AspectC++ – An AOP Extension for C++

Olaf Spinczyk, Daniel Lohmann, Matthias Urban

March 3, 2005

More and more software developers get in touch with aspect-oriented programming (AOP). By providing means to modularize the implementation of crosscutting concerns, it stands for more reusability, less coupling between modules, and better separation of concerns in general. Today, solid tool support for AOP is available, for instance, by JBoss (JBoss AOP), BEA (AspectWerkz), and IBM (AspectJ and the AJDT for Eclipse). However, all these products are based on the Java language. For C and C++ developers none of the key players offers AOP support, yet.

This article introduces AspectC++, which is an aspect-oriented language extension for C++. A compiler for AspectC++, which transforms the source code into ordinary C++ code, is developed as an open source project. The AspectC++ project began with a research prototype in 2001 that has gained maturity over the years. Today, the AspectC++ language and weaver has been successfully applied in a number of real-world projects in industry and academia and IDE integration into Eclipse and Microsoft VisualStudio.NET makes the first steps a child's play.

Our AspectC++ introduction will start with an example that can be considered the "Hello World" of AOP. It will illustrate the basic language elements like aspects, pointcuts, and advice, which some readers might already know from the AspectJ language. We will then quickly step beyond these AspectJ-like language elements by looking into an AspectC++ version of the well-known observer pattern and into "Generic Advice". This unique AspectC++ feature combines the power of aspects with generic and generative programming in C++.

Tracing - the "Hello World" of AOP

As introductory example for AOP with AspectC++ we will take a closer look at a very simple aspect. The aspect "Tracing" modularizes the implementation of output operations, which are used to trace the control flow of the program. Whenever the execution of a function starts, the following aspect prints its name:

```
#include <cstdio>
// Control flow tracing example
aspect Tracing {
  // print the function name before execution starts
  advice execution ("% ...::%(...)") : before () {
    std::printf ("in %s\n", JoinPoint::signature ());
  }
};
```

Even without fully understanding all the syntactical elements shown here, some big advantages of AOP should already be clear from the example. Without using this simple aspect, which is only a few lines long, one would have to augment *all* functions of the program with an additional `printf` statement to get the same result. In a large project, a style guide would have to document this as a requirement and all programmers would have to read and obey this global policy. As a result the AOP solution saves a lot of time, organizational efforts, and guarantees that no function will be forgotten. At the same time the code, which is affected by the aspect, is completely decoupled from the tracing code, i.e. the `printf`. Not even `<cstdio>` has to be included, because all this is done separately by the aspect.

Aspects, Advice and Pointcuts

The Tracing example shows most of the language elements that are responsible for these advantages. We will start with the "aspect", which is intended as a module for the implementation of a crosscutting concern. From the syntactical perspective an `aspect` in AspectC++ is very much like a class in C++. However, besides member functions and data elements, an aspect can additionally define "advice". After the `advice` keyword a "pointcut expression" defines *where* the advice should affect the program (the "join points"), while the part that follows the colon defines *how* the program should be affected at these points. This is a general rule for all kinds of advice in AspectC++.

Pointcut Expressions

The pointcut expression in the example (*where*) is `execution("% ...::%(...)")`. It means that the advice should affect the *execution* of all functions that match the expression `"% ...::%(...)"`. In "match expressions" the `%` and `...` characters are used as wildcards. A percent (`%`) matches any type (for example `"% *"` matches any pointer type) and also any sequence of characters in identifiers (for example `"xdr_%"` matches all classes, which have a name that starts with `xdr_`). An ellipsis (`...`) matches any sequence of types or namespaces (for example `"int foo(...)"` matches any global function which returns an `int` and is named `foo`). Eventually, the match expression `"% ...::%(...)"` matches any function in any class or namespace.

Match expressions represent sets of named program entities like functions or classes. Thus, match expressions are already primitive pointcut expressions which describe a set of

join points in the static program structure (“static join points”). However, in our example we want advice for an event in the dynamic control flow of the program, namely the execution of functions. Therefore, the “pointcut function” `execution()` is used. It yields all function execution join points for the functions given as its argument.

