PHP Runtime AOP Framework

Tony Hokayem

March 29, 2012

Abstract

We propose a design and implement a framework to support Cross-Cutting concerns in PHP using an Object-Oriented approach. The design supports matching of cross-cutting segments using a Join-Point model similar to Aspect-Oriented Programming, binding additional code to be run using delegation and weaving of the aspects at run-time mainly through publish-subscribe methodology. The system is open to extension using basic Object-Oriented techniques and practices and provides seamless API for programmers to execute code that spans across multiple classes. Finally new ideas are presented for future implementation which include using concurrent programming techniques for separate concerns.

Contents

List of Figures

List of Tables

Chapter 1

Introduction and Problem Definition

It is common to find in many programming applications some *common code* which is implemented in different separate parts of the project. The main objective of this work is to provide a way to define this code separately and then make it execute in its right place during run-time. This provide a loosely coupled approach to manage this common-code and improve overall project maintainability. Also, when providing this feature, the added layer will be used in a object-oriented manner which reduces training curve and makes it much simpler to manipulate the added layer from a programmer perspective. The exact classes will also be decoupled from the common-code , in the sense that the classes themselves need not be aware of such executions. This provides with more separation of concerns; the class then will mainly implement its exact functionality.

A common example of a common-code includes logging. Logging generally must be implemented in every class that requires the form of a logging mechanism, in a business situation, this generally include logging authentication, purchases, deposits, withdrawal and secure access. So instead of having each of these parts implement a separate logging mechanism, we would like to have that logging mechanism in one area. We would also provide the functionality to enable and disable logging during run-time. The authentication class hence for example, will not implement any logging, and will not require any direct or indirect dependence on the logging code. The logging will then be hidden and separated from the authentication class.

This reports comprises of 3 additional chapters. The next chapter deals with the background information that this reports takes into consideration and also the technology used for the implementation of the design. The third chapter will examine a design that allows the common-code to be separate from the original code. To illustrate the design, the system will be split into three different components. And finally the fourth chapter will comprise of the conclusions and possible future work to extend the design.

Chapter 2

Background

2.1 Concepts

2.1.1 Object-Oriented Programming

A programming paradigm that involves viewing code units as *objects* which have *properties* and *methods*. Object-Oriented concepts most notably include:

- *Encapsulation:* related data fields and methods are grouped together to form a unit (class) that interacts as a unit.
- *Data Abstraction:* data of an object is not arbitrarily accessed and not all data is visible for people using the object. The object provide a common interface for its behaviour.

2.1.2 Decoupling

Decoupling is a process which involves attempting to keep the unit of code restricted to its general use and be least dependant on other code modules. A component is said to be loosely coupled to other components if it has little or no knowledge of the definitions of the separate components.

2.1.3 Cross-Cutting Concerns

The common-code, that has already been previously referred to, is based on the concept of *Cross-Cutting Concerns* which states that certain parts of the code are spread across different classes. It is said that the code cross-cuts the classes but is not part of the class logic itself. Generally cross-cutting concerns create inter-dependencies in an object-oriented or procedural system due to the fact that such system deals with procedure calling and the code is scattered or duplicated across different procedures. This leads to a loss of modularity.

The re-use and management of Cross-Cutting Concerns is generally tied to *Aspect-Oriented Programming*.

Addressing Cross-Cutting concerns cleanly and allowing for their re-use improves the entire software design keeping things simple and re-usable.

The common-code found in different classes is implemented in one area, called the *advice*. And when combined with the definition of locations where it should be run (the *joinpoints*), it is referred to as *aspect*.

In the design proposed, some parts of Aspect-Oriented will be implemented using an Object-Oriented Approach.

2.1.4 Delegation

Delegation is a concept in *Object-Oriented Programming* where an object delegates some of its tasks to other specialized objects to handle it. This offsets the logic to a helper object which does it on the behalf of the object. In the design proposed, the procedure of the cross-cutting concern should be delegated to the *aspect*.

