Aspect-Oriented Programming
with
C++ and AspectC++

AOSD 2004 Tutorial

---

Presenters

- **Andreas Gal**
  - ag@aspectc.org
  - University of California, Irvine, USA
- **Daniel Lohmann**
  - d1@aspectc.org
  - University of Erlangen-Nuremberg, Germany
- **Olaf Spinczyk**
  - os@aspectc.org
  - University of Erlangen-Nuremberg, Germany

---

Schedule

<table>
<thead>
<tr>
<th>Part</th>
<th>Title</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>Introduction</td>
<td>10m</td>
</tr>
<tr>
<td>II</td>
<td>AOP with pure C++</td>
<td>40m</td>
</tr>
<tr>
<td>III</td>
<td>AOP with AspectC++</td>
<td>70m</td>
</tr>
<tr>
<td>IV</td>
<td>Tool support for AspectC++</td>
<td>30m</td>
</tr>
<tr>
<td>V</td>
<td>Real-World Examples</td>
<td>20m</td>
</tr>
<tr>
<td>VI</td>
<td>Summary and Discussion</td>
<td>10m</td>
</tr>
</tbody>
</table>

---

This Tutorial is about ...

- Writing aspect-oriented code with **pure C++**
  - basic implementation techniques
- Programming with **AspectC++**
  - language concepts, implementation, tool support
  - this is an AspectC++ tutorial
- Pros and cons of each approach
- Programming languages and concepts
  - no coverage of other AOSD topics like analysis or design
Aspect-Oriented Programming

- AOP is about modularizing crosscutting concerns
  - well modularized concern
  - badly modularized

- Examples: tracing, synchronization, security, buffering, error handling, constraint checks, ...

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)

- How can the AOP community help?
  - Part II: describe how to apply AOP with built-in mechanisms
  - Part III-V: provide special language mechanisms for AOP

Scenario: A Queue utility class

The Simple Queue Class

```cpp
namespace util
{
    class Item
    {
        friend class Queue;
        Item* next;
        public:
            Item() : next(0) {}
    };

    class Queue
    {
        Item* first;
        Item* last;
        public:
            Queue() : first(0), last(0) {}
            void enqueue( Item* item )
            {
                printf( " > Queue::enqueue\n" );
                if( last )
                {
                    last->next = item;
                    last = item;
                }
                else
                {
                    first = first->next;
                    printf( " < Queue::enqueue\n" );
                    return res;
                }
            }
        Item* dequeue()
        {
            printf( " > Queue::dequeue\n" );
            if( first == last )
            {
                first = last = 0;
            }
            else
            {
                first = first->next;
                printf( " < Queue::dequeue\n" );
                return res;
            }
        }
    } // class Queue
} // namespace util
```
**Scenario: The Problem**

Various users of Queue demand extensions:

- I want Queue to throw exceptions!
- Queue should be thread-safe!
- Please extend the Queue class by an element counter!

---

**The Not So Simple Queue Class**

```java
class Queue {
  Item *first, *last;
  int counter;
  os::Mutex lock;
  public:
  Queue () : first(0), last(0) {
    counter = 0;
  }
  void enQueue(Item* item) {
    lock.enter();
    try {
      if (item == 0) 
        throw QueueInvalidItem();
      if (last) {
        last->next = item;
        last = item;
      } else { last = first = item; }
      ++counter;
      catch (...) {
        lock.leave(); throw;
      }
      lock.leave();
    }
    lock.leave();
    return res;
  }
  int count() { return counter; }
}; // class Queue
```

---

**What Code Does What?**

```java
class Queue {
  Item *first, *last;
  int counter;
  os::Mutex lock;
  public:
  Queue () : first(0), last(0) {
    counter = 0;
  }
  void enQueue(Item* item) {
    lock.enter();
    try {
      if (item == 0) 
        throw QueueInvalidItem();
      if (last) {
        last->next = item;
        last = item;
      } else { last = first = item; }
      ++counter;
      catch (...) {
        lock.leave(); throw;
      }
      lock.leave();
    }
    lock.leave();
    return res;
  }
  int count() { return counter; }
}; // class Queue
```

---

**Problem Summary**

The component code is “polluted” with code for several logically independent concerns, thus it is ...

- hard to write the code
  - many different things have to be considered simultaneously
- hard to read the code
  - many things are going on at the same time
- hard to maintain and evolve the code
  - the implementation of concerns such as locking is scattered over the entire source base (a “crosscutting concern”)
- hard to configure at compile time
  - the users get a “one fits all” queue class
Aspect-Oriented Programming with C++ and AspectC++
AOSD 2004 Tutorial

Part II – AOP with C++

Outline

- We go through the Queue example and...
  - decompose the "one-fits-all" code into modular units
  - apply simple AOP concepts
  - use only C/C++ language idioms and elements

- After we went through the example, we...
  - will try to understand the benefits and limitations of a pure C++ approach
  - link to other related work for advanced separation of concerns in pure C++

Configuring with the Preprocessor?

```cpp
class Queue {
  Item *first, *last;
public:
  Queue();
  ~Queue();
  void enqueue(Item item);
  Item dequeue();
private:
  int size() const;
};
```

Preprocessor

- While we are able to enable/disable features
  - the code is **not expressed** in a modular fashion
  - aspactical code is spread out over the entire code base
  - the code is almost unreadable

- Preprocessor is the "typical C way" to solve problems

- Which C++ mechanism could be used instead?

  **Templates!**
**Templates**

- Templates can be used to construct **generic** code
  - To actually use the code, it has to be **instantiated**
- Just as preprocessor directives
  - Templates are evaluated at compile-time
  - Do not cause any direct runtime overhead (if applied properly)

```c
#define add(T, a, b) 
 ( ((T)a)+((T)b))

template <class T>
T add(T a, T b) { return a + b; }
printf("%d", add(int, 1, 2));
printf("%d", add(int, 1, 2));
```

**Using Templates**

Templates are typically used to implement generic abstract data types:

```c
// Generic Array class
// Elements are stored in a resizable buffer
template <class T>
class Array {
  T* buf; // allocated memory
public:
  T operator[](int i) const {
    return buf[i];
  }
  ...}
```

