

A Pattern Based Approach to Aspect-Orientation for State Based Systems

Mark Mahoney
Carthage College
Kenosha, WI

mmahoney@carthage.edu

Tzilla Elrad
Illinois Institute of Technology
Chicago, IL

elrad@iit.edu

ABSTRACT

This paper demonstrates the benefits of using Aspect-Orientation in state based systems using patterns instead of Aspect-Oriented Programming languages, frameworks, or tools. State based subsystems implemented with the State Pattern interact by binding events from state machines. Binding occurs using the well known Mediator and Abstract Factory Patterns.

Categories and Subject Descriptors

D.2.2 [Design Tools and Techniques]: *Object-oriented design methods, State diagrams*

General Terms

Design

Keywords

Aspect-Oriented Software Development, Design Patterns, State Machines.

1. INTRODUCTION

State machines are an excellent way to model the reactive behavior of objects. They can also be used to model the reactive behavior of entire subsystems. A state machine can be used to fully describe how an object or subsystem should behave in response to stimuli. State machines can easily be transformed into executable code using, for example, the State pattern [1].

Many complex systems can be created by composing smaller, self-contained state based subsystems together. One can think of a state machine as the behavioral interface to a reactive subsystem. It is a metaphor for the subsystem. When the cooperating subsystems are also state based a method is required to compose them together. However, a desirable quality is to reduce coupling between the subsystems.

Each state machine can be considered a concern because it describes an entire subsystem. Some concerns are core concerns and others are crosscutting concerns. A crosscutting concern is one that is tangled with two or more core concerns. Some crosscutting concerns are state based.

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

Our approach involves modeling the behavior of a subsystem using a state machine model. Next, the model is translated to executable code using the State Pattern. Lastly, bindings between implementations of the State Pattern are created using the Mediator and Abstract Factory [1] patterns.

The rest of this paper is organized as follows: Section 2 gives some background on the State Pattern. Section 3 describes our approach to using patterns to manage concern interactions. Section 4 discusses the tradeoffs to using the approach. Section 5 discusses related work.

2. BACKGROUND

2.1 State Pattern

The State Pattern [1] is used to create objects that are capable of responding to stimuli in a reactive manner. A typical implementation of this pattern calls for a context object to have access to all concrete state objects while maintaining a reference to the *current* state object. All events are handled by the current concrete state object. The overall state can change by assigning the context's state reference to another concrete state object.

The process of translating an informal state machine model into a set of classes using the State Pattern is straightforward and can be done without any tool support.

3. USING PATTERNS TO SEPARATE CONCERNS

When a system is composed of multiple semi-autonomous state based subsystems an obvious design goal is to decouple the subsystems from each other. Each has the potential to be reused in other applications. Of course, the subsystems must interact with each other to form a functioning system. In the case of a crosscutting concern modeled with a state machine, it may interact with many other state machines. A mechanism must be in place to allow communication between instances of the State Pattern. The key to reusability of each concern is not to tangle the concerns in the fully functioning state machine implementations.

Before any event binding takes place each state machine is isolated from all others and can be validated independently. We are proposing a method of binding events from different implementations of the State Pattern without directly referring to the bindings in the original state machine code. This will allow each state machine to remain independent and reusable in other applications.

This extends some previous work of ours [4][5] where a similar approach was suggested. In those works a state machine framework of classes was created in order to achieve the same

goals. This pattern based approach allows the same benefits without needing to use the framework. The framework approach mentioned above does allow concurrent executable state machines to be distributed. The pattern approach is designed for simpler systems with no concurrency or distribution requirements.

3.1 Mediator between States

In order to achieve interaction between different instances of the State Pattern we are proposing the use of a mediator object that recognizes when a bound event occurs in one state machine (the source state machine) and introduces a new action or event to another state machine (the target state machine).

A mediator is created that knows about the target state machine's event interface. The mediator needs this information in order to inject events in the target state machine. A derived binding aware concrete state class informs the mediator when the bound event occurs. Figure 1 shows a design where a state machine for a Two-Way Radio is bound to a Wireless Connection state machine. The derived concrete state class (IdleWithBinding) then allows the base class (Idle) to handle the event as specified in the original state machine. Depending on when the base class's event handler method is called one can specify that the target transition occur before or after the source transition. This is similar to an idea first presented in [4] and resembles the before and after advice in Aspect-Oriented Programming (AOP) languages.

