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 14 – 2014-04-24

http://www4.informatik.uni-erlangen.de/Lehre/SS14/V_KSS

Implementation Techniques: Classification

Decompositional Approaches
- Text-based filtering (untyped)
- Preprocessors

Compositional Approaches
- Language-based composition mechanisms (typed)
- OOP, AOP, Templates

Generative Approaches
- Metamodel-based generation of components (typed)
- MDD, C++ TMP, generators

About this Lecture

Solution Space
Architecture and Implementation

Problem Space
Features and Dependencies

Domain Expert

Specific Problem
Specific Solution

Domain Expert

Implementation

Configuration

Variant

System User

Conf

figuration

Variant

System User

Agenda

3.1 Motivation: Separation of Concerns
3.2 Tutorial: AspectC++
3.3 Summary and Outlook
3.4 References
3.1 Motivation: Separation of Concerns
Example: I4WeatherMon
Example: eCos
AOP
3.2 Tutorial: AspectC++
3.3 Summary and Outlook
3.4 References

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
Synchronization

Static Configurability with the CPP?

```
Cyg_Mutex::Cyg_Mutex() {
    locked = false;
    owner = NULL;
    if (defined(CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_DEFAULT) &&
        defined(CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_DEFAULT_INHERIT)) {
        protocol = CEILING;
        ceiling = CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_DEFAULT_PRIORITY;
        return;
    }
    if (defined(CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_DYNAMIC)) {
        // if there is a default priority ceiling defined, use that to initialize
        // the ceiling.
        ceiling = CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_DEFAULT_PRIORITY;
        // Otherwise set it to zero.
        ceiling = 0;
        protocol = NONE;
        return;
    }
    // DYNAMIC and DEFAULT defined
    protocol = CEILING;
    ceiling = CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_DEFAULT_PRIORITY;
    // not (DYNAMIC and DEFAULT defined)
    protocol = INHERIT;
    return;
}
```

Kernel policies:

- Tracing
- Instrumentation
- Synchronization

Aspect-Oriented Programming (AOP) [2]

**Issue**

Crosscutting Concerns

```c
Cyg_Mutex::Cyg_Mutex() {
    locked = false;
    owner = NULL;
}
```

- PROTOCOL
- CEILING
- INHERIT
- DYNAMIC

Kernel policies:

- Tracing
- Instrumentation
- Synchronization

Aspect-Oriented Programming (AOP)

- AOP is about modularizing crosscutting concerns
  ```c
  well modularized concern
  ```

- Examples: tracing, synchronization, security, buffering, error handling, constraint checks, ...
  ```c
  badly modularized
  ```

  without AOP
  ```c
  with AOP
  ```

- Examples: tracing, synchronization, security, buffering, error handling, constraint checks, ...
AOP: The Basic Idea

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

- **Join Points**  $\rightarrow$ **where**
  - positions in the static structure or dynamic control flow (event)
  - given declaratively by pointcut expressions

- **Advice**  $\rightarrow$ **what**
  - additional elements (members, ...) to introduce at join points
  - additional behavior (code) to superimpose at join points

Implementation of Crosscutting Concerns with AOP

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

    // 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();
    }
}
```

Static Configurability with the CPP?

**Crosscutting Concerns**

Can we do better with aspects?

**Result**

after refactoring into aspects [4]
AspectC++ [5, 6]

- 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

Agenda

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

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" ); 
    };
} // namespace util

Scenario: A Simple Queue

The Simple Queue Class
Scenario: A Simple Queue

Various users of Queue demand extensions:

Please extend the Queue class by an element counter!

I want Queue to throw exceptions!

Queue should be thread-safe!

Scenario: The Problem

What Code Does What?

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();
            }
            int count() { return counter; }
        }
    }
};

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**

- **Operations**
  - enqueue()
  - dequeue()
- **Counting**
  - count()
- **Thread safety**
- **Error propagation**
- **Return codes**
- **Exceptions**

---

**Agenda**

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

---

**Queue: Element Counting**

I. Element counting

II. Error handling

III. Thread safety

---

**Queue: Demanded Extensions**

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