**AOP with Templates**

- Templates allow us to encapsulate aspect code independently from the component code
- Aspect code is "woven into" the component code by instantiating these templates

```c
// component code
class Queue {
...
  void enqueue(Item* item) {
    if (last) last->next = item; last = item;
    else first = first->item;
  }
}
Item* dequeue() {
  Item* res = first;
  if (first == last) first = last = 0;
  else first = first->next;
  return res;
}
```

**Aspects as Wrapper Templates**

The counting aspect is expressed as a wrapper template class, that derives from the component class:

```c
// generic wrapper (aspect), that adds counting to any queue class
// Q, as long it has the proper interface
template <class Q>
// Q is the component class this
Counting_Aspect : public Q { // aspect should be applied on
  int counter;
public:
  void enqueue(Item* item) { // execute advice code after join point
    Q::enqueue(item); counter++;
  }
  Item* dequeue() { // again, after advice
    Item* res = Q::dequeue(item);
    if (counter > 0) counter--;
    return res;
  }
  // this method is added to the component code (introduction)
  int count() const { return counter; }
};
```
We can define a type alias (typedef) that combines both, component and aspect code (weaving):

```cpp
// component code
class Queue { ... }
// The component (wrapper class)
template <class Q>
class CountingAspect : public Q { ... }
// template instantiation
typedef CountingAspect<Queue> CountingQueue;
int main() {
    CountingQueue q;
    q.add(new Item());
    q.add(new Item());
    printf("number of items in q: \%d", q.count());
    return 0;
}
```

**Our First Aspect – Lessons Learned**

- Aspects can be implemented by template wrappers
  - Aspect inherits from component class, overrides relevant methods
  - Introduction of new members (e.g., counter variable) is easy
  - Weaving takes place by defining (and using) type aliases
- The aspect code is generic
  - It can be applied to "any" component that exposes the same interface (enqueue, dequeue)
  - Each application of the aspect has to be specified explicitly
- The aspect code is clearly separated
  - All code related to counting is gathered in one template class
  - Counting aspect and queue class can be evolved independently (as long as the interface does not change)

**Error Handling Aspect**

Adding an error handling aspect (exceptions) is straightforward. We just need a wrapper template for it:

```cpp
// another aspect (as wrapper template)
template <class Q>
class ExceptionsAspect : public Q {
    void enqueue(Item& item) { // this advice is executed before the
        if (item == 0) throw QueueInvalidItemError();
        Q::enqueue(item);
    }
    Item& dequeue() { // after advice
        Item& res = Q::dequeue();
        if (res == 0) throw QueueEmptyError();
        return res;
    }
}
```

**Combining Aspects**

We already know how to weave with a single aspect. Weaving with multiple aspects is also straightforward:

```cpp
// component code
class Queue { ... }
// wrappers (aspects)
template <class Q>
class CountingAspect : public Q { ... }
template <class Q>
class ExceptionsAspect : public Q { ... }
// template instantiation (weaving)
typedef ExceptionsAspect< CountingAspect< Queue >> ExceptionsCountingQueue;
```
Ordering

> In what order should we apply our aspects?

Aspect code is executed outermost-first:

```cpp
typedef Exceptions_Aspect< // first Exceptions, then Counting
    Counting_Aspect< Queue > > ExceptionsCountingQueue;

typedef Counting_Aspect< // first Counting, then Exceptions
    Exceptions_Aspect< Queue > > ExceptionsCountingQueue;
```

> Aspects should be independent of ordering

- For `dequeue()`, both `Exceptions_Aspect` and `Counting_Aspect` give after advice. Shall we count first or check first?
- Fortunately, our implementation can deal with both cases:

```cpp
    Item* res = Q::dequeue(item);
    // Is ok if we run before Exceptions_Wrapper
    if (counter > 0) counter--;
    return res;
```

Locking Aspect

With what we learned so far, putting together the locking aspect should be simple:

```cpp
template <class Q>
class Locking_Aspect : public Q {
    public:
        Mutex lock;
        void enqueue(Item* item) {
            lock.enter();
            try {
                Q::enqueue(item);
                lock.leave();
            } catch (...) {
                lock.leave();
                throw;
            }
            lock.leave();
        }
}
```

Locking Advice (2)

Locking_Aspect uses an **around advice**, that **proceeds** with the component code in the middle of the aspect code:

```cpp
template <class Q>
class Locking_Aspect : public Q {
    public:
        Mutex lock;
        void enqueue(Item* item) {
            lock.enter();
            try {
                Q::enqueue(item);
            } catch (...) {
                lock.leave();
                throw;
            }
            lock.leave();
        }
}
```

Advice Code Duplication

Specifying the same advice for several **joinpoints** leads to code duplication:

```cpp
template <class Q>
class Locking_Aspect : public Q {
    public:
        Mutex lock;
        void enqueue(Item* item) {
            lock.enter();
            try {
                Q::enqueue(item);
            } catch (...) {
                lock.leave();
                throw;
            }
            lock.leave();
        }
}
```

```cpp
template <class Q>
class Locking_Aspect : public Q {
    public:
        Mutex lock;
        void enqueue(Item* item) {
            lock.enter();
            try {
                Q::enqueue(item);
            } catch (...) {
                lock.leave();
                throw;
            }
            lock.leave();
        }
}
```
Dealing with Joinpoint Sets

To specify advice for a set of joinpoints, the joinpoints must have a uniform interface:

```cpp
template <class Q>
class Locking_Aspect2 { public Q { public: Mutex lock; // wrap joinpoint invocations into action classes struct EnqueueAction { Item* item; void proceed(Q* q) { q->enqueue(item); } }; struct DequeueAction { Item* res; void proceed(Q* q) { res = q->dequeue(); } ...;
```

Reusable Advice Code

The advice code is expressed as template function, which is later instantiated with an action class:

```cpp
template <class Q>
class Locking_Aspect : public Q { ... template <class action> // template inside another template void advice(action* a) { lock.enter(); try { a->proceed(this); } catch (...) { lock.leave(); throw; } lock.leave(); } ...;
```

Binding Advice to Joinpoints

Using the action classes we have created, the advice code is now nicely encapsulated in a single function:

```cpp
template <class Q>
class Locking_Aspect2 : public Q { ... void enqueue(Item* item) { EnqueueAction tjp = {item}; advice(&tjp); } Item* dequeue() { DequeueAction tjp; advice(&tjp); return tjp.res; } ...;
```

Reusing Advice – Lessons Learned

- We avoided advice code duplication, by...
  - delegating the invocation of the original code (proceed) to action classes
  - making the aspect code itself a template function
  - instantiating the aspect code with the action classes

