An Event-driven Approach for the Separation of Concerns

Hayim Makabee
Yahoo! Labs, Haifa, Israel

Keywords: Separation of Concerns, Event-driven Programming, Aspect-oriented Programming, Coupling, Cohesion.

Abstract: This paper presents an event-driven approach for the separation of concerns in software systems. We introduce the EventJ framework that provides an event-driven extension to the Java programming language. The paper describes a general methodology that can be used to identify the cross-cutting concerns and separate them from the main functionality using events and event handlers. We discuss the pre-requisites to perform this change and illustrate it with a concrete example. Finally, we make a comparison between the event-driven approach and the aspect-oriented one, and conclude that the use of events to separate concerns has a positive effect on software quality attributes such as maintainability, extensibility and reusability.

1 INTRODUCTION

One of the most important principles in Software Engineering is the Separation of Concerns (SoC) (Hursch, 1995): The idea that a software system must be decomposed into parts that overlap in functionality as little as possible. It is so central that it appears in many different forms in the evolution of all methodologies, programming languages and best practices.

Dijkstra mentions it in 1974: *"separation of concerns ... even if not perfectly possible is yet the only available technique for effective ordering of one's thoughts"* (Dijkstra, 1982). Information Hiding, (Parnas, 1972), focuses on reducing the dependency between modules through the definition of clear interfaces. A further improvement was Abstract Data Types (ADT) (Liskov, 1974), that integrated data and functions in a single definition.

In the case of Object Oriented Programming (OOP), encapsulation and inheritance proved to be essential mechanisms to support new levels of modularity. Design-by-Contract (Meyer, 1986), provides guidelines on how to improve interfaces using pre-conditions and post-conditions. Finally, the separation of cross-cutting concerns is the most important motivation for the proponents of Aspect Oriented Programming (AOP) (Kiczales, 1997).

Since the first programming systems were implemented, it was understood that it was important for them to be modular. It is necessary to follow a methodology when decomposing a system into modules and this is generally done by focusing on coupling and cohesion (Constantine, 1974):

Coupling: The degree of dependency between two modules.

Cohesion: The measure of how strongly-related is the set of functions performed by a module.

All methodologies try to reduce coupling and increase cohesion. OOP reduces coupling with the enforcement of encapsulation and the introduction of dynamic binding and polymorphism. AOP provides a solution for the problem of cross-cutting concerns, so that both the aspects and the affected methods may become more cohesive. There are many benefits that software developers expect to obtain when making a system more modular, reducing coupling and increasing cohesion:

Maintainability: A measure of how easy it is to maintain the system. As a consequence of low coupling, there is a reduced probability that a change in one module will be propagated to other modules. As a consequence of high cohesion, there is an increased probability that a change in the system requirements will affect a small number of modules.

Extensibility: A measure of how easily the system can be extended with new functionality. As a consequence of low coupling, it should be easier to introduce new modules, for example a new implementation of an existing interface. As a consequence of high cohesion, it should be easier to implement new modules without being concerned with requirements that are not directly related to

Makabee H..
An Event-driven Approach for the Separation of Concerns.
DOI: 10.5220/0003971801220127
In *Proceedings of the 7th International Conference on Evaluation of Novel Approaches to Software Engineering* (ENASE-2012), pages 122-127
ISBN: 978-989-8565-13-6

their functionality.

Reusability: A measure of how easy it is to reuse a module in a different system. As a consequence of low coupling, it should be easier to reuse a module that was implemented in the past for a previous system, because that module should be less dependent on the rest of the system. Accordingly, it should be easier to reuse the modules of the current system in new future systems. As a consequence of high cohesion, the functionality provided by a module should be well-defined and complete, making it more useful as a reusable component.

2 EVENT-DRIVEN APPROACH

Event-Driven Programming (EDP) can also be seen as a tool for the Separation of Concerns. There is a clear intention of reducing the coupling between the modules that trigger the events and the modules that handle these events. Ideally, the modules which are triggering the events should not be aware of the modules that will handle them. The modules triggering the events should not be concerned of how these events will be handled, or even if a particular event will be handled at all. EDP also helps to increase the cohesion of modules, since it allows separating the business logic of handling the event from the function that triggered it. Both the event triggering module and the event handler become more cohesive. Of course, there can be several handlers for the same type of event, each one with a very specialized way to handle it, what further decreases coupling and increases cohesion.

Some systems are essentially event-driven, for example Reactive Systems, in which their main input from the external environment is in the form of events, and in this case using EDP is a natural and almost required design decision. Some systems are implemented using EDP as a consequence of an analysis and modelling approach, for example action games, but in this case using EDP is an option and they could be modelled in a different way. Finally, some systems are hybrid, and they use EDP only to implement a specific part of their functionality, for example the Graphical User Interface, a Publish-Subscribe subsystem (Eugster, 2003) or even a simple Observer pattern (Gamma, 1995). In most of these situations, the decision of using EDP is not a consequence of its ability to separate concerns. Very often it is simply the most convenient way to model a system or some part of the system.