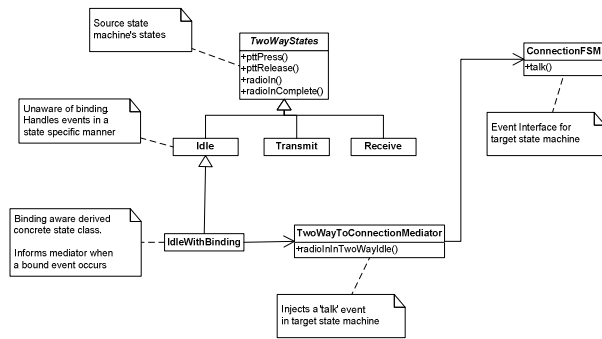


Figure 1. State Pattern with Binding

A further benefit is that one can add actions to an existing state machine transition using the mediator. This also resembles basic AOP constructs of adding functionality at well defined join points. In this case the join points are transitions in a state machine.

3.2 Abstract Factory for State Creation

The key to creating *independent* binding of state machine code is the creation of an Abstract Factory. The Abstract Factory pattern describes an interface for creating objects but allows subclasses to specify exactly how those objects are created. When it is time for a weaving developer to combine state machines the Abstract Factory will create binding aware concrete state objects in each context object along with the Mediator. This allows one to pull together disparate state machines together and specify bindings outside of the core state machine implementation.

4. DISCUSSION

The binding of events in different implementations of the State Pattern allows a development team to combine whole subsystems and features together into a functioning system. The development team can maintain a library of executable state machines and mix

and match those implementations to develop complex systems. Since it is so easy to bind different state machines together testing the interactions can be done very early in the development lifecycle.

The drawbacks of this approach are the introductions of the mediator class, factory class, specialized concrete state class, and glue code to bind them all together. For each bound event in a source state machine a single mediator and mediator aware concrete state class need to be created. These are relatively simple classes but they do muddy the waters.

It is not hard to imagine a tool to generate the binding code automatically. There are very simple tools that exist that aid in the development of state based classes using the State Pattern [2]. These tools can be extended to implement the binding between state machines. Afterwards, one might never even look at the State Pattern or binding code. They are details of the model that the development team can abstract out. One only needs to implement the state machine's action interface.

5. RELATED WORK

In [7] Aspect-Oriented and Model Driven Software Development (MDS) are examined to determine what links exist between the two. The author describes several patterns showing how AOP and MDS can be used together. Our approach most closely resembles the Pattern-Based AOP approach because it uses well known patterns to generate infrastructure for the management of concern interaction in state machine models. One difference is that our solution is specific to state based subsystems.

The idea of interacting state machines has also been addressed in Executable UML [6] and SDL [3]. These approaches define state machines for different platform independent domains. The state machines communicate with each other using an explicit signaling mechanism. Executable UML and SDL are heavyweight and require significant tool investment and a process learning curve.

In [8] crosscutting concerns are modeled with Interaction Pattern Specifications and transformed into state machines. The approach differs from ours because theirs generates state machines from scenario models.

6. CONCLUSION

We have discussed an approach to modeling state based subsystems in such a way that they can easily be translated into executable code using the State Pattern. The state machines become executable models that can be tested and validated in isolation. We have proposed a technique for combining instances of the State Pattern together that allows event bindings to take place between state machines. Using the Mediator and Abstract Factory Patterns in a straightforward way with the State Pattern one can achieve the management of state based concerns.

7. REFERENCES

- [1] Gamma, Helm, Johnson, Vlissides; Design Patterns, Elements of Reusable Software Design, Addison-Wesley 1995
- [2] <http://sourceforge.net/projects/smc>
- [3] ITU, Z. 100: Specification and Description Language (SDL), International Telecommunication Union, 2000.

- [4] Mahoney, M., Bader, A., Aldawud, O., Elrad, T., Using Aspects to Abstract and Modularize Statecharts. The 5th Aspect-Oriented Modeling Workshop in Conjunction with UML 2004.
- [5] Mahoney, M., Elrad, T. Modeling Platform Specific Attributes of a System as Crosscutting Concerns using Aspect-Oriented Statecharts and Virtual Finite State Machines. The 6th International Workshop on Aspect-Oriented Modeling as part of AOSD'05 (Chicago, USA, March 2005)
- [6] Mellor, S.J., Balcer, M.J. Executable UML: A Foundation for Model Driven Architecture, Addison-Wesley, 2002.
- [7] Volter, M. Patterns for Handling Cross-Cutting Concerns in Model-Driven Software Development, In 10th European Conference on Pattern Languages of Programs (EuroPlop 2005), Irsee, Germany, July 2005.
- [8] J. Whittle and J. Araújo, Scenario Modeling with Aspects, IEE Proceedings Software, 151(4), pp. 157-172, 2004