Patterns for Monitoring Scenarios to Handle State Based Crosscutting Concerns

Mark Mahoney
Carthage College
Kenosha, WI
mmahoney@carthage.edu

Tzilla Elrad
Illinois Institute of Technology
Chicago, IL
elrad@iit.edu

ABSTRACT
This paper describes two patterns, Scenario Monitor and Bind Completed Scenario to Event. The first allows scenarios to be monitored. The second uses scenario monitoring to address state based crosscutting concerns in traditional data transformational systems. Crosscutting concerns are tangled with core application concerns and scattered throughout a system. Core concerns are monitored for scenarios that represent events of interest to a crosscutting concern. When the monitored scenarios complete an event is injected into a crosscutting state machine that may react by introducing additional behavior. These patterns permit the monitoring and subsequent behavioral reaction in a minimally invasive way with loose coupling. No special tools or languages are required. An example using the approach is presented.

Categories and Subject Descriptors
D.2.11 [**Software Architectures**]: Data abstraction, Domain-specific architectures, Information hiding Languages (e.g., description, interconnection, definition), Patterns (e.g., client/server, pipeline, blackboard).

General Terms
Design.

Keywords
Design Patterns, crosscutting concerns, state based systems.

1. INTRODUCTION
Program monitoring is a technique that allows one to observe sequences of events from program execution and react accordingly. It has been actively used for profiling, testing, and debugging software [16]. It can also be used for handling state based crosscutting concerns. This paper documents a pattern for program monitoring, **Scenario Monitor**, that allows one to add behavior upon the completion of a scenario. An important requirement this pattern satisfies is non-invasiveness. It allows one to add or remove monitors without radically changing the core application's code or using specialized tools or programming languages. Next, we describe another pattern, **Bind Completed Scenario to Event**, that makes use of Scenario Monitor to react to the completion of scenarios with additional crosscutting behavior.

A crosscutting concern is an aspect of a system that influences many other core concerns. Crosscutting concerns cannot be easily modularized using traditional decomposition techniques. Fault tolerance, for example, is a crosscutting concern that affects many parts of a system. However, fault tolerance code is typically scattered throughout the system and tangled with the core concerns interfering with their logical flow. The field of Aspect-Oriented Software Development (AOSD) [1] addresses crosscutting concerns by separating them from core concerns at one level of abstraction and providing a means to weave them back together at a lower level of abstraction. The woven product is one step closer to an executable system. For humans analyzing the system, the separation of concerns allows one to reason about core and crosscutting concerns independently while understanding how they affect each other.

Many core concerns exist that have no state based behavior. They do not require any knowledge of the past in order to satisfy a requirement. Occasionally, a crosscutting concern requires knowledge of a core concern's state in order to function properly. Naturally, the state based crosscutting concerns should be built using state machines and the non-state based core concerns should not. Interaction between the two types of concerns is difficult because behaviorally each is so different. We propose mapping the dynamic aspects of a non-state based system (scenarios) to the dynamic aspects of a state based system (events). This mapping resolves the differences in the dynamic behavior of the concerns.

For example, consider a banking system with accounts that are accessible from a bank teller, ATM, or online. From a security standpoint, repeated transfers in a single day through an ATM or online rather than through a bank teller requires that the transaction be logged as suspicious activity. The core transfer behavior is not state based and therefore should not be implemented with state machines. The crosscutting security logging concern, however, does require knowledge of the core's state. In particular, it needs to know how many transfers have been attempted in a day and by what means the transfers took place. A state machine can model this behavior and keep the transfer and logging concerns separate. The purpose of the state machine is to model the relevant state of the core. To promote reuse and to reduce coupling the state machine should not have direct access to the core concerns. The main problem this paper addresses is how to weave the two separated concerns together while maintaining loose coupling. Our approach uses a program monitor to observe when certain scenarios occur, like a transfer

PLoP'08, October 18-20, Nashville, TN, USA.

from an ATM, and map those completed scenarios to events in the logging state machine. The crosscutting state machine may then react by introducing new behavior, like logging a transfer.

There are tools [16][6], specialized programming languages [7][12][17], and frameworks [18] that can be used to accomplish similar results. The tools and frameworks are typically heavyweight and proprietary and may not be appropriate for production software. The specialized programming languages include new syntax and require a significant learning curve to become proficient. We prefer an approach using standard object oriented techniques with patterns so that all OO developers can use it regardless of programming language or environment.

The patterns discussed in this paper map scenarios to events in such a way that neither type of concern is directly coupled with the monitoring code. This makes all the concerns reusable in different contexts. Non-state based core concerns can be used with or without the state based crosscutting concerns, and state based crosscutting concerns can be used with different core concerns. In order to accomplish monitoring, decorator objects are created that wrap the monitored objects and track when certain messages are passed. The monitor is responsible for translating completed scenarios into events that are fed into a crosscutting concern state machine.

The rest of this paper is organized as follows: section two discusses the **Scenario Monitor** and **Bind Completed Scenario to Event** patterns and how they can be used to track the state of a set of core concerns. Section three provides an example of using the pattern to handle a crosscutting concern.

2. SCENARIO MONITORING FOR HANDLING EVENTS

In our approach, a crosscutting concern developer models the relevant state of the core concerns with a state machine. Figure 1 shows the state machine for the security logging concern from the introduction. The states represent periods of time in the core when the crosscutting logging concern reacts to events. Logging behavior is added on certain transitions in the state machine.

