

Konfigurierbare Systemsoftware (KSS)

VL 3 – Aspect-Oriented Programming (AOP)

Daniel Lohmann

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

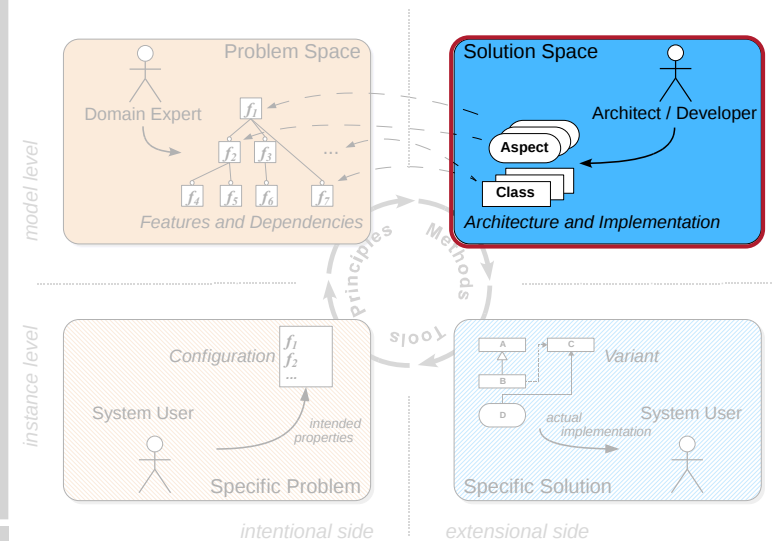
Friedrich-Alexander-Universität
Erlangen-Nürnberg

SS 13 – 2013-05-02

http://www4.informatik.uni-erlangen.de/Lehre/SS13/V_KSS



About this Lecture

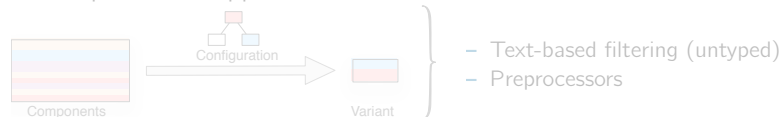


© dl KSS (VL 3 | SS 13) 3 AOP

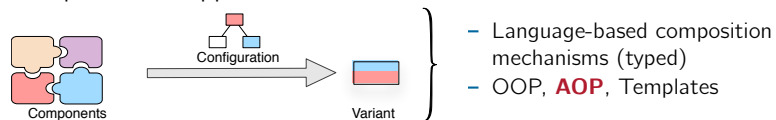
3-2

Implementation Techniques: Classification

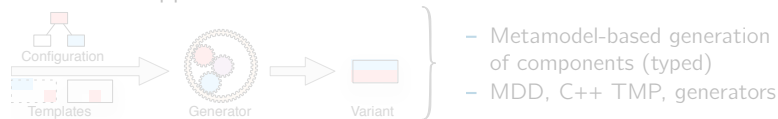
■ Decompositional Approaches



■ Compositional Approaches



■ Generative Approaches



© dl KSS (VL 3 | SS 13) 3 AOP

3-3

Agenda

- 3.1 Motivation: Separation of Concerns
- 3.2 Tutorial: AspectC++
- 3.3 Summary and Outlook
- 3.4 References



© dl KSS (VL 3 | SS 13) 3 AOP

3-4

Agenda

3.1 Motivation: Separation of Concerns

Example: I4WeatherMon

Example: eCos

AOP

Static Configurability with the CPP?

14WeatherMon (C++): Implementation (Excerpt)

l4WeatherMon
example from
last lecture

[illegible]

Case Study eCos [4]



- The **embedded Configurable operating system**
 - Operating system for embedded applications
 - Open source, maintained by eCosCentric Inc.
 - Many 16-bit and 32-bit platforms supported
 - Broadly accepted real-world system
- More than **750** configuration options (kernel)
 - Feature-based selection
 - **Preprocessor-based** implementation



Static Configurability with the CPP?

```

Cyg_Mutex() {
    CYG_REPORT_FUNCTION();
    locked      = false;
    owner       = NULL;

    #if defined(CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_DEFAULT) && \
    defined(CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_DYNAMIC)
    #ifdef CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_DEFAULT_INHERIT
        protocol = INHERIT;
    #endif
    #ifdef CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_DEFAULT_CEILING
        protocol = CEILING;
        ceiling  = CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_DEFAULT_PRIORITY;
    #endif
    #ifdef CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_DEFAULT_NONE
        protocol = NONE;
    #endif
    #else // not (DYNAMIC and DEFAULT defined)
    #ifdef CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_CEILING
    #ifdef CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_DEFAULT_PRIORITY
        // if there is a default priority ceiling defined, use that to initialize
        // the ceiling.
        ceiling = CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_DEFAULT_PRIORITY;
    #else
        // Otherwise set it to zero.
        ceiling = 0;
    #endif
    #endif
    #endif // DYNAMIC and DEFAULT defined
    CYG_REPORT_RETURN();
}

```

Mutex options:

PROTOCOL

CEILING

INHERIT

DYNAMIC

Kernel policies:

Tracing

Instrumentation

Synchronization

Static Configurability with the CPP?

```

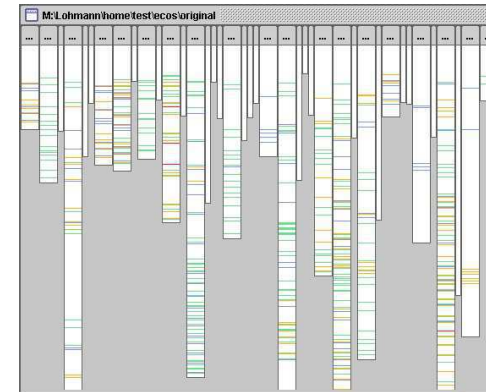
Cyg_Mutex::Cyg_Mutex() {
  CYG_REPORT_FUNCTION();
  locked = false;
  owner = NULL;
  #if defined(CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_DEFAULT) && \
  defined(CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_DYNAMIC)
  #ifdef CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_DEFAULT_INHERIT
    protocol = INHERIT;
  #endif
  #ifdef CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_DEFAULT_CEILING
    protocol = CEILING;
    ceiling = CYGSEM_...
  #endif
  #ifdef CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_DEFAULT_PRIORITY
    protocol = CEILING;
    ceiling = CYGSEM_...
  #endif
  #ifdef CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_DEFAULT_PRIORITY
    // if there is a default priority ceiling defined, use that to initialize
    // the ceiling.
    ceiling = CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_DEFAULT_PRIORITY;
  #else
    // Otherwise set it to zero.
    ceiling = 0;
  #endif
  #endif
  #endif // DYNAMIC and DEFAULT defined
  CYG_REPORT_RETURN();
}
  
```

Mutex options:

PROTOCOL
CEILING
INHERIT
DYNAMIC

Kernel policies: Tracing Instrumentation Synchronization

Static Configurability with the CPP?



Mutex options:

PROTOCOL
CEILING
INHERIT
DYNAMIC

Kernel policies: Tracing Instrumentation Synchronization

Static Configurability with the CPP?



