Aspect-Oriented Programming

with

AspectC++

Olaf Spinczyk (os@aspectc.org)
Daniel Lohmann (dl@aspectc.org)

www.aspectc.org

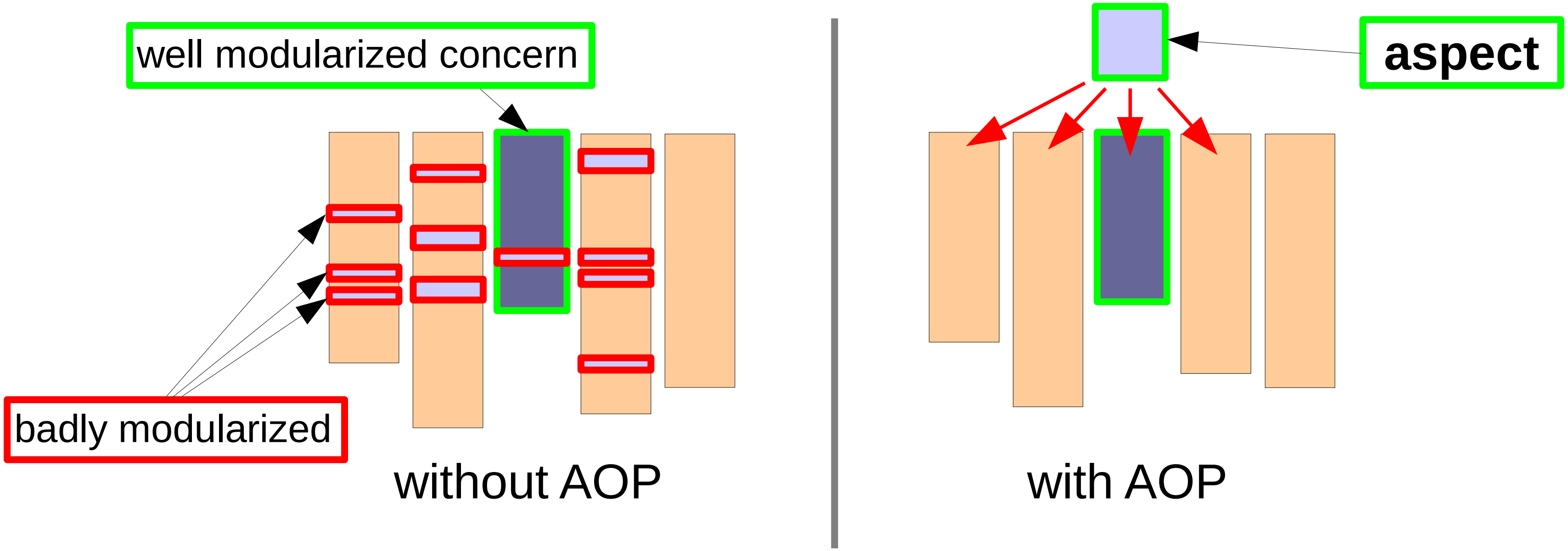
2017-05-22

Schedule

Part	Title	Time
I	Introduction	10
II	AspectC++ Language	70
III	Tool Support	30
IV	Real-World Examples	20
V	Summary	10