```cpp
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() { -=counter;
    printf(" Aspect ElementCounter: # of elements = \d\n", counter );
}
};
```

ElementCounter1 - Elements

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

Like a class, an aspect can define data members, constructors and so on.
Queue: Element Counting

ElementCounter1 - Elements

```cpp
aspect ElementCounter {
    int counter;
    ElementCounter() {
        counter = 0;
    }
    advice execution("% until::Queue::enqueue(...)") : after() {
        ++counter;
        printf(" Aspect ElementCounter: # of elements = %d\n", counter);
    }
    advice execution("% until::Queue::dequeue(...)") : after() {
        --counter;
        printf(" Aspect ElementCounter: # of elements = %d\n", counter);
    }
};
```

ElementCounter1.ah

---

Queue: Element Counting

ElementCounter1 - Elements

```cpp
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 = queue.dequeue();
    printf("main(): dequeued item:\n");
    item = queue.dequeue();
}
```

main.cc
Queue: Element Counting

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

```cpp
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;
}:
```

Queue: Element Counting

ElementCounter2 - Elements

```cpp
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
Queue: Element Counting

**ElementCounter2 - Result**

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

**Queue: Element Counting**

**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

**AspectC++ Language Elements**

**Syntactic Elements**

- **aspect name**
- **pointcut expression**
- **advice type**

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

**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
Queue: Error Handling

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

**Error Exception**

```cpp
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();
    }
};
```
namespace util {
    struct QueueInvalidItemError {};
    struct QueueEmptyError {};
}

aspectErrorException {
    advice execution("%util::Queue::enqueue(...)" ) && \textcolor{red}{\textbf{args}}(item)
        : \textbf{before}(util::Item* item) {
            if (item == 0)
                throw util::QueueInvalidItemError();
        }
    advice execution("%util::Queue::dequeue(...)" ) && result(item)
        : \textbf{after}(util::Item* item) {
            if (item == 0)
                throw util::QueueEmptyError();
        };
}

\textbf{ErrorException.ah}

---

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

aspectErrorException {
    advice execution("%util::Queue::enqueue(...)" ) && \textcolor{red}{\textbf{args}}(item)
        : \textbf{before}(util::Item* item) {
            if (item == 0)
                throw util::QueueInvalidItemError();
        }
    advice execution("%util::Queue::dequeue(...)" ) && result(item)
        : \textbf{after}(util::Item* item) {
            if (item == 0)
                throw util::QueueEmptyError();
        };
}

\textbf{ErrorException.ah}

---

\textbf{ErrorException - Elements}

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

---

\textbf{ErrorException - Elements}

A context variable \textbf{item} is bound to the first argument of type \textbf{util::Item*} passed to the matching methods
Queue: Thread Synchronization

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

```cpp
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

```cpp
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
Queue: Thread Synchronization

**LockingMutex - Elements**

```cpp
aspect LockingMutex {
  advice "util::Queue" : slice class { os::Mutex lock; };
  pointcut sync_methods() = % util::Queue::Xqueue(...) ;

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

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

---

**LockingMutex - Elements**

```cpp
aspect LockingMutex {
  advice "util::Queue" : slice class { os::Mutex lock; }:
  pointcut sync_methods() = % util::Queue::Xqueue(...) ;

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

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

---

**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: IRQ Synchronization

Queue: A new Requirement

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

---

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();
}

LockingIRQ1.ah

---

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();
}

LockingIRQ1.ah
Queue: IRQ Synchronization

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

Queue: IRQ Synchronization

aspect LockingIRQ2
{
  pointcut sync_methods() = "% util::Queue::%queue(...);"
  pointcut kernel_code() = "% kernel::%();"
  advice execution(sync_methods()) && within(kernel_code()) : around()
  { os::enable_int();
    try {
      tjp->proceed();
    } catch(...) {
      os::enable_int();
    }
  }
}

LockingIRQ1 – Elements

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

LockingIRQ1 – Problem

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

LockingIRQ1 – Result

LockingIRQ2

Solution Using the eflow pointcut function

main.cc

<Output>
A First Summary

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 Scenario
   - First Steps And Language Overview
   - Advanced Concepts
   - Weaver Transformations
   - Further Examples
3.3 Summary and Outlook
3.4 References