Mutex options:

PROTOCOL
CEILING
INHERIT
DYNAMIC

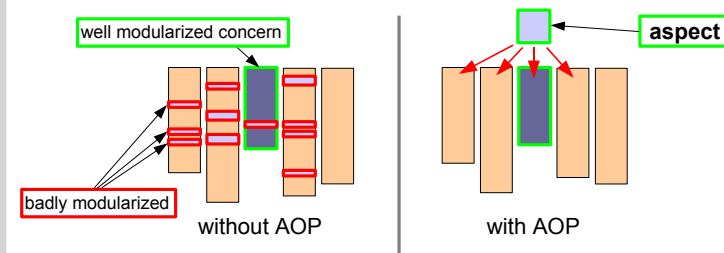
Issue
Crosscutting Concerns

Kernel policies: Tracing Instrumentation Synchronization

Aspect-Oriented Programming (AOP) [2]

Aspect-Oriented Programming

- AOP is about modularizing crosscutting concerns



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

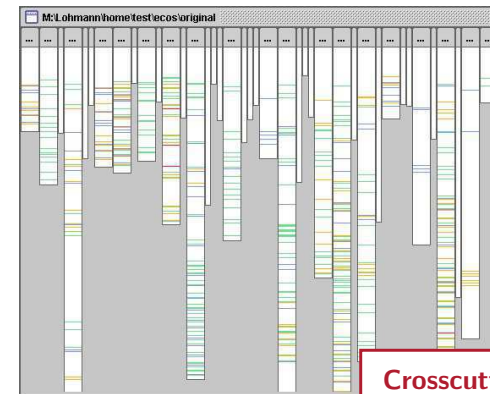
AOP: The Basic Idea

Separation of **what** from **where**:

- **Join Points** $\mapsto$ **where**
 - positions in the static structure or dynamic control flow (event)
 - given declaratively by pointcut expressions
- **Advice** $\mapsto$ **what**
 - additional elements (members, ...) to introduce at join points
 - additional behavior (code) to superimpose at join points



Static Configurability with the CPP?



Mutex options:

PROTOCOL

CEILING

INHERIT

DYNAMIC

Crosscutting Concerns

Can we do better with aspects?

Kernel policies:

Tracing

Instrumentation

Synchronization



Implementation of Crosscutting Concerns with AOP

```
aspect int_sync {
```

```
    pointcut sync() = execution(...) // kernel calls to sync
    || construction(...)
    || destruction(...);
```

where

```
    // advise kernel code to invoke lock() and unlock()
```

```
    advice sync() : before() {
```

```
        Cyg_Scheduler::lock();
```

```
    }
```

```
    advice sync() : after() {
```

```
        Cyg_Scheduler::unlock();
```

```
    }
```

```
    // In eCos, a new thread always starts with a lock value of 0
```

```
    advice execution(
```

```
        "%Cyg_HardwareThread::thread_entry(...)" : before() {
```

```
        Cyg_Scheduler::zero_sched_lock();
```

```
    }
```

```
    ...
```

```
};
```

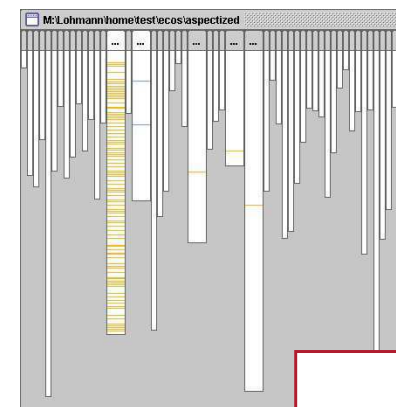
what



Synchronization



Static Configurability with the CPP?



Result

after refactoring into aspects [4]

Kernel policies:

Tracing

Instrumentation

Synchronization



- AspectC++ is an AOP language extension for C++
 - superset of ISO C++ 98 [1]
 - ↪ every C++ program is also an AspectC++ program
 - additionally supports AOP concepts
- Technical approach: source-to-source transformation
 - ac++ weaver transforms AspectC++ code into C++ code
 - resulting C++ code can be compiled with any standard-compliant compiler (especially gcc)
 - ag++ weaver wrapper works as replacement for g++ in makefiles
- Language and weaver are open source (GPL2)



<http://www.aspectc.org>



3.1 Motivation: Separation of Concerns

3.2 Tutorial: AspectC++

Example Scenario

First Steps And Language Overview

Advanced Concepts

Weaver Transformations

Further Examples

3.3 Summary and Outlook

3.4 References

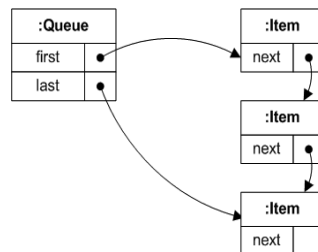


Scenario: A Simple Queue

Scenario: A Queue utility class

util::Queue
-first : util::Item
-last : util::Item
+enqueue(in item : util::Item)
+dequeue() : util::Item

util::Item
-next



Scenario: A Simple Queue

The Simple Queue Class

```

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
            last = first = item;
        printf( " < Queue::enqueue()\n" );
    }

    Item* dequeue() {
        printf( " > Queue::dequeue()\n" );
        Item* res = first;
        if( first == last )
            first = last = 0;
        else
            first = first->next;
        printf( " < Queue::dequeue()\n" );
        return res;
    }
}; // class Queue
} // namespace util

```



Scenario: A Simple Queue

Scenario: The Problem

Various users of Queue demand extensions:



I want Queue to throw exceptions!

Please extend the Queue class by an element counter!



Queue should be thread-safe!



Scenario: A Simple Queue

The Not So Simple Queue Class

```
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 QueueInvalidItemError();
            if (last) {
                last->next = item;
                last = item;
            } else { last = first = item; }
            ++counter;
        } catch (...) {
            lock.leave(); throw;
        }
        lock.leave();
    }
};
```

```
Item* dequeue() {
    Item* res;
    lock.enter();
    try {
        res = first;
        if (first == last)
            first = last = 0;
        else first = first->next;
        if (counter > 0) --counter;
        if (res == 0)
            throw QueueEmptyError();
    } catch (...) {
        lock.leave();
        throw;
    }
    lock.leave();
    return res;
}
int count() { return counter; }
}; // class Queue
```



Scenario: A Simple Queue

What Code Does What?

<pre>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 QueueInvalidItemError(); if (last) { last->next = item; last = item; } else { last = first = item; } ++counter; } catch (...) { lock.leave(); throw; } lock.leave(); } };</pre>	<pre>Item* dequeue() { Item* res; lock.enter(); try { res = first; if (first == last) first = last = 0; else first = first->next; if (counter > 0) --counter; if (res == 0) throw QueueEmptyError(); } catch (...) { lock.leave(); throw; } lock.leave(); return res; } int count() { return counter; } }; // class Queue</pre>
--	---



Scenario: A Simple Queue

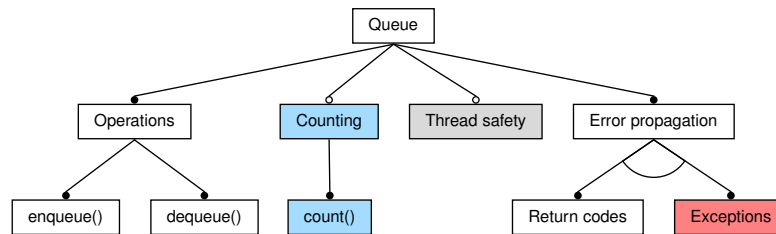
Problem Summary

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



Goal: A configurable Queue



Agenda

- 3.1 Motivation: Separation of Concerns
- 3.2 Tutorial: AspectC++
 - Example Szenario
 - First Steps And Language Overview
 - Advanced Concepts
 - Weaver Transformations
 - Further Examples
- 3.3 Summary and Outlook
- 3.4 References