- Compilers will probably generate less efficient code
  - Additional overhead for storing argument/result values
Putting Everything Together

We can now instantiate the combined Queue class, which uses all aspects:
(For just 3 aspects, the typedef is already getting rather complex)

```cpp
typedef Locking_Aspect<Exceptions_Aspect<Counting_Aspect<Queue>>> CountingQueueWithExceptionsAndLocking;
// maybe a little bit more readable ...
typedef Counting_Aspect<Queue> CountingQueue;
typedef Exceptions_Aspect<CountingQueue> CountingQueueWithExceptions;
typedef Locking_Aspect<CountingQueueWithExceptions> CountingQueueWithExceptionsAndLocking;
```

“Obliviousness”

... is an essential property of AOP: the component code should not have to be aware of aspects, but ...
- the extended Queue cannot be named “Queue”
  - our aspects are selected through a naming scheme (e.g. CountingQueueWithExceptionsAndLocking).
- using wrapper class names violates the idea of obliviousness

Preferably, we want to hide the aspects from client code that uses affected components.

Hiding Aspects

- Aspects can be hidden using C++ namespaces
- Three separate namespaces are introduced
  - namespace components: component code for class Queue
  - namespace aspects: aspect code for class Queue
  - namespace configuration: selection of desired aspects for class Queue
- The complex naming schemes as seen on the previous slide is avoided

```cpp
namespace components {
    class Queue { ... };
}
namespace aspects {
    template <class Q>
    class Counting_Aspect : public Q { ... };
}
namespace configuration {
    // select counting queue
    typedef aspects::Counting_Aspect<components::Queue> Queue;
}
```

Hiding Aspects (2)

```cpp
// client code can import configuration namespace and use // counting queue as "Queue"
using namespace configuration;

void client_code () {
    Queue queue; // Queue with all configured aspects
    queue.enqueue (New MyItem);
}
Obliviousness – Lessons Learned

- Aspect configuration, aspect code, and client code can be separated using C++ namespaces
  - name conflicts are avoided
- Except for using the configuration namespace the client code does not have to be changed
  - obliviousness is (mostly) achieved on the client-side

What about obliviousness in the extended classes?

Limitations

For simple aspects the presented techniques work quite well, but a closer look reveals limitations:

- Joinpoint types
  - no distinction between function call and execution
  - no advice for attribute access (as known from AspectJ)
  - no advice for private member functions
- Quantification
  - no flexible way to describe the target components (like AspectJ/AspectC++ pointcuts)
  - applying the same aspect to classes with different interfaces is impossible or ends with excessive template metaprogramming

Limitations (continued)

- Scalibility
  - the wrapper code can easily outweigh the aspect code
  - explicitly defining the aspect order for every affected class is error-prone and cumbersome
  - excessive use of templates and namespaces makes the code hard to understand and debug

“AOP with pure C++ is like OO with pure C”

Conclusions

- C++ templates can be used for separation of concerns in C++ code without special tool support
- However, the lack of expressiveness and scalability restricts these techniques to projects with ...
  - only a small number of aspects
  - few or no aspect interactions
  - aspects with a non-generic nature
  - component code that is “aspect-aware”
- However, switching to tool support is easy!
  - aspects have already been extracted and modularized.
  - transforming template-based aspects to code expected by dedicated AOP tools is only mechanical labor
References/Other Approaches

- A comprehensive analysis of doing AOP with pure C++: what's possible and what not
  - http://www.heise.de/ix/artikel/2001/08/143/

A. Alexandrescu: "Modern C++ Design – Generic Programming and Design Patterns Applied", Addison-Wesley, C++ in depth series, 2001
- Introduces "policy-based design", a technique for advanced separation of concerns in C++
- Policy-based design tries to achieve somewhat similar goals as AOP does
  - http://www.moderncppdesign.com/

Other suggestions towards AOP with pure C++:
- C. Diggins: "Aspect Oriented Programming in C++"
- D. Vollmann: "Visibility of Join-Points in AOP and Implementation Languages"
Part III – Aspect C++

Queue: Demanded Extensions

I. Element counting

II. Error handling
(signaling of errors by exceptions)

III. Thread safety
(synchronization by mutex variables)

Element counting: The Idea

- Increment a counter variable after each execution of util::Queue::enqueue()
- Decrement it after each execution of util::Queue::dequeue()
aspect ElementCounter1 {
    int counter;
    ElementCounter() {
        counter = 0;
    }

    advice execution("% util::Queue::enqueue(...)") : after() {
        ++counter;
        printf(" Aspect ElementCounter: # of elements = %d\n", counter);
    }

    advice execution("% util::Queue::dequeue(...)") : after() {
        if (counter > 0) --counter;
        printf(" Aspect ElementCounter: # of elements = %d\n", counter);
    }
}
ElementCounter1 - Elements

This pointcut expression denotes where the advice should be given. (After execution of methods that match the pattern)

```java
aspect ElementCounter {
    int counter;
    ElementCounter() {
        counter = 0;
    }

decide execution("% util::Queue::enqueue(...)") : after() {
    ++counter;
    printf( " AspectElementCounter: # of elements = %d\n", counter );
}
decide execution("% util::Queue::dequeue(...)") : after() {
    if( counter > 0 ) --counter;
    printf( " AspectElementCounter: # of elements = %d\n", counter );
}
}
```

ElementCounter1.ah

Aspect member elements can be accessed from within the advice body

```java
aspect ElementCounter {
    int counter;
    ElementCounter() {
        counter = 0;
    }

decide execution("% util::Queue::enqueue(...)") : after() {
    ++counter;
    printf( " AspectElementCounter: # of elements = %d\n", counter );
}
decide execution("% util::Queue::dequeue(...)") : after() {
    if( counter > 0 ) --counter;
    printf( " AspectElementCounter: # of elements = %d\n", counter );
}
}
```

ElementCounter1.ah

ElementCounter1 - Result

```c
int main() {
    util::Queue queue;
    printf("main(): enqueueing an item\n");
    queue.enqueue( new util::Item );
    printf("main(): dequeueing two items\n");
    Util::Item item;
    item = queue.dequeue();
    item = queue.dequeue();
}
```

main.cc

