

Position Paper: A Different Need for Sequencing Contracts Using State-Based Aspects

James Heliotis

Rochester Institute of Technology
Rochester, NY, 14623-5608 USA
01.585.475.6133

jeh@cs.rit.edu

ABSTRACT

Aspect technology has been used to describe concerns that had been thought to be unmodularizable. Some of those concerns have to do with workflows in an application's domain and sequencing of operation invocations within the application. Another principle of software engineering that has generated some interest and controversy within the aspect community is *design by contract*. Over the years some have suggested that a component's contracts be expressed through one or several aspects attached to it. This paper will describe a use of aspects that is a hybrid of the above ideas: the enforcement of operation invocation ordering through the component's contract and expressed as an aspect. The aspect is in turn modeled with a state machine whose description can be used to enhance the clarity of a component's description at the analysis and design phases, and whose implementation can be used to monitor invocations on the component for proper sequencing.

Categories and subject descriptors: D.2.5 [Software Engineering]: Testing and Debugging; D.2.3 [Software Engineering]: Coding Tools and Techniques; I.6.5 [Simulation and Modeling]: Model Development; K.6.3 [Computing Milieux]: Software Management

General Terms: Design, Languages.

Additional Key Words and Phrases: State-base modeling, Aspect-oriented

1. INTRODUCTION

In this section the intended meanings of the key terms of the paper, sequence checking, design by contract, and aspect, will be explained.

1.1 Sequence Checking

In the software solution spaces for embedded systems, state-controlled sequencing has always been a large part of any software system designed to control machinery or to coordinate

components of a distributed system. Actions must be done in an order that does not damage physical hardware or that does not somehow confuse other software components. At the other end of the spectrum, the proper sequence of software actions is determined by actor-based workflows, of which the software may only be a small part.

In this paper we focus on operational sequencing that is necessary for proper software operation but does not need to be realized in the software's design in order for the software to function. Some examples of this category follow.

- When reading data from an input device via a buffered channel, one must fill the buffer first before reading the data from it.
- If a recursive algorithm is implemented in such a way that it must allocate work areas on a stack, variables within those work areas should not be used in computations before they have been assigned values.
- There are counting problems such as restricting the number of instances of a class or requiring that only two threads interact with a given instance of a synchronization primitive.

In order to enforce rules like these, additional overhead must be added in the form of flags or some kind of null value. As long as the client software follows the rule, i.e. is bug-free, this overhead is technically unnecessary.

- Data structures can often be in states where certain operations are inappropriate. A simple example is removing an element from an empty queue. In this example, the state necessary to determine the usage error is already present, but checking it is still execution overhead. Of course this is the classic design-by-contract notion of a precondition.
- Finally, in the control software for a movie film projector, it is very important to power down the projector lamp before stopping the film drive mechanism when film is loaded in the projector to avoid damage to the film. This would fit into the first category of there being no need for the state information except for the fact that the risk of loss from an error is high enough that the overhead in the design would probably be acceptable.

In section 3.1 of the paper I elaborate on another example regarding proper sequence of steps in a distributed, computer-controlled game.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Workshop BPOAOSD '07, March 12-13, 2007
Vancouver, British Columbia, Canada.

Copyright 2007 ACM 1-59593-662-2/07/03...\$5.00.

1.2 Design by Contract

The design-by-contract principle, or DBC, promotes the idea that software components are not well specified until each operation is completely specified in terms of the conditions under which it may be called and the resulting condition of the component and/or its environment that is promised by the operation.¹

Many designers embrace this principle in a limited way, writing preconditions about non-null parameters and state of a data structure (as in the queue example above) and postconditions involving rewriting one-line methods as logical clauses or use of the *const* attribute in languages like C++[10].

Some researchers have suggested that DBC can be applied completely by building a parallel model of a component's behavior [8]. In general this could prove to be problematic from a practical standpoint in that the model might be just as prone to design errors as the actual component.² We will however be applying this idea in a limited fashion by suggesting an external state-based model to enforce not the pre- and post-conditions involving state including those that are too often ignored: those enforcing the sequencing of operation invocations.

Of course, DBC techniques can be applied to the analysis of the problem space as well. It is a fairly common practice to include preconditions and postconditions in use cases [2].

1.3 Aspects

The meaning of the term *aspect* in this paper is fairly traditional. An aspect is an easily specified concern in the problem space whose cohesion is maintained in the solution space of design and implementation, even if traditional design and programming paradigms do not provide a way for this to happen.

In the parts of the paper where programming issues are discussed, we use the term in the way pioneered by AspectJ [4] wherein *advice* is woven in to the core software at *join points*.

2. SEQUENCE CONTROL ASPECTS

State-based models have often been mentioned as a strong candidate for application in aspect-oriented software. One example is the State pattern [1] where the state model, as an aspect, is used to modify the behavior of a component during system execution. Another example is to build a state chart from a pre-existing aspect-oriented design to aid in software verification [11].