Goal: A configurable Queue

Configuring with the Preprocessor?

```
class Queue {
    Item *first, *last;
#ifdef COUNTING_ASPECT
    int counter;
#endif
#ifdef LOCKING_ASPECT
    os::Mutex lock;
#endif
public:
    Queue () : first(0), last(0) {
#ifdef COUNTING_ASPECT
        counter = 0;
#endif
    }
    void enqueue(Item* item) {
#ifdef LOCKING_ASPECT
        lock.enter();
#endif
        try {
#ifdef ERRORHANDLING_ASPECT
            if (item == 0)
                throw QueueInvalidItemError();
#endif
            if (last) {
                last->next = item;
                last = item;
            } else { last = first = item; }
#ifdef COUNTING_ASPECT
            ++counter;
#endif
        }
#ifdef LOCKING_ASPECT
        } catch (...) {
            lock.leave(); throw;
        }
        lock.leave();
    }
#ifdef LOCKING_ASPECT
    Item* dequeue() {
        Item* res;
#ifdef LOCKING_ASPECT
        lock.enter();
        try {
            res = first;
            if (first == last)
                first = last = 0;
            else first = first->next;
#ifdef COUNTING_ASPECT
            if (counter > 0) --counter;
#endif
        } catch (...) {
            lock.leave();
            throw;
        }
        lock.leave();
        return res;
    }
#ifdef COUNTING_ASPECT
    int count() { return counter; }
#endif
}; // class Queue
```



Queue: Element Counting

Queue: Demanded Extensions

- I. Element counting
- II. Errorhandling
(signaling of errors by exceptions)
- III. Thread safety
(synchronization by mutex variables)

Please extend
the Queue class
by an element
counter!



Element counting: The Idea

- Increment a counter variable after each execution of `util::Queue::enqueue()`
- Decrement it after each execution of `util::Queue::dequeue()`



ElementCounter1

```
aspect ElementCounter {
    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.ah



ElementCounter1 - Elements

```
aspect ElementCounter {
    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.ah

We introduced a new **aspect** named *ElementCounter*.
An aspect starts with the keyword **aspect** and is syntactically much like a class.



ElementCounter1 - Elements

```
aspect ElementCounter {
    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.ah

Like a class, an aspect can define data members, constructors and so on



ElementCounter1 - Elements

```

aspect ElementCounter {
    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 );
    }
};

```

We give **after advice** (= some crosscutting code to be executed after certain control flow positions)

ElementCounter1.ah



ElementCounter1 - Elements

```

aspect ElementCounter {
    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 );
    }
};

```

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

ElementCounter1.ah



ElementCounter1 - Elements

```

aspect ElementCounter {
    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 );
    }
};

```

Aspect member elements can be accessed from within the advice body

ElementCounter1.ah



ElementCounter1 - Result

```

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

```

main(): enqueueing an item
> Queue::enqueue(00320FD0)
< Queue::enqueue(00320FD0)
Aspect ElementCounter: # of elements = 1
main(): dequeueing two items
> Queue::dequeue()
< Queue::dequeue() returning 00320FD0
Aspect ElementCounter: # of elements = 0
> Queue::dequeue()
< Queue::dequeue() returning 00000000
Aspect ElementCounter: # of elements = 0
<Output>

```



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

```
aspect ElementCounter {
    advice "util::Queue" : slice class {
        int counter;
    public:
        int count() const { return counter; }
    };
    advice execution("% util::Queue::enqueue(...)")
        && that(queue) : after( util::Queue& queue ) {
        ++queue.counter;
        printf( " Aspect ElementCounter: # of elements = %d\n", queue.count() );
    }
    advice execution("% util::Queue::dequeue(...)")
        && that(queue) : after( util::Queue& queue ) {
        if( queue.count() > 0 ) --queue.counter;
        printf( " Aspect ElementCounter: # of elements = %d\n", queue.count() );
    }
    advice construction("util::Queue")
        && that(queue) : before( util::Queue& queue ) {
        queue.counter = 0;
    }
};
```

ElementCounter2.ah



ElementCounter2 - Elements

```
aspect ElementCounter {
    advice "util::Queue" : slice class {
        int counter;
    public:
        int count() const { return counter; }
    };
    advice execution("% util::Queue::enqueue(...)")
        && that(queue) : after( util::Queue& queue ) {
        ++queue.counter;
        printf( " Aspect ElementCounter: # of elements = %d\n", queue.count() );
    }
    advice execution("% util::Queue::dequeue(...)")
        && that(queue) : after( util::Queue& queue ) {
        if( queue.count() > 0 ) --queue.counter;
        printf( " Aspect ElementCounter: # of elements = %d\n", queue.count() );
    }
    advice construction("util::Queue")
        && that(queue) : before( util::Queue& queue ) {
        queue.counter = 0;
    }
};
```

Introduces a **slice** of members into all classes denoted by the pointcut "util::Queue"

ElementCounter2.ah



ElementCounter2 - Elements

```
aspect ElementCounter {
    advice "util::Queue" : slice class {
        int counter;
    public:
        int count() const { return counter; }
    };
    advice execution("% util::Queue::enqueue(...)")
        && that(queue) : after( util::Queue& queue ) {
        ++queue.counter;
        printf( " Aspect ElementCounter: # of elements = %d\n", queue.count() );
    }
    advice execution("% util::Queue::dequeue(...)")
        && that(queue) : after( util::Queue& queue ) {
        if( queue.count() > 0 ) --queue.counter;
        printf( " Aspect ElementCounter: # of elements = %d\n", queue.count() );
    }
    advice construction("util::Queue")
        && that(queue) : before( util::Queue& queue ) {
        queue.counter = 0;
    }
};
```

We introduce a private *counter* element and a public method to read it

ElementCounter2.ah



ElementCounter2 - Elements

```

aspect ElementCounter {
    advice "util::Queue" : slice class {
        int counter;
    public:
        int count() const { return counter; }
    };
    advice execution("% util::Queue::enqueue(...)")
        && that(queue) : after( util::Queue& queue ) {
        ++queue.counter;
        printf( " Aspect ElementCounter: # of elements = %d\n", queue.count() );
    }
    advice execution("% util::Queue::dequeue(...)")
        && that(queue) : after( util::Queue& queue ) {
        if( queue.count() > 0 ) --queue.counter;
        printf( " Aspect ElementCounter: # of elements = %d\n", queue.count() );
    }
    advice construction("util::Queue")
        && that(queue) : before( util::Queue& queue ) {
        queue.counter = 0;
    }
};

```

A context variable *queue* is bound to *that* (the calling instance).
The calling instance has to be an *util::Queue*

ElementCounter2.ah



ElementCounter2 - Elements

```

aspect ElementCounter {
    advice "util::Queue" : slice class {
        int counter;
    public:
        int count() const { return counter; }
    };
    advice execution("% util::Queue::enqueue(...)")
        && that(queue) : after( util::Queue& queue ) {
        ++queue.counter;
        printf( " Aspect ElementCounter: # of elements = %d\n", queue.count() );
    }
    advice execution("% util::Queue::dequeue(...)")
        && that(queue) : after( util::Queue& queue ) {
        if( queue.count() > 0 ) --queue.counter;
        printf( " Aspect ElementCounter: # of elements = %d\n", queue.count() );
    }
    advice construction("util::Queue")
        && that(queue) : before( util::Queue& queue ) {
        queue.counter = 0;
    }
};

```

The context variable *queue* is used to access the calling instance.

