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

AOSD 2007 Tutorial

University of Erlangen-Nuremberg
Computer Science 4

Presenters

- **Daniel Lohmann**
  `dl@aspectc.org`
  – University of Erlangen-Nuremberg, Germany

- **Olaf Spinczyk**
  `os@aspectc.org`
  – University of Erlangen-Nuremberg, Germany

Schedule

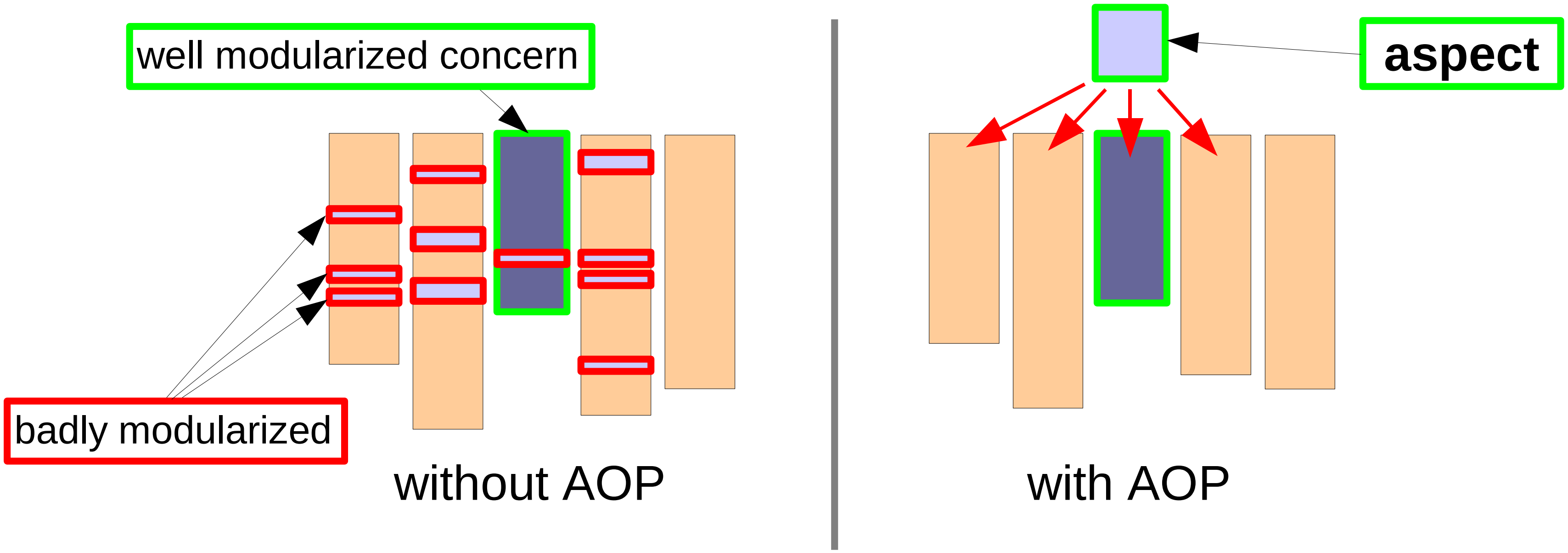
Part	Title	Time
I	Introduction	10m
II	AOP with pure C++	40m
III	AOP with AspectC++	70m
IV	Tool support for AspectC++	30m
V	Real-World Examples	20m
VI	Summary and Discussion	10m

This Tutorial is about ...