2.1 Sequencing in Analysis

As stated above, sequencing information has long been included in requirements analysis work products. It usually appears through preconditions attached to individual use cases. A common example is privileged access to a system and logging in. The latter must be performed successfully before the former will be allowed.

In more complex sequence requirements such as explicitly acknowledging a log-off use case or describing several levels of

¹ Traditionally DBC includes the notion of component *invariants*, but these are simply postconditions that can be applied to all the operations of the component.

² This is of course the same argument some make against formal methods for program correctness.

access to a system, the state chart approach could be employed. This was made very popular in the Object Life Cycles work [9] where the entire behavior of a system was expressed in terms of a state machine.

Ideally the analysis model, like the architecture, has multiple perspectives [5]. There is a use case perspective, from which scenarios and then the collaboration perspective are created, and there is the static class perspective. State models are often employed to aid in a dynamic perspective such as collaboration. What I suggest here is that there be a state model that can possibly be the engineering formalism of the user's instruction manual. From the user's point of view, there are two types of information of prime importance. The first is the internal structural model of the system (often neglected in user literature). The second is the step-wise how-to for a specific task. This information can be seen as an analysis phase aspect that advises (literally!) the user on the proper sequences of operation invocations on objects in the model to accomplish tasks.

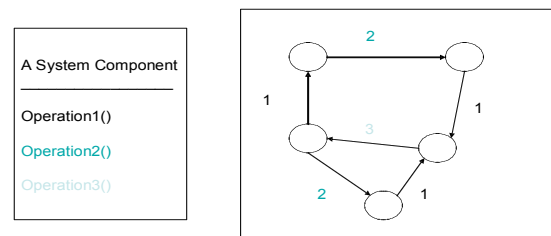


Figure 1. A State Machine as a Sequence-Controlling Aspect
Operations trigger transitions, or are blocked because the transition is not specified.

2.2 Sequencing in Design

Sequencing can be done much the same way in design. But if a state-based approach is not employed, an object-oriented design can become overly cluttered with operations preconditioned on history described by conglomerations of state variables set by operations. If care is taken to express operation sequencing rules in a separately designed state model aspect, then by the nature of aspect-oriented design the core objects will lose their confusing clutter.

The reader may recall from the introduction that we are mostly concerned with sequence controls that are not inherently present in the core design of the functioning system. These controls actually only exist to explain proper use of the design to the implementers and to the designers themselves.

As software implementation is in many senses simply the most detailed form of software design, the same issues that affect design apply in code, with one wrinkle, from DBC. If we want to treat the sequence controls as sanity checks, like assertions, in the executing system, then aspect-oriented language tools that support code weaving would be highly desirable. The sequence controls could then be written as an attachable aspect that could be detached once testing has raised the confidence in the correctness

of the client software to an acceptable level. (For a counterexample, see the example above about movie projectors.)

Another approach would be the stronger linguistic one taken by languages such as Eiffel [7]. In these languages pre- and post-conditions are not just in the documentation but in the code. The compiler can generate assertions for those conditions so that they are actually checked at run time if desired. Pioneers of DBC have stated their opposition to use of woven advice to serve as pre- and post-condition checkers because the approach visually separates an operation from its contract. It does not appear that the argument would be as strong for sequence-enforcing preconditions. The preconditions would either have to invent, change, and check flags and counters that have no use in the core model of the component, or make calls into an external state machine to authorize the current operation invocation. The aspect-oriented approach simply flips the second option around and in so doing completely isolates operation sequencing as a separate concern in the design.

3. IMPLEMENTATION OF SEQUENCING CONTRACTS IN ASPECTJ

In AspectJ, the aspect would be in charge of the state chart. Each operation in the class to which it is attached would get **before** advice. That advice would inform the actual state machine of the operation being performed. If this step is successful, the aspect would update its current state. If not, a `RuntimeException` (for example `AssertionException`) could be raised to suspend the program and allow for debugging.

An enhancement not shown in the example below would be useful for the case where an operation could fail in its execution, and that failure is interpreted as preventing the state change. **around**

```
public interface Player< Move, Outcome > {
    /**
     * Enable user interface to input a move.
     * That is, the view may only prompt its
     * player only if allow() is called
     * first.
     */
    void allow ();

    /**
     * Retrieve the move.
     */
    Move getMove ();

    /**
     * Inform this player about the other
     * player's move. The view should not
     * reveal this move until the player's
     * own move has been collected.
     */
    void setOthersMove (Move move);

    /**
     * The view is told it's now OK to
     * show other players' moves.
     */
    void present ();

    /**
     * Inform about the outcome of a round
     * of play.
     */
    void outcome (Outcome outcome);
}
```

Figure 2. The Player interface

advice could then be used to (a) validate the operation before it starts and (b) effect the state change after the operation has succeeded.