ElementCounter2.ah



ElementCounter2 - Elements

```

aspect ElementCounter {
    advice "util::Queue" : slice class {
        int counter;
    public:
        int count() const { return counter; }
    };
    advice execution("% util::Queue::enqueue(...)")
        && that(queue) : after( util::Queue& queue ) {
        ++queue.counter;
        printf( " Aspect ElementCounter: # of elements = %d\n", queue.count() );
    }
    advice execution("% util::Queue::dequeue(...)")
        && that(queue) : after( util::Queue& queue ) {
        if( queue.count() > 0 ) --queue.counter;
        printf( " Aspect ElementCounter: # of elements = %d\n", queue.count() );
    }
    advice construction("util::Queue")
        && that(queue) : before( util::Queue& queue ) {
        queue.counter = 0;
    }
};

```

By giving **construction advice** we ensure that counter gets initialized

ElementCounter2.ah



ElementCounter2 - Result

```

int main() {
    util::Queue queue;
    printf("main(): Queue contains %d items\n", queue.count());
    printf("main(): enqueueing some items\n");
    queue.enqueue(new util::Item);
    queue.enqueue(new util::Item);
    printf("main(): Queue contains %d items\n", queue.count());
    printf("main(): dequeueing one items\n");
    util::Item* item;
    item = queue.dequeue();
    printf("main(): Queue contains %d items\n", queue.count());
}

```

main.cc



ElementCounter2 - Result

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

main.cc

```
main(): Queue contains 0 items
main(): enqueueing some items
> Queue::enqueue(00320FD0)
< Queue::enqueue(00320FD0)
Aspect ElementCounter: # of elements = 1
> Queue::enqueue(00321000)
< Queue::enqueue(00321000)
Aspect ElementCounter: # of elements = 2
main(): Queue contains 2 items
main(): dequeuing one item
> Queue::dequeue()
< Queue::dequeue() returning 00320FD0
Aspect ElementCounter: # of elements = 1
main(): Queue contains 1 items
```

<Output>



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 structs: new elements (*introductions*)
 - Joinpoints are described by *pointcut expressions*
- We will now take a closer look at some of them



Syntactic Elements

aspect name pointcut expression advice type

```
aspect ElementCounter {
    advice execution("% util::Queue::enqueue(...)") : after()
    {
        printf( " Aspect ElementCounter: after Queue::enqueue!\n" );
    }
    ...
};
```

ElementCounter1.ah

advice body



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++ programm entities → name joinpoints
 - support wildcards
 - pointcut functions**, e.g. execution(...), call(...), that(...)
 - 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

- A *slice* of additional methods, types, etc. is added to the class
- Can be used to extend the interface of a class



Before / After Advice

with execution joinpoints:

```
advice execution("void ClassA::foo()") : before()
advice execution("void ClassA::foo()") : after()
```

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

with call joinpoints:

```
advice call("void ClassA::foo()") : before()
advice call("void ClassA::foo()") : after()
```

```
int main(){
    printf("main()\n");
    ClassA a;
    a.foo();
}
```



Around Advice

with execution joinpoints:

```
advice execution("void ClassA::foo()") : around()
before code
tjp->proceed()
after code
```

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

with call joinpoints:

```
advice call("void ClassA::foo()") : around()
before code
tjp->proceed()
after code
```

```
int main(){
    printf("main()\n");
    ClassA a;
    a.foo();
}
```



Introductions

```

advice "ClassA" : slice class {
  element to introduce

public:
  element to introduce
};

```

```

class ClassA {
public:

  void foo(){
    printf("ClassA::foo()\n");
  }
}

```



Queue: Demanded Extensions

I. Element counting

II. Errorhandling
(signaling of errors by exceptions)III. Thread safety
(synchronization by mutex variables)

I want Queue to
throw exceptions!



Queue: Error Handling

Errorhandling: 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



Queue: Error Handling

ErrorException

```

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();
  }
}

```

ErrorException.ah



Exception - Elements

```

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

aspect Exception {
    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();
        }
}

```

Exception.ah

We give advice to be executed *before* enqueue() and *after* dequeue()



Exception - Elements

```

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

aspect Exception {
    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();
        }
}

```

Exception.ah

A context variable *item* is bound to the first argument of type *util::Item** passed to the matching methods



Exception - Elements

```

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

aspect Exception {
    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();
        }
}

```

Exception.ah

Here the context variable *item* is bound to the **result** of type *util::Item** returned by the matching methods



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. Errorhandling (signaling of errors by exceptions)

III. Thread safety (synchronization by mutex variables)

Queue should be thread-safe!



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

```
aspect LockingMutex {
    advice "util::Queue" : slice class { 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.ah



LockingMutex - Elements

```
aspect LockingMutex {
    advice "util::Queue" : slice class { 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.ah



LockingMutex - Elements

```

aspect LockingMutex {
  advice "util::Queue" : slice class { 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.ah

Pointcuts can be named.
sync_methods describes all
 methods that have to be
 synchronized by the mutex



LockingMutex - Elements

```

aspect LockingMutex {
  advice "util::Queue" : slice class { 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.ah

sync_methods is used to give
 around advice to the execution
 of the methods



LockingMutex - Elements

```

aspect LockingMutex {
  advice "util::Queue" : slice class { 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.ah

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



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()*



Queue: A new Requirement

- I. Element counting
- II. Errorhandling
(signaling of errors by exceptions)
- III. Thread safety
(synchronization by mutex variables)
- IV. Interrupt safety
(synchronization on interrupt level)

We need Queue to be
synchronized 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



LockingIRQ1

```
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.ah



LockingIRQ1 – Elements

```
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();
    }
};
```

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

LockingIRQ1.ah



LockingIRQ1 – Elements

```

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();
    }
};

```

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

LockingIRQ1.ah



LockingIRQ1 – Result

```

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

```

main()
os::disable_int()
> Queue::enqueue(00320FD0)
< Queue::enqueue()
os::enable_int()
kernel::irq_handler()
> Queue::enqueue(00321030)
< Queue::enqueue()
do_something()
os::disable_int()
> Queue::enqueue(00321060)
< Queue::enqueue()
os::enable_int()
back in main()
os::disable_int()
> Queue::dequeue()
< Queue::dequeue() returning 00320FD0
os::enable_int()

```

<Output>



LockingIRQ1 – Problem

```

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();
}

```

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

```

< Queue::enqueue()
os::enable_int()
kernel::irq_handler()
> Queue::enqueue(00321030)
< Queue::enqueue()
do_something()
os::disable_int()
> Queue::enqueue(00321060)
< Queue::enqueue()
os::enable_int()
back in main()
os::disable_int()
> Queue::dequeue()
< Queue::dequeue() returning 00320FD0
os::enable_int()

```

main.cc

<Output>



LockingIRQ2

```

aspect LockingIRQ {
    pointcut sync_methods() = "% util::Queue::%queue(...)";
    pointcut kernel_code() = "% kernel::%(...)";

    advice execution(sync_methods())
    && !cflow(execution(kernel_code())) : around() {
        os::disable_int();
        try {
            tjp->proceed();
        }
        catch(...) {
            os::enable_int();
            throw;
        }
        os::enable_int();
    }
};

```

LockingIRQ2.ah

Solution
Using the `cflow` pointcut function



LockingIRQ2 – Elements

