Design of a Joint Point Model for Aspect Oriented Programming in C++

Oscar Calvo  
Artinsoft LLC, Research,  
FL, United States, 33166  
Oscar.Calvo@Artinsoft.com

Francisco J. Torres-Rojas  
Instituto Tecnológico de Costa Rica, Escuela de Ingeniería en Computación,  
Cartago, Costa Rica  
torres@ic-itcr.ac.cr

Abstract

Current technologies for implementing Aspect Oriented Programming in the C++ language required changes to the language, non standard extensions or the use of external pre-processors. In this paper we explore a technique to provide C++ with Aspect Oriented Programming by means of a C++ Library using the same approach as the Spring Framework. We define a joint point model for C++ and a working prototype is described that allows an advice to be created around virtual method execution. The need for a runtime type information system for C++ is discussed and the Microsoft Type Library mechanism from the COM specification is discussed as an alternative.

Keywords: Software Engineering, Programming Languages, Aspect-Oriented Programming, C++, AOP.
1 Introduction

Aspect Oriented Programming (AOP) is an exciting and relatively new paradigm. One can say that AOP started with the birth of AspectJ by Gregor Kiczales and his team at Xerox PARC [1]. Since then, the paradigm has been in constant evolution. New tools have been created and support for multiple programming languages have been added, in particular for C++. Even though there are AOP extensions for C++, none of the current implementations of AOP for C++ works without modifying or extending the C++ language or by the introduction of a specialized Aspect language for C++.

It is the objective of this work to find a way to introduce AOP techniques into the C++ language without modifying the compiler and without designing nor building an Aspect language from scratch. In this paper, we propose a novel a way to bring AOP techniques into the C++ language, satisfying the constraints established above. At this point, we have a working simplified prototype that allows us to verify the soundness of our model.

Section 2 of this paper explains why AOP is important; in section 3, we discuss the bare minimum requirements that a programming language must have in order to support AOP techniques. Section 4 discusses the difference between an AOP implementation that uses only the base programming language and one that uses a specialized aspect language. The current state of AOP for the C++ language is briefly described in section 5. Section 6 explains our proposed joint point model (JPM) for C++ and the working prototype is discussed in section 7.

2 The case for AOP

What is AOP and why it is important? To answer both questions we have to start by understanding that there are software requirements that, because of its nature, are crosscutting across an entire software system. That is to say, they affect multiple components in every layer of the system [2].

Typical cross cutting concerns are logging, infrastructure, distribution, and transaction management. A simplistic way to resolve this problem is to “pollute” the source code of the existing components with calls to other components that abstract and implement the crosscutting concern. However, there is a fundamental flaw with this solution. Components in an object oriented design are meant to be reusable and to deal with one concern. The solution mentioned above breaks this fundamental law of the Object Oriented Design. There are also more pragmatic reasons why this solution does not work for all cases. For example, what if my system is built using components for which there is not source code available? Or the source code is available but suddenly one needs to include the source code in the building process of the entire system? What happens if the component is upgraded with new required functionality by a third party?

According to Jacobson [2] there are limitations to the object orientation paradigm; one of them is dealing with crosscutting concerns. AOP is established to overcome this limitation.

3 Language Requirements for the Aspect Orientation Paradigm

A programming language must satisfy certain key requirements in order to support a programming paradigm. For instance, in the case of the Object Oriented Programming (OOP), a real OOP language must support inheritance, modularity, polymorphism and encapsulation [3].

Now, for the Aspect Orientation paradigm, a language must satisfy two requirements:
   1. A technique for separation of crosscutting concerns from the components.
   2. A composition mechanism for crosscutting concerns.

By considering these requirements, one can begin to use the aspect oriented paradigm.

There are some particular concepts that AOP defines. The first one is the concept of join point, which basically is a point in the flow of a computer program. Conceptually a join point exists in the same place
where an assertion would exist in Hoare logic [4]. A *point cut* is a set of join points. A point cut is associated with a piece of logic. This piece of logic is called *advice* and it is executed when one or more of the join points in the point cut are executed. The advice describes a point cut and also invokes an *aspect*. The aspect implements a *crosscutting concern*. The composition of aspects in the final program is called *weaving* and it can happen at compilation time, run time or both [5].

**4 Using an Aspect Language**

The characteristics of a specific programming language define in some degree the way AOP is implemented in such language. Some implementations of AOP introduce a programming language on top of an existing one, e.g., *AspectJ* or *AspectC++* [6]. Most of the implementations of AOP for C++ are made this way. The use of an additional language on top of the traditional language gives to the AOP framework more power of expression; however in general the complexity is increased [7].