Advice for Dynamic Join Points

For “dynamic join points” AspectC++ supports three types of code advice called `before()`, `after()`, and `around()`. They all implement an additional piece of program behaviour. In our aspect `Trace` this behaviour is implemented by the `printf()` statement, which follows `before()`. Syntactically this looks like a function body and, indeed, we can understand the “advice body” as an anonymous member function of the aspect. Instead of `before()` we could also use `after()` advice (or both) in the example. In this case the advice body would be run *after* the execution of a function has been finished. An `around()` advice body is executed *instead of* the control flow, which would normally follow the dynamic join point.

Combined Pointcut Expressions

Pointcut expressions can be combined by using the set operators `&&` (intersection), `||` (union), and `!` (inversion). For instance, the expression `"% foo(int, ...)" || "int bar(...)"` matches any global function named `foo` which takes an `int` as first parameter and any global function named `bar` which returns an `int`. In conjunction with pointcut functions, we thereby get quite powerful expressions to describe *where* advice should affect the program. For instance, we might change the pointcut expression for our `Tracing` aspect as follows:

```
advice call ("% ...::%(...)")
       && within ("Client") : before () {
  std::printf ("calling %s\n", JoinPoint::signature ());
}
```

The `call()` function yields all function call join points for the given functions. In contrast to execution join points, call join points take effect on the caller side, that is before, after, or around an actual function call. The `within()` pointcut function simply returns all join points in the given classes or functions. By giving advice to the intersection of `call ("% ...::%(...)")` (any function call) and `within ("Client")` (any join point in class `Client`), the aspect will now trace only those function calls that are made from a method in class `Client`.

Join Point API

In the advice body the expression `JoinPoint::signature()` still waits for an explanation. As we know from the example, it yields the function name that we print before the function execution starts. The static member function `signature()` is defined by the “join point API”. This is an API defined by AspectC++ that allows aspect code to retrieve context information from or about the join point for which it is running. We will later see, that such context information is an indispensable feature for many real-world aspects.

Lessions Learned

By the `Tracing` example, although it is implemented by a few lines of code only, we introduced a lot of AspectC++ concepts. Let’s summarize them:

- *Crosscutting Concern*: a concern of an implementation, which affects many different parts of a program.
- *Aspect*: provides a modular implementation of a crosscutting concern by defining advice.
- *Join Point*: either an event in the control flow (dynamic join point) or an element of the static program structure (static join point) at which advice affects the program.
- *Pointcut*: a set of join points.
- *Match Expression*: pattern which is matched against the signatures of named program entities, i.e. elements of the static program structure. Thus, match expressions are primitive pointcut expressions, which yield static join points.
- *Pointcut Expression*: is used to define a pointcut. Pointcut expressions are composed by match expressions and pointcut functions. They define *where* advice should affect the program.
- *Advice*: defines *how* an aspect affects the program at a given pointcut. In the case of advice for dynamic join points `before()`, `after()`, or `around()` advice can be used to implement additional behavior.
- *Join Point API*: can be used in advice code to retrieve context information from the current join point via the built-in pointer `JoinPoint *tjp`.

What’s next

“Tracing” is a typical “development aspect”. In contrast to “production aspects” these aspects are only used during the development of a program, e.g. for the purpose of debugging, quality assurance and optimization. Production aspects are part of the final software product, which is shipped to the users. Therefore, we recommend to start AOP with development aspects and gather some experience first. However, in this article we will, of course, not stop after the “Hello World” program of AOP! Our next example will be a production aspect that will show some more advanced AspectC++ features and especially deals with crosscutting in the static program structure.

Observer Pattern in AspectC++

Today, it is state of the art to use design patterns from the “Gang of Four” to develop object-oriented software. One of the most popular patterns is “Observer”, which is illustrated by the class diagram in figure 1. This pattern can be applied if an object manages a state (the “Subject” – a `ClockTimer` object)