```

aspect LockingIRQ {
    pointcut sync_methods() = "% util::Queue::%queue(...)";
    pointcut kernel_code() = "% kernel::%(...)";

    advice execution(sync_methods())
    && !cflow(execution(kernel_code())) : around() {
        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

```

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()
os::disable_int()
> Queue::enqueue(00320FD0)
< Queue::enqueue()
os::enable_int()
kernel::irq_handler()
> Queue::enqueue(00321030)
< Queue::enqueue()
do_something()
> Queue::enqueue(00321060)
< Queue::enqueue()
back in main()
os::disable_int()
> Queue::dequeue()
< Queue::dequeue() returning 00320FD0
os::enable_int()

```

main.cc

<Output>



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++: A First Summary

- The Queue example has presented the most important features of the AspectC++ language
 - aspect, advice, joinpoint, pointcut expression, pointcut function, ...
- Additionally, 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 interdependence and ordering
- In the following, we give a short overview on these advanced language elements



Agenda

- 3.1 Motivation: Separation of Concerns
- 3.2 Tutorial: AspectC++
 - Example Szenario
 - First Steps And Language Overview
 - Advanced Concepts
 - Weaver Transformations
 - Further Examples
- 3.3 Summary and Outlook
- 3.4 References



Advanced Concepts

AspectC++: Advanced Concepts

- Join Point 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
- Generic Advice
 - exploits static type information in advice code
- Aspect Ordering
 - allows to specify the invocation order of multiple aspects
- Aspect Instantiation
 - allows to implement user-defined aspect instantiation models



Advanced Concepts

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**



Advanced Concepts

The Join Point API (Excerpt)

Types (compile-time)

```
// object type (initiator)  
That  
// object type (receiver)  
Target  
// result type of the affected function  
Result  
// type of the i'th argument of the affected  
// function (with 0 <= i < ARGS)  
Arg<i>::Type  
Arg<i>::ReferredType
```

Consts (compile-time)

```
// number of arguments  
ARGS  
// unique numeric identifier for this join point  
JPID  
// numeric identifier for the type of this join  
// point (AC::CALL, AC::EXECUTION, ...)  
JPTYPE
```

Values (runtime)

```
// pointer to the object initiating a call  
That* that()  
// pointer to the object that is target of a call  
Target* target()  
// pointer to the result value  
Result* result()  
// typed pointer the i'th argument value of a  
// function call (compile-time index)  
Arg<i>::ReferredType* arg()  
// pointer the i'th argument value of a  
// function call (runtime index)  
void* arg( int i )  
// textual representation of the joinpoint  
// (function/class name, parameter types...)  
static const char* signature()  
// executes the original joinpoint code  
// in an around advice  
void proceed()  
// returns the runtime action object  
AC::Action& action()
```



Abstract Aspects and Inheritance

- Aspects can inherit from other aspects...
 - Reuse aspect definitions
 - Override methods and 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

```
#include "mutex.h"
aspect LockingA {
    pointcut virtual sync_classes() = 0;
    pointcut virtual sync_methods() = 0;

    advice sync_classes() : slice class {
        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();
    };
};
```

LockingA.ah

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

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

LockingQueue.ah



Abstract Aspects and Inheritance

```
#include "mutex.h"
aspect LockingA {
    pointcut virtual sync_classes() = 0;
    pointcut virtual sync_methods() = 0;

    advice sync_classes() : slice class {
        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();
    };
};
```

LockingA.ah

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

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

LockingQueue.ah



Generic Advice

Uses static JP-specific type information in advice code

- in combination with C++ overloading
- to instantiate C++ templates and template meta-programs

```
aspect TraceService {
    advice call(...) : after() {
        ...
        cout << *tjp->result();
    };
};
```

... operator <<(..., int)

... operator <<(..., long)

... operator <<(..., bool)

... operator <<(..., Foo)



Generic Advice

Uses static JP-specific type information in advice code

- in combination with C++ overloading

Resolves to the **statically typed** return value of template meta-programs

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

```

aspect TraceService {
  advice call(...) : after() {
    ...
    cout << *tjp->result();
  }
};
  
```



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-expr* : **order**(*high*, ..., *low*)
- Different aspect orders can be defined for different pointcuts

Example

```

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

```

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

MyAspect.ah

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

(Visual C++)



3.1 Motivation: Separation of Concerns

3.2 Tutorial: AspectC++

Example Szenario

First Steps And Language Overview

Advanced Concepts

Weaver Transformations

Further Examples

3.3 Summary and Outlook

3.4 References



Aspect Transformation

```

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

```

→

```

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

```



Aspect Transformation

```

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

```

Aspects are transformed into **ordinary classes**

→

```

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

```



Aspect Transformation

```

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

```

One global aspect **instance** is created by default

→

```

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

```



Aspect Transformation

```

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

```

Advice becomes a **member function**

→

```

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

```



Aspect Transformation

```

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

```

"Generic Advice" becomes a **template member function**

```

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

```



Joinpoint Transformation

```

int main() {
    foo();
    return 0;
}
main.cc

```

```

int main() {
    struct __call_main_0_0 {
        static inline void invoke () {
            AC::..._a0_before ();
            ::foo();
        }
    };
    __call_main_0_0::invoke ();
    return 0;
}
main.cc'

```



Joinpoint Transformation

```

int main() {
    foo();
    return 0;
}
main.cc

```

the function call is replaced by a call to a wrapper function

```

int main() {
    struct __call_main_0_0 {
        static inline void invoke () {
            AC::..._a0_before ();
            ::foo();
        }
    };
    __call_main_0_0::invoke ();
    return 0;
}
main.cc'

```



Joinpoint Transformation

```

int main() {
    foo();
    return 0;
}
main.cc

```

a local class invokes the advice code for this joinpoint

```

int main() {
    struct __call_main_0_0 {
        static inline void invoke () {
            AC::..._a0_before ();
            ::foo();
        }
    };
    __call_main_0_0::invoke ();
    return 0;
}
main.cc'

