Aspect-oriented Programming

Mateus Krepsky Ludwich
mateus@lisha.ufsc.br
http://www.lisha.ufsc.br

November 03, 2010

Introduction

- Why encoding important issues in a cleanly localized way (single code section)?
  - Because is better to:
    - Understand, analyze, modify, extend, debug, reuse, maintain, ...
- Object-oriented, generic, and component-oriented programming allow us that
- However there are issues that are difficult or impossible to express in a cleanly and localized way
  - these cross-cut the system and affect many classes

Cross-cutting concerns

- Shotgun surgery:
  "You whiff this when every time you make a kind of change, you have to make a lot of little changes to a lot of different classes."
- Cross-cut concerns
  - Examples: synchronization, security control, exception handling, logging, caching, persistence

Composition mechanisms

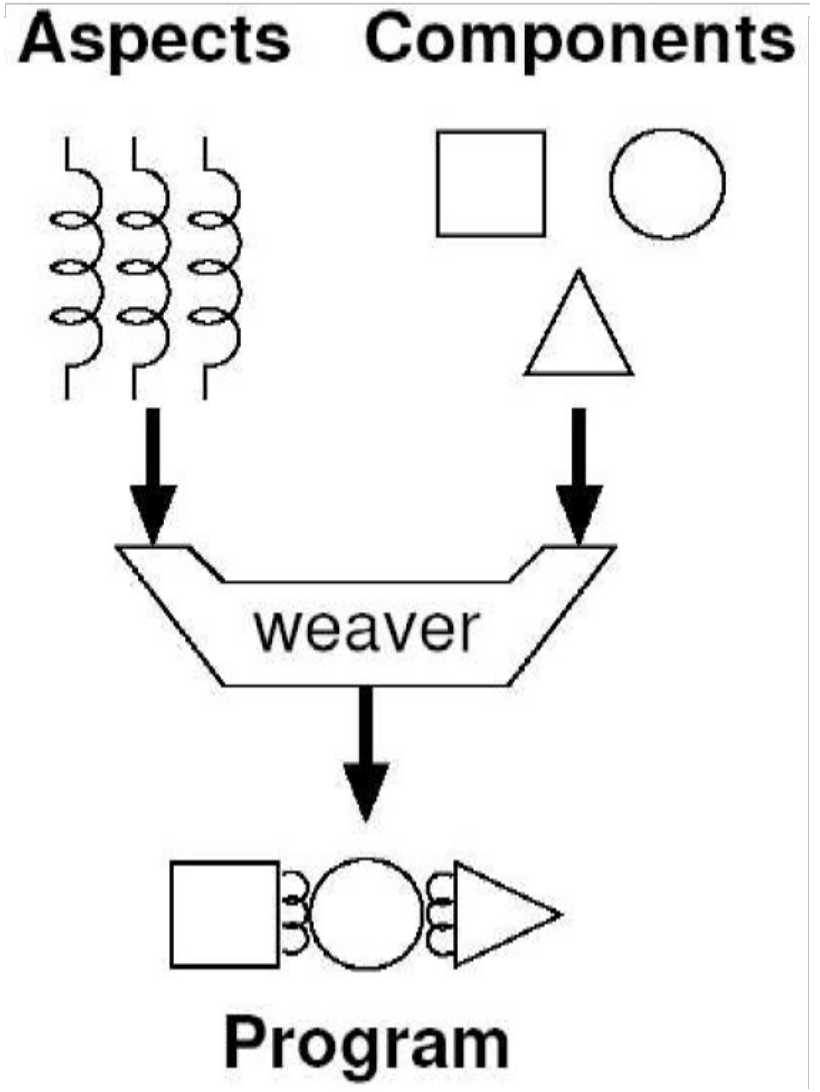
- Conventional
  - function calls
  - dynamic and static parametrization
  - Inheritance
- Aspect-oriented
  - Composition rules in SOP - Subject-oriented programming
  - Message filters in CF - Composition Filters
  - Transversal strategies in Demeter
  - Join point models

