme where some form of dispatch is used to construct an object of the expected type. It is also known as the factory pattern. No language lets you declare a virtual constructor because it doesn’t make much sense, you can’t dispatch on the type of an object until you have an object (after construction). Therefore, a virtual constructor refers to a scheme such as the following (from memory, this is approximately an example from some Stroustrup book I once read):

```java
abstract class Shape {
    public static final int CIRCLE_SHAPE = 1;
    public static final int TRIANGLE_SHAPE = 2;
    public static final int SQUARE_SHAPE = 3;

    // ... 
    // Shape constructor, various shape related abstract or otherwise 
    // methods 
    // ...

    public static constructShape (int type) // This is the "virtual" 
        // constructor 
    {
        switch (type)
        {
            case CIRCLE_SHAPE:
                return new CircleShape ();
            case TRIANGLE_SHAPE:
                return new TriangleShape ();
            case SQUARE_SHAPE:
                return new SquareShape ();
        }
    }
}
```

So I claim this would be a solution to the constructor problem by allowing the client to register a virtual constructor for each class (leaf class – because of the interior node problem) and the library could use those to construct the data structure.
0.0.3.3 Multiple Clients

Even a virtual constructor mechanism only solves the problem for a single library client, because two such solutions do not compose.

0.0.4 Proposed Solution

This solution attempts to work around the problems above. Namely, that programmers cannot add virtual methods to class hierarchies for which they do not have permission or access to the source code.

As with any modification to the language, there are tradeoffs in the new description language, but I would like a program written for this system to remain at the very least syntactically valid. I propose something like the following:

class MyAugmentor extends Augmentor {
    private void super_m (TreeNode n) { } // body ignored

    void m (TreeNode n) {
        // body for TreeNode
    }

    void m (ExprNode n) {
        // body for ExprNode

        super_m (n);
    }
}

In the above program, the super_m method will act similar to super.m and declares the root of the class hierarchy that the set of m methods will operate on.

The idea is that if each of the above methods matching

\[
T \text{ METHOD } (T_0 \; P_0, \; T_1 \; P_1, \ldots \; T_N \; P_N)
\]

were added to class T0 (the class on which to dispatch) as:
T METHOD (T1 P1, ..., TN PN)

(I’ve removed the first argument) and then replaced each super METHOD call with super . METHOD (and similarly removed the first argument to super METHOD since it is implicitly THIS), then you would simply be adding each method as a true virtual function. For example, the following method body inside an augmentor:

```java
void x (TreeNode n, int y) { }
```

will effectively be added to the TreeNode class as:

```java
void x (int y) { }
```

In the ideal scenario, I might call a method like so:

```java
TreeNode n = EXPR;
n.METHOD (A1, ... AN);
```

Using my new system, I can instead write:

```java
TreeNode n = EXPR;
MyAugmentor a = new MyAugmentor ();
a.METHOD (n, A1, ... AN);
```

The set of m methods which appear as a set of ordinary (statically) overloaded methods will be translated into a single method with dynamic dispatch based on the statically overloaded type. The methods to be translated may contain additional arguments – these do not interfere with the construction. Methods with type signatures which differ in positions other than the first argument will be translated as truly overloaded methods – they are logically separate and require a separate super method. A function named iAmSuperOf serves as an annotation:

```java
abstract class Augmentor {
    final protected void iAmSuperOf (String methodName) { }
}
```

The iAmSuperOf method is used in the example below to indicate that the super_m method is to be treated as the super-method-call for the group of methods with the same signature named m.
class MyAugmentor extends Augmentor {
    private void super_m (TreeNode n) { iAmSuperOf ("m"); }
    // The above indicates that it is to be treated as a supercall inside
    // the method named "m".

    void m (TreeNode n)
    {
        // body for TreeNode
    }

    void m (ExprNode n)
    {
        // body for ExprNode

        super_m (n);
    }
}

One benefit of this construction is that temporary variables used during the computation can
be stored as fields of the containing class (in the example above, in MyAugmentor). Some people
feel that fields of the augmented class should somehow be added to the classes in question, but I
feel this is not appropriate and don’t think it makes sense since it is an unnecessary complication
and violation of Java’s syntax and semantics.