```c
main(): enqueueing an item
    > Queue::enqueue(00320FDO)
    < Queue::enqueue(00320FDO)
    Aspects ElementCounter: # of elements = 1
main(): dequeueing two items
    > Queue::dequeque()
    < Queue::dequeque() returning 00320FDO
    Aspects ElementCounter: # of elements = 0
    > Queue::dequeque()
    < Queue::dequeque() returning 00000000
    Aspects ElementCounter: # of elements = 0
```

ElementCounter1 - What's next?

- The aspect is not the ideal place to store the counter, because it is shared between all Queue instances
- Ideally, counter becomes a member of Queue
- In the next step, we
  - move counter into Queue by introduction
  - expose context about the aspect invocation to access the current Queue instance

ElementCounter1 - What's next?

- The aspect is not the ideal place to store the counter, because it is shared between all Queue instances
- Ideally, counter becomes a member of Queue
- In the next step, we
  - move counter into Queue by introduction
  - expose context about the aspect invocation to access the current Queue instance
ElementCounter2 - Elements

```cpp
aspect ElementCounter {
    private:
    advice "util::Queue": int counter;
    public:
    advice "util::Queue": int count { return counter; } const
    advice execution("% util::Queue::enqueue(...)")
        & that(queue) : after(util::Queue& queue) {
            queue.enqueue("a");
            printf("\nQueue contains \%d items\n", queue.count());
        }
    advice execution("% util::Queue::dequeue(...)")
        & that(queue) : before(util::Queue& queue) {
            if (queue.count() > 0) queue.dequeue();
            printf("\nQueue contains \%d items\n", queue.count());
        }
    advice execution("util::Queue::Queue(...)")
        & that(queue) = before(util::Queue& queue) {
            queue.counter = 0;
        }
};
}
```

ElementCounter2 - Result

```cpp
int main() {
    util::Queue queue;
    printf("main(): Queue contains \%d items\n", queue.count());
    queue.enqueue(new UnixFile);  
    queue.enqueue(new UnixFile);  
    printf("main(): Queue contains \%d items\n", queue.count());
    printf("main(): dequeuing one item\n");  
    queue.dequeue();  
    printf("main(): Queue contains \%d items\n", queue.count());
}
```

<Output>
```
main(): Queue contains 0 items
main(): dequeuing some items
> Queue::enqueue(00320F00)
< Queue::enqueue(00320F30)  
main(): Queue contains 2 items
main(): dequeuing one item
> Queue::dequeue()  
< Queue::enqueue(00321000)  
main(): Queue contains 1 items
```
ElementCounter – Lessons Learned

You have seen...
- the most important concepts of AspectC++
  - Aspects are introduced with the keyword `aspect`
  - They are much like a class, may contain methods, data members, types, inner classes, etc.
  - Additionally, aspects can give advice to be woven in at certain positions (joinpoints). Advice can be given to
    - Functions/Methods/Constructors: code to execute (code advice)
    - Classes or namespaces: new elements (introductions)
  - Joinpoints are described by pointcut expressions
- We will now take a closer look at some of them

Joinpoints

- A joinpoint denotes a position to give advice
  - **Code** joinpoint
    a point in the control flow of a running program, e.g.
    - execution of a function
    - call of a function
  - **Name** joinpoint
    a named C++ program entity (identifier)
    - class, function, method, type, namespace
- Joinpoints are given by pointcut expressions
  - a pointcut expression describes a set of joinpoints

Pointcut Expressions

- Pointcut expressions are made from ...
  - **match expressions**, e.g. "% util::queue::enqueue( ... )"
    - are matched against C++ program entities → name joinpoints
    - support wildcards
  - **pointcut functions**, e.g. `execution(...)`, `call(...)`, `that(...)`, `within(...)`
    - execution: all points in the control flow, where a function is about to be executed → code joinpoints
    - call: all points in the control flow, where a function is about to be called → code joinpoints
- Pointcut functions can be combined into expressions
  - using logical connectors: &&, ||, !
  - Example: `call("% util::queue::enqueue( ... )") && within("% main( ... )")`
Advice

Advice to functions
- **before advice**
  - Advice code is executed before the original code
  - Advice may read/modify parameter values
- **after advice**
  - Advice code is executed after the original code
  - Advice may read/modify return value
- **around advice**
  - Advice code is executed instead of the original code
  - Original code may be called explicitly: `tjp->proceed()`

Introductions
- Additional methods, data members, etc. are added to the class
- Can be used to extend the interface of a class or namespace

Before / After Advice

with execution joinpoints:
```cpp
class ClassA {
  public:
    void foo(){
      printf("ClassA::foo()\n");
    }
}
```

```cpp
decision execution("void ClassA::foo()": before){
  printf("ClassA::foo()\n");
}
decision execution("void ClassA::foo()": after){
}
```

with call joinpoints:
```cpp
int main(){
  printf("main()\n");
  ClassA a;
  a.foo();
}
```

```cpp
decision call("void ClassA::foo()": before){
}
decision call("void ClassA::foo()": after){
}
```

Around Advice

with execution joinpoints:
```cpp
decision execution("void ClassA::foo()": around){
  before code
  tjp->proceed()
  after code
}
```

Introductions

```cpp
private:
  decision advice "ClassA": element to introduce
```

```cpp
class ClassA {
  public:
    void foo(){
      printf("ClassA::foo()\n");
    }
}
```

```cpp
public:
  decision advice "ClassA": element to introduce
```

```cpp
int main(){
  printf("main()\n");
  ClassA a;
  a.foo();
}
```
Queue: Demanded Extensions

I. Element counting

II. Error handling
   (signaling of errors by exceptions)

III. Thread safety
   (synchronization by mutex variables)

---

Error Handling: The Idea

- We want to check the following constraints:
  - enqueue() is never called with a NULL item
  - dequeue() is never called on an empty queue
- In case of an error an exception should be thrown

- To implement this, we need access to ...
  - the parameter passed to enqueue()
  - the return value returned by dequeue() ...
  - from within the advice

---

```
namespace util {
    struct QueueInvalidItemError {};
    struct QueueEmptyError {};
}

aspect ErrorException {
    advice execution("% util::Queue::enqueue(...)") && args(item) {
        before(util::Item* item) {
            if (item == 0) {
                throw util::QueueInvalidItemError();
            }
        }
    }
    advice execution("% util::Queue::dequeue(...)") && result(item) {
        after(util::Item* item) {
            if (item == 0) {
                throw util::QueueInvalidItemError();
            }
        }
    }
}
```

---