2.2 Technology

2.2.1 General

The design is proposed as an abstract design that can be implemented on any language having the following properties:

- *Object-Oriented*: The language must implement Object-Oriented programming as the design is based on OO concepts and supports extendibility through Object-Oriented concepts such as inheritance, interfaces and objects.
- *Dynamic Dispatch*: Mapping of message (function) to an object at runtime as opposed to compile-time binding. (While this is basically available in C++ through polymorphism), the need in this design is the usage of an arbitrary message on an arbitrary object.
- *Reflection*: There is a basic need to access meta-data of the objects defined, this is necessary to be able to determine data on the object at run-time.

2.2.2 PHP

The design suggested will be implemented using PHP (Hypertext Preprocessor) which is $Object - Oriented$, supports *Dynamic Dispatch*, and has $Reflection$ using the PHP Reflection API

Chapter 3

Handling Cross-Cutting Concerns

This chapter will illustrate the design that will provide programmers with a set of classes. These classes would allow them to tackle cross-cutting concerns and separation of the different layers (Logic, Business, Database).

Therefore this section comprises of generally three major areas. First the method that allows the programmer to define which part of the code to select will be examined, this will be the matching section. Second, the procedure that allows the programmer to execute his code will be detailed, this is the binding section. And third, an explanation will be given as to how the system weaves the code together.

To understand better the part of each aspect of the system, a general view will be provided next.

Implementation Strategy

To be able to create such a system, we need a way to define the bindings; which generally points out where and when the common-code should be included. Generally we will be using *public functions* as locations to insert before or after the common-code itself. That is why we will be intercepting the function calls themselves then determining as per class and function call, when to call the common-code which is also in the form of an function (which may be part of an object).

A general use-case for how the main functionality will work is found below:

User	**System**
1. Call Class.function()	
	2. Find code that should run before Class.function() 3. Execute code that should be run before the function
	4. Execute the Class.function()
	5. Find code that should run after Class.function() 6. Execute code that should run after Class.function()
	7. Return Class.function()'s result (affected by the interception)

Table 3.1: General Use-Case displaying the major usage of the system

In the table, the rows which depict normal behaviour have been highlighted. The additional functionality, should be implemented without interfering with the class call itself. And to accomplish this, we will be creating a wrapper that intercepts the function call and provides the additional functionality (while acting the same way as the class itself) is needed.

3.1 Matching

This section illustrates to the programmer where and how he can define which segments of the program to target for his code.

3.1.1 Matching Scope

The set of classes that are created operate on the foundation of Object-Oriented programming. Therefore it is important to note that all protected or private functions will not be intercepted by design, this is of course to ensure the proper paradigms applications of Object-Oriented Programming. What we will be intercepting thus is everything public, that includes:

1. $Classes$: the programmer will define which classes to attach his code on.
2. $Functions$: the programmer will define which public functions are of importance for his code.
3. $Interface$: the programmer can even target whole interfaces (Logging Objects, Database objects)

3.1.2 Join-Point Model

The programmer will then detail a list of conditions on which it is desired to call his code and will supply them to the framework ($Aspect\_Handler$) to notify that the code should be run in these conditions.

JoinPoints Sequence

The conditions are provided as a Sequence of Joint-Points where a Joint-Point is a basic condition. Examples of Joint-Points are class name or function names. However we will illustrate later the versatility of Join-Points. The sequence of joint-points is provided as a list containing other lists of Joint-Points, let S be the sequence and let $J_1...J_n$, $K_1...K_n$ be two lists of joint points. The structure of the sequence is a two dimensional array where the outer dimension arrays are evaluated using OR and the inner dimensions are evaluated using AND. As an illustration of the Model, consider $S = ((J_1, J_2..J_n), (K_1, K_2, ..K_n))$. The final expression to be evaluated is: (J_1 AND J_2 AND .. J_n) OR (K_1 AND K_2 AND .. K_n). For practical considerations, if the inner dimension is a single element then we have one simple array, the entire array is processed with the OR. If we have $S_2 = (J_1$, J_2, ... $J_n)$, then the evaluation is (J_1 OR J_2 OR .. J_n). This sequence is stored in the *Aspect_Hook* object, part of the Aspect class.