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

Aspect-Oriented Programming

- AOP is about modularizing crosscutting concerns

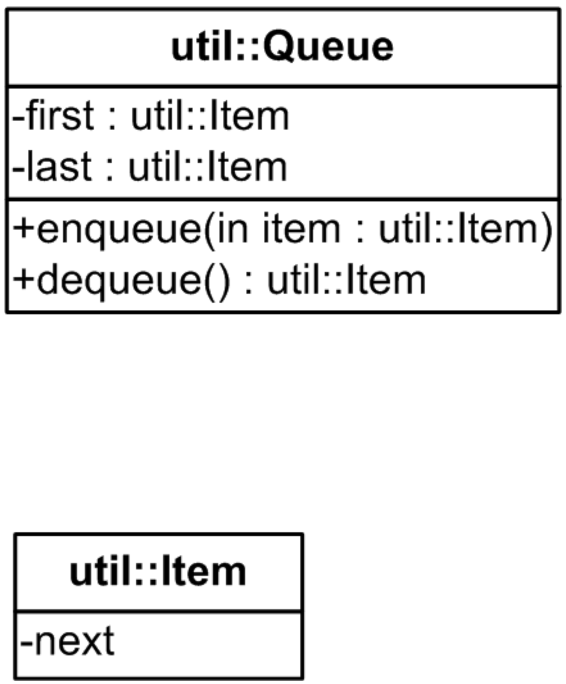


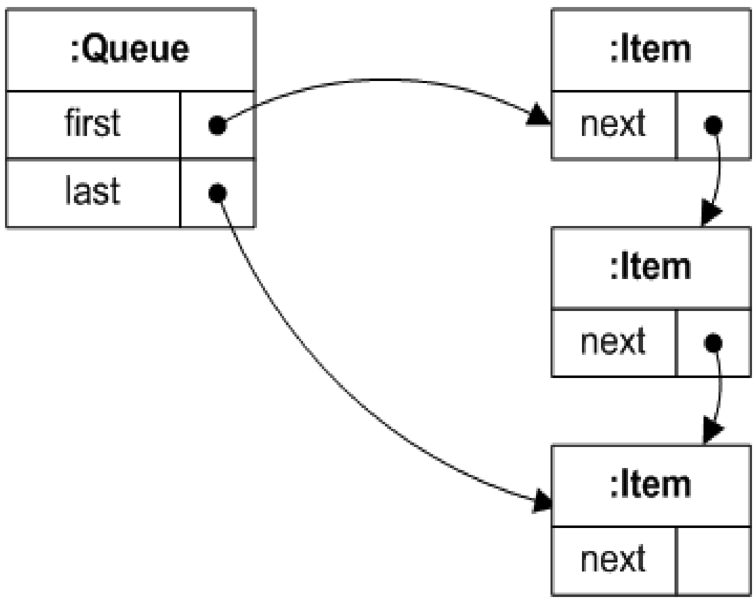
- Examples: tracing, synchronization, security, buffering, error handling, constraint checks, ...

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

Scenario: A Queue utility class





The Simple Queue Class

```
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

```
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 C++ and AspectC++

AOSD 2007 Tutorial

Part II – AOP with C++

Outline

- We go through the Queue example and...
  - decompose the "one-fits-all" code into modular units
  - apply simple AOP concepts
  - use only C/C++ language idioms and elements
- After we went through the example, we...
  - will try to understand the benefits
    and limitations of a pure C++ approach
  - motivate the need for an advanced language
    with built-in AOP concepts: AspectC++

Configuring with the Preprocessor?

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

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

Preprocessor

- While we are able to enable/disable features
  - the code is **not expressed in a modular fashion**
  - aspectual code is spread out over the entire code base
  - the code is almost unreadable
- Preprocessor is the "typical C way" to solve problems
- Which C++ mechanism could be used instead?

  Templates!

Templates

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

```
#define add1(T, a, b) \
  (((T)a)+((T)b))

template <class T>
T add2(T a, T b) { return a + b; }

printf("%d", add1(int, 1, 2));
printf("%d", add2<int>(1, 2));
```

Using Templates

Templates are typically used to implement generic abstract data types:

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

AOP with Templates

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

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

Aspects as Wrapper Templates

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

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

Weaving

We can define a type alias (typedef) that combines both, component and aspect code (**weaving**):

```
// component code
class Queue { ... }
// The aspect (wrapper class)
template <class Q>
class Counting_Aspect : public Q { ... }
// template instantiation
typedef Counting_Aspect<Queue> CountingQueue;

int main() {
  CountingQueue q;
  q.enqueue(new Item);
  q.enqueue(new Item);
  printf("number of items in q: %u\n", q.count());
  return 0;
}
```

Our First Aspect – Lessons Learned

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

Error Handling Aspect

Adding an error handling aspect (exceptions) is straight-forward. We just need a wrapper template:

```
// another aspect (as wrapper template)
template <class Q>
class Exceptions_Aspect : public Q {
  void enqueue(Item* item) { // this advice is executed before the
    if (item == 0)           // component code (before advice)
      throw QueueInvalidItemError();
    Q::enqueue(item);
  }

  Item* dequeue() { // after advice
    Item* res = Q::dequeue();
    if (res == 0) throw QueueEmptyError();
    return res;
  }
}
```

Combining Aspects

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

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

Ordering

- In what order should we apply our aspects?