If the AOP is implemented only using the programming language there is less power of expression but the new paradigm is fairly easy to learn and understand. Our research has limited its scope to find solutions for the C++ programming language without the introduction of a new aspect language or modification of the compiler. Enough work and research has been done using aspect languages and compile time weaving at Arachne project [8], AspectC++ [9] and AspectC [10].

**5 AOP for C++ today**

There are multiple projects dealing with aspect orientation on the C++ programming language. For instance, we can mention:

- **Aspect C++**: introduces a new AOP language that compiles to C++ [9].
- **AspectC**: uses a simple aspect language on top of the standard C language [11].
- **XWeaver Project**: uses XML as language for representing aspects and produces modified C++ programs [12].
- **FeatureC++**: is a C++ language extension to support AOP [13].
- **AspeCt-oriented C**: is C compiler with extensions to support AOP [14].
- **Aspicere2**: is an aspect language based on prolog for the C programming language [15].
- **Arachne**: is a dynamic weaver for C programs, it also defines and implements an aspect language [8].

From the studied projects we were not able to find an AOP solution for C++ that did not modified the compiler or introduced a new aspect language.

**6 Proposed JPM**

So, if our solution for this problem is limited to the ones that require no modification whatsoever to the language then how will the solution look like? In order to answer that question we need to examine how the joint point model for C++ will look like. A JPM defines three things:

1. Join points
2. Point cuts
3. Advice.

6.1 Joint Points

Given the characteristics of the C++ language, a virtual method invocation is the only viable join point. But how can we intercept a virtual method invocation in C++? The answer lies on how the virtual method table (VTBL) is built for a C++ object.

By definition, a virtual method in C++ is a method whose actual implementation is only known at runtime when the object has been created. Let’s see an example:

```cpp
class CFooBar {
  int m;
  public:
    virtual void Foo() { ... }
    virtual void Bar() { ... }
};
```

Figure 1 shows the memory layout of a pointer to a CFooBar object instance. A square represents a word in memory. Notice that the first word in the layout of the CFooBar instance is the pointer to the virtual method table. This is true for every C++ object that has virtual methods.

It is clear that in order to create a join point at a virtual method invocation we need to modify somehow the virtual table method. There are two ways to modify the virtual table:

1. Change the pointer in each of the entries in the table. The problem with this approach is that the table is located in a memory segment marked as execution only, so this is read only memory. Trying to modifying the memory will generate an operating system trap.

2. Create a shadow table and change the pointer of the original table to point to the new table. This is a valid memory operation because the table pointer is always located in a data segment.

We implemented successfully approach number 2 in our prototype.

---

1 On 32 bit Windows Vista and XP platforms
6.2 POINT CUTS

Implementing point cuts in C++ without the help of an additional aspect language poses a challenging problem. The Spring Framework [16] easily solves this problem because it only deals with languages that provide rich type information at runtime. In Spring.NET [7] a point cut is defined by the programmer by overiding the method “Matches”. This new method receives meta type classes that are not available in C++. In order to provide more useful point cuts for C++ a rich runtime type system (RTS) must be used. Several RTS for C++ exist, in particular:

1. Qt Meta object system [17].
2. Standard C++ RTTI [18].
3. Microsoft Type Library (TLB) from the COM specification [19].

Let’s talk first about 1 and 2. Those RTS systems are available for almost any standard implementation of C++. However, most of the third party components in C++ are compiled without any kind of RTS. Also support for RTTI is very poor in some C++ compiler implementations. Thus, options 1 and 2 limit the scope of the solution for AOP in C++.

Option 3 limits the platform to Windows based operating systems. There are COM implementations on certain UNIX based platforms, however, at this time; we will not work on that direction. The biggest advantages provided by option 3 are:

1. There are many third party components that ships with TLB.
2. There are free software development tools with support for TLB.

For the time being, we will assume a RTS based on Microsoft TLB, however we plan to make our library easily extensible so that other RTS systems can be used. This is a high priority requirement.

6.3 ADVICE

Given our definition of the join point and that C++ provides support for structured exception handling we can define the following kinds of Advices:

1. **Around Advice**: Executes before and after the execution of a point cut. This kind of advice is showcased it in the prototype.
2. **Before Advice**: Executes before the invocation of a point cut.
3. **After Advice**: Executes after the invocation of a point cut.
4. **Throws Advice**: Executes after the invocation of a point cut if such throws an exception.