Now each Joint-Point follows the *Aspect_JoinPoint_Abstract* abstract class which has one basic method called *test*() this method takes three parameters which can be used to match, these parameters are: *class*, $function$, *instance*. So for the object on which the function is being called the programmer gets to add conditions based on the:

1. $Class$: the class of the object. (example: *DepositAccount*)
2. $Function$: the function being called. (example: *deposit*())
3. $Instance$: the actual object on which it is called. This is left for the programmer to determine instance specific triggers. This can be used to get to see if the object implements a given interface so that triggering can be done on the interface level.

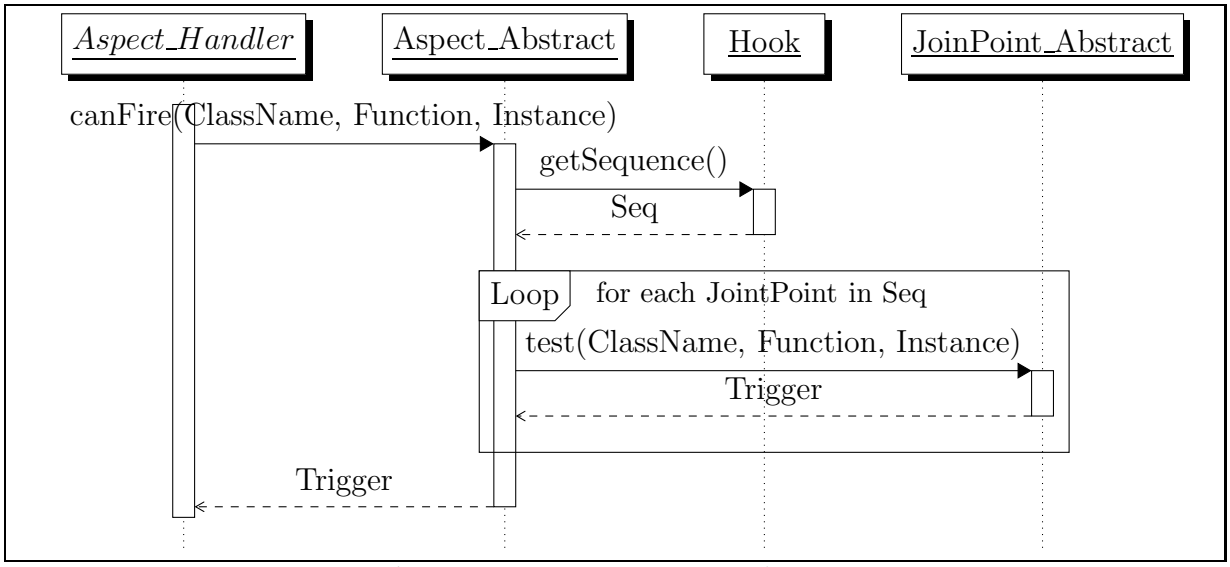

Figure 3.1: Join-Points evaluation

Implemented types of JoinPoints

As we have seen so far, a JoinPoint is a simple boolean function that takes as parameters the *ClassName*, *FunctionName* and *InstanceObject* and provide us with the trigger output. A true output indicates that the JoinPoint has matched one part of the code that should be matched. Enumerated in the table below is the set of Classes that implements the basic JoinPoints needed, programmers may extend the available JoinPoint by simply inheriting from the *Aspect_JoinPoint_Abstract* class.

Class Name	**Functionality**
JoinPoint_True	Matches always true.
JoinPoint_Instanceof	Matches if the *InstanceObject* is an instance of *InstanceName*. Useful in PHP to detect inheritance or interfaces, we check if the object has a given parent or implements an interface.
JoinPoint_Regex_Class	Matches if the object's ClassName matches a given regular expression.
JoinPoint_Regex_Function	Matches if the Function being called on an object matches a given regular expression.
JoinPoint_Regex_ClassFunction	Shorthand for both Regex_Class and Regex_Function

Table 3.2: Table listing the implemented JoinPoints, the *Aspect_* prefix has been removed from class names for readability

3.1.3 Simple Matching

So far we noticed that conditions can be complex and can span across multiple classes, functions or objects. Using the *JoinPoint_Regex_Class* a programmer can define to attach code to all classes starting with *a*. Sometimes however programmers have a certain set of classes in mind or function. When the programmer has an idea of the exact class or function that needs to be part of the trigger, there exists a way to provide the framework with a list of classes or functions to match exactly while avoiding JoinPoints.