```



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



3.1 Motivation: Separation of Concerns

3.2 Tutorial: AspectC++

Example Scenario

First Steps And Language Overview

Advanced Concepts

Weaver Transformations

Further Examples

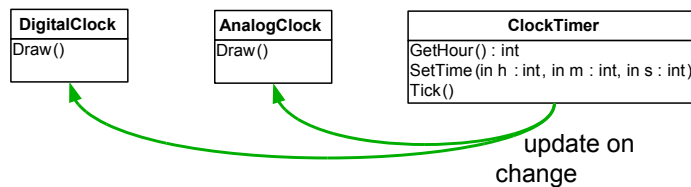
3.3 Summary and Outlook

3.4 References



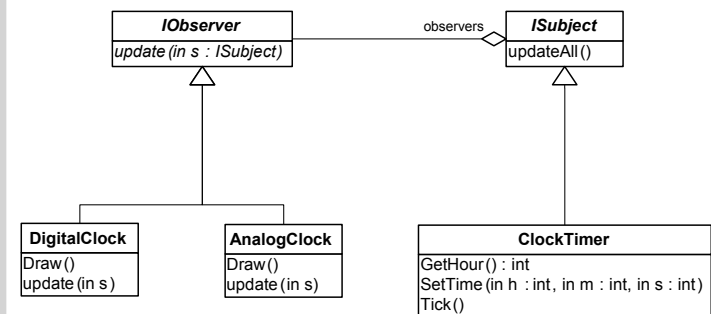
Further Examples

Observer Pattern: Scenario



Further Examples

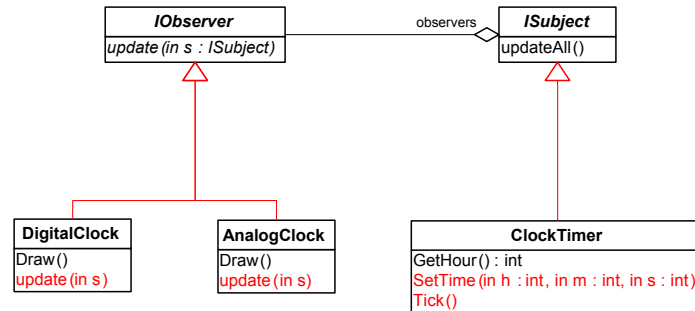
Observer Pattern: Implementation



Further Examples

Observer Pattern: Problem

The 'Observer Protocol' Concern...



...crosscuts the module structure



Further Examples

Solution: Generic Observer Aspect

```

aspect ObserverPattern {
  ...
public:
  struct ISubject {};
  struct IObserver {
    virtual void update (ISubject *) = 0;
  };

  pointcut virtual observers() = 0;
  pointcut virtual subjects() = 0;

  pointcut virtual subjectChange() = execution( "% ..::%(...)"
    && !" % ..::%(...) const" ) && within( subjects() );

  advice observers() : slice class : public ObserverPattern::IObserver;
  advice subjects() : slice class : public ObserverPattern::ISubject;

  advice subjectChange() : after () {
    ISubject* subject = tjp->that();
    updateObservers( subject );
  }

  void updateObservers( ISubject* subject ) { ... }
  void addObserver( ISubject* subject, IObserver* observer ) { ... }
  void removeObserver( ISubject* subject, IObserver* observer ) { ... }
};
  
```



Further Examples

Solution: Generic Observer Aspect

```

aspect ObserverPattern {
  ...
public:
  struct ISubject {};
  struct IObserver {
    virtual void update (ISubject *) = 0;
  };

  pointcut virtual observers() = 0;
  pointcut virtual subjects() = 0;

  pointcut virtual subjectChange() = execution( "% ..::%(...)"
    && !" % ..::%(...) const" ) && within( subjects() );

  advice observers() : slice class : public ObserverPattern::IObserver;
  advice subjects() : slice class : public ObserverPattern::ISubject;

  advice subjectChange() : after () {
    ISubject* subject = tjp->that();
    updateObservers( subject );
  }

  void updateObservers( ISubject* subject ) { ... }
  void addObserver( ISubject* subject, IObserver* observer ) { ... }
  void removeObserver( ISubject* subject, IObserver* observer ) { ... }
};
  
```

Interfaces for the
subject/observer roles



Further Examples

Solution: Generic Observer Aspect

```

aspect ObserverPattern {
  ...
public:
  struct ISubject {};
  struct IObserver {
    virtual void update (ISubject *) = 0;
  };

  pointcut virtual observers() = 0;
  pointcut virtual subjects() = 0;

  pointcut virtual subjectChange() = execution( "% ..::%(...)"
    && !" % ..::%(...) const" ) && within( subjects() );

  advice observers() : slice class : public ObserverPattern::IObserver;
  advice subjects() : slice class : public ObserverPattern::ISubject;

  advice subjectChange() : after () {
    ISubject* subject = tjp->that();
    updateObservers( subject );
  }

  void updateObservers( ISubject* subject ) { ... }
  void addObserver( ISubject* subject, IObserver* observer ) { ... }
  void removeObserver( ISubject* subject, IObserver* observer ) { ... }
};
  
```

abstract pointcuts that
define subjects/observers
(need to be overridden by a
derived aspect)



Further Examples

Solution: Generic Observer Aspect

```
aspect ObserverPattern {
    ...
    public:
        struct ISubject {};
        struct IObserver {
            virtual void update (ISubject *) = 0;
        };
        pointcut virtual observers() = 0;
        pointcut virtual subjects() = 0;
        pointcut virtual subjectChange() = execution( "% ...::%(...)"
            && !" % ...::%(...)" const" ) && within( subjects() );
        advice observers () : slice class : public ObserverPattern::IObserver;
        advice subjects() : slice class : public ObserverPattern::ISubject;
        advice subjectChange() : after () {
            ISubject* subject = tjp->that();
            updateObservers( subject );
        }
        void updateObservers( ISubject* subject ) { ... }
        void addObserver( ISubject* subject, IObserver* observer ) { ... }
        void removeObserver( ISubject* subject, IObserver* observer ) { ... }
};
```

virtual pointcut defining all state-changing methods.

(Defaults to the execution of any non-const method in subjects)



Further Examples

Solution: Generic Observer Aspect

```
aspect ObserverPattern {
    ...
    public:
        struct ISubject {};
        struct IObserver {
            virtual void update (ISubject *) = 0;
        };
        pointcut virtual observers() = 0;
        pointcut virtual subjects() = 0;
        pointcut virtual subjectChange() = execution( "% ...::%(...)"
            && !" % ...::%(...)" const" ) && within( subjects() );
        advice observers () : slice class : public ObserverPattern::IObserver;
        advice subjects() : slice class : public ObserverPattern::ISubject;
        advice subjectChange() : after () {
            ISubject* subject = tjp->that();
            updateObservers( subject );
        }
        void updateObservers( ISubject* subject ) { ... }
        void addObserver( ISubject* subject, IObserver* observer ) { ... }
        void removeObserver( ISubject* subject, IObserver* observer ) { ... }
};
```

Introduction of the role interface as additional **baseclass** into subjects / observers



Further Examples

Solution: Generic Observer Aspect

```
aspect ObserverPattern {
    ...
    public:
        struct ISubject {};
        struct IObserver {
            virtual void update (ISubject *) = 0;
        };
        pointcut virtual observers() = 0;
        pointcut virtual subjects() = 0;
        pointcut virtual subjectChange() = execution( "% ...::%(...)"
            && !" % ...::%(...)" const" ) && within( subjects() );
        advice observers () : slice class : public ObserverPattern::IObserver;
        advice subjects() : slice class : public ObserverPattern::ISubject;
        advice subjectChange() : after () {
            ISubject* subject = tjp->that();
            updateObservers( subject );
        }
        void updateObservers( ISubject* subject ) { ... }
        void addObserver( ISubject* subject, IObserver* observer ) { ... }
        void removeObserver( ISubject* subject, IObserver* observer ) { ... }
};
```

After advice to update observers after execution of a state-changing method



Further Examples

Solution: Putting Everything Together

Applying the Generic Observer Aspect to the clock example

```
aspect ClockObserver : public ObserverPattern {
    // define the participants
    pointcut subjects() = "ClockTimer";
    pointcut observers() = "DigitalClock" || "AnalogClock";
    public:
        // define what to do in case of a notification
        advice observers() : slice class {
            public:
                void update( ObserverPattern::ISubject* s ) {
                    Draw();
                }
        };
};
```



Further Examples

Errorhandling in Legacy Code: Scenario

```
HRESULT 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( "example.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;
    }
}
```

A typical Win32 application



Further Examples

Errorhandling in Legacy Code: Scenario

```
HRESULT 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( "example.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;
    }
}
```

These Win32 API functions may fail!