Join point models

- Join points
  - Points in a running program where additional behavior can be usefully joined
- Pointcuts
  - A way to specify (or quantify) join points
- Advices
  - Code that runs at pointcuts
- Aspect
  - The combination of the pointcut and the advice

Aspect-Oriented Programming

- Deals with cross-cutting concerns
  - abstracts non-functional properties
  - reduces replicated code
  - are reusable
- A new construct: *Aspect*
  - are woven with components



Synchronized stack example

- Constrains
  - Push only when is not full
  - Pop only when is not empty
  - Push, self exclusive
  - Pop, self exclusive
  - Push and pop, mutually exclusive

Tangled version

- Tangled version
  - Aspect (synchronization) code manually coded and mixed with the functional code
  - Non-intentional representation of the synchronization aspect
  - Unnecessary overhead in a single-thread scenario

```
#include "ace\Synch.h"                                                    //line  1
                                                                         //line  2
template<class Element, int S_SIZE>                                      //line  3
class Sync_Stack                                                         //line  4
{ public:                                                                //line  5
    enum {                                                               //line  6
      EMPTY         = -1,       // top value for empty stack           //line  7
      ONE_TOP       = EMPTY+1,  // top value if one element in stack   //line  8
      MAX_TOP       = S_SIZE-1, // maximum top value                   //line  9
      UNDER_MAX_TOP = MAX_TOP-1 // just under the maximum top value    //line 10
    };                                                                   //line 11
    Sync_Stack()                                                         //line 12
      : top (EMPTY),                                                     //line 13
        push_wait (lock),                                                //line 14
        pop_wait  (lock) { };                                            //line 15
    void push(Element *element)                                          //line 16
    { ACE_Guard<ACE_Thread_Mutex> monitor (lock);                        //line 17
      while (top == MAX_TOP) push_wait.wait();                           //line 18
      ACE_DEBUG ((LM_DEBUG, "(%t) push: top = %d\n", top));
      elements [++top] = element;                                        //line 19
      if (top == ONE_TOP) pop_wait.signal();    //signal if was empty    //line 20
      // the lock is unlocked automatically                              //line 21
      // by the destructor of the monitor                                //line 22
    }                                                                    //line 23
    Element *pop()                                                       //line 24
    { Element *return_val;                                               //line 25
      ACE_Guard<ACE_Thread_Mutex> monitor (lock);                        //line 26
      while (top == EMPTY) pop_wait.wait();                              //line 27
      ACE_DEBUG ((LM_DEBUG, "(%t) pop : top = %d\n", top));
      return_val = elements [top--];                                     //line 28
      if (top == UNDER_MAX_TOP) push_wait.signal(); //signal if was full//l. 29
      return return_val;                                                 //line 30
    }                                                                    //line 31
  private:                                                               //line 32
    // synchronization variables                                        //line 33
    ACE_Thread_Mutex lock;                                               //line 34
    ACE_Condition_Thread_Mutex push_wait;                                //line 35
    ACE_Condition_Thread_Mutex pop_wait;                                 //line 36
                                                                         //line 37
    // stack variables                                                  //line 38
    int top;                                                             //line 39
    Element *elements [S_SIZE];                                          //line 40
};                                                                       //line 41
```