```
namespace util {
    struct QueueInvalidItemError {};
    struct QueueEmptyError {};
}

aspect ErrorException {
    advice execution("% util::Queue::enqueue(...)") && args(item) {
        before(util::Item* item) {
            if (item == 0) {
                throw util::QueueInvalidItemError();
            }
        }
    }
    advice execution("% util::Queue::dequeue(...)") && result(item) {
        after(util::Item* item) {
            if (item == 0) {
                throw util::QueueEmptyError();
            }
        }
    }
}
```
Error Exception - Elements

```cpp
namespace util {  
    struct QueueValidItem {  
        A context variable item is bound to the first argument of type util::Item* passed to the matching methods  
    };
}

aspect ErrorException {  
    advice execution("% util::Queue::enqueue(...)") & args(item)  
        before(util::Item* item) {  
            if (item == 0)  
                throw util::QueueValidItemError();  
        }
        advice execution("% util::Queue::dequeue(...)") & result(item)  
        after(util::Item* item) {  
            if (item == 0)  
                throw util::QueueValidItemError();  
    }
};
```

Error Exception - Elements

```cpp
namespace util {  
    struct QueueValidItem {  
        Here the context variable item is bound to the result of type util::Item* returned by the matching methods  
    };
}

aspect ErrorException {  
    advice execution("% util::Queue::enqueue(...)") & args(item)  
        before(util::Item* item) {  
            if (item == 0)  
                throw util::QueueValidItemError();  
        }
        advice execution("% util::Queue::dequeue(...)") & result(item)  
        after(util::Item* item) {  
            if (item == 0)  
                throw util::QueueValidItemError();  
    }
};
```

Error Exception – Lessons Learned

You have seen how to ...

- use different types of advice
  - before advice
  - after advice

- expose context in the advice body
  - by using args to read/modify parameter values
  - by using result to read/modify the return value

Queue: Demanded Extensions

I. Element counting

II. Error handling
   (signaling of errors by exceptions)

III. Thread safety
    (synchronization by mutex variables)
Thread Safety: The Idea

- Protect enqueue() and dequeue() by a mutex object

- To implement this, we need to
  - introduce a mutex variable into class Queue
  - lock the mutex before the execution of enqueue() / dequeue()
  - unlock the mutex after execution of enqueue() / dequeue()

- The aspect implementation should be exception safe!
  - in case of an exception, pending after advice is not called
  - solution: use around advice

LockingMutex

```java
aspect LockingMutex {
  advice "util::Queue": os::Mutex lock;
  pointcut sync_methods() = "% util::Queue::Queue(...)";

  advice execution(sync_methods()) & that(queue) : around( util::Queue& queue ) {
    queue.lock.enter();
    try {
      tjp->proceed();
    } catch(...) {
      queue.lock.leave();
      throw;
    }
    queue.lock.leave();
  };
}
```

LockingMutex - Elements

```java
aspect LockingMutex {
  advice "util::Queue": os::Mutex lock;
  pointcut sync_methods() = "% util::Queue::Queue(...)";

  advice execution(sync_methods()) & that(queue) : around( util::Queue& queue ) {
    queue.lock.enter();
    try {
      tjp->proceed();
    } catch(...) {
      queue.lock.leave();
      throw;
    }
    queue.lock.leave();
  };
}
```

We introduce a mutex member into class Queue
LockingMutex - Elements

```java
aspect LockingMutex {
  advice "util::Queue": os::Mutex lock;
  pointcut sync_methods() = "% util::Queue::queue(...)";

  advice execution(sync_methods()) & that(queue)
  : around( util::Queue& queue ) {
    try {
      tjp->proceed();
    } catch(...) {
      queue.lock.leave();
      throw;
    }
    queue.lock.leave();
  }
};
```

LockingMutex.ah

LockingMutex - Elements

```java
aspect LockingMutex {
  advice "util::Queue": os::Mutex lock;
  pointcut sync_methods() = "% util::Queue::queue(...)";

  advice execution(sync_methods()) & that(queue)
  : around( util::Queue& queue ) {
    queue.lock.enter();
    try {
      tjp->proceed();
    } catch(...) {
      queue.lock.leave();
      throw;
    }
    queue.lock.leave();
  }
};
```

By calling `tjp->proceed()` the original method is executed.

LockingMutex.ah

LockingMutex – Lessons Learned

You have seen how to ... 

- use named pointcuts
  - to increase readability of pointcut expressions
  - to reuse pointcut expressions
- use around advice
  - to deal with exception safety
  - to explicit invoke (or don't invoke) the original code by calling `tjp->proceed()`
- use wildcards in match expressions
  - `"% util::Queue::queue(...)"` matches both `enqueue()` and `dequeue()`

Scenario (A new requirement)

Requirements tend to change...

- Please extend the Queue class by an element counter.
- Queue should be thread-safe.
- We need `Queue` to be synchronized on interrupt level!
Queue: Demanded Extensions

I. Element counting

II. Error handling
   (signaling of errors by exceptions)

III. Thread safety
   (synchronization by mutex variables)

IV. Interrupt safety
   (synchronization on interrupt level)

Interrupt Safety: The Idea

- Scenario
  - Queue is used to transport objects between
    kernel code (interrupt handlers) and application code
  - If application code accesses the queue, interrupts must be disabled first
  - If kernel code accesses the queue, interrupts must not be disabled

- To implement this, we need to distinguish
  - if the call is made from kernel code, or
  - if the call is made from application code

LockingIRQ

aspect LockingIRQ {
  pointcut sync_methods() = "% util::Queue::queue(...)";
  pointcut kernel_code() = "% kernel::%(...)";
  advice call(sync_methods()) & & !within(kernel_code()) : around() {
    os::disable_int();
    try {
      tjp->proceed();
    } catch(...) {
      os::enable_int();
      throw;
    } os::enable_int();
  }
};

LockingIRQ1

aspect LockingIRQ1 {
  pointcut sync_methods() = "% util::Queue::queue(...)";
  pointcut kernel_code() = "% kernel::%(...)";
  advice call(sync_methods()) & & !within(kernel_code()) : around() {
    os::disable_int();
    try {
      tjp->proceed();
    } catch(...) {
      os::enable_int();
      throw;
    } os::enable_int();
  }
};

We define two pointcuts. One for the methods to be synchronized and one for all kernel functions.
LockingIRQ1 – Elements