Advanced Concepts

The Joinpoint API

- Inside an advice body, the current joinpoint context is available via the implicitly passed $tjp$ variable:
  ```c
  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)

<table>
<thead>
<tr>
<th>Types (compile-time)</th>
<th>Values (runtime)</th>
</tr>
</thead>
<tbody>
<tr>
<td>object type (initiator)</td>
<td>pointer to the object initiating a call</td>
</tr>
<tr>
<td>object type (receiver)</td>
<td>That* that()</td>
</tr>
<tr>
<td>Target</td>
<td>pointer to the object that is target of a call</td>
</tr>
<tr>
<td>Result</td>
<td>Target* target()</td>
</tr>
<tr>
<td>type of the 1st argument of the affected function</td>
<td>pointer to the result value</td>
</tr>
<tr>
<td>Result* result()</td>
<td>typed pointer the 1st argument value of a</td>
</tr>
<tr>
<td>function with 0 &lt;= i &lt; ARGS</td>
<td>function call (compile-time index)</td>
</tr>
<tr>
<td>Arg&lt;i&gt;::Type</td>
<td>Arg&lt;i&gt;::ReferredType* arg()</td>
</tr>
<tr>
<td>Arg&lt;i&gt;::ReferedType</td>
<td>pointer the 7th argument value of a</td>
</tr>
<tr>
<td>Arg&lt;i&gt;::ReferedType* arg()</td>
<td>function call (runtime index)</td>
</tr>
<tr>
<td>static const char* signature()</td>
<td>void* arg(int i)</td>
</tr>
<tr>
<td>textual representation of the joinpoint</td>
<td>static context</td>
</tr>
</tbody>
</table>
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 a **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

```cpp
#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();
    }
  }
}
```

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

### Advanced Concepts

#### Abstract Aspects and Inheritance

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

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

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

#### Generic Advice

Uses static JP-specific type information in advice code
- in combination with C++ overloading
- to instantiate C++ templates and template meta-programs

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

... operator<<(..., int)
... operator<<(..., long)
... operator<<(..., bool)
... operator<<(..., Foo)
Advanced Concepts

Generic Advice

Uses static JP-specific type information in advice code
- in combination with C++ overloading

Resolves to the *statically typed* return value
- no runtime type checks are needed
- unhandled types are detected at compile-time
- functions can be inlined

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

... operator <(..., int)
... operator <(..., long)
... operator <(..., bool)
... operator <(..., Foo)

Advanced Concepts

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" );
  ```

Advanced Concepts

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++)

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
Weaver Transformations

Aspect Transformation

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

```
class Transform {
    static Transform __instance;
    // ...
    void __a0 before () { printf("before foo call\n");
    }
    template<class JoinPoint>
    void __a0_after (JoinPoint *tjp) {
        printf (tjp->signature () );
    };
}
```

Aspect Transformation

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

```
class Transform {
    static Transform __instance;
    // ...
    void __a0 before () { printf("before foo call\n");
    }
    template<class JoinPoint>
    void __a0_after (JoinPoint *tjp) {
        printf (tjp->signature () );
    };
}
```

One global aspect instance is created by default

Advice becomes a member function

Aspects are transformed into ordinary classes
### Weaver Transformations

**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 __a3 after (JoinPoint *tjp) {
            printf (tjp->signature () );
    }
};
```

*"Generic Advice" becomes a template member function*

**Joinpoint Transformation**

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

*The function call is replaced by a call to a wrapper function*

**Joinpoint Transformation**

```cpp
int main() {
    struct __call_main_0_0 {
        static inline void invoke (){
            AC::a0 before ();
        ::foo();
        };
        __call_main_0_0::invoke ();
        return 0;
    }
}
```

*A local class invokes the advice code for this joinpoint*
Weaver Transformations

Translation Modes

- **Whole Program Transformation-Mode**
  - e.g. `act++ -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. `act++ -c a.cc -o a-gen.cc -p`
  - easier integration into build processes

Agenda

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

Observer Pattern: Implementation
Further Examples

Observer Pattern: Problem

The 'Observer Protocol' Concern...

...crosscuts the module structure

Further Examples

Solution: Generic Observer Aspect

```cpp
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 remObserver( ISubject* subject, IObserver* observer ) { ... }
};
```

Further Examples

Solution: Generic Observer Aspect