In this paper we claim that EDP should be adopted as an alternative tool for the Separation of Concerns. This means that when a software developer is confronted with the problem of reducing the coupling and increasing the cohesion of a system, he should consider adopting an EDP approach, based on the explicit definition of Events and Event Handlers. Further yet, we believe that an EDP approach has advantages and can provide benefits that cannot be easily obtained with OOP or AOP.

In the remaining sections of this paper we describe a simple framework to support the introduction of EDP in a system, we provide a brief example of an application in which EDP is effectively used to separate concerns, and we present a more detailed comparison of the advantages and disadvantages of EDP when compared to AOP.

2.1 The EventJ Framework

In order to analyze the efficacy of the event-driven approach for the separation of concerns, we implemented a framework in Java called EventJ. The two main concepts in the EventJ framework are the Events and their respective EventHandlers.

Events: An Event is an immutable object which has state and may have functions that perform some computation over this state. Events have type and are organized in an inheritance hierarchy.

EventHandlers: An EventHandler is responsible for executing some action in response to an Event. A single EventHandler may subscribe to different types of Events. If an EventHandler subscribes to an Event type, it handles also all instances of its subtypes. EventHandlers may be stateless or stateful. An EventHandler may trigger Events itself. EventHandlers receive Events asynchronously and should not depend on the results of other EventHandlers. Each EventHandler runs on a separate thread, and manages its own queue of Events.

EventDispatcher: The EventDispatcher is a Singleton object (Gamma, 1995) which is responsible for the propagation of all Events to the appropriate EventHandlers. When an Event is triggered anywhere in the system, it is initially stored in the EventDispatcher's central queue. Then, according to the Event type, each Event is asynchronously propagated to all the EventHandlers that have subscribed to its type (or to its supertypes). When an EventHandler starts its execution, the first step is to call the EventDispatcher in order to subscribe to all types of Events it intends to handle. The EventDispatcher runs on a separate thread.

Of course, the appropriate usage of the EventJ

framework requires discipline from software developers. For example, the programmer must be aware that EventHandlers will handle Events asynchronously. If there are several EventHandlers associated to the same Event type, one should not expect that these handlers will be executed in any particular order.

In our view, the separation of concerns using the event-driven approach follows three main steps:

Identification: Identify the concerns that may be separated from the main functionality. This means locating the specific pieces of code that we would like to move to some other module in the system.

Triggering: For each concern, define an appropriate type of Event and insert the triggering of this Event in the suitable places in the code.

Handling: For each type of Event, implement the associated EventHandler(s). This means explicitly separating the pieces of code that were previously found in the Identification step and moving them to the respective handlers.

In the case of EventJ, there are two pre-conditions to be possible to separate a specific piece of code from its original context:

Concurrency: Since Events are handled asynchronously, it must be possible to execute this piece of code in parallel with the rest of the original code.

Non-dependency: Since EventHandlers should not modify any external data, the execution of the original code must not depend on the results of the execution of the piece of code that was separated.

For example, the printing of a log message can be easily transformed in a log event. In general it is not necessary to print log messages synchronously: it is enough for the message to contain the exact timestamp of when it was created. Accordingly, the act of printing a log message does not generate any response that is required by the original function that triggered that log event.

We have applied this event-driven approach in real systems, using the EventJ framework. From our concrete experience, the following benefits were obtained:

Readability: It is easy to understand the system, because triggering an Event and subscribing to an Event type are explicit actions that can be clearly traced.

Maintainability: It is easy to maintain the system, because it is possible to precisely identify the EventHandlers that will be executed in response to some Event type.

Extensibility: It is easy to extend the system by adding new EventHandlers without any need to modify the code that triggers the Events. Conversely, it is possible to add a new function that triggers some existing Event type, and all existing EventHandlers will apply to it as well.

Testability: Thanks to the decoupling between the code that triggers the Events and the EventHandlers, it is possible to test them separately. One unit test may trigger Events of some type and check that the right EventHandlers were executed. Another unit test may define test-specific EventHandlers and check that they are activated by the right Events.

Reusability: The Event hierarchy and associated EventHandlers are highly cohesive and independent from the rest of the system, and as such are potential candidates for reuse. It is possible to model the Events to be as generic as the exception types commonly found in Java libraries.

In the next section we provide a more detailed example of a system that was improved using the event-driven approach for the separation of concerns.