7 Prototype

For our prototype we implemented an *around advice* for adding logging in the invocation of the virtual method Execute from the command interface. The command interface looks like:

```cpp
class ICommand
{
public:
  virtual void Execute() = 0;
};
```

2 Some of the names are based on the Spring Framework
We have a component called MyCommand implemented by the concrete class CMyCommand. The source for the component looks something like:

```cpp
class CMyCommand: public ICommand
{
public:
    void Execute()
    {
        ...
    }
};
```

Please notice that we may have only a binary dependency to ICommand and CMyCommand. Let’s say now that we have a requirement which is to log to the console every time a command is executed. A clean way to implement this requirement is to create a class that handles logging into the console, so let’s define ConsoleLoggingAspect as the class that implements such requirement.

```cpp
class CConsoleLoggingAspect: public IMethodInterceptorAdvice
{
public:
    void Before()
    {
        ...
    }
    void After();
    {
        ...
    }
};
```

Now, the idea is that we don’t modify the component CMyCommand so we need to add an advice around the execution of the method “Execute”.

```cpp
AspectManager->AddAdvice(new CConsoleLoggingAspect());
```

The method AddAdvice adds an advice to a globaly defined container called AspectManager.

```cpp
CMyCommand* aCommand = AspectManager->Create<CMyCommand>();
```

We ask the container to create and new instance of MyCommand for us. During the creation the aspect is “injected” into the new instance. In order to do this we need to modify the virtual table method of the instance aCommand. The following declaration in C++ will give us the memory layout of the virtual method table:

```cpp
struct CObject
{
    Vtlb* pVtbl;
};
struct Vtlb
{
    void __stdcall * Func00)(CObject* pObject);
};
```
Notice that Vtlb is limited to a single function pointer, no return value, no additional parameters and one single calling convention: __stdcall. Here we have identified several challenging problems:

1. What if the object does not have virtual methods? We will need a way to get meta data about the object type. A RTS will give us that, but we would like a solution that does not require one for this particular problem.
2. What if the calling convention is different between entries in the table? We suspect this is a not very common scenario. We may limit our framework to a single kind of calling convention, if we choose to do that, then we will use the standard calling convention as the default.
3. How do we get access to the return value to the around and after advices?
4. How do we get access to the parameters to the advices?

Using this declaration and the reinterpret_cast operator its now possible to change the virtual method table of a C++ object instance.

```cpp
CObject* pObject = reinterpret_cast<CObject*>(_pObject);
Vtlb* vtlb = pObject->pVtbl;
pObject->pVtbl = &aspectVtlb;
```

The aspectVtlb is a custom built virtual method table designed to manage the aspects associated with a running instance. The Vtlb is built to point to a method called CAspectManager::Invoke.

We haven’t implemented the joint cut concept yet, mainly because a rich join cut system will require a RTS. In our prototype the advices are global and we only support the around advice.

```cpp
void __stdcall CAspectManager::Invoke(CObject* pObject)
{
    CAspectInstaller* pThis = CAspectInstaller::Instance();
    Vtlb* vtlb = pThis->objects[pObject];
    std::list<IMethodInterceptorAdvice*>::iterator pIt = pThis->aspects.begin();
    while ( pIt!=pThis->aspects.end() )
    {
        (*pIt)->Before();
        pIt++;
    }

    // Perform invocation of the target method
    vtlb->Func00(pObject);

    pIt = pThis->aspects.begin();
    while ( pIt!=pThis->aspects.end() )
    {
        (*pIt)->After();
        pIt++;
    }
}
```

Thus when something invokes ICommand::Execute, our around advice CConsoleLoggingAdvice will be invoked before and after. The logic that controls execution of aspects is implemented by CAspectManager::Invoke.

The complete source code of our prototype can be obtained at: [http://www.codeplex.com/Aspector](http://www.codeplex.com/Aspector).
8 Conclusions and future work

We have studied the current state of the art for AOP in C++ and we concluded that an AOP system for C++ not based on an aspect language is missing. Java, C# and VB.NET have a similar system with the Spring Framework [16].

In order to provide a programming language with AOP capabilities a JPM must be proved to exist. We have shown that given our constraints such JPM for C++ exists. Even though this model exists the implementation of such model poses challenging problems and requirements:

1. Modification and management of the virtual method tables.
2. A rich point cut model requires the introduction of a RTS for C++. We will use the Microsoft Type Library to base our implementation of the RTS.
3. The RTS system must be pluggable and easily extensible.
4. How can one detect that an object does not have a virtual method table?
5. What if the calling convention if different for different classes of objects? What if the calling convention if different between methods in the same class?
6. How can one get access to the return value to the around and after advices?
7. How can one get access to the parameters to the advices?
8. What’s the performance impact of using AOP in C++?

Much work is still left. Most of it will be based on resolving the problems mentioned above and others that we still don’t know about. We plan on using the results of this work to build a proof of concept and report the results back to the community.

References


http://www.aspectc.org/.

[10]. **Kiczales, Gregor and Coady, Yvonne.** AspectC. [Online]


[12]. AspectX Language Guide. XWeaver. [Online]


http://www.aspectc.net/acc_manual.pdf. v0.5.


[19]. **Microsoft Corp.** Type Libraries and the Object Description Language. *MSDN.* [Online]