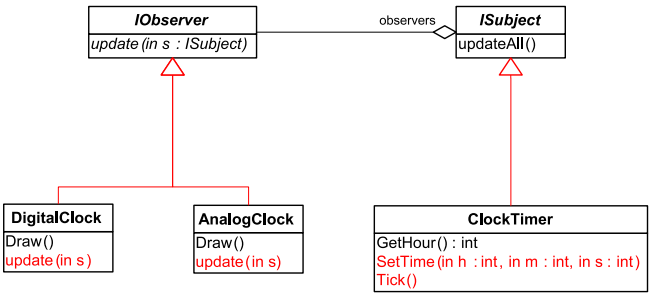

Figure 1: Crosscutting in the Observer Pattern

and an arbitrary number of other objects (the "Observers" – `DigitalClock` and `AnalogClock` instances) should be informed when the state changes. As the class diagram shows, the subject/observer relationship between our three application classes can be established by deriving the `ClockTimer` from a reusable `ISubject`, which manages the list of observer objects, and by deriving the observers from the abstract `IObserver` class. Furthermore, all state changing functions (`SetTime()` and `Tick()`) have to be extended by a call to `updateAll()` to notify all observers about the change. On the observer side the `DigitalClock` and `AnalogClock` have to be extended by an `update()` function that is required by the abstract base class. Overall, a quite high number of error-prone modifications has to be performed on the code of our three classes. Figure 1 illustrates by red color, which parts of the implementation are affected. From the aspect-oriented point of view the "observer protocol concern" statically and dynamically crosscuts the participating classes `ClockTimer`, `DigitalClock` and `AnalogClock`. Hence, it is better separated out into an aspect.

Dealing with the Dynamic Crosscutting

We already know the necessary AspectC++ language elements to implement the dynamic crosscutting in this example. The observer protocol requires all state-changing methods in the subject class to call `updateAll()` before returning. In C++, all non-`const` member functions of a class can be considered as state-changing. The following advice definition inserts the necessary calls to `updateAll()` into our `ClockTimer` class.

```
advice execution("% ClockTimer::%(...)") &&
       !execution("% ClockTimer::%(...) const") : after () {
  updateAll ();
}
```

You can read the pointcut expression of this advice as "A join point is an element of the resulting pointcut if it is the execution of a `ClockTimer` member function *and not* the execution of a `ClockTimer` member function declared as `const`". The reason for this *and not* kind of expression is that `const` in a match expression is interpreted as a restriction. If `const` is not given, both, `const` and non-`const` functions are matched.

Introductions – Implementing Static Crosscutting

The static crosscutting in the example can be implemented by an AspectC++ feature called "introductions". An introduction is another kind of advice for which the *where* is a pointcut expression, which represent a set of classes, while the *how* is a declaration, which should be introduced into the classes. For example, the `update()` function could be introduced into the observer classes as shown here:

```
advice "DigitalClock"||"AnalogClock" : void update() {
  Draw();
}
```

Note that introduced members are not only visible by the aspect. The `update()` function can be called, for instance, by other members of `DigitalClock` or `AnalogClock` as if it were an ordinary member function. However, introductions are not restricted to member functions. They can be used to introduce data members, nested classes and anything else that is syntactically allowed inside a class definition.

"Base class introductions" are a special kind of introduction, which introduce new classes into the list of base classes. They are very helpful in our example, as the subject and the observers have to derive from the `ISubject` and `IObserver` roles, respectively:

```
advice "DigitalClock"||"AnalogClock" : baseclass(IObserver);
advice "ClockTimer" : baseclass(ISubject);
```

Virtual Pointcuts and Abstract Aspects

Now we have all elements together to assemble an `ObserverPattern` aspect for our example. However, applying the observer protocol to a set of classes is a recurring task. We want to achieve a reusable implementation. For this purpose we need two further AspectC++ language features.

The first feature is the ability to give pointcuts a name. For example, the pointcut expression `"DigitalClock" || "AnalogClock"`, which has been used several times, can become a named pointcut `observers()`:

```
pointcut observers() = "DigitalClock"||"AnalogClock";
```

An even more interesting feature of named pointcuts is, that they can be declared "virtual" or "pure virtual". Pure virtual pointcuts can be used by advice as ordinary pointcuts. An aspect that uses pure virtual pointcuts only defines *how* a crosscutting concern is implemented, but not *where* it will affect the program.

```
pointcut virtual observers() = 0;
pointcut virtual subjects() = 0;
advice observers() : baseclass(IObserver);
advice subjects() : baseclass(ISubject);
```

As a consequence the aspect is incomplete, it is an "abstract aspect". This is very similar to abstract classes with pure virtual member functions, which can not be used for instantiation. Abstract aspects do not affect the program as long as there is no derived aspect, which defines the pure virtual pointcut of its base aspect.