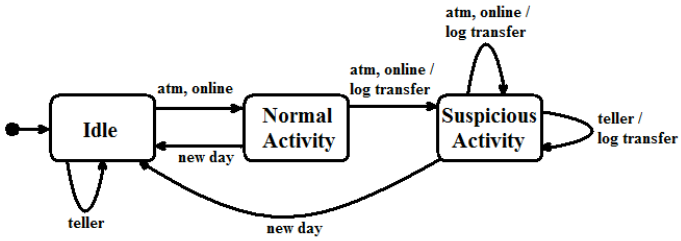


Figure 1. Security logging state machine.

The state machine in figure 1 is abstract in that the events are not directly bound to a core concern's implementation. Transferring funds using an ATM, for example, is a complex core process that involves many objects interacting and can't be mapped to a single method call. The fact that the state machine is abstract makes the crosscutting concerns reusable in many different contexts. We limit the mapping of events to the completion of scenarios. A scenario is an ordered set of messages (method calls) sent and received from objects in the system. Traditionally, scenarios are modeled with sequence diagrams [14].

Scenario monitoring is the crucial element in this approach. One of our goals is to allow scenario monitoring to occur without the use of special tools, programming languages or complex logics. Our target audience is the average OO developer working on systems where introducing new tools or complex logics would be impossible due to language incompatibilities, legacy environments, or insufficient formal background in Computer Science. We have developed a straightforward pattern that allows relatively complex scenario monitoring to occur in a non-invasive manner.

Scenario monitoring is accomplished by breaking the problem up into two main parts:

1. Decorators and Observers [4] that notify the monitor when specified object interactions take place
2. Simple state machines used to track the progress of each active scenario

The state machines listed in item two are used to track the completeness of a scenario and are different from the crosscutting concern state machines that introduce additional behavior (logging). Scenario monitoring state machines ensure that object interactions happen in the specified order. The Bind Completed Scenario to Event pattern describes state machines responsible for adding behavior to a set of core concerns.

2.1 Scenario Monitor Pattern

Name

Scenario Monitor

Context

A complex Object-Oriented system is made up of many different objects sending many different messages to each other. Imagine a system that must react in a particular way to the occurrence of an ordered sequence of messages within the system. This set of messages is called a scenario. In a scenario each message has a sender and a receiver. If a single message is missing or received out of order then the scenario is not complete and no reaction occurs.

In the simplest case a scenario is made up of a collection of sequential messages. Figure 2 shows several different sequence diagrams that contain sequential messages. Each sending point and receiving point is highlighted with a small circle. A series of messages are sequential if every sequential pair shares either a sending or receiving point. A similar idea was presented in David Harel's Play-Engine work [6].

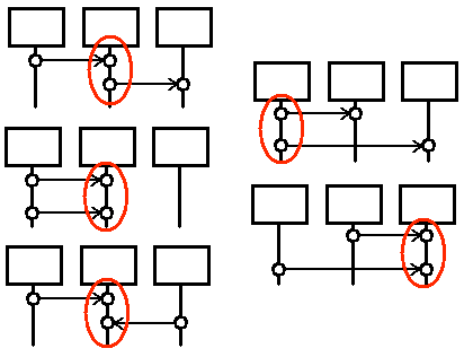


Figure 2. Sequential Messages

A scenario might also be made up of disjoint sequences of ordered messages, see figure 3. In this case the two disjoint sets of messages can occur in any order as long as neither violates the sequential ordering required in each. The scenario can be considered satisfied with several different orderings of the messages. For example, the order ‘m1’, ‘m2’, ‘m3’, ‘m4’ satisfies the scenario as does the order ‘m3’, ‘m1’, ‘m4’, ‘m2’ because all the messages are handled and no violation of sequential ordering occurs.

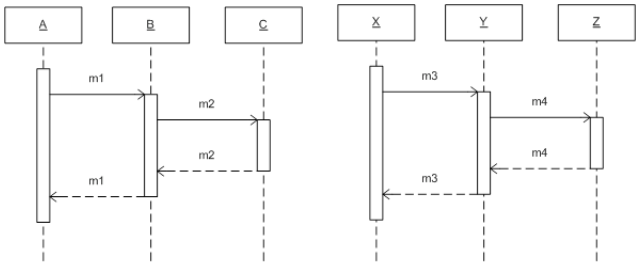


Figure 3. Scenario With Disjoint Sequences of Messages

To complicate matters even further a scenario might be made up of *mostly* disjoint messages that rendezvous at one or more points, see figure 4. In this case, there are two sets of relatively independent messages. For the most part, messages from either set can occur in any order (as long as there are no violations of order in each collection). However, at the rendezvous point where C sends a message to M and X sends a message to M the two independent collections of messages join together and there is an order that must be adhered to.