```cpp
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 remObserver( ISubject* subject, IObserver* observer ) { ... }
};
```

Further Examples

Solution: Generic Observer Aspect

```cpp
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 remObserver( ISubject* subject, IObserver* observer ) { ... }
};
```
Further Examples

Solution: Generic Observer Aspect

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

Further Examples

Solution: Generic Observer Aspect

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

Further Examples

Solution: Generic Observer Aspect

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

Further Examples

Solution: Putting Everything Together

Applying the Generic Observer Aspect to the clock example

```cpp
aspect ClockObserver : public ObserverPattern {
    public:
    // define the participants
    pointcut subjectChange() = "ClockTimer";
    pointcut observers() = "DigitalClock" || "AnalogClock";
    advice observers () : slice class : public ObserverPattern:IObserver;
    advice subjectChange() : after () {
        ISubject* subject = tjp->that();
        updateObservers( subject );
    }
    void addObserver( ISubject* subject, IObserver* observer ) { ... }
    void removeObserver( ISubject* subject, IObserver* observer ) { ... }
};
```
Further Examples

Errorhandling in Legacy Code: Scenario

```cpp
RESULT WINAPI WinProc( HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam ) {
    HDC dc = NULL; PAINTSTRUCT ps = {0};

    switch( uMsg ) {
    case WM_PAINT:
        dc = BeginPaint( hwnd, &aps );
        ...  
        EndPaint( hwnd, &aps );
        break;
    ...  
    }
    int WINAPI WinMain( ... ) {
        HANDLE hConfigFile = CreateFile( "example.config", GENERIC_READ, ..., NULL );
        DWORD wc = { 0, WinProc, 0, 0, ..., "Example_Class"};
        RegisterClass( &wc );
        HINSTANCE hInst = GetModuleHandle( "Example", NULL );
        UpdateWindow( hwndMain );
        MSG msg;
        while( GetMessage( &msg, NULL, 0, 0 ) ) {
            TranslateMessage( &msg );
            DispatchMessage( &msg );
        }
        return 0;
    }
```  

A typical Win32 application

These Win32 API functions may fail!

Further Examples

Win32 Errorhandling: Goals

- Detect failed calls of Win32 API functions
  - by giving after 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

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

Further Examples

Detecting the failure: Generic Advice

```cpp
advice call(win32API) :  
  after() {  
    if(isError("tjp->result()"))  // throw an exception
      bool isError(ATOM);  
      bool isError(BOOL);  
      bool isError(HANDLE);  
      bool isError(HWND);  
    ...  
  }
```
Further Examples

Describing the failure: Generative Advice

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

```cpp
advice call(win32API()) : after() {
  // throw an exception
  stringstream s;
  DWORD code = GetLastError();
  s << "WIN32 ERROR " << code << "...
  << win32::GetErrorText(code) << ... <<
  << tjp->signature() << "WITH: " << ...;
  ArgPrinter<0>:: joinPoint::ARGS:: work(*tjp, s);
  throw win32::Exception(s.str());
}
```

Reporting the Error

```cpp
RESULT WINAPI WndProc(HWND HWND, UINT nMsg, WPARAM wParam, LPARAM lParam) {
  if (nMsg == WM_DESTROY)
    PostQuitMessage(0);
  switch(nMsg) {
    case WM_CREATE:
      // do something
      break;
    default:
      return FALSE;
  }
}  
```

Agenda

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

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

- Basic idea: separation of what from where
  - Join Points  → where
    - positions in the static structure or dynamic control flow (event)
  - 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

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

Referenzen
### 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 match 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
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 JPT 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 ≤ 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() [type] gives a textual description of the join point (function name, class name, ...

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

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

That *that() [type] 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 ≤ 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

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() << "(";
    for (unsigned i = 0; i < JoinPoint::ARGS; ++i)  
      cout << (i ? " , " : "") << JoinPoint::argtype(i);
    cout << " )" << endl;
    jpt->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 V3 conforms to the C++ ABI V3 specification
‡‡ http://www.codesourcery.com/cxx-abi/abi.html#mangling
†† AC::DESTRUCTION, AC::CONSTRUCTION, only object Construction, only object Destruction); N, NC, Nc, Nr, Nf: Names (any, only Class, only Function, only Type)