Aspect Inheritance

If we put everything together we end up with the reusable `ObserverPattern` aspect shown in figure 2. `ObserverPattern` is completely independent of our three classes in the example. It only defines *how* the observer pattern crosscuts an implementation, but not *where*. This has to be done by a derived aspect, shown in listing 3. Our derived `ClockObserver` aspect does so by defining the two inherited pure virtual pointcuts. It also implements the introduced `update()` function in an observer-specific way.

Lessions Learned

So we finally end up with two aspects. The reusable abstract base aspect `ObserverPattern` encapsulates the implementation of the observer protocol. This is a clear advantage, as the corresponding design decision would otherwise be hard-wired in dozens of classes. For instance, by modifying this aspect we could easily switch between an implementation, which stores an observer list in each subject instance, and an implementation, which manages a central data structure for that purpose.

The aspect `ClockObserver` inherits from `ObserverPattern` and thereby performs the binding of the abstract observer and subject roles to the concrete application classes `ClockTimer`, `DigitalClock` and `AnalogClock`. This is another advantage, as it is thereby no longer necessary to pollute the application classes themselves with pattern-specific code. They can remain untouched, thus keep their comprehensibility and reusability.

```
aspect ObserverPattern {
  // Data structures to manage subjects and observers
  ...
public:
  // Interfaces for each role
  struct ISubject {};
  struct IObserver {
    virtual void update(ISubject *) = 0;
  };

  // To be defined by the concrete derived aspect
  // subjectChange() matches all non-const methods
  pointcut virtual observers() = 0;
  pointcut virtual subjects() = 0;
  pointcut virtual subjectChange() =
      execution("% ...::%(...)" && !"% ...::%(...) const")
      && within(subjects());

  // Add new baseclass to each subject/observer class
  // and insert code to inform observers
  advice observers() : baseclass(IObserver);
  advice subjects() : baseclass(ISubject);

  advice subjectChange() : after() {
    ISubject* subject = tjp->that();
    updateObservers(subject);
  }
  // Operations to add, remove and notify observers
  void updateObservers(ISubject* sub) { ... }
  void addObserver(ISubject* sub, IObserver* ob) { ... }
  void remObserver(ISubject* sub, IObserver* ob) { ... }
};
```

Figure 2: The Abstract Aspect `ObserverPattern`

```
#include "ObserverPattern.ah"
#include "ClockTimer.h"

aspect ClockObserver : public ObserverPattern {
  // define the pointcuts
  pointcut subjects()   = "ClockTimer";
  pointcut observers()  = "DigitalClock"||"AnalogClock";
public:
  advice observers() :
          void update( ObserverPattern::ISubject* sub ) {
      Draw();
    }
};
```

Figure 3: Concrete Observer Implementation

By the `ObserverPattern` example, we introduced some more important concepts of AspectC++. Let's again summarize them:

- *Introduction*: advice for static join points. Introductions can be used to extend the static structure of classes by additional elements like member functions, data members, or local classes.
- *Baseclass Introduction*: a special form of introduction to extend the list of base classes a class inherits from.
- *Named Pointcut*: a pointcut expression that can be referred by an identifier. Named pointcuts can be declared virtual or pure virtual, thus allowing to override them in an inherited aspect.
- *Abstract Aspect*: an aspect which contains at least one pure virtual pointcut or method. Abstract aspects are incomplete, thus do not affect the program until completed by a derived aspect.
- *Aspect Inheritance*: like class inheritance, aspects can inherit from other aspects.

What's next

Applications of aspects or not restricted to tracing and patterns. There are many crosscutting concerns and soon after learning the AOP basics, programmers automatically identify them and long for tools to implement them in a modular way. We will now look into our last example, which describes another production aspect. From the technical perspective it will show how aspect implementations can benefit from the various information provided by the join point API together with generic and generative programming techniques.