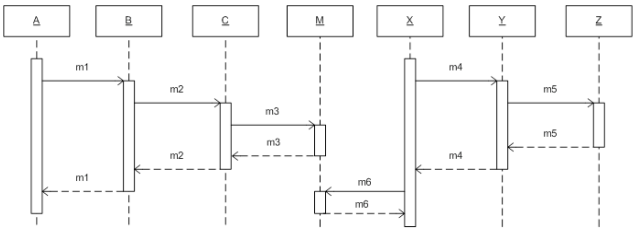


Figure 4. Scenario With Mostly Disjoint Sequences of Messages

The most obvious use of scenario monitoring is for testing and debugging. Scenario monitoring can provide a trace of an executing system that can be used to verify that use cases are being completed. Scenario monitoring can also be used to debug a system where conventional debugging tools are not available.

Perhaps a less obvious use of scenario monitoring is to deal with state based crosscutting concerns. The following pattern emphasizes the use of scenario monitoring in this context but this pattern can be applied in other contexts as well.

Problem

The difficulty is that some sort of scenario monitoring must occur. There are tools and languages [6][2][12] that might aid in this monitoring but they require a significant investment in acquiring and learning a proprietary technology. In addition, they cannot be used with existing and legacy systems without retrofitting the systems to these new tools and technologies. Ideally, scenario monitoring should occur using standard Object-Oriented languages and techniques such that the objects being monitored are loosely coupled to the monitor. Then, monitoring can be added and removed easily.

Forces

The difficulty in introducing scenario monitoring is to do so in a way that is minimally invasive to the objects being monitored. It would be easy to go into every class where monitoring is required and add code to those methods. However, this scatters the monitoring code and tangles the code from two competing concerns. Further, if the monitoring strategy was to change or monitoring was no longer required one would have to update or remove all the monitoring code.

Monitoring a scenario is akin to tracing the state of a scenario. Therefore a state machine is an ideal way to monitor a scenario. In fact, this is not a new idea and there have been state based scenario monitoring algorithms introduced in the past [5][8][9][15]. In the simplest case when all messages are sequential a single state machine is an easy solution. The states in the state machine each represent the processing of a message. The states are connected by transitions where the next expected message is an event that will move the scenario forward. For each state, all messages beyond the next expected message are events that will take the scenario to an invalid state due to a violation of the order imposed by the scenario. A monitor should no longer track invalid scenarios.

One difficulty, however, is resolving how disjoint scenarios are monitored. When a scenario is broken up into disjoint sets of sequential messages a single state machine can grow to be incredibly complex to handle all the different combinations of received messages.

In the course of an executing system the same scenario might occur multiple times. It is possible that several instances of a scenario might be advancing with each one at a different state. For example, in figure 3 if the monitor recognizes that A sends the message ‘m1’ to B, then scenario monitoring should begin. If before that scenario is completed or invalidated the same message is sent again, this can be considered another instance of the same scenario. The scenario monitor must be able to manage multiple scenario instances and react appropriately when each instance successfully completes.

Solution

In the pattern a scenario is made up of Scenarios, Scenario Instances, and Scenario Fragments, see figure 5. A Scenario Fragment is a simple state machine. The state machine represents a collection of sequential messages sent synchronously where there is no possibility of delayed message reception. The first message in a Scenario Fragment has no required messages that must be handled before it. Every other message in a Scenario Fragment must come in a strict order specified when creating it. Figure 6 shows the state machine for one of the Scenario Fragments from figure 4. One could use the State Pattern [4] to implement Scenario Fragments but since each one models such an uninteresting state machine a state table based approach is clearer [13].

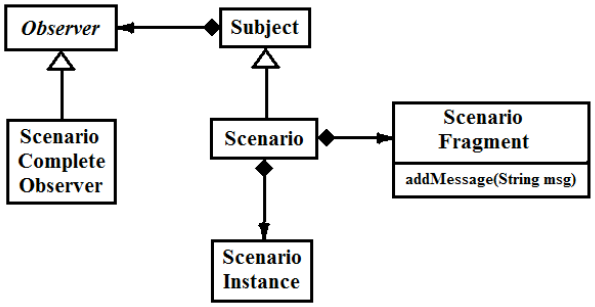


Figure 5.

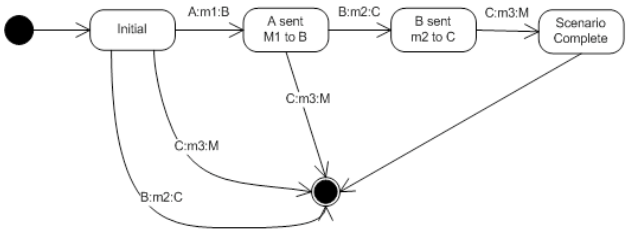


Figure 6.

A Scenario Fragment is defined by the messages and the order that the corresponding method calls have completed. Therefore, when defining a scenario one creates a sequence diagram and maps out both the method calls and the method returns. The order of messages in the fragment is taken from the order of method call returns. This is required to track only the messages that have been sent, received, and fully processed.

In figure 7 a simple scenario is laid out. In this case the order of the calls is ‘m1’, ‘m2’, ‘m3’, and ‘m4’. However, the order that the methods complete their processing is ‘m3’, ‘m2’, ‘m4’, and ‘m1’. Only after those methods calls have been sent and received in that order should the scenario be considered satisfied. The code that reacts to the scenario’s completion will find the monitored objects in a state where all the messages have been fully processed.

