Aspect-Oriented Programming

with

AspectC++

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

www.aspectc.org

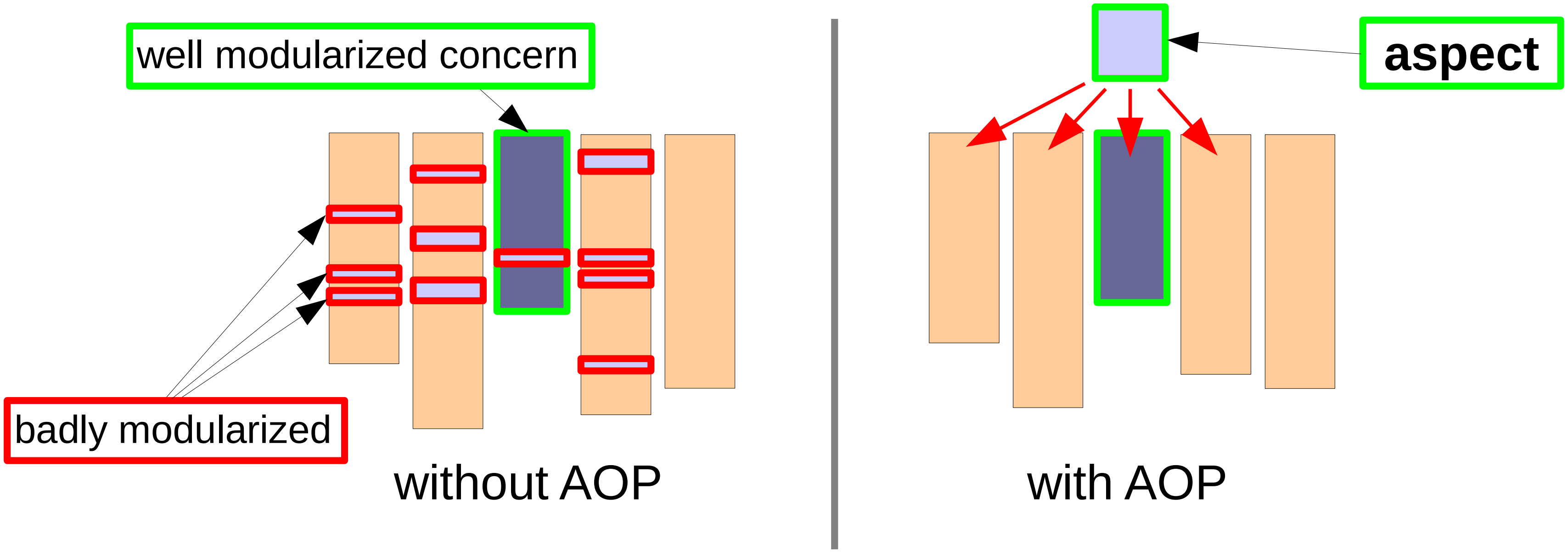
2017-05-22

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

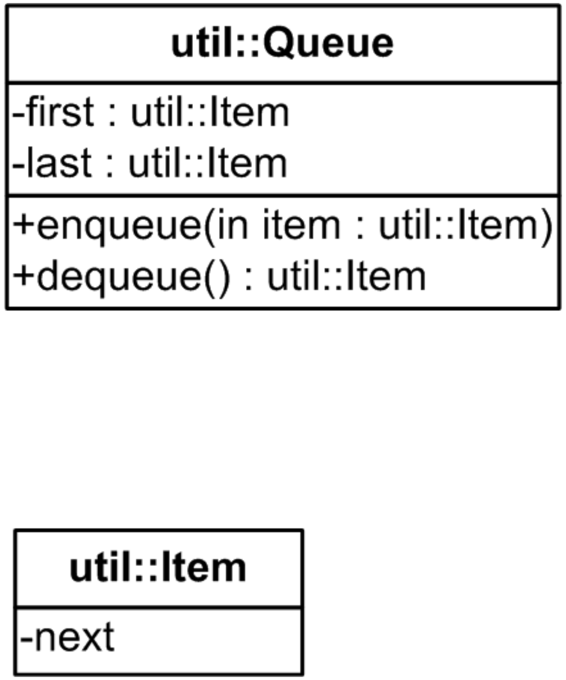


- Examples: tracing, synchronization, security, buffering, error handling, constraint checks, ...

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



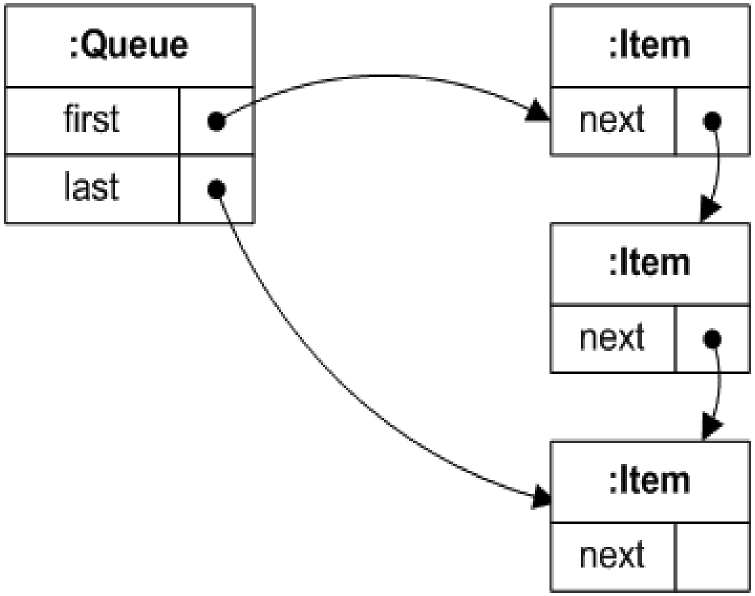
util::Queue

-first : util::Item
-last : util::Item

+enqueue(in item : util::Item)
+dequeue() : util::Item

util::Item

-next



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