Aspects with Advanced C++

C++ programmers often have to deal with legacy C libraries like the Win32 API. Besides the fact that the API is not object-oriented, the error handling of the library functions does not

```
#include <strstream>
#include "win32-helper.h"
aspect ThrowWin32Errors {
  using namespace std;

  // template metaprogram to generate code for
  // streaming a comma-separated sequence of arguments
  template< class TJP, int N >
  struct stream_params {
    static void process( ostream& os, TJP* tjp ) {
      os << *tjp->arg< TJP::ARGS - N >() << ", ";
      stream_params< TJP, N - 1 >::process( os, tjp );
  } };
  // specialization to terminate the recursion
  template< class TJP >
  struct stream_params< TJP, 1 > {
    static void process( ostream& os, TJP* tjp ) {
      os << *tjp->arg< TJP::ARGS - 1 >();
  } };

  advice call( win32::Win32API() ) : after() {
    if( win32::IsErrorResult( *tjp->result() ) ) {
      ostringstream os;
      DWORD code = GetLastError();

      os << "WIN32 ERROR " << code << ": "
         << win32::GetErrorText(code) << endl;
      os << "WHILE CALLING: "
         << tjp->signature() << endl;
      os << "WITH: " << "(";

      // Generate joinpoint-specific sequence of
      // operations to stream all argument values
      stream_params< JoinPoint,
                     JoinPoint::ARGS >::process( os, tjp );
      os << ")";
      throw win32::Exception( os.str(), code );
  } }
};
```

Figure 4: An Aspect to Throw Win32 Errors as Exceptions

fit into exception-based error handling, which is favoured by many programmers today. The transformation of the C-style error handling towards an exception-based approach would be a laborious and error-prone task. Furthermore, it is a cross-cutting concern, because all the Win32 API functions would have to be wrapped by a function that checks the result and raises an exception if an error was detected.

The `ThrowWin32Errors` aspect shown in listing 4 does the same with less work for the programmer. By compiling the application with this aspect the Win32 API behaves as if it were reporting errors by throwing exceptions. However, the implementation is not trivial and should therefore be explained.

Detecting Win32 Errors

The first step towards an exception-based propagation of errors is to detect if the invocation of Win32 function has failed. Win32 API functions indicate an error situation by returning a special “magic value”. Detecting failed API calls is therefore, once again, a problem of dynamic crosscutting. The general

```
namespace win32 {
  struct Exception {
    Exception( const std::string& w, DWORD c ) { ... }
  };

  // Check for "magic value" indicating an error
  inline bool IsErrorResult( HANDLE res ) {
    return res == NULL || res == INVALID_HANDLE_VALUE;
  }
  inline bool IsErrorResult( HWND res ) {
    return res == NULL;
  }
  inline bool IsErrorResult( BOOL res ) {
    return res == FALSE;
  }
  ...

  // Translates a Win32 error code into a readable text
  std::string GetErrorText( DWORD code ) { ... }

  pointcut  Win32API() =  "% CreateWindow%(...)"
                        || "% BeginPaint(...)"
                        || "% CreateFile%(...)"
                        || ...
} // namespace Win32
```

Figure 5: The `win32helper.h` File

idea is to give after advice for all calls to Win32 functions. In the advice body, the return value should be checked to throw an exception in case of an error:

```
aspect ThrowWin32Errors{
advice call(win32::Win32API()) : after() {
  if(<magic value> == *tjp->result())  throw ...
}};
```

The advice affects all API functions that are described by the (externally defined) named pointcut `win32::Win32API()`, which contains all Win32 API functions (listing 5). In the advice body, the return value of the called Win32 function is retrieved via the `tjp->result()` method of the join point API. This method returns a pointer to the actual result value, thus makes it even possible to modify the result. Here we just compare it with the “magic value” that is returned by the API function to indicate an error.

The advice definition, however, does not work yet. The problem is that the actual “magic value” to be compared with the result is not always the same. It depends on the return type of the called API function. Many Win32 functions are simply of type `BOOL` and indicate an error by returning `FALSE`. However, other API functions use types like `HWND`, `ATOM`, `HDC`, or `HANDLE`. For each of these types there is some associated “magic value” that is returned in case of an error. `ATOM` functions, for instance return `0`; `HANDLE` functions return either `NULL` or `INVALID_HANDLE_VALUE`.

