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"

template<class Element, int S_SIZE>
class Sync_Stack
{
  public:
    enum {
      EMPTY = -1, // top value for empty stack
      ONE_TOP = EMPTY+1, // top value if one element in stack
      MAX_TOP = S_SIZE-1, // maximum top value
      UNDER_MAX_TOP = MAX_TOP-1 // just under the maximum top value
    };
    Sync_Stack()
      : top (EMPTY),
        push_wait (lock),
        pop_wait (lock) {}
    void push(Element *element)
    {
      ACE_Guard<ACE_Thread_Mutex> monitor (lock);
      while (top == MAX_TOP) push_wait.wait();
      ACE_DEBUG ((LM_DEBUG, "(%t) push: top = %d\n", top));
      elements [++top] = element;
      if (top == ONE_TOP) pop_wait.signal(); // signal if was empty
      // the lock is unlocked automatically
      // by the destructor of the monitor
    }
  Element *pop()
  {
    Element *return_val;
    ACE_Guard<ACE_Thread_Mutex> monitor (lock);
    while (top == EMPTY) pop_wait.wait();
    ACE_DEBUG ((LM_DEBUG, "(%t) pop : top = %d\n", top));
    return_val = elements [top--];
    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;

    // stack variables
    int top;
    Element *elements [S_SIZE];
};
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 Unsynchronized>

class Sync Stack Wrapper : public Unsynchronized
{
    public:
        // get the element type and empty and maximum top value
        typedef typename Unsynchronized::Element Element;
        enum { EMPTY = Unsynchronized::EMPTY,
            MAX_TOP = Unsynchronized::MAX_TOP};
        // declare ONE_TOP and UNDER_MAX_TOP
        enum { ONE_TOP = EMPTY+1,
            UNDER_MAX_TOP = MAX_TOP-1};

        Sync_Stack_Wrapper()
            : Unsynchronized (),
                push_wait (lock),
                pop_wait (lock) {}
        void push(Element *element)
        { ACE_Guard<ACE_Thread_Mutex> monitor(lock);
            while (top == MAX_TOP) push_wait.wait();
            Unsynchronized::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 = Unsynchronized::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 Smalltalk
  - 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