Aspect-Oriented Programming

- AOP is about modularizing crosscutting concerns

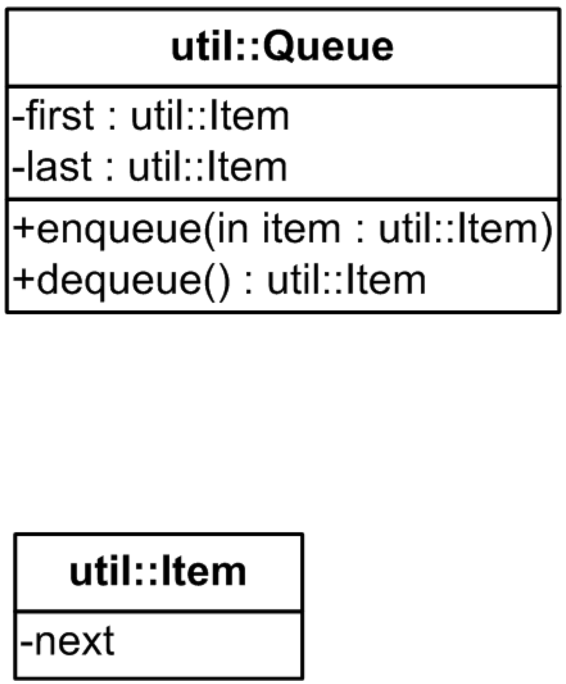


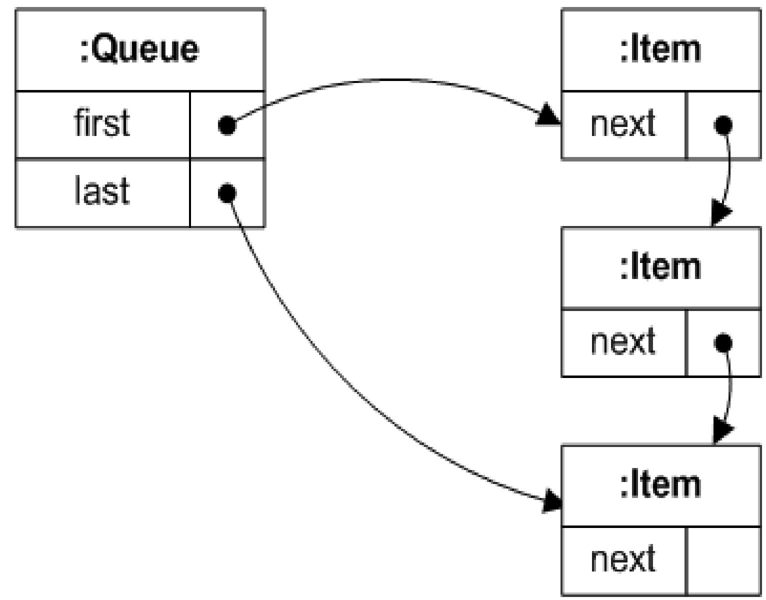
- Examples: tracing, synchronization, security, buffering, error handling, constraint checks, ...

Why AOP with C++?

- Widely accepted benefits from using AOP
  - avoidance of code redundancy, better reusability, maintainability, configurability, the code better reflects the design, ...
- Enormous existing C++ code base
  - maintainance: extensions are often crosscutting
- Millions of programmers use C++
  - for many domains C++ is the adequate language
  - they want to benefit from AOP (as Java programmers do)
- How does AspectC++ help?
  - it is the only actively maintained AOP extension for C++
  - combines AOP and C++ language features in a unique way

Scenario: A Queue utility class





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: 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!

The Not So Simple Queue Class

```cpp
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
```

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();
    return res;
  }
  int count() { return counter; }
}; // class Queue
```

Problem Summary

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

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

Aspect-Oriented Programming

with

AspectC++

Part II – AspectC++ Language

The Simple Queue Class Revisited

```
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
```

Queue: Demanded Extensions

I. Element counting

II. Errorhandling
(signaling of errors by exceptions)

III. Thread safety
(synchronization by mutex variables)

Element counting: The Idea

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

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 );
  }
};
```

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

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 );
  }
};
```

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

ElementCounter1.ah

ElementCounter1 - Elements

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

```
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 );
  }
};
```

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 am 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
  printf("main(): dequeueing one
  util::Item* item;
  item = queue.dequeue();
  printf("main(): Queue contains
}
```

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(): dequeueing one items
  > 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.
  - Additionaly, 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.
    - **e**xecution 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
    - nested entities are matched implicitly, e.g. “util” matches util::queue
    - 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