3.1.4 The *Aspect_Hook* object

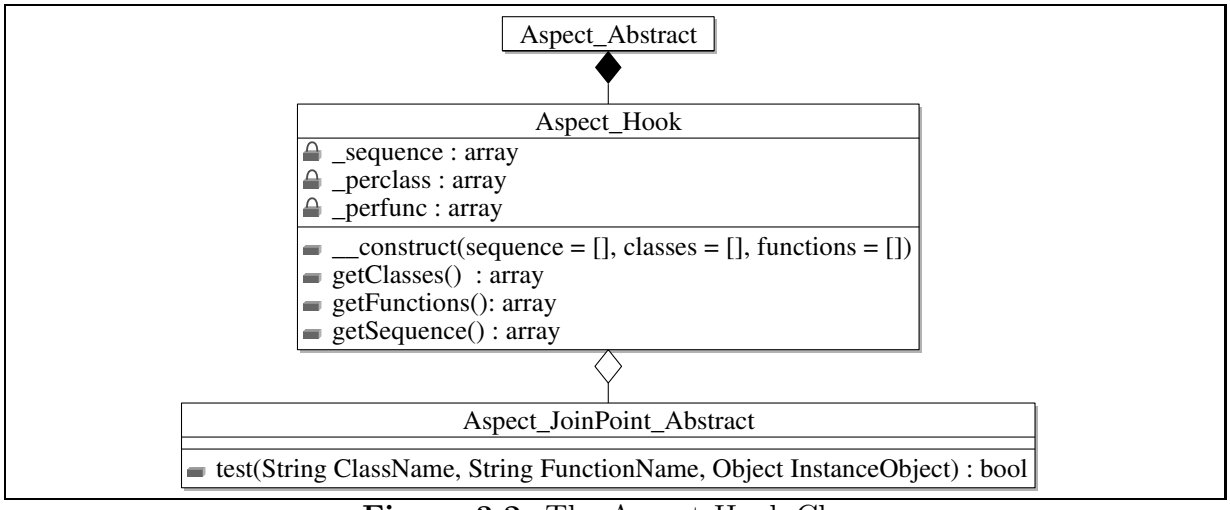

Figure 3.2: The Aspect Hook Class

The *Aspect_Hook* object is a basic object that tells the code where it should execute. It provides all the matching logic. At the construction of the object the programmer can define 3 optional parameters. The first is the array containing the sequence of JoinPoints, the second is a list of the classes for Simple Matching and the third is a list of functions for Simple Matching. The remaining accessors provide the necessary interface for the greater unit that will define what code will execute. In the case of simple matching, the *handler* will match the function or class name exactly but a sequence still has to apply. In a general sense if a programmer wants to match class name or function name only, then the *JoinPoint_True* can be used as the sole element of the sequence.

3.1.5 Examples of Matching

In this small section, an example of matching will be provided. Let us assume that our class list consists of the following: Two classes are presented, a *CreditAccount* and a

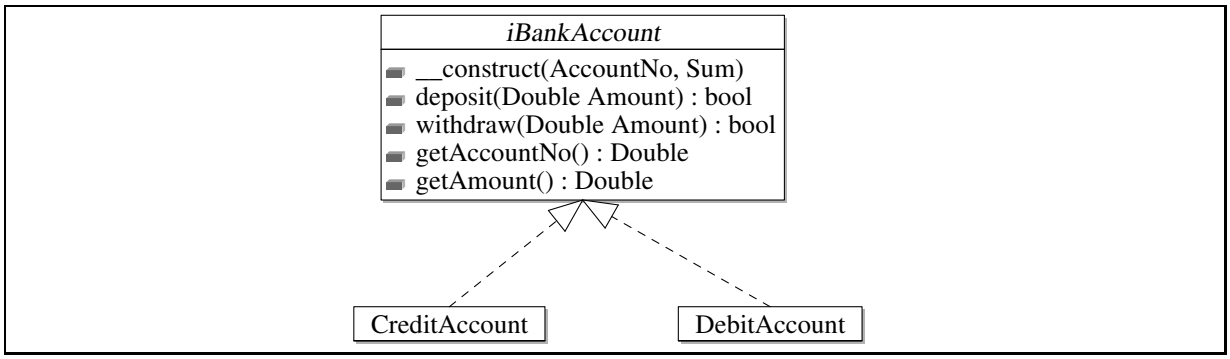

Figure 3.3: Class Diagram of an Example set of classes

DebitAccount class which implement the interface *iBankAccount*, assuming that Credit and Debit account have other functions that have been hidden (will not interest the example).