```c
aspect LockingIRQ {
    pointcut sync_methods() = “% util::Queue::Queue(...)”;
    pointcut kernel_code() = “% kernel:K(...)”;
    advice call(sync_methods()) & l within(kernel_code()) : around()
    if (tjp->proceed());
    catch(...) {}
    os::disable_int();
    throw;
    os::enable_int();
};
```

LockingIRQ1.ah

This pointcut expression matches any call to a sync_method that is not done from kernel_code.

LockingIRQ1 – Result

```c
util::Queue queue;
void do_something()
    if (queue.enqueue(new util::Item));

namespace kernel {
    void irq_handler()
        queue.enqueue(new util::Item);
do_something();
}

int main()
    if (queue.enqueue(new util::Item);
        kernel:irq_handler(); // irq
        printf("back in main\n\n");
        queue.dequeue();
};
```

main.cc

LockingIRQ1 – Problem

```c
util::Queue queue;
void do_something()
    if (queue.enqueue(new util::Item));

namespace kernel {
    void irq_handler()
        queue.enqueue(new util::Item);
do_something();
}

int main()
    if (queue.enqueue(new util::Item);
        kernel:irq_handler(); // irq
        printf("back in main\n\n");
        queue.dequeue();
};
```

main.cc

The pointcut within(kernel_code) does not match any indirect calls to sync_methods.

LockingIRQ2

```c
aspect LockingIRQ {
    pointcut sync_methods() = “% util::Queue::Queue(...)”;
    pointcut kernel_code() = “% kernel:K(...)”;
    advice execution(sync_methods()) & l within(kernel_code()) : around()
    if (tjp->proceed());
    catch(...) {}
    os::disable_int();
    throw;
    os::enable_int();
};
```

LockingIRQ2.ah

Solution Using the cflow pointcut function.
LockingIRQ2 – Elements

```c
aspect LockingIRQ {
    pointcut sync_methods() = "% util::Queue::enqueue(...)";
    pointcut kernel_code() = "% kernel::%(...)";
    advice execution(sync_methods())
        & cflow(kernel_code()) ; aroundC() {
        os::disable_int();
        try {
            tjp->proceed();
        } catch(...) {
            os::enable_int();
            throw;
        } os::enable_int();
    }
}
```

This pointcut expression matches the execution of `sync_methods` if no `kernel_code` is on the call stack. `cflow` checks the call stack (control flow) at runtime.

LockingIRQ2.ah

LockingIRQ2 – Result

```c
util::Queue queue;
void do_something() {
    printf("do_something()\n");
    queue.enqueue(new util::Item);
    namespace kernel {
        void irq_handler () {
            printf("kernel::irq_handler()\n");
            queue.enqueue(new util::Item);
            do_something();
        }
    }
    int main() {
        printf("main()\n");
        queue.enqueue(new util::Item);
        kernel::irq_handler(); // irq
        printf("back in main()\n");
        queue.dequeue();
    }
}
```

main.cc

AspectC++: A First Summary

> The Queue example has presented the most important features of the AspectC++ language
> - aspect, advice, joinpoint, pointcut expression, pointcut function, ...
> - Additionaly, AspectC++ provides some more advanced concepts and features
  > - to increase the expressive power of aspectual code
  > - to write broadly reusable aspects
  > - to deal with aspect independence and ordering

> In the following, we give a short overview on these advanced language elements

LockingIRQ – Lessons Learned

You have seen how to...

- restrict advice invocation to a specific calling context
- use the within(...) and cflow(...) pointcut functions
  - `within` is evaluated at compile time and returns all code joinpoints of a class' or namespaces lexical scope
  - `cflow` is evaluated at runtime and returns all joinpoints where the control flow is below a specific code joinpoint
AspectC++: Advanced Concepts

- **The Joinpoint API**
  - provides a uniform interface to the aspect invocation context, both at runtime and compile-time
- **Abstract Aspects and Aspect Inheritance**
  - comparable to class inheritance, aspect inheritance allows to reuse parts of an aspect and overwrite other parts
- **Aspect Ordering**
  - allows to specify the invocation order of multiple aspects
  - important in the case of inter-aspect dependencies
- **Aspect Instantiation**
  - allows to implement user-defined aspect instantiation models (default: singleton), e.g. per thread, per client

The Joinpoint API

- Inside an advice body, the current joinpoint context is available via the implicitly passed `tjp` variable:
  ```
  advice ... {
    struct JoinPoint {
      ...
      *tjp; // implicitly available in advice code
      ...
    }
  }
  ```
- You have already seen how to use `tjp`, to...
  - execute the original code in around advice with `tjp->proceed()`
- The joinpoint API provides a rich interface
  - to expose context independently of the aspect target
  - this is especially useful in writing reusable aspect code

Abstract Aspects and Inheritance

- Aspects can inherit from other aspects...
  - Reuse aspect definitions
  - Override aspect pointcuts
- Pointcuts can be pure virtual
  - Postpone the concrete definition to derived aspects
  - An aspect with a pure virtual pointcut is called abstract aspect
- Common usage: Reusable aspect implementations
  - Abstract aspect defines advice code, but pure virtual pointcuts
  - Aspect code uses the joinpoint API to expose context
  - Concrete aspect inherits the advice code and overrides pointcuts
Abstract Aspects and Inheritance

```c
#include "mutex.h"
aspect LockingA {
    pointcut virtual sync_classes() = 0;
    pointcut virtual sync_methods() = 0;
    advice sync_classes() : os::Mutex lock;

    advice execution(sync_methods()) : around() {
        tjp->that()->lock.enter();
        try {
            tjp->proceed();
        }
        catch(...) {
            tjp->that()->lock.leave();
        throw;
        }
        tjp->that()->lock.leave();
    }
};

aspect LockingQueue : public LockingA {
    pointcut sync_classes() = "% util::Queue::queue(...)";
    pointcut sync_methods() = "% util::Queue::Queue(...)";
};
```

This abstract locking aspect declares two pure virtual pointcuts and uses the joinpoint API for an context-independent advice implementation.

Abstract Aspects and Inheritance