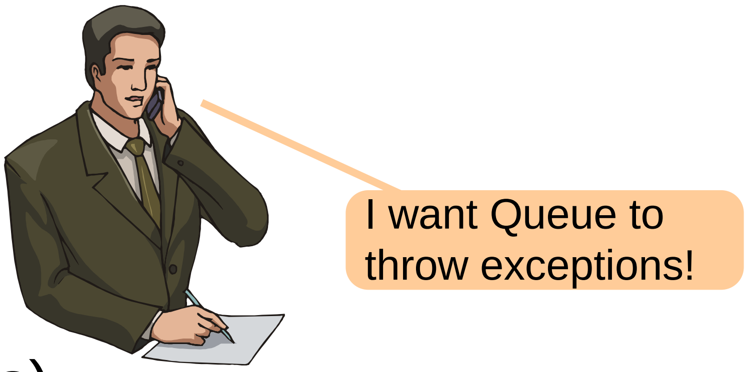
 public:
  element to introduce
};

```
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)

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

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

ErrorException - Elements

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

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

ErrorException.ah

ErrorException - Elements

```
namespace util {
  struct QueueInvalidItemEr
  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();
  }
};
```

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

ErrorException.ah

ErrorException - Elements

```
namespace util {
  struct QueueInvalidItemEr
  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();
  }
};
```

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

ErrorException.ah

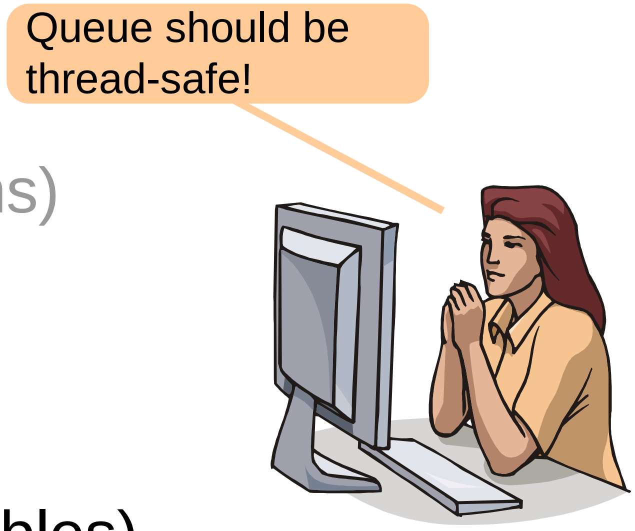
ErrorException – Lessons Learned

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)

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();
  }
};
```

LockingMutex.ah

We introduce a mutex member into class Queue

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();
  }
};
```

LockingIRQ1.ah

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

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();
}
```

main.cc

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

```
mai
os:
  > 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>

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();
  }
};
```

LockingIRQ2.ah

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 – 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()
  > Queue::enqueue(00321060)
  < Queue::enqueue()
back in main()
os::disable_int()
  > Queue::dequeue()
  < Queue::dequeue() returning 00320FD0
os::enable_int()
```

<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, ...
- Additionaly, AspectC++ provides some more advanced concepts and features
  - to increase the expressive power of aspectual code
  - to write broadly reusable aspects
  - to deal with aspect interdependence and ordering
- In the following, we give a short overview on these advanced language elements

AspectC++: Advanced Concepts

- Attributes
  - an alternative to named pointcuts; machine-readable intentions
- Join Point API
  - provides a uniform interface to the aspect invocation context
- Abstract Aspects and Aspect Inheritance
  - reuse parts of an aspect and overwrite others
- 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

Attributes

- can be used to annotate code (C++ 11 syntax)

```
[[OS::uninterrupted]] void enqueue( Item* item ) {
  printf( "  > Queue::enqueue()\n" );
...
}
```

The execution of this function must never be interrupted. We specifify **what** we want, but **not how** it is achieved.

- must be declared

(User-defined) attributes can be declared in a **namespace**, **class**, or **aspect**.

