

Aspect of Life-Cycle Control in a C++ Framework

Lutz Dominick

Siemens AG, Corporate Technology, ZT SE 1

D-81730 Munich, Germany

Lutz.Dominick@mchp.siemens.de

April 1999

Abstract

This paper presents some of the experiences made with AOP. In an object oriented C++ framework, aspects have been identified which now have been separated from the components of the framework. In this paper, the focus is on life-cycle control aspects when applying the Command Processor pattern. Because no weaver technology was available, preprocessor macros were used. They simulate both operation and statement-level join points. Making the aspects of the code explicit yielded positive results.

Introduction

Recently, I came across AOP and wanted to evaluate the idea. Even more, after having read a couple of papers I wanted to profit from AOP immediately, although I am a novice in AOP.

From experience it was well known, that the API of a framework never reflected the internal complexity. Growing complexity meant to make relationships among components more complex and/or put more and more code into methods that actually already did their job. The latter option often seemed to allow higher performance, and more inheritance can hamper concurrency.

The problem for code reviews, debugging, maintenance, and enhancements now was evident: Those extra pieces of code were tied together in no way any more, although they had been introduced for very specific reasons and could have been taken out at any time also. Those very specific reasons form the aspects that need to be separated.

For example, aspects of frameworks that result in operation join points are error detection and handling, logging, and tracing [2], or synchronization issues [4]. Here, the focus is on the life-cycle issues imposed by the Command Processor Pattern. My colleagues describe the pattern in detail in [1]. The implementation uses the Adaptive Communication Environment (ACE) framework, which is freely available, see [3].

The investigated C++ framework is used in the products of one of the Siemens business units as a core component, the software for those products can contain several million lines of code.

AOP terms

Aspect Oriented Programming (AOP) allows to “express the different aspects of a software system in a separate and natural form, and then automatically combine those separate descriptions into a final executable” [5]. Those aspects

crosscut the software, they are not captured as a single design entity that results from decomposition, but they can be looked at as some of the properties of the software.

Such a property “must be implemented as a component, if it can be clearly encapsulated in a generalized procedure”, and as “an aspect, if it can not be clearly encapsulated in a generalized procedure. Aspects tend not to be units of the system’s functional decomposition but rather to be properties that affect the performance or semantics of the components in systemic ways” [6].

Join points are those places in the software, where aspect code gets inserted into a component. Aspect weavers process the component and aspect languages in order to build the complete software system. They work by “generating a join point representation of the component program, and then executing (or compiling) the aspect programs with respect to it” [6].

Types of join points describe how a cross-cutting concern affects code at one or more join points. Cross-cutting concerns that “affect the behavior of groups of operations typically require the use of operation join points”, like error detection after opening a file, while “statement-level join points indicate that they can occur in any part of code”. The latter can mean changing an existing single line of code [2],

Life-Cycle Aspects

In C++, watching the life-cycle of objects and components is vital. Errors result in crashes or memory leaks. In multi-threaded systems, the control flow for life-cycles spans many locations that execute independent of each other. At the same time, there are several possibilities of memory allocation, for example:

- From stack, or from the heap using new/delete, alloc/free
- From a memory pool where the memory is bound or unbound
- Allocating and releasing memory can be hosted by the same thread or by different threads

For objects, instances of classes in C++, several ways of caching are available, for example:

- Caches for one type or multiple types
- Caches that are global to the process
- Caches that are private to one object or a set of objects
- Caches that are bound to one thread

The state of an object can also require the application of rules covered in aspects. For example:

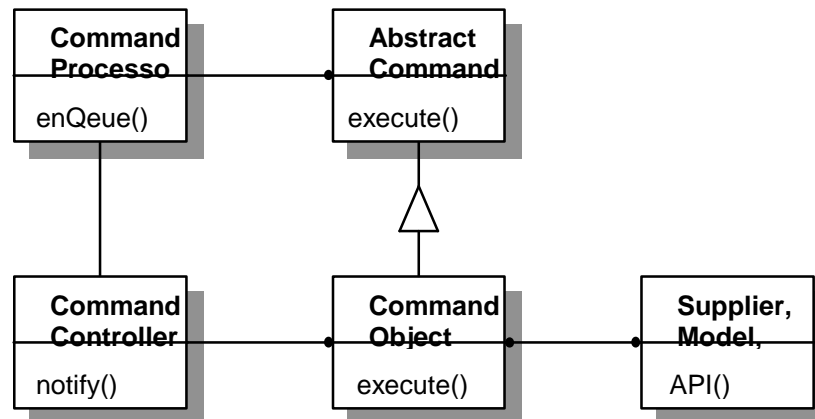
- A method can re-occur in a call stack due to nesting operations. Then, state needs to be organized independent of the nesting level. State needs to be “local” to each nesting level.
- An object is required to allow multiple threads to call the same or different methods simultaneously without serialization on locks. Then, state needs to be organized to appear “local” to each of the callers.