2.2 Example: An Instant-Messaging System

In order to provide a more detailed example of the usage of the event-driven approach for the separation of concerns, we analyze an Instant-Messaging (IM) system. The purpose of this type of system is to allow users to exchange a series of short messages online, what is typically called a “chat session”.

In an IM system, each user has a list of contacts (friends), and it is important for him to know who among his contacts is available to chat. Thus, the system must manage the user status, and whenever there is a change in this status the system must notify all his contacts that are currently online. This is normally done using a subscription model, in which each online user is subscribed to all his friends.

To illustrate our approach, it is sufficient to consider the Login and Logout use-cases: A Login request occurs when a user enters the system and becomes online. A Logout request occurs when a user leaves the system and becomes offline. Whenever a user Logs-in, all his online contacts must be notified, and he must be subscribed to all friends in his contacts list. Conversely, whenever a user Logs-out, his online contacts must also be notified and he must be unsubscribed to all friends in his contacts list. Using a traditional object-oriented

approach, these would be the steps of these operations:

```
Login(User u)
{
u.SetStatus(Online);
NotificationMgr.NotifyContacts(u);
SubscriptionMgr.SubscribeContacts(u);
}

Logout(User u)
{
u.SetStatus(Offline);
NotificationMgr.NotifyContacts(u);
SubscriptionMgr.UnsubscribeContacts(u);
}
```

As it is clear in this example, the main function to be performed is the change in the user status. Notification Management and Subscription Management are secondary concerns; they are almost a side-effect of the modification in the user status.

Now imagine that the system also requires the client application to periodically send "keep alive" requests, informing that the user is still online. In this case, the user may "time-out" if the connection was lost for some reason without appropriate Logout. But the code for the Timeout operation would be identical to a Logout:

```
Timeout(User u)
{
u.SetStatus(Offline);
NotificationMgr.NotifyContacts(u);
SubscriptionMgr.UnsubscribeContacts(u);
}
```

This example illustrates that Notification and Subscription Management are cross-cutting concerns. Whenever, for any reason, there is a change in the user status, these modules must be activated. The consequences of the implementation above are that there is too much coupling between the code that changes the status and the modules that manage notifications and subscriptions. Accordingly, the functions that should handle the change in the status become less cohesive.

This is not a rare example. Frequently, software systems have functional requirements that are implemented as cross-cutting concerns. Most often these are operations that do not represent the essence of the business logic. They can be seen as secondary functions that happen as a consequence of the primary one.

In this example, the cross-cutting concerns satisfy the pre-requisites that allow us to adopt an event-driven approach. Regarding Concurrency, both Notification and Subscription Management can be done asynchronously and in parallel with the main flow. It is not necessary to immediately send notifications when a user changes his status; it just needs to happen soon enough. The same is true about updating subscriptions to friends. Regarding Non-dependency, the main functions of Login, Logout and Timeout do not need the results of the Notification and Subscription Management operations.

Thus, after we have successfully identified the cross-cutting concerns, and after we have assured that they satisfy our pre-requisites, it is possible to execute the steps of Triggering and Handling to separate them from the main code:

```
Login(User u)
{
u.SetStatus(Online);
}

Logout(User u)
{
u.SetStatus(Offline);
}

Timeout(User u)
{
u.SetStatus(Offline);
}

User.SetStatus(Status status)
{
…
EventDispatcher.Trigger(new
UserStatusChangedEvent(this));
}

NotificationHandler.Handle(
UserStatusChangedEvent e)
{ … }

SubscriptionHandler.Handle(
UserStatusChangedEvent e)
{ … }
```

In the example above, the triggering of the event was moved to the User class, because it should be the responsibility of the User to trigger an event whenever its status is changed.

There are several benefits obtained by adopting the event-driven approach to separate the concerns in the example above. In terms of system performance, it is possible to make it more efficient, reducing the latency of the Login and Logout operations and increasing their throughput, because more operations are executed in parallel instead of serially. In a software quality perspective, the system is now more modular, since we both reduced coupling and increased cohesion. The Login and

Logout functions are not coupled to anything related to Notification or Subscription Management, and neither is the User class. The Notification and Subscription Handlers are themselves highly cohesive.

In the next section we compare this event-driven approach for the separation of concerns with the widely accepted aspect-oriented approach.

2.3 Comparison between the Event-driven and Aspect-oriented Approaches

For each of the characteristics below, we briefly present a comparison of the EDP approach using EventJ and the AOP approach.

Encapsulation:
EDP does not violate encapsulation. Event Handlers have no access to the data members of other classes.
AOP allows the violation of encapsulation. An advice may change the value of any variable anywhere in the code.

Inheritance:
EDP can use existing inheritance mechanisms. Event types and EventHandlers can be organized in hierarchies. This increases the potential for reuse and extensibility.
AOP does not use inheritance. Pointcuts cannot be organized in a hierarchy, since they are defined by name. Aspects cannot inherit from other aspects. This reduces the opportunities for reuse or extension.