```
namespace OS {
  attribute uninterrupted();
}
```

- are an alternative to named pointcuts

```
advice execution(OS::uninterrupted())
&& !cflow(execution(kernel_code())) : around() { … }
```

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**

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
- nplate meta-programs

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 << *tjp->result();
  }
};
```

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

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

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

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

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++)

Summary

- AspectC++ facilitates AOP with C++
  - AspectJ-like syntax and semantics + C++-style generic code
- Full obliviousness and quantification
  - aspect code is given by **advice**
  - joinpoints are given declaratively by **pointcuts**
  - implementation of crosscutting concerns is fully encapsulated in **aspects**
  - **attributes** let programmers express their intentions in an aspect-readable manner
- Good support for reusable and generic aspect code
  - **aspect inheritance** and **virtual pointcuts**
  - rich **joinpoint API**

And what about tool support?

Aspect-Oriented Programming

with

AspectC++

Part III – Tool Support

Overview

- ac++ compiler
  - open source and base of the other presented tools
- ag++ wrapper
  - easy to use wrapper around g++ for make-based projects
- AspectC++ plugin for Eclipse®
  - sophisticated environment for AspectC++ development

About ac++

- Available from **www.aspectc.org**
  - Linux, Win32, MacOS binaries + source (GPL)
  - documentation: Compiler Manual, Language Reference, ...
- Transforms AspectC++ to C++ code
  - machine code is created by the back-end (cross-)compiler
  - supports g++ language extensions
- Current version: 2.2
  - front end is based on Clang 3.9.2