Generic Advice

As a possible solution for this problem we might filter the functions in the `win32::Win32API()` pointcut for each return type and give specific advice for it:

compile-time types and enumerators:	
`That`	type of the affected class
`Target`	type of the destination class (for call join-points)
`Arg<`*i*`>::Type` `Arg<`*i*`>::ReferredType`	type of the i'th argument with $0 \leq i <$ ARGS
`Result`	result type
`ARGS`	number of arguments
runtime static methods:	
`const char *signature()`	signature of the affected function
runtime non-static methods:	
`void proceed()`	execute original code (around advice)
`That *that()`	object instance referred to by this
`Target *target()`	target object instance of a call (for call join-points)
`Arg<`*i*`>::ReferredType *arg()`	argument value instance of the i'th argument
`Result *result()`	result value instance

Table 1: Excerpt from the AspectC++ Join Point API

```
aspect ThrowWin32Errors {
advice call(win32::Win32API() && "BOOL %(...)") : after() {
  if(FALSE == *tjp->result())  throw ...
}
advice call(win32::Win32API() && "HANDLE %(...)") : after() {
  if((NULL == *tjp->result())
     || (INVALID_HANDLE_VALUE == *tjp->result()))throw ...
}
... // and so on
};
```

This solution has some drawbacks, though. We have to write almost the same advice definition again and again. Even worse, if we forget a type or Microsoft introduces a new one, the related API functions would silently be missed by the aspect, as they are not matched by any of the existing advice definitions. Therefore, we strive for a better, less fragile solution:

```
aspect ThrowWin32Errors {
advice call(win32::Win32API()) : after() {
  if(win32::IsErrorResult(*tjp->result()))  throw ...
}};
```

Now we have separated out all type-dependend code (the comparison with the type-dependent "magic values") into an own `win32::IsErrorResult()` function, which has to be overloaded for each return type (listing 5). Depending on the actual static type of `*tjp->result()`, the *compiler* looks for a compatible version of `win32::IsErrorResult()` and, even more important, complains if no one can be found. It will no longer happen that we silently miss some functions, just because their return type was forgotten.

The above advice definition is an example for "Generic Advice". It is generic, because it adapts its actual implementation ("magic value" to test for) with respect to some type information (return type of the matched function) of the current join point context. This is very similar to the techniques used in templates libraries for generic programming, like the STL.

Reporting Win32 Errors

After we know how to detect failed Win32 API calls by our aspect reliably, the next step is to report them as an exception. The exception object should include all context information that can be helpful to figure out the reason for the actual failure. Besides the Win32 error code, this should include a human readable string describing the error, the signature of the called function (retrieved with `JoinPoint::signature()`) and the actual parameter values that were passed to the function.

Generative Advice

The tricky part is the generation of a string of the actual parameter values. The idea is to stream each parameter into a `std::ostringstream` object. However, as the advice affects functions with very different signatures, its implementation has to be generic with respect to the number and types of function arguments. The sequence of `operator <<(std::ostream&, T)` calls has therefore to be *generated* according to the affected function's signature. This is realized (listing 4) by feeding the information provided by the join point API (table 1) into a small template meta-program. This template meta-program is instantiated by the advice code with the `JoinPoint` type and iterates, by recursive instantiation of the template, over the join-point-specific argument type list `JoinPoint::Arg<`*I*`>`. For each argument type, a `stream_params` class with a `process()` method is generated, which later at runtime will stream the typed argument value (retrieved via `tjp->arg<`*I*`>()`) and recursively call `stream_params::process()` for the next argument.

Lessions Learned

As demonstrated by `ThrowWin32Errors`, aspects can not only used with object-oriented software, but provide benefits for improving legacy C-style code, too. They can furthermore be implemented in a very generic way by exploiting other advanced C++ techniques like generic and generative programming. The AspectC++ concepts for this combination are:

- *Generic Advice*: advice that uses static type information from the current join point context to instantiate or bind generic code.
- *Generative Advice*: advice with an implementation that is partly generated by the instantiation of template-metaprograms using static type information from the current join point context.

What's next

You have now seen the most important language constructs of AspectC++. The `Tracing`, `ObserverPattern`, and `ThrowWin32Error` aspects are, of course, just some examples for the very different flavors of crosscutting that can be addressed by AOP. Probably you already have some ideas for

using AOP in your own C++ projects. We will now take a look at the available AspectC++ tools for this purpose.

Tool Support

AOP provides means to modularize the implementation of crosscutting concerns into aspects. As a consequence, the aspect code has to be *woven* into the affected components to build the final program. For this task an *aspect weaver* is required.