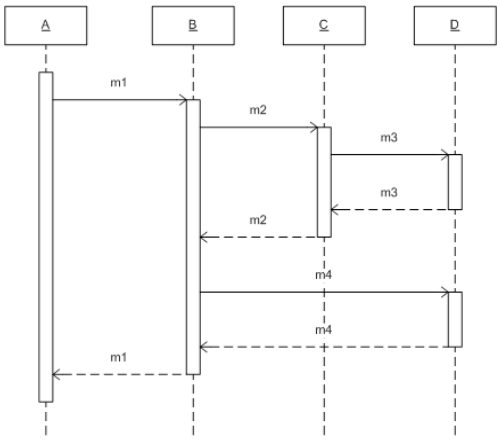


Figure 7.

A Scenario is made up of a collection of Scenario Fragments. It is possible that in a single Scenario there will be two or more fragments that have no objects in common. In this case the ordering between the two fragments is not important (although the ordering within each fragment must be maintained). For example, in figure 4 three separate Scenario Fragments are specified. This is highlighted in figure 8, which shows the same sequence diagram three times with different fragments highlighted. Each individual fragment has strict sequential ordering requirements. Notice, however, that there is some overlap between the messages in the different fragments. This overlap provides a way for disparate fragments to rendezvous. As long as all messages are received and no fragment order is violated the scenario is considered complete.

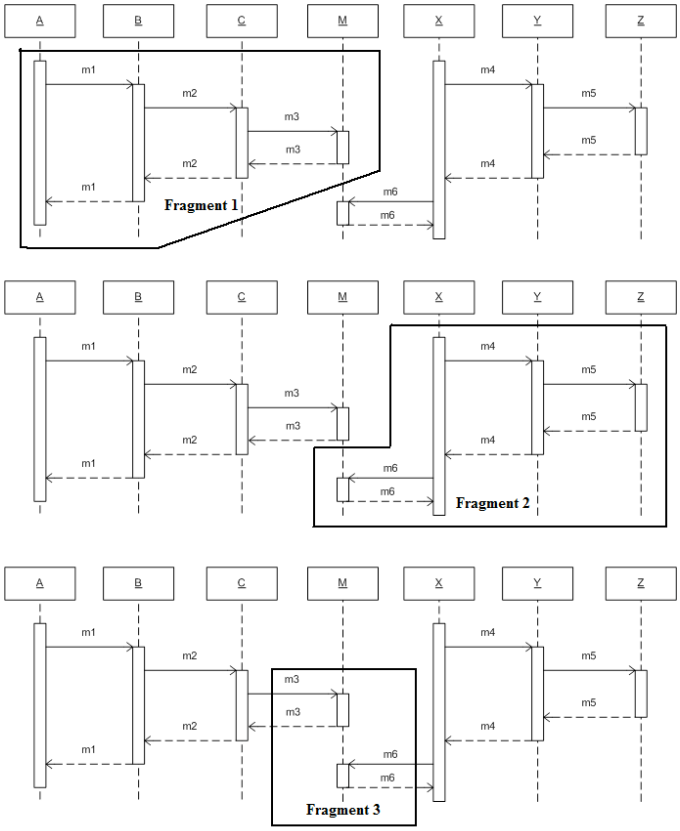


Figure 8.

A Scenario Instance tracks the state of a Scenario. It remembers in each of the Scenario Fragments what the last message handled was and notifies the Scenario when all the fragments are completed. Scenario Instances are needed because the same Scenario may be active multiple times where each instance is in a different state.

Each Scenario object is notified when a monitored message is handled. The Scenario object responds by either creating a new Scenario Instance (if the event represents the first message in any Scenario Fragment) or feeds every Scenario Instance the event. Each Scenario Instance will check with all the Scenario Fragments for the Scenario and either ignore the event, move forward, or be in violation of the Scenario and be removed from monitoring. To handle the completion of Scenarios the Observer

Pattern [4] is used. The Scenario acts as the subject and one or more observers are notified when a scenario successfully completes. The observers can then react to the completion of the scenario.

So far we have not discussed in detail how Scenario objects are notified when a particular message is passed. To achieve this in a non-invasive way we use the Decorator [4] and Observer Patterns [4], see figure 9. Each object that receives a message in a monitored scenario will become wrapped inside a Decorator. Each sender in a scenario has its reference to the monitored object changed to a Decorator. The Decorator will store a reference to a monitored object and delegate the object's responsibility to it. The Monitor Decorator class will store the name of the sender and receiver of the message to be used to create an event that specifies the names of the sender, the message, and the receiver. This combination of patterns is not totally original, in fact some AOP compilers and runtimes may use it. We are highlighting that many features available in AOP languages can also be accomplished using good OO design.

The Decorator will also be a subject from the Observer Pattern. After each delegation the Decorator will inform all observers that the delegate has handled a particular message. The observers are Scenario objects that have registered interest in a particular message because it is part of one of the Scenario's fragments.

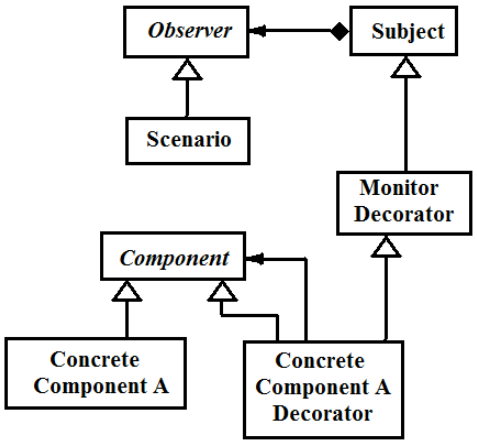


Figure 9.

