Advances in AOP with AspectC++

Olaf Spinczyk\textsuperscript{1}, Daniel Lohmann\textsuperscript{1}, and Matthias Urban\textsuperscript{2}

\textsuperscript{1}Friedrich-Alexander University Erlangen-Nuremberg, Germany
\textsuperscript{2}pure-systems GmbH Magdeburg, Germany

http://www4.cs.fau.de/~spinczyk
olaf.spinczyk@cs.fau.de
Outline

1. Introduction
   ▪ AOP with C++

2. AspectC++
   ▪ main goals and language introduction

3. Example: Win32 Error Handling
   ▪ advice and genericity
   ▪ the AspectC++ join point API
   ▪ generic advice

4. Evaluation

5. Summary and Future Work
Aspect-Oriented Programming

AOP is about modularizing crosscutting concerns

examples: tracing, synchronization, security, buffering, error handling, constraint checks, ...

well modularized concern

badly modularized

without AOP

with AOP

aspect
Why AOP with C++?

- widely accepted benefits from using AOP
  - avoidance of code redundancy, better reusability, maintainability, configurability, the code better reflects the design, ...

- enormous existing C++ code base
  - maintainance: extensions are often crosscutting

- millions of programmers use C++
  - for many domains C++ is the adequate language
  - they want to benefit from AOP (as Java programmers do)
AspectC++ Overview

- general-purpose AOP language extension for C++
  - AspectJ-like syntax and semantics
  - source-level weaver ac++

- goals of AspectC++
  - AOP with C++ should become easy
    - language level / tool support
    - integration into IDEs (Visual Studio / Eclipse)
  - AspectC++ should be strong where C++ is strong
    - facilitate “aspect-izing” the enormous C/C++ code base
    - no overhead
    - do as much as possible at compile-time
aspect ElementCounter {
    advice call("% Queue::enqueue(...)") : after() {
        tjp->target ()->_elements++;
    }
}
...
Example: A typical Win32 Application

LRESULT WINAPI WndProc( HWND hWnd, UINT nMsg, WPARAM wParam, LPARAM lParam) {
    HDC dc = NULL; PAINTSTRUCT ps = {0};

    switch( nMsg ) {
        case WM_PAINT:
            dc = BeginPaint( hWnd, &ps );
            ..
            EndPaint(hWnd, &ps);
            break;
        ..
    }
}

int WINAPI WinMain( ... ) {
    HANDLE hConfigFile = CreateFile( "app.config", GENERIC_READ, .. );

    WNDCLASS wc = {0, WndProc, 0, 0, .. , "Example_Class"};
    RegisterClass( &wc );
    HWND hwndMain = CreateWindowEx( 0, "Example_Class", "Example", .. );
    UpdateWindow( hwndMain );

    MSG msg;
    while( GetMessage( &msg, NULL, 0, 0 ) ) {
        TranslateMessage( &msg );
        DispatchMessage( &msg );
    }
    return 0;
}
Example: A typical Win32 Application

```c
LRESULT WINAPI WndProc( HWND hWnd, UINT nMsg, WPARAM wParam, LPARAM lParam) {
    HDC dc = NULL; PAINTSTRUCT ps = {0};

    switch( nMsg ) {
    case WM_PAINT:
        dc = BeginPaint( hWnd, &ps );
        ..
        EndPaint(hWnd, &ps);
        break;
    ..
    }

    int WINAPI WinMain( ... ) {
        HANDLE hConfigFile = CreateFile( "app.config", GENERIC_READ, .. );

        WNDCLASS wc = {0, WndProc, 0, 0, .. , "Example_Class"};
        RegisterClass( &wc );
        HWND hwndMain = CreateWindowEx( 0, "Example_Class", "Example", .. );
        UpdateWindow( hwndMain );

        MSG msg;
        while( GetMessage( &msg, NULL, 0, 0 ) ) {
            TranslateMessage( &msg );
            DispatchMessage( &msg );
        }
        return 0;
    }
```
Example: Goals

- detect failed Win32 API calls
  - checking all these call highly crosscuts our application
  - a case for Aspect-Oriented Programming with AspectC++

- throw a *helpful* exception in case of a failure
  - describing the exact circumstances and reason of the failure
  - a default exception handler can show an error message and exit()

Problem: *generic* aspect code needed

- function argument and result types differ
- errors are indicated by a “magic” return value

<table>
<thead>
<tr>
<th>return type</th>
<th>magic value</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOL</td>
<td>FALSE</td>
</tr>
<tr>
<td>ATOM</td>
<td>(ATOM) 0</td>
</tr>
<tr>
<td>HANDLE</td>
<td>INVALID_HANDLE_VALUE or NULL</td>
</tr>
<tr>
<td>HWND</td>
<td>NULL</td>
</tr>
</tbody>
</table>
Advice & Genericity

conceptual perspective:

```cpp
aspect Trace {
    advice call("% f%(...)") : after() {
        print all arguments and result
    }
};
```

join points

- `int f1()`
- `bool f2(bool,double)`
- `long f3(const C&)`
Advice & Genericity

technical perspective:

advice

```cpp
aspect Trace {
    advice call("% f%(...)") : after() {
        print all arguments and result
    }
};
```