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

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

main.cc

```cpp
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

Tool Demo

- AspectC++ plugin for Eclipse®
  - sophisticated environment for AspectC++ development

Summary

- Tool support for AspectC++ programming is based on the ac++ command line compiler
  - full "obliviousness and quantification"
  - delegates the binary code generation to your favorite compiler
- Non-commercial IDE integration is available
  - Eclipse®

Aspect-Oriented Programming

with

AspectC++

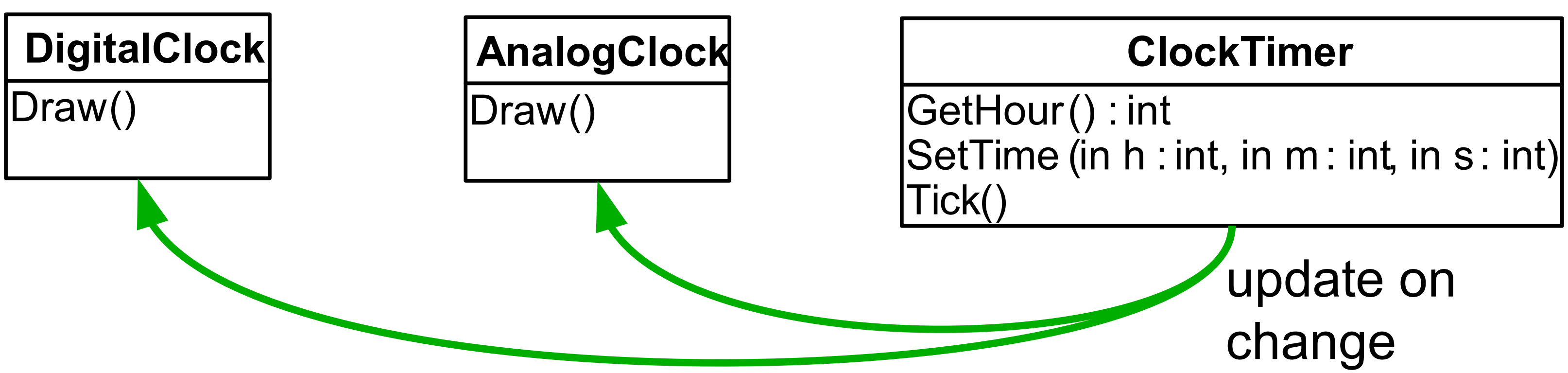
Part IV – Examples

AspectC++ in Practice - Examples

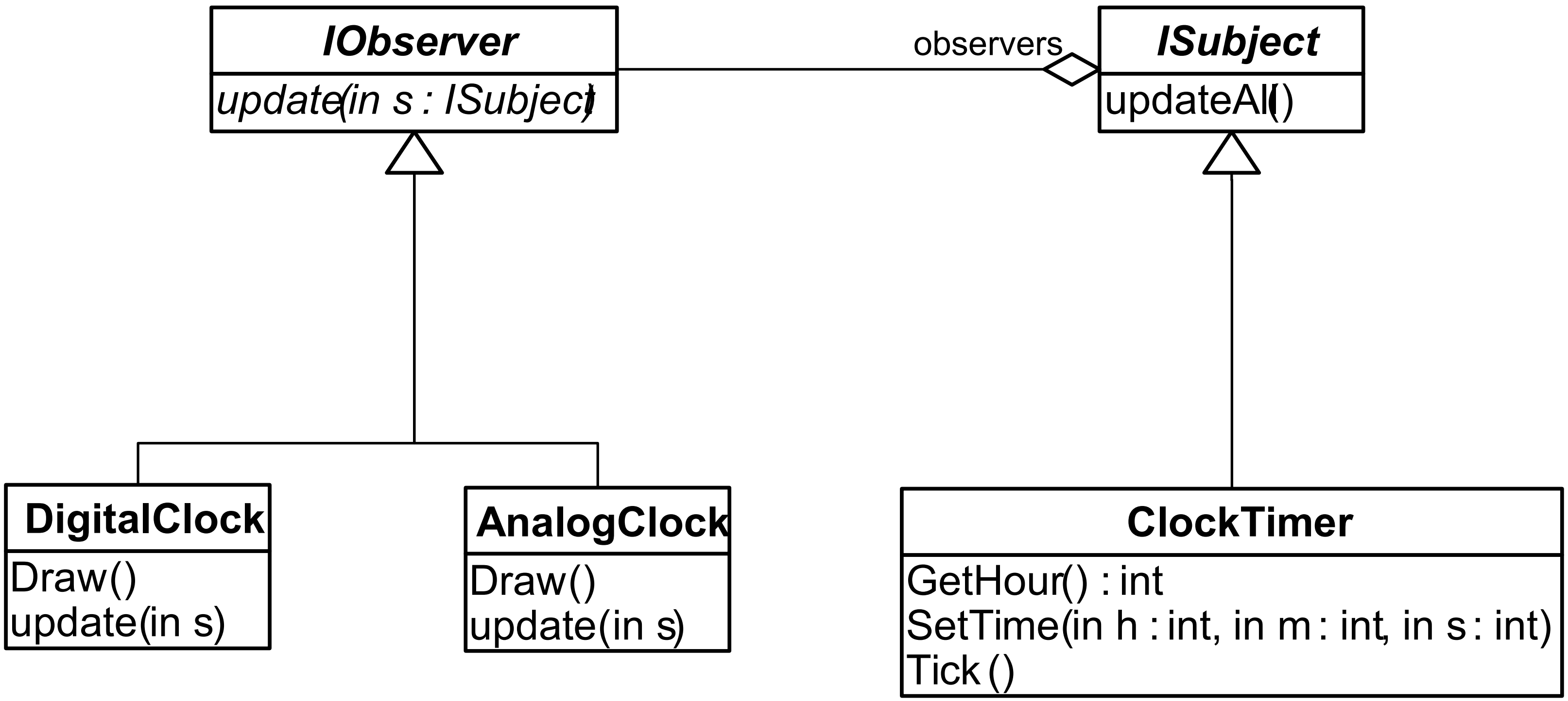
- **Applying the observer protocol**
  - Example: a typical scenario for the widely used observer pattern
  - Problem: implementing observer requires several design and code transformations

- **Errorhandling in legacy code**
  - Example: a typical Win32 application