Several Decorator instances may exist for a single monitored object because a Scenario needs to know not only what message was sent but also who sent it. In figure 10 there are two scenarios. In the first an object of type A is sending the message m1 to an object of type C. In the next scenario an object of type B is sending the message m1 to the same object. Even though the receiver is receiving the same message the senders are of different types, therefore these are different scenarios. Both sending objects require distinct Decorators so that when the Decorator notifies the observers the observers know who the sender is. This information is stored in each MonitorDecorator and is passed to observers in the notifyAllObservers() method, see figure 11. These Decorators can be used by many different Scenario objects, all that is required is that each Scenario object register with the correct Decorator to be notified when a monitored message occurs. A hierarchy of decorators can be created if the same objects participate in many scenarios.

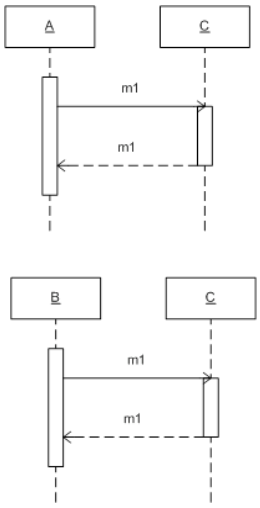


Figure 10.

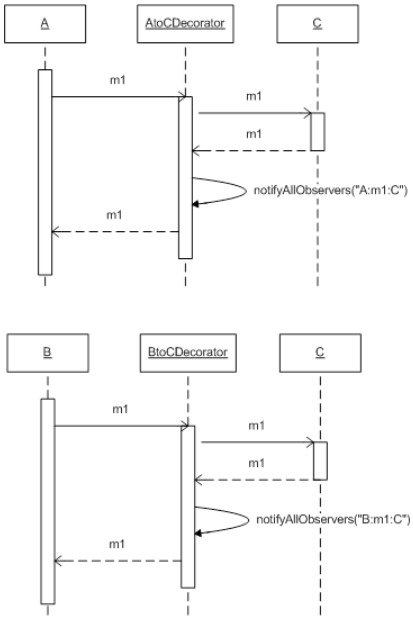


Figure 11.

Forces Resolved

The Decorator Pattern allows one to add behavior to an object dynamically. This allows monitored objects to remain unchanged and unaware that they are actively being monitored. This solution does require that someone do the wrapping of monitored objects into Decorators, but that code can be localized outside of any monitored class. Also, because we are using the Observer Pattern monitoring can be turned off as easily as it is turned on.

Scenario Fragments represent simple state machines, one for each collection of sequential messages. Scenarios with disjoint collections of messages are handled by including a Scenario Fragment for each one. The overlapping of messages in the fragments guarantee that synchronization occurs between

Scenario Fragments. As long as no individual fragment is violated, the scenario is not violated and a single complex state machine is avoided.

The Scenario object creates and tracks Scenario Instance objects so that multiple occurrences of a scenario can be active and monitored at once. When a scenario is violated the Scenario object removes the Scenario Instance so that it is no longer monitored. Similarly, when a scenario completes the Scenario Instance is removed.

This approach is somewhat complex and does require additional code to be created to set up monitoring. There is overhead when monitoring scenarios. Each scenario requires some memory to store the scenario specification along with the state of each scenario instance. Each monitored message involves notifying a monitor when the system could be executing the behavior associated with the message instead. This overhead might become overwhelming in a massively monitored system. However, no code inside the monitored objects needs to change and monitoring can be turned on and off relatively easily.

Known Uses

The Spring Framework [18] uses a similar collection of decorators to permit AOP

2.2 Bind Completed Scenario to Event Pattern

Name

Bind Completed Scenario to Event

Context

During the development of a traditional data transformational system one or more state based requirements are discovered that need to monitor the state of the core concerns. In other words, some state based behavior is recognized but the *state* in question belongs to the part of the system that was built without using state based decomposition techniques. A state machine must be built so that additional behavior can occur during transitions between states but it should not interfere with the development of the non-state based concerns. In particular the developers of the non-state based core should not have to instrument their code in order for the state based behavior to be added. Ideally, the state based behavior will be reusable in different contexts with different sets of core code. Further, it is not possible to add new tool support or change programming language because either the system is already in development or the cost associated with such tools.

Problem

How do we resolve two radically different types of interacting concerns using standard OO deign techniques while maintaining reusability and loose coupling?

Forces

Loose coupling is important in this pattern because each concern should be able to exist in isolation. It is likely that under certain situations the core concerns will exist without the crosscutting concerns and vice versa. For example, handling a crosscutting concern might be an additional feature that is provided at a premium. The standard version of the software might exist without such a feature. Similarly, a crosscutting concern that adds behavior might be useful in many different contexts. A state based security concern, for example, might be applicable in many different non-state based systems as long as a mapping can occur from the core to the crosscutting concerns.

Solution

The solution involves using the Scenario Monitor pattern (or some other scenario monitoring technique) to specify how the dynamic parts of the two concern types interact. In a data transformational system the dynamics are in the messages sent between objects. In a state based system the transitions between states represent the dynamic behavior. A mapping between these two is required to add additional behavior in a loosely coupled way.