Join points

- **Context 1**: `int f1()`
  - `res: int, 0`

- **Context 2**: `bool f2(bool, double)`
  - `res: bool, true`
  - `arg 1: bool, false`
  - `arg 2: float, 3.14`

- **Context 3**: `long f3(const C&)`
  - `res: long, 42L`
  - `arg 1: C&, cobj`
Advice & Genericity

technical perspective:

```
aspect Trace {
  advice call("% f%(...)") : after() {
    print all arguments and result
  }
}
```
AspectC++ Joinpoint API

**Compile-Time Joinpoint API**
- `JoinPoint::That` Type of affected class (call/execution)
- `JoinPoint::Target` Type of the target class (call)
- `JoinPoint::Result` Type of the function result
- `JoinPoint::Arg<i>::Type` Type of the \(i^{th}\) function argument (with \(0 \leq i < \text{ARGS}\))
- `JoinPoint::Arg<i>::ReferredType` value of \(i^{th}\) argument
- `JoinPoint::ARGS` Number of arguments
- `JoinPoint::JPID` Unique identifier for this joinpoint
- `JoinPoint::JPTYPE` Type of the joinpoint (call/execution)

**Runtime Joinpoint API**
- `That* that()` current object instance
- `Target* target()` target object instance (call)
- `Result* result()` result value
- `Arg<i>::ReferredType* arg<i>()` value of \(i^{th}\) argument
- ...

AspectC++ Joinpoint API

SoMeT 2005: Advances in AOP with AspectC++
AspectC++ Joinpoint API

Compile-Time Joinpoint API

- JoinPoint::That
- JoinPoint::Target
- JoinPoint::Result
- JoinPoint::Arg< i >::Type
- JoinPoint::Arg< i >::ReferredType
- JoinPoint::ARGS
- JoinPoint::JPID
- JoinPoint::JPTYPE

Runtime Joinpoint API

- That* that()
- Target* target()
- Result* result()
- Arg< i >::ReferredType* arg< i >()
- ...
AspectC++ Joinpoint API

Compile-Time Joinpoint API
JoinPoint::That
Type of affected class (call/execution)
JoinPoint::Target
Type of the target class (call)
JoinPoint::Result
Type of the function result
JoinPoint::Arg< i >::Type
Type of the $i^{th}$ function argument (with $0 \leq i < \text{ARGS}$)
JoinPoint::Arg< i >::ReferredType
Number of arguments
JoinPoint::ARGS
JoinPoint::JPID
JoinPoint::JPTYPE

Runtime Joinpoint API
That* that()
current object instance
Target* target()
target object instance (call)
Result* result()
result value
Arg< i >::ReferredType* arg< i >()
value of $i^{th}$ argument

Type-safe access to actual values at runtime
Generic Advice

We call an advice a *generic advice*,
if its implementation depends on
joinpoint-specific static type information
Generic Advice for Error Detection

```
advice call(win32API ()) :
  after () {
    if (isError (*tjp->result()))
      // throw an exception
  }
```

- `bool isError(ATOM);`
- `bool isError(BOOL);`
- `bool isError(HANDLE);`
- `bool isError(HWND);`
- ...
Generic Advice for Error Detection

- no runtime type checks are needed
- unhandled types are detected at compile-time
- functions can be inlined

```cpp
advice call(win32API ()) : after () {
  if (isError (*tjp->result()))
    // throw an exception
}

bool isError(ATOM);
bool isError(BOOL);
bool isError(HANDLE);
bool isError(HWND);
```
Error Message Generation by Advice

```cpp
template <int I> struct ArgPrinter {
    template <class JP> static void work (JP &tjp, ostream &s) {
        ArgPrinter<I-1>::work (tjp, s);
        s << “, “ << *tjp. template arg<I-1>();
    }
};

advice call(win32API ()) : after () {
    ostringstream s;
    DWORD code = GetLastError();
    s << “WIN32 ERROR ” << code << ...
        << win32::GetErrorMessage( code ) << ...
        << tjp->signature() << “WITH: ” << ...;
    ArgPrinter<JoinPoint::ARGS>::work (*tjp, s);
    // throw an exception
    throw win32::Exception( s.str() );
}
```
template <int I> struct ArgPrinter {
    template <class JP> static void work (JP &tjp, ostream &s) {
        s << ArgPrinter<I-1>::work (tjp, s) << tjp->signature() << "WITH: ";
        ArgPrinter<JoinPoint::ARGS>::work (*tjp, s);
        throw win32::Exception (s.str());
    }
};

full power of template metaprogramming is available for aspects
Achieved Goals

the Win32 error handling aspect ...