```c
#include "mutex.h"
aspect LockingA {
    pointcut virtual sync_classes() = 0;
    pointcut virtual sync_methods() = 0;
    advice sync_classes() : os::Mutex lock;

    advice execution(sync_methods()) : around() {
        tjp->that()->lock.enter();
        try {
            tjp->proceed();
        }
        catch(...) {
            tjp->that()->lock.leave();
        throw;
        }
        tjp->that()->lock.leave();
    }
}
```

The concrete locking aspect derives from the abstract aspect and overrides the pointcuts.

Aspect Ordering

- Aspects should be independent of other aspects
  - However, sometimes inter-aspect dependencies are unavoidable
    - Example: Locking should be activated before any other aspects
- Order advice
  - The aspect order can be defined by `order advice`
    - `advice pointcut-exp : order(high,..., low)`
  - Different aspect orders can be defined for different pointcuts
- Example
  ```c
  advice "% util::Queue::queue(...)"
     : order("LockingIRQ", "%" && !"LockingIRQ" );
  ```

Aspect Instantiation

- Aspects are singletons by default
  - `aspectof()` returns pointer to the one-and-only aspect instance
- By overriding `aspectof()` this can be changed
  - e.g. one instance per client or one instance per thread

```c
aspect MyAspect {
    // ...
    static MyAspect* aspectof() {
        static _declspec(thread) MyAspect* theAspect;
        if( theAspect == 0 )
            theAspect = new MyAspect;
        return theAspect;
    }
}
```

Example of an user-defined `aspectof()` implementation for per-thread aspect instantiation by using thread-local storage.

(Visual C++)
Summary

- AspectC++ facilitates AOP with C++
  - AspectJ-like syntax and semantics
- Full obliviousness and quantification
  - aspect code is given by advice
  - joinpoints are given declaratively by pointcuts
  - implementation of crosscutting concerns is fully encapsulated in aspects
- Good support for reusable and generic aspect code
  - aspect inheritance and virtual pointcuts
  - rich joinpoint API

And what about tool support?
About ac++

- Available from [www.aspectc.org](http://www.aspectc.org)
  - Linux, Win32, Solaris, MacOS X binaries + source (GPL)
  - documentation: Compiler Manual, Language Reference, ...
- Transforms AspectC++ to C++ code
  - machine code is created by the back-end (cross-)compiler
  - supports g++ and Visual C++ specific language extensions
- Current version < 1.0
  - no optimizations for compilation speed
  - no weaving in templates
  - but already more than a proof of concept, examples follow
Aspect Transformation

```cpp
aspect Transform {
    advice call("% foo()") : before() {
        printf("before foo call\n");
    }
    advice execution("% C::%()") : after() {
        printf(tjp->signature());
    }
};

class Transform {
    static Transform __instance;
    // ...
    void __a0_before () {
        printf("before foo call\n");
    }
    template<class JoinPoint>
    void __a1_after (JoinPoint *tjp) {
        printf (tjp->signature());
    }
};
```

Tool Support

IV/5

Aspect Transformation

```cpp
aspect Transform {
    advice call("% foo()") : before() {
        printf("before foo call\n");
    }
    advice execution("% C::%()") : after() {
        printf(tjp->signature());
    }
};

class Transform {
    static Transform __instance;
    // ...
    void __a0_before () {
        printf("before foo call\n");
    }
    template<class JoinPoint>
    void __a1_after (JoinPoint *tjp) {
        printf (tjp->signature());
    }
};
```

Tool Support

IV/6

Aspect Transformation

```cpp
aspect Transform {
    advice call("% foo()") : before() {
        printf("before foo call\n");
    }
    advice execution("% C::%()") : after() {
        printf(tjp->signature());
    }
};

class Transform {
    static Transform __instance;
    // ...
    void __a0_before () {
        printf("before foo call\n");
    }
    template<class JoinPoint>
    void __a1_after (JoinPoint *tjp) {
        printf (tjp->signature());
    }
};
```

Tool Support

IV/7

Aspect Transformation

```cpp
aspect Transform {
    advice call("% foo()") : before() {
        printf("before foo call\n");
    }
    advice execution("% C::%()") : after() {
        printf(tjp->signature());
    }
};

class Transform {
    static Transform __instance;
    // ...
    void __a0_before () {
        printf("before foo call\n");
    }
    template<class JoinPoint>
    void __a1_after (JoinPoint *tjp) {
        printf (tjp->signature());
    }
};
```

Tool Support

IV/8
Joinpoint Transformation

```
int main() {
    foo();
    return 0;
}
```

```
int main() {
    struct __call_main_0_0 {
        static inline void invoke () {
            AC::a0_before ();
            ::foo();
        }
    };
    __call_main_0_0::invoke ();
    return 0;
}
```

Joinpoint Transformation

```
int main() {
    foo();
    return 0;
}
```

```
int main() {
    struct __call_main_0_0 {
        static inline void invoke () {
            AC::a0_before ();
            ::foo();
        }
    };
    __call_main_0_0::invoke ();
    return 0;
}
```

Joinpoint Transformation