Further Examples

Win32 Errorhandling: Goals

- Detect failed calls of Win32 API functions
 - by giving advice for any call to a Win32 function
- Throw a *helpful* exception in case of a failure
 - describing the exact circumstances and reason of the failure

Problem: Win32 failures are indicated by a “magic” return value

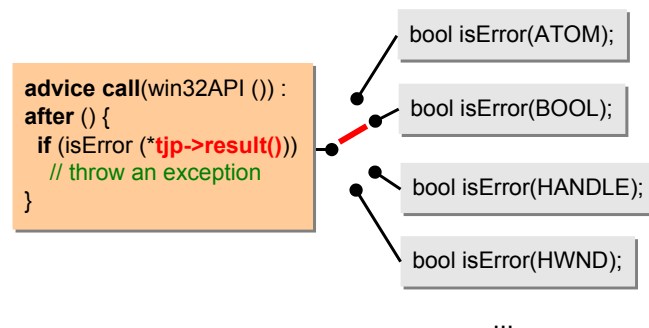
- magic value to compare against depends on the **return type** of the function
- error reason (GetLastError()) only valid in case of a failure

return type	magic value
BOOL	FALSE
ATOM	(ATOM) 0
HANDLE	INVALID_HANDLE_VALUE or NULL
HWND	NULL



Further Examples

Detecting the failure: Generic Advice



Further Examples

Describing the failure: Generative Advice

```
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 () {
    // throw an exception
    ostream s;
    DWORD code = GetLastError();
    s << "WIN32 ERROR " << code << ...
    << win32::GetErrorText( code ) << ... <<
    << tjp->signature() << "WITH: " << ...;
    ArgPrinter<JoinPoint::ARGS>::work (*tjp, s);

    throw win32::Exception( s.str() );
}
```



Agenda

3.1 Motivation: Separation of Concerns

3.2 Tutorial: AspectC++

3.3 Summary and Outlook

3.4 References



Further Examples

Reporting the Error

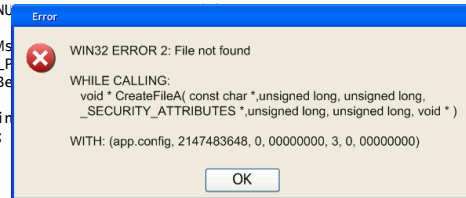
```
LRESULT WINAPI WndProc( HWND hWnd, UINT nMsg, WPARAM wParam, LPARAM lParam ) {
    HDC dc = NULL;

    switch( nMsg )
    case WM_PAINT:
        dc = GetDC( hWnd );
        ...
        EndPaint( hWnd, dc );
        break;
    ...
}

int WINAPI WinMain( ... ) {
    HANDLE hConfigFile = CreateFile( "example.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;
}
```



Aspect-Oriented Programming: Summary

- AOP aims at a better separation of crosscutting concerns
 - Avoidance of code tangling
 - implementation of optional features
 - Avoidance of code scattering
 - implementation of nonfunctional features
- Basic idea: separation of **what** from **where**
 - **Join Points** → **where**
 - positions in the static structure or dynamic control flow (event)
 - given declaratively by pointcut expressions
 - **Advice** → **what**
 - additional elements (members, ...) to introduce at join points
 - additional behavior (code) to superimpose at join points
- AspectC++ brings AOP concepts to the C++ world
 - Static source-to-source transformation approach



Aspect-Oriented Programming: Summary

■ AOP aims at a better separation of crosscutting concerns

- Avoidance of code tangling
→ implementation of optional features
- Avoidance of code scattering
→ implementation of nonfunctional features

Next Lecture:

How to use AOP to achieve loose coupling, granularity and variability for feature implementations in configurable system software
~ aspect-aware design

■ Basic idea: separation of **what** from **where**

- **Join Points** → **where**
 - positions in the static structure or dynamic control flow (event)
 - given declaratively by pointcut expressions
- **Advice** → **what**
 - additional elements (members, ...) to introduce at join points
 - additional behavior (code) to superimpose at join points

■ AspectC++ brings AOP concepts to the C++ world

- Static source-to-source transformation approach



Referenzen

- [1] The British Standards Institute. *The C++ Standard (Incorporating Technical Corrigendum No. 1)*, second. Printed version of the ISO/IEC 14882:2003 standard. John Wiley & Sons, Inc., 2003. ISBN: 0-470-84674-7.
- [2] Gregor Kiczales, John Lamping, Anurag Mendhekar, et al. "Aspect-Oriented Programming". In: *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP '97)*. (Finland). Ed. by Mehmet Aksit and Satoshi Matsuoka. Vol. 1241. Lecture Notes in Computer Science. Springer-Verlag, June 1997, pp. 220–242.
- [3] Daniel Lohmann, Georg Blaschke, and Olaf Spinczyk. "Generic Advice: On the Combination of AOP with Generative Programming in AspectC++". In: *Proceedings of the 3rd International Conference on Generative Programming and Component Engineering (GPCE '04)*. Ed. by G. Karsai and E. Visser. Vol. 3286. Lecture Notes in Computer Science. Springer-Verlag, Oct. 2004, pp. 55–74. ISBN: 978-3-540-23580-4. DOI: 10.1007/978-3-540-30175-2_4.
- [4] Daniel Lohmann, Fabian Scheler, Reinhard Tartler, et al. "A Quantitative Analysis of Aspects in the eCos Kernel". In: *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2006 (EuroSys '06)*. (Leuven, Belgium). Ed. by Yolande Berbers and Willy Zwaenepoel. New York, NY, USA: ACM Press, Apr. 2006, pp. 191–204. ISBN: 1-59593-322-0. DOI: 10.1145/1218063.1217954.



Referenzen (Cont'd)

- [5] Olaf Spinczyk and Daniel Lohmann. "The Design and Implementation of AspectC++". In: *Knowledge-Based Systems, Special Issue on Techniques to Produce Intelligent Secure Software 20.7* (2007), pp. 636–651. DOI: 10.1016/j.knosys.2007.05.004.
- [6] Olaf Spinczyk, Daniel Lohmann, and Matthias Urban. "AspectC++: An AOP Extension for C++". In: *Software Developers Journal* 5 (May 2005), pp. 68–76. URL: <http://www.aspectc.org/fileadmin/publications/sdj-2005-en.pdf>.



AspectC++ Quick Reference

Concepts

aspect
Aspects in AspectC++ implement in a modular way cross-cutting concerns and are an extension to the class concept of C++. Additionally to attributes and methods, aspects may also contain *advice declarations*.

advice
An advice declaration is used either to specify code that should run when the *join points* specified by a *pointcut expression* are reached or to introduce a new method, attribute, or type to all *join points* specified by a *pointcut expression*.

slice
A slice is a fragment of a C++ element like a class. It may be used by introduction advice to implemented static extensions of the program.

join point
In AspectC++ join points are defined as points in the component code where aspects can interfere. A join point refers to a method, an attribute, a type (class, struct, or union), an object, or a point from which a join point is accessed.

pointcut
A pointcut is a set of join points described by a *pointcut expression*.

pointcut expression
Pointcut expressions are composed from *match expressions* used to find a set of join points, from pointcut functions used to filter or map specific join points from a pointcut, and from algebraic operators used to combine pointcuts.

match expression
Match expressions are strings containing a search pattern.

order declaration
If more than one *aspect* affects the same *join point* an *order declaration* can be used to define the order of advice code execution.