Coupling:
EDP supports coupling by type. An EventHandler is coupled to a type of Event.
AOP allows coupling by name. An aspect can execute over a specific function or variable, by name. If the name of this variable is changed, the aspect must be changed as well.

Order of Execution:
EDP uses EventHandlers which are executed asynchronously and in no particular order, since their execution should be independent of each other and should not directly affect the rest of the system.
AOP has aspects whose execution order is not well-defined and thus if several aspects execute as a consequence of the same code, the results may be unpredictable.

Invocation:
EDP is based on explicit invocations. The Events are triggered explicitly. For each method it is possible to know which Events are triggered by it, and for each Event type it is clear which EventHandlers handle it.
AOP uses implicit invocations. By observing a piece of code there is nothing that indicates that an aspect may be executed. Given an aspect, it is hard to find in the system all pieces of code affected by it.

Extensibility:
EDP makes it easy to add a new EventHandler for some existing Event type or to trigger an Event of an existing type in some new function.
AOP is not easily extensible. If a new advice must be added to some existing pointcut it is necessary to repeat the pointcut definition in a new aspect. If we want to extend an advice to be applied to more pointcuts, it is necessary to change the code of the original aspect.

Reusability:
EDP supports reusability of Events and EventHandlers. Handlers are modular and reusable since they are coupled to Event types, and are independent of the rest of the system.
AOP defines aspects which have small potential for reuse in other systems, since they are coupled to functions and variables by name.

Concurrency:
EDP supports EventHandlers that can be executed in parallel, since they are encapsulated and do not affect code outside them.
AOP does not support concurrency. Advices cannot be executed in parallel. Their execution must be serialized since they can potentially affect the same piece of code.

3 RELATED WORK

To the best of our knowledge, no previous work has proposed the use of an event-driven approach for the separation of concerns. Other frameworks such as EventJava (Eugster, 2009) have been created to add support for events in Java, but their goal was not the separation of concerns. The Ptolemy language (Rajan, 2008) was proposed as an implementation of AOP using events, but it does not discuss an event-driven approach to separate concerns. Other works on event-based AOP (Douence, 2002) have proposed the use of events as an extension to AOP, but not as an alternative approach.

4 CONCLUSIONS

In this paper we have proposed the adoption of an Event-Driven approach for the Separation of Concerns. We described a general methodology that

can be used to identify the cross-cutting concerns and separate them from the main functionality using events and event handlers. We presented the pre-requisites to perform this change and illustrated it with a concrete example, based on our real-world experience using the EventJ framework. We compared our approach with AOP, and concluded that that usage of events has many potential benefits which improve software quality attributes. We hope that this work will help promote the adoption of Event-Driven Programming as an alternative tool to increase the modularity of software systems.

REFERENCES

Constantine, L., Stevens, W., Myers, G., 1974. Structured Design. *IBM Systems Journal*, 13 (2), 115-139.

Dijkstra, E. W., 1982. On the role of scientific thought. In Selected writings on Computing: A Personal Perspective. *New York, NY, USA: Springer-Verlag New York*, Inc. pp. 60–66. ISBN 0-387-90652-5.

Douence, R., Fradet, P., Sudholt, M., 2002. A framework for the detection and resolution of aspect interactions. *In Proc. of the Conf. on Generative Programming and Component Engineering*, pages 173–188.

Eugster, P. T., Felber, P. A, Guerraoui, R., Kermarrec, A., 2003. The Many Faces of Publish/Subscribe. *ACM Computing Surveys*, Vol. 35, No. 2, pp. 114–131.

Eugster, P., Jayaram, K. R., 2009. EventJava: An Extension of Java for Event Correlation. *In Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Springer-Verlag.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. Design Patterns: Elements of Reusable Object-Oriented Software. *Addison-Wesley*. ISBN 0-201-63361-2.

Hursch, W. L., Lopes, C. V., 1995. Separation of Concerns. *Technical Report, Northeastern University*.

Kiczales, G., Lamping, J., Mehdhekar, A., Maeda, C., Lopes, C. V., Loingtier, J., Irwin, J., 1997. Aspect-Oriented Programming. *In Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Springer-Verlag.

Liskov, B., 1974. Programming with Abstract Data Types. *In Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages*, pp. 50--59.

Meyer, B., 1986. Design by Contract. Technical Report TR-EI-12/CO, *Interactive Software Engineering Inc.*

Parnas, D. L., 1972. On the Criteria To Be Used in Decomposing Systems into Modules. Comm ACM 15 (12): 1053–8.

Rajan, H., Leavens, G. T., 2008. Ptolemy: A language with quantified, typed events. *In Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Springer-Verlag.