Now our task will be to create matchers to log the *Amount* passed to deposit or withdraw for both classes, we can do this in several ways.

The first way uses a regular expression to match both classes and function names:

```
1 new Aspect_Hook(
2     //Sequence
3     array(
4       new Aspect_JoinPoint_Regex_Class('/(Credit|Debit)Account/'),
5       new Aspect_JoinPoint_Regex_Function('(Deposit|Withdraw)')
6     ),
7     null, //No simple matching for Classes
8     null  //No simple matching for Functions
9   );
```

The second way illustrates usage of only simple matching:

```
new Aspect_Hook(
    //Sequence (needs to be true)
    array( new Aspect_JoinPoint_True() ),
    //Simple classes matching
    array('CreditAccount', 'DebitAccount'),
    //Simple Function matching
    array('withdraw', 'deposit')
  );
```

If we want to match all get functions of the given classes we can use:

```
new Aspect_Hook(
    //Add a condition for function
    array( new Aspect_JoinPoint_Regex_Function('get.*') ),
    //Restrict to classes using Simple Match
    array('CreditAccount', 'DebitAccount')
  );
```

3.2 Binding

This section illustrates to the programmer the possible ways to execute his code once the matching has been completed. In this area the main object we will be dealing with is the *Aspect_Abstract* object as an abstract class and the implemented aspect styles. When binding the programmer does not necessarily have to just execute one portion of code, there must be some data about the context in which he is executing his code in. This is necessary because if a cross-cutting concern implements logging, it has to log specific parameters pertaining to the initial function call, in a more general sense, the additional code might need to *interact* with the original matching segment.

In this section we will first illustrate how exactly is the code joined to the matching code, and the possible interaction of this code with the *context*.

3.2.1 The *Aspect_Abstract* Class

The *Aspect_Abstract* parent class provides the binding point and allows for interfacing with the *Aspect_Handler* which is the core library class. Below is a class diagram representing it:

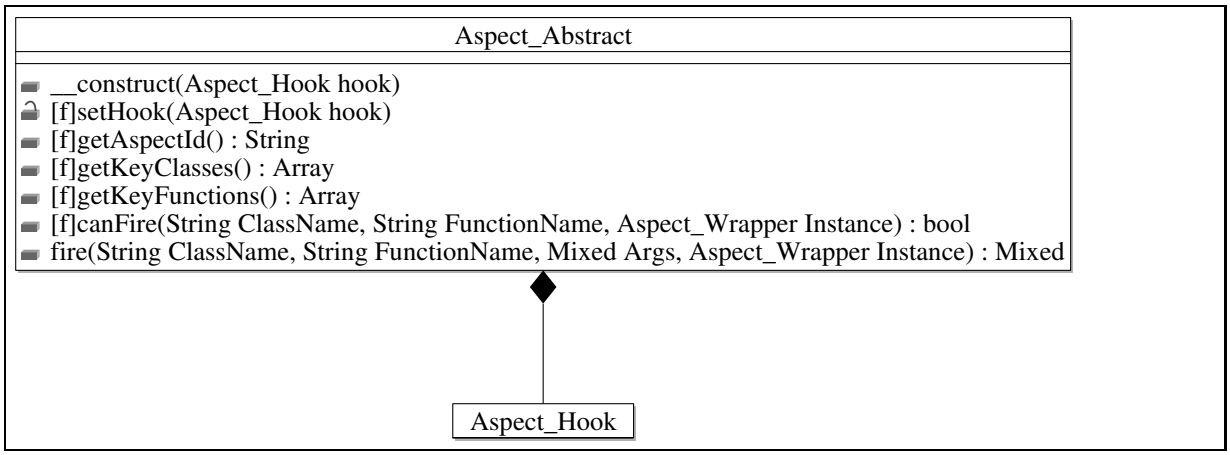

Figure 3.4: The Aspect Abstract Class

A very basic version of an aspect is constructed with an *Aspect_Hook* that determines where it should be fired. Note that the prefixed functions with $[f]$ in the diagram are final functions and cannot be overwritten, these functions provide the basic functionalities to

integrate with the framework of any binding. What generally changes is the constructor (though there is a need to use $setHook()$ in the new constructor to determine the hook. The $canFire()$ procedure has been mentioned before and loops through the sequence to do the matching. What is of interest here is the $fire()$ procedure which is the method that has the code that should be executed.