  Aspect code is executed outermost-first:

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

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

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

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

Locking Aspect

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

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

```
  Item* dequeue() {
    Item* res;
    lock.enter();
    try {
      res = Q::dequeue(item);
    } catch (...) {
      lock.leave();
      throw;
    }
    lock.leave();
    return res;
  }
};
```

Locking Advice (2)

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

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

```
  Item* dequeue() {
    Item* res;
    lock.enter();
    try {
      res = Q::dequeue(item);
    } catch (...) {
      lock.leave();
      throw;
    }
    lock.leave();
    return res;
  }
};
```

Advice Code Duplication

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

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

```
  Item* dequeue() {
    Item* res;
    lock.enter();
    try {
      res = Q::dequeue(item);
    } catch (...) {
      lock.leave();
      throw;
    }
    lock.leave();
    return res;
  }
};
```

Dealing with Joinpoint Sets

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

```
template <class Q>
class Locking_Aspect2 : public Q {
public:
  Mutex lock;

  // wrap joinpoint invocations into action classes
  struct EnqueueAction {
    Item* item;
    void proceed(Q* q) { q->enqueue(item); }
  };

  struct DequeueAction {
    Item* res;
    void proceed(Q* q) { res = q->dequeue(); }
  };
  ...
```

Reusable Advice Code

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

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

Binding Advice to Joinpoints

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

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

Reusing Advice – Lessons Learned

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

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

Putting Everyting Together

We can now instantiate the combined Queue class, which uses all aspects:

(For just 3 aspects, the typedef is already getting rather complex)

```
typedef Locking_Aspect2<Exceptions_Aspect<Counting_Aspect
   <Queue> > > CountingQueueWithExceptionsAndLocking;

// maybe a little bit more readable ...

typedef Counting_Aspect<Queue> CountingQueue;
typedef Exceptions_Aspect<CountingQueue> CountingQueueWithExceptions;
typedef Locking_Aspect<CountingQueueWithExceptions>
   CountingQueueWithExceptionsAndLocking;
```

“Obliviousness”

... is an essential property of AOP: the component code should not have to be aware of aspects, but ...

- the extended Queue cannot be named “Queue”
  - our aspects are selected through a naming scheme (e.g. `CountingQueueWithExceptionsAndLocking`).
- using wrapper class names violates the idea of obliviousness

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

Hiding Aspects

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

Hiding Aspects (2)

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

// client code can import configuration namespace and use
// counting queue as “Queue”
using namespace configuration;

void client_code () {
  Queue queue; // Queue with all configured aspects
  queue.enqueue (new MyItem);
}
```

Obliviousness – Lessons Learned

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

What about obliviousness in the extended classes?

Limitations

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

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

Limitations (continued)

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

"AOP with pure C++ is like OO with pure C"

Conclusions

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

References/Other Approaches

K. Czarnecki, U.W. Eisenecker et. al.: *"Aspektorientierte Programmierung in C++"*, iX – Magazin für professionelle Informationstechnik, 08/09/10, 2001

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

A. Alexandrescu: *"Modern C++ Design – Generic Programming and Design Patterns Applied"*, Addison-Wesley, C++ in depth series, 2001

- Introduces "policy-based design", a technique for advanced separation of concerns in C++
- Policy-based design tries to achieve somewhat similar goals as AOP does
- http://www.moderncppdesign.com/

Other suggestions towards AOP with pure C++:

- **C. Diggins:** *"Aspect Oriented Programming in C++"*
  Dr. Dobb's Journal August, 2004
- **D. Vollmann**: *"Visibility of Join-Points in AOP and Implementation Languages"*
  http://i44w3.info.uni-karlsruhe.de/~pulvermu/workshops/aosd2002/submissions/vollmann.pdf