Parametrized inheritance version

- Reuse the synchronization wrapper for different stack implementations (e.g. stacks using different data structures for storing their elements)
- Multi-thread scenario stills checks for error (in push and pop) operations, although the checking is not needed in this case. Can be solved by another wrapper level...

```
template<class Element, int S_SIZE>
class Stack
{  public:
      // export element type and empty and maximum top value
      typedef Element Element;
      enum { EMPTY   = -1,
             MAX_TOP = S_SIZE-1};

      // classes used as exceptions
      class Underflow {};
      class Overflow  {};

      Stack() : top (EMPTY) {}
      void push(Element *element)
      {   if (top == MAX_TOP) throw Overflow(); //stack full!
          elements [++top] = element;
      }
      Element *pop()
      {   if (top == EMPTY) throw Underflow(); //stack empty!
          return elements [top--];
      }
   protected:
      int top;
   private:
      Element *elements [S_SIZE];
};
```

```
template<class UnsyncStack>
class Sync_Stack_Wrapper : public UnsyncStack
{  public:
     // get the element type and empty and maximum top value
     typedef typename UnsyncStack::Element Element;
     enum { EMPTY   = UnsyncStack::EMPTY,
            MAX_TOP = UnsyncStack::MAX_TOP};
     // declare ONE_TOP and UNDER_MAX_TOP
     enum { ONE_TOP       = EMPTY+1,
            UNDER_MAX_TOP = MAX_TOP-1};

     Sync_Stack_Wrapper()
           : UnsyncStack (),
             push_wait (lock),
             pop_wait (lock) { }
     void push(Element *element)
     {    ACE_Guard<ACE_Thread_Mutex> monitor(lock);
          while (top == MAX_TOP) push_wait.wait();
          UnsyncStack::push(element);
          if (top == ONE_TOP) pop_wait.signal(); // signal if was empty
     }
     Element *pop()
     {   Element *return_val;
          ACE_Guard<ACE_Thread_Mutex> monitor (lock);
          while (top == EMPTY) pop_wait.wait();
          return_val = UnsyncStack::pop();
          if (top == UNDER_MAX_TOP) push_wait.signal();  // signal if was full
          return return_val;
     }
   private:
     // synchronization variables
     ACE_Thread_Mutex lock;
     ACE_Condition_Thread_Mutex push_wait;
     ACE_Condition_Thread_Mutex pop_wait;
};
```

AspectJ + Cool version

- Cool
  - an aspect language for expressing synchronization in concurrent OO programs
  - Implemented in AspectJ 0.1.0 (October 2010: 1.6.10)
- One language for each aspect it addressed
  - Cool – synchronization
  - Ridl – remote invocation

```
//in a separate Java file
public class Stack
{    private int s_size;

     public Stack(int size)
     {    elements = new Object[size];
          top = -1;
          s_size = size;
     }
     public void push(Object element)
     {    System.out.println("push: top = " + top);
          elements[++top] = element;
     }
     public Object pop()
     {    System.out.println("pop : top = " + top);
          return elements[top--];
     }

     private int top;
     private Object [] elements;
}
//in a separate Cool file
coordinator Stack
{    selfex push, pop;
     mutex {push, pop};
     condition full=false, empty=true;

     guard push:
         requires !full;
         onexit
         {     if (empty) empty=false;
               if (top==s_size-1) full=true;
         }
     guard pop:
         requires !empty;
         onexit
         {     if (full) full=false;
               if (top==-1) empty=true;
         }
}
```

Expressing Aspects in Programming Languages

- Implementing aspect-specific abstractions
  - Conventional library
    - Sometimes is the only choice
    - E.g. Dynamic Cool in Smaltalk
  - Design a separated language for the aspect
    - E.g. Cool, Ridl
  - Design a language extension for the aspect
    - Differs from the previous one in technology rather than at the language level
    - Uses the same compilation infrastructure that the “conventional” language

Expressing Aspects in Programming Languages

- Implementing weaving
  - source-to-implementation transformation
    - Tangle code, containing aspect and functional code generated at compile time
    - E.g. AspectJ + Cool
  - dynamic reflection
    - Interpreted at runtime and the control is transferred between the aspects as often as necessary
    - E.g. Dynamic Cool in Smaltalk

Conclusions

- AOP provides a way for capturing important aspects of systems in a cleanly localized way, that generalized procedures aren't capable of
- Introduces a new style of decomposition
  - aspects componentization
- Multiparadigm view: OO + AO

References

- Czarnecki, K. and Eisenecker, U. W. 2000 Generative Programming: Methods, Tools, and Applications. ACM Press/Addison-Wesley Publishing Co.
- Fowler et al., 1999 Refactoring: Improving the Design of Existing Code. Addison-Wesley.