3.1 Sample Application

As a more complex example than the trivial ones listed in the introduction, consider a framework supporting turn-taking games. An interface called `Player` provides the interface to an individual player's View component. The `Referee` class creates two objects that implement `Player`. It must call their methods in a proper order to allow the game to work. For example, `getMove()` may only be called after `allow()` is called and `present()` must be called after `setOthersMove()`. The details of this design are presented in another paper [3]. Figure 2 lays out the `Player` interface.

Depending on if and when a particular game allows one player to see the other player's move, the algorithm in the `Referee` class could vary. Figure 3 shows one possibility.

An aspect is then set up to realize the state machine. In our solution, the state machine is a regular class whose instances are introduced into `Player` classes and whose methods are called by advice within the aspect proper. An example of one of the advice code bodies can be found in Figure 4. When a particular operation is called, a token (enumeration literal) representing that operation is passed to the state machine `fsa` as a candidate stimulus for a transition. If the stimulus is acceptable, `fsa` advances to its new state and returns true. If not, `fsa` returns false.

We had already implemented several games in this framework before the aspect was added. When the aspect was added, an error was discovered in one of the `Referee` algorithms we were using.

```
player[first].allow();
a = player[first].getMove();
player[next].setOthersMove(a);

player[next].allow();
b = player[next].getMove();
player[first].setOthersMove(b);

player[first].present();
player[first].present();

player[first].outcome( ... );
player[next].outcome( ... );
```

Figure 3. One possible Player operation sequence

```
before( Player player ) :
call ( * Player*.getMove() ) &&
target( player ) {

    assert player.fsa.transition(
        PlayerAction.GET_MOVE ) :
        error( action, player.fsa );
}
```

Figure 4. Advice for getMove() operation

3.2 Whether to Automate

The next logical step in the development of this idea would be to take the expression of a sequencing state machine from the design phase and automatically generate the aspect in code. There is actually an argument to be made against automating this step.

Automating the code for the state machine itself may be possible, but I noted that even in the small example shown in section 3.1, the machine could be greatly simplified by recognizing that the sequences could be expressed more easily as two parallel machines that only merged in the `outcome()` operation call. If the tools available demanded that the machine be described as a single automaton, the number of states would be higher and therefore require more work by the designer and also result in a less comprehensible design.

Many suggestions for DBC realization in Java have included the use of annotations that have been available since the release of 1.5. This would obviate the need for us to write an aspect, but also put us right back to the position we would be in using `require` clauses in Eiffel.³ I have already made the argument that it is, at least in some cases, preferable to cleanly separate the sequencing concern from the component.

4. CONCLUSION

This paper represents one attempt to apply aspects to the principle of design by contract. We revealed an inherent strength of aspects by using them to enforce contracts involved with the proper sequence of invocations on object operations. These contracts cannot be enforced cleanly, from a separation of concerns viewpoint, using other approaches.

Future work will involve readdressing the automation issue by taking a close look at design tools that offer strong support for state-based specification for objects and seeing if they offer sufficient flexibility for our needs.

5. REFERENCES

- [1] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc.
- [2] Hamilton, K., and Miles, R. 2006. *Learning UML 2.0*. O'Reilly.
- [3] Heliotis, J. and Schreiner, A. 2006. A Pattern for Distributing Turn-Based Games. *Killer Examples Workshop, 21st Annual Conference on Object-Oriented Systems, Languages, and Applications* (Portland, OR, USA).
- [4] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. 2001. Getting started with ASPECTJ. *Communications of the ACM* 44, 10 (Oct. 2001), 59-65. DOI= <http://doi.acm.org/10.1145/383845.383858>.
- [5] Krutchen, Phillippe. The 4+1 view model of software architecture. *IEEE Software*, 12(6):42--50, November 1995.
- [6] Meyer, B. 1992. Applying "Design by Contract". *Computer* 25, 10 (Oct. 1992), 40-51. DOI= <http://dx.doi.org/10.1109/2.161279>
- [7] Meyer, B. 1997. *Object-Oriented Software Construction (2nd Ed.)*. Prentice-Hall, Inc.
- [8] Mitchell, R., McKim, J., and Meyer, B. 2002. *Design by Contract, by Example*. Addison Wesley Longman Publishing Co., Inc.
- [9] Shlaer, S., and Mellor, S., 1992. *Object Lifecycles: Modeling the World in States*. Prentice-Hall.
- [10] Stroustrup, Bjarne. 1997. *The C++ Programming Language (3rd ed.)*. Addison Wesley Longman Publishing Co., Inc.
- [11] Xu, D. and Xu, W. 2006. State-based incremental testing of aspect-oriented programs. In *Proceedings of the 5th international Conference on Aspect-Oriented Software Development* (Bonn, Germany, March 20 - 24, 2006). AOSD '06. ACM Press, New York, NY, 180-189. DOI= <http://doi.acm.org/10.1145/1119655.1119680>.

³ Method annotations are an inferior solution compared to Eiffel preconditions anyway, since the former cannot be inherited.