This pattern accomplishes this by mapping completed scenarios in the core concerns to events that are injected into a crosscutting concern state machine. The state machine can then react to the event and possibly add behavior on a transition (if one takes place).

A mechanism is required for the scenario monitor to inject events into a state machine. If one uses the Scenario Monitor pattern this behavior exists because the Scenario is the subject in the Observer Pattern [4] and it is responsible for notifying all observers waiting for the completion of a scenario. The observers are objects that interact with state machines that react to the notification by injecting an event into the state machine. It is these interacting objects that bind the loosely coupled concerns together. It would be the responsibility of a *weaving developer* to bring together the core and crosscutting concerns and specify how the completion of a scenario is mapped to an event in a state machine.

Forces Resolved

This approach is superior to AOSD approaches because it provides all of the loose coupling that is present with specialized programming languages without having to adopt new tools or language extensions. One can use this pattern in any OO environment whether being created from scratch or several years (and releases) into the development process.

In most AOSD technologies the weaving is done automatically with language extensions and a tool. In this pattern the weaving is done manually rather than with a tool. In order to avoid new tools it is up to a human to specify which completed scenario maps to which event in a state machine. In addition, there is overhead involved with the use of the Observer Pattern.

3. EXAMPLE USING THE PATTERNS

The following describes a set of requirements that we use to elucidate our approach (for a more complete description see [11]). The system is for a financial advisor that generates and sells reports to his customers about potential companies to invest in. The financial advisor gets some of his financial data from a much larger financial services organization referred to as the Investment Warehouse.

3.1 Report Generating System

Requirement R1: Financial Advisor Attempts to Sell Report

A financial advisor requests an investment report from the Report Generating System to sell to his customer. If the Report Generating System does not have the requested report it will ask an independent Investment Warehouse for information in order to generate the report. The Report Generating System will then generate a report and send a summary to the financial advisor. The financial advisor shows the customer the summary and tries

to sell it to her. If the customer wishes she may purchase the full report. If that happens the report will be stored by the system and presented to the customer. When a customer requests a report that already exists, the Report Generating System will pull the report from storage and display a summary to the customer. If the customer chooses to purchase the report it will be presented to her.

In the early design phase the Sequence Diagrams in figures 12 and 13 are created.