  - Problem: errorhandling often “forgotten” as too much of a bother

Observer Pattern: Scenario

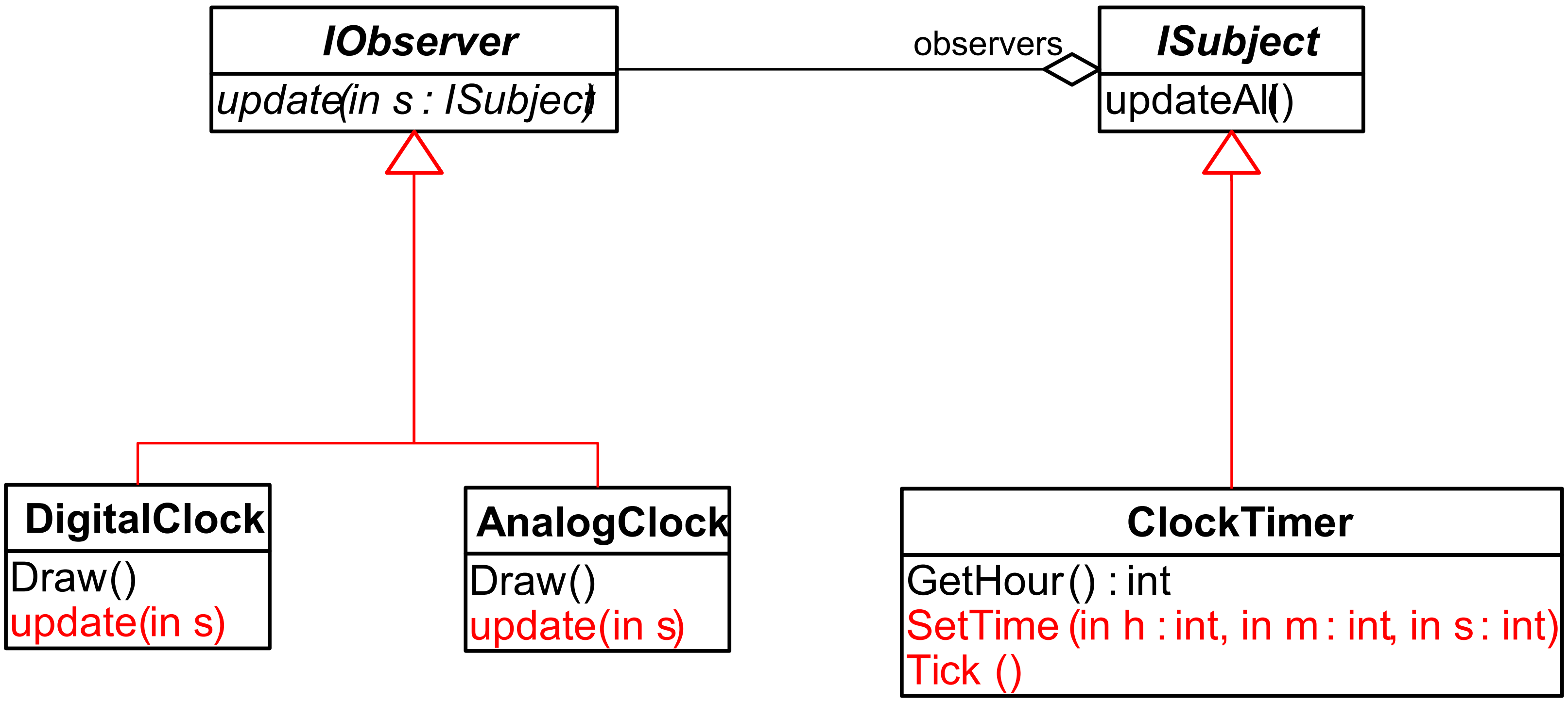


Observer Pattern: Implementation



Observer Pattern: Problem

The 'Observer Protocol' Concern...



...crosscuts the module structure

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

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

Interfaces for the subject/observer roles

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

abstract pointcuts that define subjects/observers

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

virtual pointcut defining all state-changing methods.

(Defaults to the execution of any

pinczyk

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

Introduction of the role interface as additional **baseclass**

aczyk

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

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

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();
    }
  };
};
```