- converts the error handling of the Win32-API
- provides context information about the error
- will not be compiled if a case is not properly handled

⇒ „do as much as possible at compile time“

in the component code neither template nor aspect programming is required

⇒ „facilitate 'aspectizing' the enormous C/C++ code base“
Problems

- (generic) aspects depend on types
  - would fail if all Win32 functions returned int objects with different error semantics
  - AOP with C++ makes more sense than with C

- code bloating
  - advice is „instantiated“ per join point
  - C++ compilers are still not very clever in detecting duplicates
  - advice adds an additional level of indirection

→ „minimal overhead“?
### Evaluation: Micro Benchmarks

Measured on Pentium III with g++-3.3 -O?

```plaintext
advice ... {
    global++;
}
```

```plaintext
advice ... {
    (*tjp->arg<0>()++)
}
```

<table>
<thead>
<tr>
<th>Runtime [clock cycles]:</th>
<th>O3</th>
<th>Os</th>
<th>O0</th>
<th>O3</th>
<th>Os</th>
<th>O0</th>
</tr>
</thead>
<tbody>
<tr>
<td>call advice</td>
<td>5</td>
<td>1</td>
<td>44</td>
<td>8</td>
<td>11</td>
<td>69</td>
</tr>
<tr>
<td>hand-written</td>
<td>1</td>
<td>3</td>
<td>23</td>
<td>1</td>
<td>15</td>
<td>12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Advice Code Size [bytes]:</th>
<th>O3</th>
<th>Os</th>
<th>O0</th>
<th>O3</th>
<th>Os</th>
<th>O0</th>
</tr>
</thead>
<tbody>
<tr>
<td>call advice</td>
<td>17</td>
<td>23</td>
<td>36</td>
<td>29</td>
<td>41</td>
<td>174</td>
</tr>
<tr>
<td>hand-written</td>
<td>17</td>
<td>23</td>
<td>26</td>
<td>17</td>
<td>32</td>
<td>36</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stack Consumption [bytes]:</th>
<th>O3</th>
<th>Os</th>
<th>O0</th>
<th>O3</th>
<th>Os</th>
<th>O0</th>
</tr>
</thead>
<tbody>
<tr>
<td>call advice</td>
<td>0</td>
<td>0</td>
<td>64</td>
<td>4</td>
<td>4</td>
<td>72</td>
</tr>
<tr>
<td>hand-written</td>
<td>0</td>
<td>0</td>
<td>20</td>
<td>0</td>
<td>12</td>
<td>16</td>
</tr>
</tbody>
</table>
Evaluation: AO Product Lines

- case study: an embedded weather station product line
  - separation of concerns is imperative for component plug&play

![Diagram of a weather station product line with components such as Actors, Sensors, Alarm, Display, PC Connection, Temperature, Air Pressure, Wind Speed, RS232Line, USBLine, Protocol, SNGProto, and XMLProto.]
Evaluation: AO Product Lines

- two implementations compared: AO vs. OO

- Flash [bytes]
  - AO Version
  - OO Version

- RAM [bytes]
  - AO Version
  - OO Version

<table>
<thead>
<tr>
<th>RAM</th>
<th>AO Version</th>
<th>OO Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>P+Display</td>
<td>64</td>
<td>96</td>
</tr>
<tr>
<td>P+Display+Serial+XML</td>
<td>96</td>
<td>128</td>
</tr>
<tr>
<td>PW+Display</td>
<td>128</td>
<td>160</td>
</tr>
<tr>
<td>T+Display</td>
<td>160</td>
<td>192</td>
</tr>
<tr>
<td>TWP+Serial+XML</td>
<td>192</td>
<td>224</td>
</tr>
<tr>
<td>TWP+Serial+XML+Display</td>
<td>224</td>
<td>256</td>
</tr>
</tbody>
</table>

- Flash [bytes]
  - AO Version
  - OO Version

- P+Display | 64 | 96 |
- P+Display+Serial+XML | 96 | 128 |
- PW+Display | 128 | 160 |
- T+Display | 160 | 192 |
- TWP+Serial+XML | 192 | 224 |
- TWP+Serial+XML+Display | 224 | 256 |
Evaluation: AO Product Lines

two implementations compared: AO vs. OO

advantages of the AO version:
- OO flash consumption: 79%-114% higher
- OO RAM consumption: 55%-70% higher

„no overhead“ and better than OO ✓
Summary: AspectC++ ...

- provides fully-fledged AOP support for C++
- combines the strengths of aspects and templates
  - highly generic (and efficient) aspect implementations
  - transparent injection of template-metaprograms
- is strong where C++ is strong
  - static typing
  - transparently applicable
  - minimal overhead
Future Work

Version 1.0:
- Weaving in templates and template instances
- Weaving in C code and mixed projects
- improved performance

- further research on AOP & Generic/Generative Programming

http://www.aspectc.org