Each choice from the above points affects the overall system performance, memory footprint over time, heap fragmentation and others.

For a certain type of object, the life-cycle control flow is designed into the system, specifying both the location in the code and the operation that is chosen. But performance tests, for example, might show that the choice was unacceptable. Changing the code by hand is dangerous in large systems. Changing the implementation of the adherent aspect is not. Everything is modeled in one place and in a concise way.

Example: Command Processor Pattern

The static diagram for the Command Processor Pattern contains the following participants:



The command controller gets notified of an incoming command in one thread. The command then is enqueued with the command processor in that thread. The command processor has a pool of threads that dequeue commands and execute command objects which can access the application model and suppliers.

For this example, the code of the classes is a sketch only:

```

class CommandController {
    notify() { // get a command
        cmd = new Command( obj );
        ::readBuffer( cmd );
        CommandProcessor::instance()->enqueue( cmd );
        // Command Processor is a process singleton
    }
    open() {
        obj = new CommandObject;
    }
    close() {
        delete obj;
    }
};

class CommandProcessor: public ACE_Task {
    enqueue( Command* cmd ) {
        mb = new ACE_Message_Block( cmd );
        putQueue( mb );
    }
    svc() { // run per thread of thread pool
        while( true ) {

```

```

        mb = getQueue() )
        execute( mb->command );
        delete mb;
    }
}
execute( Command* cmd ) {
    cmd->obj->execute( cmd ); // run Command Object
    delete cmd;
}
};

```

Each command contains the parameters for its execution and contains a reference to its command object.

In the above code sketch, a couple of creation and destruction aspects are covered:

- The message queue in ACE always contains message blocks of ACE which contain the data to be queued. The rules for the aspect are: A message block is created in *enQueue()* before *putQueue()*, and it is destroyed in *svc()* after the execution of the command. The design leaves both creation and destruction in the command processor, but two threads are involved.
- The controller creates a new command and fills its data from a buffer, before *enQueue()*, and the command is deleted after its execution in command processor *execute()*, which are the rules for the life-cycle aspect of a command. Here, two objects and two threads are involved.
- The controller creates and destroys the Command Object
- The command processor is a singleton, according to the singleton pattern [1].

The command processor life-cycle is now left out. For simplicity, the above code has created all objects from the heap and destroyed them in the same way. But the scenario allows for or even requires some enhancements, for example:

- Message Blocks could be kept in a global message block pool that is accessed by all threads or the process. This allows for re-using empty blocks for new commands in order to gain higher performance.
- Commands could be kept in a small pool in every command object, because their type is specific to that type of command object. Commands that are done go into the pool.
- The command object has a history mechanism that applies to some of the commands. It first stores the commands on the history stack and then keeps those in the pool that drop out of the history stack (the scenario becomes more complex if the command object decides to overtake the life-cycle of only some of the commands, instead of copying commands).
- All command objects need to be stored somewhere to allow safe shut-down.
- Creation and destruction of a command object can be much more complicated. All command objects might announce shut-down first, then we wait for pending commands to be executed on each command object, and then all get destroyed.

All of the above points are aspects of the command processing system. But they go “extra” because they are only aspects of the implementation. The code should not be tangled.

The mere creation and destruction aspects require statement-level changes in the above code, the bold lines form the statement-level join points.

The command history aspect and the command object shut-down aspect require additional operations, whenever a command or a command object needs to be destroyed. They form operation join points.