Observer Pattern: Conclusions

- Applying the observer protocol is now very easy!
  - all necessary transformations are performed by the generic aspect
  - programmer just needs to define participants and behaviour
  - multiple subject/observer relationships can be defined
- More reusable and less error-prone component code
  - observer no longer “hard coded” into the desing and code
  - no more forgotten calls to update() in subject classes
- Full source code available at www.aspectc.org

Errorhandling in Legacy Code: Scenario

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

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

int WINAPI WinMain( ... ) {
  HANDLE hConfigFile = CreateFile( "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

Errorhandling in Legacy Code: Scenario

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

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

int WINAPI WinMain( ... ) {
  HANDLE hConfigFile = CreateFile( "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!

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

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

Examples

Detecting the Failure: Generic Advice

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

bool isError(ATOM);

bool isError(BOOL);

bool isError(HANDLE);

bool isError(HWND);

...

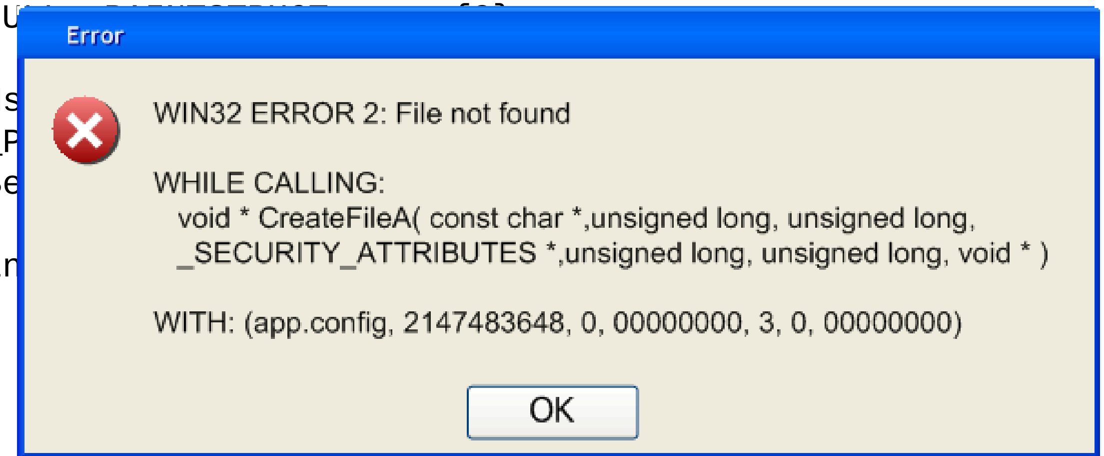
Error Reporting: Generative Advice

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

Error Reporting

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

  switch( nMs
    case WM_P
      dc = Be
      ...
      EndPain
      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;
}
```

Win32-Errorhandling: Conclusions

- Easy to apply errorhandling for Win32 applications
  - previously undetected failures are reported by exceptions
  - rich context information is provided
- Uses advanced AspectC++ techniques
  - error detection by generic advice
  - context propagation by generative advice
- Full source code available at www.aspectc.org

Aspect-Oriented Programming

with

AspectC++

Part V – Summary

Pros and Cons

- \+ A powerful AO language combining aspects with C++ concepts
- \+ Easy to learn
- \+ No overhead at runtime
- \+ Suitable for embedded and cross-platform development
- \+ IDE support available

- \- special tool (ac++/ag++) required
- \- longer compilation times

Summary – This Tutorial ...

- introduced the AspectC++ language extension for C++
  - AspectJ-like language extension
  - ac++ transforms AspectC++ into C++
  - supports AOP even in resource constrained environments
- demonstrated the AspectC++ tools
- discussed the pros and cons of the approach

Future Work – Roadmap

- Language design
  - consider/support/exploit new features of C++ >= 11
  - control flow patterns
  - free variables in pointcut expressions
- Aspect weaver
  - attributes for parameters
  - weaving in templates
  - aspect/advice templates
- Tools
  - file dependency handling (faster incremental builds)
  - `acmake` and whole-program analysis

Thank you for your attention!