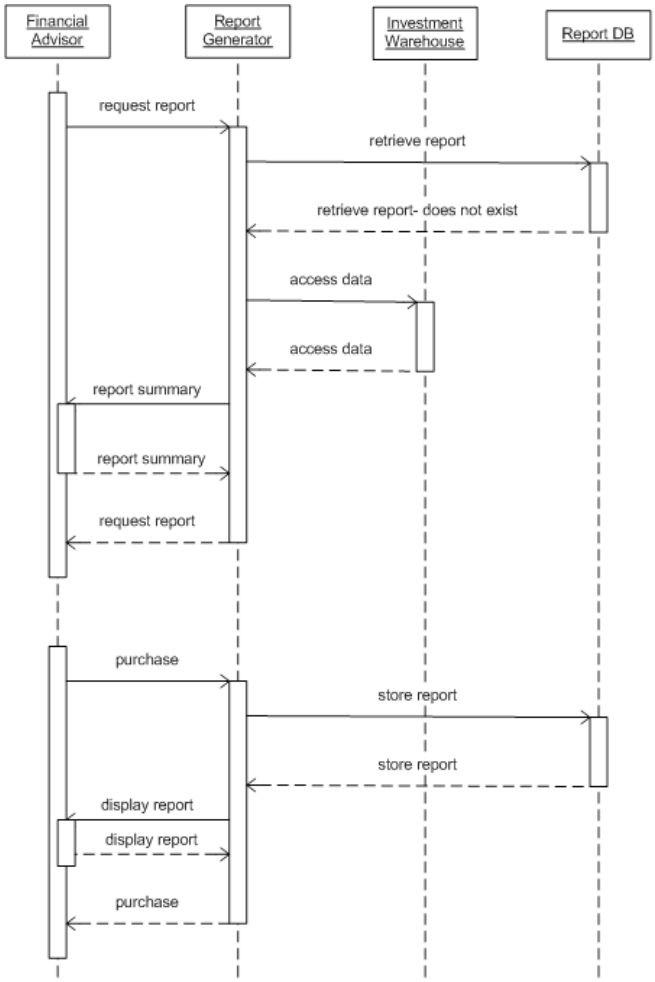


Figure 12. Financial Advisor Attempts to Sell Report

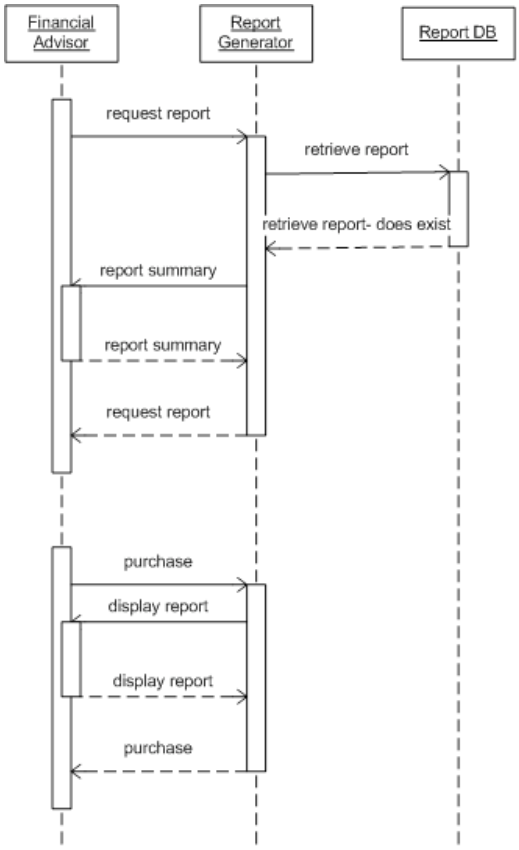


Figure 13. Financial Advisor Attempts to Sell Report (Report Already Exists).

3.2 Crosscutting Billing Concern

Imagine the financial advisor has the option of becoming a franchisee of the Investment Warehouse or a pay-as-you-go customer. If the financial advisor becomes a franchisee, he is charged a relatively high flat yearly rate for access to financial services information. A pay-as-you-go customer is charged per access to the Investment Warehouse. The financial advisor has negotiated an additional term in the contract. The Investment Warehouse will only charge the financial advisor once when he accesses data the first time a report is sold. If the financial advisor does not sell the report, or the report is already in his database of sold reports, he is not charged for the data access.

To keep costs down the financial advisor would initially like to be a pay-as-you-go customer but would like the flexibility to switch to a franchisee. If he switches he would like to keep his existing Report Generating System in place. The main difference is that a franchisee does not need to handle billing.

The state machine in figure 14 describes how normal franchisees are charged for access to the Investment Warehouse's data. The requirement is to charge $10 for the first 5000 accesses in a month, $5 for the next 5000 accesses in a month, and charge nothing for more than 10000 accesses in a month.

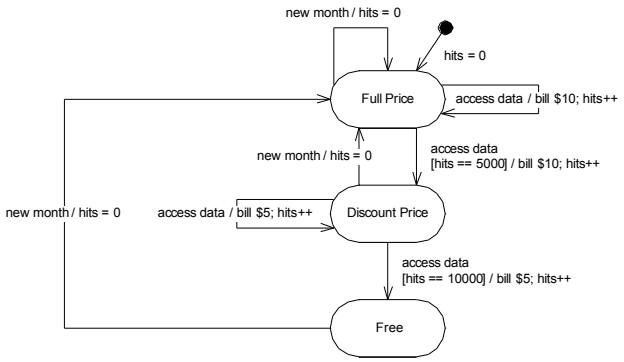


Figure 14. 'Access Data' State Machine

The problem is determining when a billable access has been performed. The only time a billable access occurs is when a new report is actually purchased. One cannot simply bill after accessing the data from the Investment Warehouse because there is no charge unless the customer purchases a report. Further, one cannot simply bill after the customer purchases a report because they may be purchasing a report that already exists. The billing system needs to know the state of the transaction.

The important states to the billing concern are Idle, Pending Purchase New Report, Pending Purchase Existing Report, and New Report Purchased. The important events are when a new report is generated, when an existing report is requested, when a report is purchased, and when a report is abandoned. The state machine in figure 15 describes these states and events.

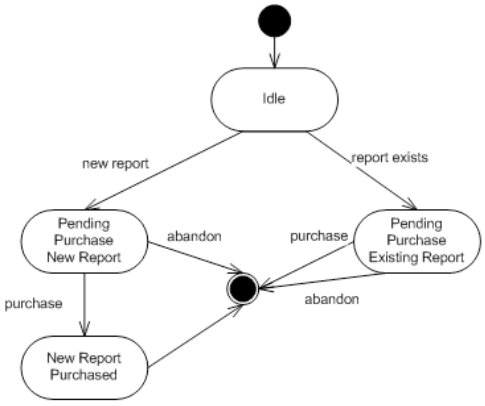
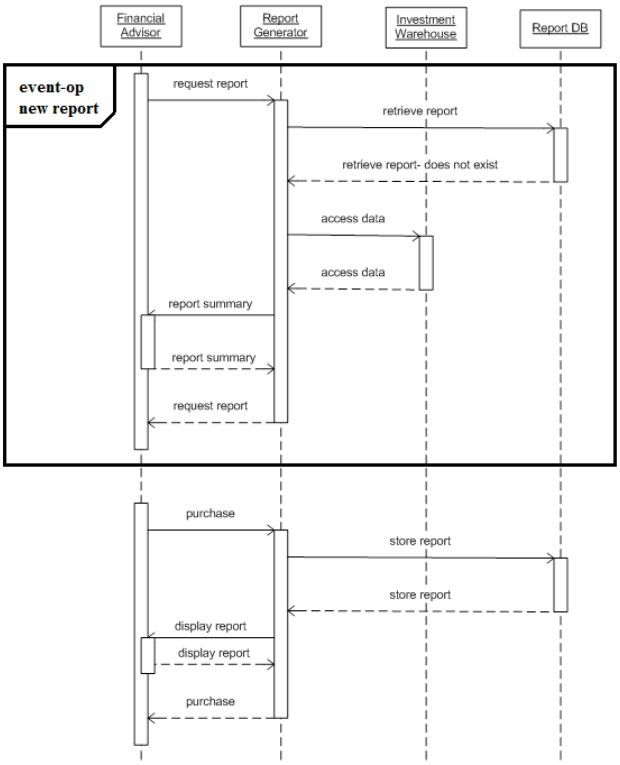