The aspects are defined as pieces of code that get woven with the program code according to the aspect rules. Here, they are modeled using a macro style language because no specific weaver or aspect oriented language has been applied yet. The rules are only comments and the macros are inserted by hand appropriately. As for the program code, only sketches for the macros are presented that are simplified. The intended caches are available from a class library, operations inside the caches are not modeled.

```
//// Aspect: Command Object life-cycle
// apply at end of CommandController::open method per command object
CMD_OBJ_CREATION( type )      CmdObjCache ->set(new type);
// apply at begin of CommandController::close method
CMD_OBJ_DESTRUCTION()      CmdObjCache->clear();

//// Aspect: Command life-cycle
// apply before calling CommandProcessor::enqueue()
CMD_CREATION( cmd, obj )      cmd = obj->cache.alloc();
// apply after command execution
CMD_DESTRUCTION( cmd )      cmd->obj->cache.free( cmd );

//// Aspect: Message Buffer life-cycle
// apply at begin of CommandProcessor::enqueue method
MSG_BLOCK_CREATION( mb, param )      mb = MBCache->alloc( param );
// apply after call to CommandProcessor::execute method
MSG_BLOCK_DESTRUCTION( mb )      MBCache->free( mb );

//// Aspect: Safe shut-down of Command Objects
// apply before destruction of command objects
CMD_OBJ_SHUT_DOWN()
    for( all obj = CmdObjCache->iterate() ) obj->announceShutDown();
    for( all obj = CmdObjCache->iterate() ) obj->waitForPending();

//// Aspect: History mechanism
// apply at end of CommandObject::execute after a positive check that
// command gets stored in history stack (sample code in comments);
// uses a singleton in thread-specific storage as the mediator
HIST_OVERTAKE_LIFE_CYCLE( cmd )
    //If( cmd->noHistory() ) return;
    TSS_State::instance()->dontDestroy( cmd );
    //if( history.len() > max ) CMD_DESTRUCTION( history.get() );
    //history.push( cmd )
// apply before command destruction
HIST_CHECK_LIFE_CYCLE( cmd )
    if( TSS_State::instance()->dontDestroy( cmd ) ) return;
```

```

// apply at begin of CommandProcessor::execute method
// this is required if nested command calls occur, the state guard
// stores the TSS state on the call stack and restores it when
// leaving the method, TSS state might be more complex
HIST_GUARD_TSS_STATE()
    TSS_Guard mon( TSS_State::instance()->getState() );
    TSS_State::instance()->doDestroy(cmd); // inits TSS state

```

The implementation of the aspects can change independently of the system implementation, aspects can also be removed.

The history aspect affects the control flow for the command life-cycle. The rules for the control flow are still obvious (If the command was copied, the copy got onto the history stack, but the new implementation might decrease the overall performance). The history mechanism requires additional design steps and changes in the system code, because the aspect is modeling only additional life-cycle issues and not the entire mechanism, which should also be possible.. Then, removing the aspect meant removing a feature.

What we got so far is woven into the code sketches below:

```

class CommandController {
    notify() { // get a command
        CMD_CREATION( cmd, obj )
        ::readBuffer( cmd );
        CommandProcessor::instance()->enqueue( cmd );
        // Command Processor is a process singleton
    }
    open() {
        CMD_CREATION( PrintService ) // just a sample
    }
    close() {
        CMD_OBJ_SHUT_DOWN()
        CMD_OBJ_DESTRUCTION()
    }
};

class CommandProcessor: public ACE_Task {
    enqueue( Command* cmd ) {
        MSG_BLOCK_CREATION( mb, cmd )
        putQueue( mb );
    }
    svc() { // run per thread of thread pool
        while( true ) {
            mb = getQueue() );
            execute( mb->command );
            MSG_BLOCK_DESTRUCTION( mb )
        }
    }
    execute( Command* cmd ) {
        HIST_GUARD_TSS_STATE()
        cmd->obj->execute( cmd ); // run Command Object
        HIST_CHECK_LIFE_CYCLE()
        CMD_DESTRUCTION( cmd )
    }
};

```

```

class CommandObject: public AbstractCommand {
    execute( Command* cmd ) {
        // do the work
        If( cmd->noHistory() ) return;
        HIST_OVERTAKE_LIFE_CYCLE( cmd )
        if( history.len() > max )
            CMD_DESTRUCTION( history.get() );
        history.push( cmd )
    }
};

```

Concluding Remarks

Aspects help to organize program code where design could tangle the system code. At the same time, control is re-gained over the aspect code. This allows for more flexible evolution of the software, and, to my opinion, is one of the means against software's "death thru maintenance". The existing aspects can be reused in the same or other software systems.

Perhaps aspects could function as the natural extension to code generation tools like wizards and case tools.

The current state of work lacks the application of weaver technology. So more complex aspects cannot be handled at all. Macros cannot be treated as an equivalent to a weaver. Other issues also need to be answered, like debugging the complete system.

References

- [1] Frank Buschmann, Regine Meunier, Hans Rohnert Peter Sommerlad, Michael Stal: A System of Patterns, Wiley and Sons Ltd, 1996
- [2] Hardol Ossher, Peri Tarr: Operation-Level Composition: A Case in (Join) Point, Proceedings of ECOOP98
- [3] Douglas C. Schmidt, Washington University of St. Louis, home page at <http://www.siesta.cs.wustl.edu/~schmidt>
- [4] David Holmes, James Noble, John Potter: Aspects of Synchronization, Proceedings of ECOOP97
- [5] Gregor Kiczales et al.: Aspect Oriented Programming: A Position paper from the Xerox PARC Aspect Oriented Programming Project", Xerox PARC, 1996
- [6] Gregor Kiczales, John Lamping, Anurag Medbeker, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, John Irving: Aspect Oriented Programming, ECCOP97