Not required, but strongly recommended, is furthermore tool support for *join point visualization*. Aspects can potentially modify the program at any place. In larger projects, this implies the danger of "surprising" program behavior, if developers who work on the component code are not aware of the aspects. Therefore, all join points which are actually affected by an aspect should be marked automatically in the code. Then developers can easily see where aspects affect their code.

AspectC++ Weaver

The AspectC++ weaver `ac++` is a source-to-source weaver that transforms AspectC++ programs into C++ programs. Hence, it can be used in conjunction with any standard-compliant C++ compiler as back end, g++ (3.x) and Microsoft C++ (VisualStudio.NET) are particulary supported.

In order to identify join points correctly, `ac++` performs a complete syntactical and semantical analysis of its AspectC++ input. Considering the complexity of the C++ language the project can be regarded as highly ambitious. Nevertheless, `ac++` can already parse real-world C++ code and even complex templates as defined by the STL or Microsofts ATL are no problem anymore. More advanced template libraries like Boost will be supported in the near future.

ACDT Plugin for Eclipse

The AspectC++ Development Tool for Eclipse (ACDT) is an Eclipse plugin based on the code of the CDT project. It extends the C++ Development Tools by adding syntax highlighting of the AspectC++ keywords, an extended outline (shows aspects, advice, and pointcuts, see figure 6), a builder for "Managed Make" projects, and join point visualization in the outline view and the source code editor even in "Standard Make Projects" based on your own Makefile.

AspectC++ Add-In for VisualStudio.NET

A commercial VisualStudio.NET extension for AspectC++ is available from `www.pure-systems.com`. It supports various Visual C++ specific language extensions and is therefore the first choice for users accustomed to the VisualStudio IDE and the Microsoft Visual C++ compiler. A free evaluation version is available.

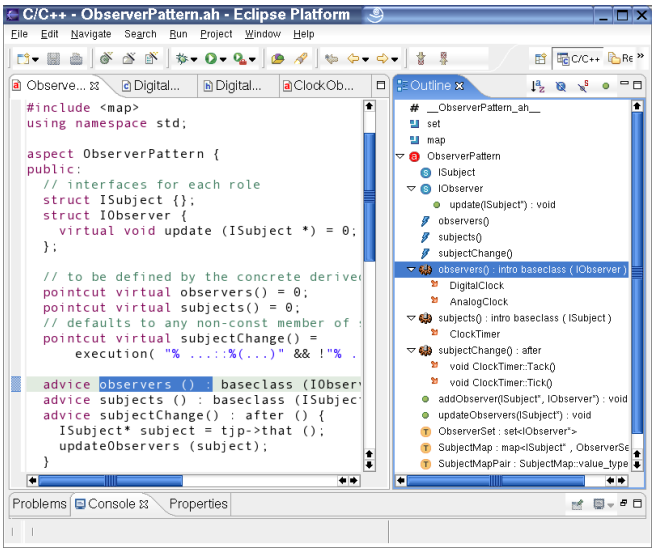

Figure 6: ACDT in Action

Summary and Conclusions

Mostly known only from the Java world, AOP is suitable for C++ projects as well. This article introduced the most important concepts and language features of AspectC++. Programmers can benefit from an aspect-oriented language extension in various ways. Development aspects like Tracing are a good start for using AOP and can already safe programmers a lot of work. In some real-world projects we measured that about 25% of the lines of code were related to tracing, profiling, or constraint checks. Production aspects can be found everywhere. As the examples showed, they can simplify the design, the implementation, and even the handling of legacy libraries.

After reading this article you have already mastered the first steps of "going AOP". The evaluation version of the AspectC++ Addin for VisualStudio.NET by pure-systems GmbH and the open source AspectC++ Development Tools (ACDT) for Eclipse are available on the accompanying CD.

On the net

http://www.aspectc.org/ AspectC++ project homepage

http://acdt.aspectc.org/ AspectC++ Development Tools (ACDT) for Eclipse

http://www.aosd.net/ Web portal for everything related to aspect-oriented software development (AOSD)

http://www.pure-systems.com/ Company, which offers the AspectC++ Add-In for Visual Studio .NET and commercial support for AspectC++ users