```
int main() {
    foo();
    return 0;
}
```

```
int main() {
    struct __call_main_0_0 {
        static inline void invoke () {
            AC::a0_before ();
            ::foo();
        }
    };
    __call_main_0_0::invoke ();
    return 0;
}
```

Translation Modes

- **Whole Program Transformation-Mode**
  - e.g. ac++ -p src -d gen -e cpp -Iinc -DDEBUG
  - transforms whole directory trees
  - generates manipulated headers, e.g. for libraries
  - can be chained with other whole program transformation tools

- **Single Translation Unit-Mode**
  - e.g. ac++ -c a.cc -o a-gen.cc -p
  - easier integration into build processes
  - more precise dependency handling possible
Tool Demos

- AspectC++ Add-In for Microsoft® Visual Studio®
  - by pure-systems GmbH (www.pure-systems.com)

- AspectC++ plugin for Eclipse®
  - recently started open source development effort

Summary

- Tool support for AspectC++ programming is based on the ac++ command line compiler
  - full “obliviousness and quantification”
  - delegates the binary code generation to your favorite compiler

- There is a commercial and a non-commercial IDE integration available
  - advice visualization is an essential feature for AOP

- There is still a lot of work, but it’s a start!
Part V – Examples

Redundancy Management (old)

// global data the must be kept consistent on the standby host
enum { IDLE, ORIG, BUSY };
type def struct {
  int call_id;
  int call_status; // IDLE, ORIG, or BUSY
  Job_Type;
  Job_Type JobBlk[MAX_CALLS];
  int no_of_calls;
}

// one of many functions manipulating the state
void call_origin (int id) {
  JobBlk[no_of_calls].call_id = id;
  JobBlk[no_of_calls].call_status = ORIG;
  // inform the redundancy management system about the changes
  rmsBackupData ((unsigned long)&JobBlk[no_of_calls], sizeof(Job_Type));
  no_of_calls++;
  rmsBackupData ((unsigned long)&no_of_calls, sizeof(no_of_calls));
  // update the standby node
  rmsCommitData ();
}

Redundancy Management (old)

// global data the must be kept consistent on the standby host
enum { IDLE, ORIG, BUSY };
type def struct {
  int call_id;
  int call_status; // IDLE, ORIG, or BUSY
  Job_Type;
  Job_Type JobBlk[MAX_CALLS];
  int no_of_calls;
}

// one of many functions manipulating the state
void call_origin (int id) {
  JobBlk[no_of_calls].call_id = id;
  JobBlk[no_of_calls].call_status = ORIG;
  // inform the redundancy management system about the changes
  rmsBackupData ((unsigned long)&JobBlk[no_of_calls], sizeof(Job_Type));
  no_of_calls++;
  rmsBackupData ((unsigned long)&no_of_calls, sizeof(no_of_calls));
  // update the standby node
  rmsCommitData ();
  Calls to rms" crosscut the whole application. The ‘update policy’ is hardwired.
Developer Requirements

Find a way to...

- get rid of these rms* calls
- implement different update policies
  - without needing to change the whole application
  - just to play with the performance

but without...

- other changes in functions and data structures
  - no changes to the C-like program structure!
- a significant overhead

Redundancy Management (new)

```c
// global data the must be kept consistent on the standby host
enum { IDLE, ORIG, BUSY };
typedef struct {
  RedInt call_id;
  RedInt call_status; // IDLE, ORIG, or BUSY
} Job_Type;
Job_Type JobBlk[MAX_CALLS];
RedInt no_of_calls;

// one of many functions manipulating the state
void call originate (int id) { 
  JobBlk[no_of_calls].call_id = id;
  JobBlk[no_of_calls].call_status = ORIG;
  no_of_calls++;
}
```

Update Policy Aspect

```c
aspect RMS_function_based_with_modification_check {
  bool modified;
  protected:
    virtual transactions () = 0;
    // commit after each function execution that modified something
    advice execution (transactions () : around () { 
      modified = false;
      tjp->proceed ();
      if (modified)
        rmsCommitData ();
    } // call rmsBackupData after every modification and set flag
    advice execution ("void RedInt::set(...)") & that (redint) : 
      after (RedInt &redint) { 
        modified = true;
        rmsBackupData ((unsigned long)&redint, sizeof (RedInt));
      };
}
```
Example Summary

- An existing (C-style) code base was refactored
- Programmers were freed from the burden of a global policy
  - errors are avoided, e.g. forgetting the redundancy management
- The update policy is now modularized
  - can be configured at compile time or runtime
  - can be changed easily
  - specialists can concentrate on policy development
  - the implementation better reflects the design
- An overhead was introduced (more rmsBackupData calls)
  - ... but was considered acceptable

Weather Station Example: Platform

Weather Station Variants

- Thermometer: LCD, Temperature
- Home use: LCD, Temperature, Pressure
- Outdoor use: LCD, Temp., Pressure, Wind
- Deluxe outdoor: + PC data recording
- Networked: + TCP/IP
- Serial connection option
- USB connection option
- ...

Functional Decomposition

```c
int main() {
    init();
    while (1) {
        measure();
        wait();
    }
}
```
The ‘Display' Concern

... *crosscuts* the module structure!

The 'Trace' Concern

... *crosscuts* the module structure!

Separation of Concerns by AOP

The Display Aspect

```cpp
aspect Display : public LCD_OStream {
    uint8 _line;
    void prepare (const char *name, const char *unit);
    advice execution("% main(...)") : before () {
        command (LCD_ON);
        "this << "AOSD 2004";
    }
    advice execution("% Measurement::execute()") : before () {
        _line = 1;
    }
    advice execution("% Sensor::%:measure()") : after () {
        prepare (JoinPoint:: THAT::name (), JoinPoint:: THAT::unit ());
        "this << *tp->result () ;
    };
}
```

The Display Aspect

```
aspect Display : public LCD_Output {
    uint8 _line;
    void prepare (const char *name, const char *unit);

    advice execution("% main(...)") : before () {
        command (LCD_ON);
        "this << "A0SD 2004";
    }
    advice execution("% Measurement::execute()") : before () {
        _line -= 1;
    }
    advice execution("% Sensor::%::measure()") : after () {
        prepare (JoinPoint::That::name (), JoinPoint::That::unit ());
        "this << "tjp->result () ;
    }
};
```

AOP Benefits for Product Lines

- Better modularity:
  - Reduced number of configuration points in case of crosscutting concerns

- Reuse:
  - The same aspect can be used for various component code configurations

- Separation of Concerns:
  - Static configuration of aspects is a very powerful mechanism

Code Sizes (in Bytes)
Aspect-Oriented Programming with C++ and AspectC++
AOSD 2004 Tutorial

Part VI – Summary

Pros and Cons
AOP with pure C++
+ no special tool required
- requires in-depth understanding of C++ templates
- lack of “obliviousness” the component code has to be aspect-aware
- lack of “quantification” no pointcut concept, no match expressions

AspectC++
+ the ac++ compiler transforms AspectC++ into C++
+ various supported joinpoint types, e.g. execution and calls
+ built-in support for advanced AOP concepts: cflow, joinpoint-API
- longer compilation times

Summary – This Tutorial ...
- showed basic techniques for AOP with pure C++
  - using templates to program generic wrapper code
  - using action classes to encapsulate the “proceed-code”
  - using namespaces to substitute types transparently
- introduced the AspectC++ language extension for C++
  - AspectJ-like language extension
  - ac++ transforms AspectC++ into C++
  - supports AOP even in resource constrained environments
- demonstrated the AspectC++ tools
- discussed the pros and cons of each approach

Future Work – Roadmap
- Parser improvements
  - full template support
  - speed optimization
  - full g++ 3.x and Visual C++ compatibility
- Language design/weaver
  - weaving in templates
  - advice for object access (set/get pointcut functions)
  - advice for object instantiations
- Tools
  - dependency handling
Thank you for your attention!