Aspects

Writing aspects works very similar to writing C++ class definitions. Aspects may define ordinary class members as well as advice.

```
aspect A { ... };
    defines the aspect A

aspect A : public B { ... };
    A inherits from class or aspect B
```

Advice Declarations

```
advice pointcut : before(...) {...}
    the advice code is executed before the join points in the
    pointcut

advice pointcut : after(...) {...}
    the advice code is executed after the join points in the point-
    cut

advice pointcut : around(...) {...}
    the advice code is executed in place of the join points in the
    pointcut

advice pointcut : order(high, ...low);
    high and low are pointcuts, which describe sets of aspects.
    Aspects on the left side of the argument list always have a
    higher precedence than aspects on the right hand side at the
    join points, where the order declaration is applied.

advice pointcut : slice class : public Base {...}
    introduces a new base class Base and members into the target
    classes matched by pointcut.

advice pointcut : slice ASlice ;
    introduces the slice ASlice into the target classes matched by
    pointcut.
```

Pointcut Expressions

Type Matching

```
"int"
    matches the C++ built-in scalar type int

"% *"
    matches any pointer type
```

Namespace and Class Matching

```
"Chain"
    matches the class, struct or union Chain

"Memory%"
    matches any class, struct or union whose name starts with
    "Memory"
```

Function Matching

```
"void reset()"
    matches the function reset having no parameters and return-
    ing void

"% printf(...)"
    matches the function printf having any number of parame-
    ters and returning any type

"% ...::%(...)"
    matches any function, operator function, or type conversion
    function (in any class or namespace)
```

```
"% ...::Service::%(...) const"
    matches any const member-function of the class Service
    defined in any scope

"% ...::operator %(...)"
    matches any type conversion function

"virtual %C::%(...)"
    matches any virtual member function of C
```

Template Matching[†]

```
"std::set<...>"
    matches all template instances of the class std::set

"std::set<int>"
    matches only the template instance std::set<int>

"% ...::%<...>::%(...)"
    matches any member function from any template class in-
    stance in any scope
```

Predefined Pointcut Functions

Functions

```
call(pointcut) N→CC##
    provides all join points where a named entity in the pointcut
    is called.

execution(pointcut) N→CE
    provides all join points referring to the implementation of a
    named entity in the pointcut.

construction(pointcut) N→CCons
    all join points where an instance of the given class(es) is con-
    structed.

destruction(pointcut) N→CDes
    all join points where an instance of the given class(es) is de-
    structed.
```

pointcut may contain function names or class names. A class name is equivalent to the names of all functions defined within its scope combined with the `||` operator (see below).

Control Flow

```
cflow(pointcut) C→C
    captures join points occuring in the dynamic execution con-
    text of join points in the pointcut. The argument pointcut
    is forbidden to contain context variables or join points with
    runtime conditions (currently cflow, that, or target).
```

Types

```
base(pointcut) N→NC,F
    returns all base classes resp. redefined functions of classes in
    the pointcut

derived(pointcut) N→NC,F
    returns all classes in the pointcut and all classes derived from
    them resp. all redefined functions of derived classes
```

Scope

within(*pointcut*) N→C
filters all join points that are within the functions or classes in the *pointcut*

Context

that(*type pattern*) N→C
returns all join points where the current C++ `this` pointer refers to an object which is an instance of a type that is compatible to the type described by the *type pattern*

target(*type pattern*) N→C
returns all join points where the target object of a call is an instance of a type that is compatible to the type described by the *type pattern*

result(*type pattern*) N→C
returns all join points where the result object of a call/execution is an instance of a type described by the *type pattern*

args(*type pattern*, ...) (N,...)→C
a list of *type patterns* is used to provide all joinpoints with matching argument signatures

Instead of the *type pattern* it is possible here to pass the name of a **context variable** to which the context information is bound. In this case the type of the variable is used for the type matching.

Algebraic Operators

pointcut && *pointcut* (N,N)→N, (C,C)→C
intersection of the join points in the *pointcuts*

pointcut || *pointcut* (N,N)→N, (C,C)→C
union of the join points in the *pointcuts*

! *pointcut* N→N, C→C
exclusion of the join points in the *pointcut*

JoinPoint-API

The JoinPoint-API is provided within every advice code body by the built-in object **tjp** of class **JoinPoint**.

Compile-time Types and Constants

That [type]
object type (object initiating a call)

Target [type]
target object type (target object of a call)

Result [type]
type of the object, which is used to *store* the result of the affected function

Res::Type, **Res::ReferredType** [type]
result type of the affected function

Arg<i>::Type, **Arg<i>::ReferredType** [type]
type of the i^{th} argument of the affected function (with $0 \leq i < ARGS$)

ARGS [const]
number of arguments

JPID [const]
unique numeric identifier for this join point

JPTYPE [const]
numeric identifier describing the type of this join point (**AC::CALL**, **AC::EXECUTION**, **AC::CONSTRUCTION**, or **AC::DESTRUCTION**)

Runtime Functions and State

static const char *signature()
gives a textual description of the join point (function name, class name, ...)

static const char *filename()
returns the name of the file in which the joinpoint shadow is located

static int line()
the source code line number that is associated with the joinpoint shadow

That *that()
returns a pointer to the object initiating a call or 0 if it is a static method or a global function

Target *target()
returns a pointer to the object that is the target of a call or 0 if it is a static method or a global function

Result *result()
returns a typed pointer to the result value or 0 if the function has no result value

Arg<i>::ReferredType *arg<i>()
returns a typed pointer to the i^{th} argument value (with $0 \leq i < ARGS$)

void *arg(int i)
returns a pointer to the memory position holding the argument value with index *i*

void proceed()
executes the original code in an around advice (should be called at most once in around advice)

AC::Action &action()
returns the runtime action object containing the execution environment to execute (*trigger()*) the original code encapsulated by an around advice

Runtime Type Information

static AC::Type resulttype()

static AC::Type argtype(int i)
return a C++ ABI V3 conforming string representation of the result type / argument type of the affected function

Example

A reusable tracing aspect.

```
aspect Trace {
    pointcut virtual functions() = 0;
    advice execution(functions()) : around() {
        cout << "before " << JoinPoint::signature() << "\n";
        for (unsigned i = 0; i < JoinPoint::ARGS; i++)
            cout << (i ? ", " : "") << JoinPoint::argtype(i);
        cout << "\n" << endl;
        tjp->proceed();
        cout << "after" << endl;
    }
};
```

In a derived aspect the *pointcut functions* may be redefined to apply the aspect to the desired set of functions.

```
aspect TraceMain : public Trace {
    pointcut functions() = "% main(...)";
};
```

This is a reference sheet corresponding to AspectC++ 1.0. Version 1.12, April 18, 2011.

(c) Copyright 2011 pure-systems GmbH, Olaf Spinczyk and Daniel Lohmann. All rights reserved.

[†] support for template instance matching is an experimental feature
^{††} <http://www.codesourcery.com/cxx-abi/abi.html#mangling>
^{‡‡} C, C_C, C_E, C_{Cons}, C_{Des}: Code (any, only *Call*, only *Execution*, only object *Construction*, only object *Destruction*); N, N_N, N_C, N_F, N_T: Names (any, only *Namespace*, only *Class*, only *Function*, only *Type*)