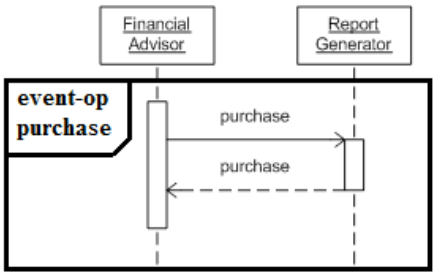
Figure 15. 'Billing' State Machine

Using a variation of sequence diagrams described in [11] one can capture the scenarios of interest, see figure 16. The models specify which scenarios map to events in the state machine using the event-op operator. The semantics of this model are that when the scenario in the Combined Fragment of the sequence diagram completes the event specified in the event-op operator will be injected into the state machine. This decouples the crosscutting concern's required state information (specified in figure 15) from a particular implementation.

Using the Scenario Monitor Pattern from above, an implementation can be created that will treat the completed scenarios as events in the crosscutting concern state machine. Billing can then occur by combining the state machines using the approach in [10]. In that work state machines can be combined and events from one state machine are bound to events in another. In this case the 'purchase' event from the Billing state machine of figure 15 is bound to the 'access data' event from the Access Data state machine from figure 14.



a. New Report Event



b. Purchase Event

c. Report Exists Event

Figure 16. Specification of bindings between Sequence Diagrams and events.

4. CONCLUSION

History sensitive crosscutting concerns are difficult to implement when the history lives in the core implementation. We have provided a way to monitor the core concerns and create events that can be used by a crosscutting concern state machine. This is done in a non-invasive manner without using any proprietary tools or programming languages. We combined simple design patterns to create a new Scenario Monitor Pattern.

The primary benefit of our approach is the loose coupling between core and crosscutting concerns. The specification of binding between the core and crosscutting concerns is at a higher level of abstraction than other approaches. The consequences are that developers can specify state based behaviors required for crosscutting concerns in an abstract way that is reusable in different contexts. Crosscutting concern developers can emphasize the state based nature of concerns without requiring the core concern developers to create a state machine model- they are oblivious.

5. ACKNOWLEDGEMENTS

We would like to thank Marcelo d'Amorim for his insightful comments during the shepherding phase of this year's PLoP. In addition, the members of our shepherding group provided invaluable insight into the strengths and weaknesses of the patterns.

6. REFERENCES

[1] AOSD web page. http://aosd.net

[2] Cottenier, T., van den Berg, A., Elrad, T. The Motorola WEAVR: Model Weaving in a Large Industrial Context. Industry Track paper at AOSD'07

[3] Filman R.E., Friedman, D.P. "Aspect-Oriented Programming is Quantification and Obliviousness", Workshop on Advanced Separation of Concerns, OOPSLA 2000, October 2000, Minneapolis.

[4] Gamma, Helm, Johnson, Vlissides; Design Patterns, Elements of Reusable Software Design, Addison-Wesley 1995

[5] D. Harel, H. Kugler. "Synthesizing State based Object Systems from LSC Specifications", Proceedings of Fifth International Conference on Implementation and Application of Automata (CIAA2000), Lectutre Notes in Computer Science, Springer-Verlag, July 2000.

[6] Harel, D., Marelly, R. Specifying and Executing Behavioral Requirements: The Play In/Play-Out Approach Software and System Modeling (SoSyM) 2 (2003), pp. 82-107.

[7] Kiczales, G. et al., Aspect-Oriented Programming Proc. European Conf. Object-Oriented Programming, Lecture Notes in Computer Science, no. 1241, Springer-Verlag, Berlin, June 1997, pp. 220-242.

[8] K. Koskimies, E. Makinen. "Automatic Synthesis of State Machines from Trace Diagrams". Software-Practice and Experience, vol.24, No. 7, pp. 643-658 (July 1994).

[9] S. Leue, L. Mehrmann, M. Rezai. Synthesizing ROOM Models From Message Sequence Charts Specifications. TR98-06, Department of Electric and Computer Engineering, University of Waterloo, Waterloo, Canada, 1998.

[10] M. Mahoney, T. Elrad. A Pattern Story for Combining Crosscutting Concern State Machines. Pattern Language of Programs Conference. Monticello, Illinois, 2007.

[11] M. Mahoney, T. Elrad. Using Scenario Monitoring to Address State Based Crosscutting Concerns. Software Engineering Knowledge Engineering Conference. Redwood California, 2008.

[12] Maoz, S. Harel, D. From multi-modal scenarios to code: compiling LSCs into aspectJ Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering Portland, Oregon, USA 2006: 219 – 230

[13] Samek, M. Practical Statecharts in C/C++. CMP Books. 2002.

[14] UML Specification. http://www.uml.org/

[15] J. Whittle, R. Kwan, and J. Saboo. From scenarios to code: An air traffic control case study. Software and System Modeling, 4(1):71–93, 2005

[16] Telelogic Tau. http://www.telelogic.com

[17] JASCO Home Page. http://ssel.vub.ac.be/jasco/.

[18] Spring Framework, http://www.springframework.org/