As we can see, the programmer needs to inherit from $Aspect\_Abstract$ to use his code in the $fire()$ method however this is not very useful unless the programmer wants to inject an arbitrary code. This is why a set of children classes is provided, that use several methods to handle the $fire()$ and forward the call to the programmer. We examine next the context in which the $fire()$ method is called which are of great importance for the programmer.

3.2.2 The Different Contexts

In the design proposed we provide the programmer, with two possible contexts a read-only context, and a read-write context.

1. **The read-only context** as the name implies can only be accessed by the programmer to get status of the current call. A programmer in this case will be able to pull data of the context such as the class of the object and the parameters passed to its method.

2. In a **read-write context**, the programmer has the same possibility of reading, but with more privileges. These privileges include changing the control flow, changing the return value or the parameters passed to the object's method.

The *Aspect_Meta* Object

The *Aspect_Meta* object provides context data for any other code that will be additionally executed to handle the matching segment. The *Aspect_Meta* object provide read-only context data and is passed in the *args* array to the *fire*() method of *Aspect_Abstract* as the first element. A class diagram of the public methods of the class are shown below:

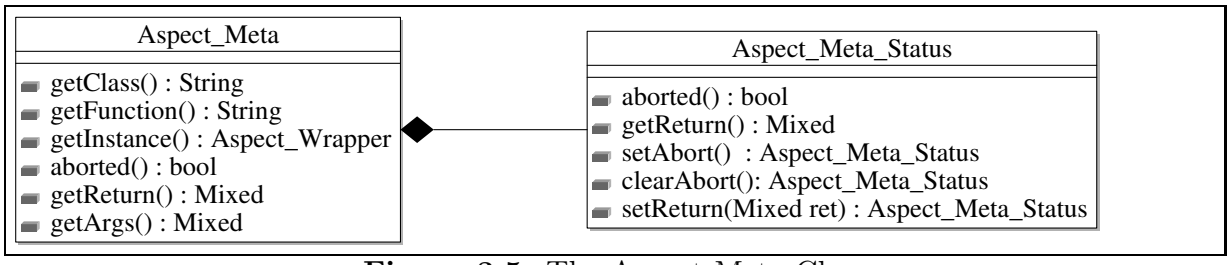

Figure 3.5: The Aspect Meta Class

The *Aspect_Meta* status is passed to the programmer and can be queried as have seen for (to provide info on matched point):

- Class Name (using *getClass*())
- Function Name (using *getFunction*())
- Instance Object (using *getInstance*())
- Arguments originally passed to the function (using *getArgs*())

The other methods (*aborted*(), *getReturn*()) and the *Aspect_Meta_Status* will be discussed in the next section.

The *Aspect_Result* Object

The *Aspect_Result* object is the object that handles the read-write context by providing write functionality. It is used as a return value to the *fire*() function of the *Aspect_Abstract* and detected by the *Aspect_Handler* to make necessary modifications. If such an object is returned in a read-only context then it is ignored. The *Aspect_Result* is an immutable object (i.e. does not support write operations to it) and is used as a

tool to detect changes from higher classes. When a programmer needs to make changes, a new object is constructed with the change and returned. The *Aspect_Result* has the following methods:

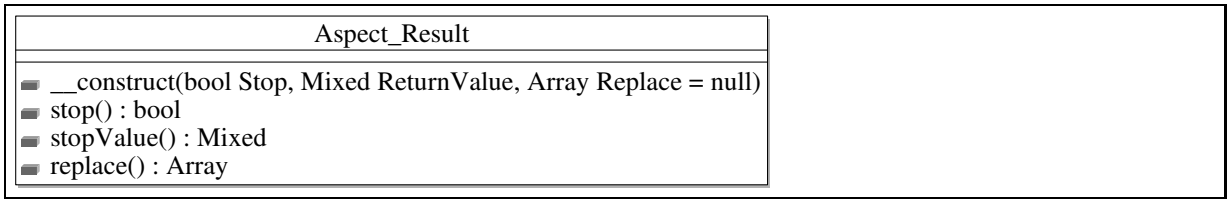

Figure 3.6: The Aspect Result Class

In the constructor a *Stop* flag determines whether to short-circuit control-flow (will be discussed in later sections), a $ReturnValue$ overrides the matching-code segment's return value, it replaces the return value of the function on the given object with $ReturnValue$. The *Replace* array is a dictionary (hash-table) where the key represent the index of the argument and the value represents the value to swap the argument with. This allows to change the actual arguments passed to the function. In the case of different bindings on the same location, this affects the parameters passed to the subsequent Aspects. The other functions are used to read the data in the object.

3.2.3 Binding Locations

The binding locations specify where the $fire()$ function will be called with respect to the original function of the object. In this design we provide $four$ binding locations as illustrated in the table (Actual function, represents the actual method call for the object):

Location	**Context Type**	**Call Priority**
Before	Read-Only	0
AroundBefore	Read-Write	1
Actual Function	-	2
AroundAfter	Read-Write	3
After	Read-Only	4

Table 3.3: The Binding Locations Relative to Method Call

As we can see the binding locations determine the context of the $fire()$. We will examine next how the *read-write context* operates in the $AroundBefore$ and $AroundAfter$ locations.

3.2.4 Replacement and Control-Flow Modifications

In the case of an *read-write context* it is possible to change the actual parameters passed to the method, its return value and the control flow. An important point to consider is that more than one aspect might be bound on the same location, and so these are organized in a list manner based on *First Come First Serve* ($FCFS$) basis any effect done by one will cascade to the rest using $FCFS$. Additionally the locations are ordered, any change cascades to the rest of the locations.

Replacement

Any aspect requesting the replacement of the arguments in a *read-write context* changes the arguments passed to subsequent aspects in the same location, and to the location afterwards. As such if a replace is returned in AroundBefore it will actually change the parameters passed to the intercepted Object's Method. Therefore parameter replacement generally should happen in $AroundBefore$ although it is possible to do it in $AroundAfter$. It is still possible however to access the original arguments by using the $Aspect\_Meta$ object passed to the $fire()$ function by calling the $getArgs()$ method.

Stop and Return

The return value is never changed unless a *Stop* flag is set in the *Aspect_Result* object. The *Stop* flags causes the interruption of the current execution of aspects in the location, sets the Return Value to the $ReturnValue$ set in the *Aspect_Result* object and moves to another Location. The control flow skip is done as follows:

1. **If the** *Stop* **is requested in the** $AroundBefore$ **location**, the execution jumps to the $After$ Location (which **still executes**). Call to the actual method is skipped and the actual method is assumed to have returned the $ReturnValue$ set in the *Aspect_Result* object.

2. **If the** *Stop* **is requested in the** $AroundAfter$ **location**, the execution jumps to the $After$ Location. The value of the return is changed to the $ReturnValue$ set in the *Aspect_Result* object.

3.2.5 Delegation

In order to keep with best practices and provide practical ways for programmers to use their code with this system. We provide with several classes that extend the *Aspect_Abstract* class and provide a certain *delegation* of the $fire()$ function. The programmer is not expected to implement the $fire()$ function for each aspect of his code. We provide 4 classes to simplify the API: *Aspect_Object*, *Aspect_Function*, *Aspect_BuiltinFunction* and *Aspect_OneToOne*. In *Object-Oriented Programming*, *Delegation*, in short, is the offsetting of the code to be executed by another entity specialized in that. And this is exactly what we want to do, we need to keep the programmer's code isolated from the *Aspect_Abstract* class.

We will examine each in this section and provide an example as to how they are used. In all of our example we will assume the *Aspect_Hook* to have been defined somewhere and will be referred to as $\$h$.

Aspect_Object Delegator

The *Aspect_Object* delegator allows an instance of an arbitrary object to execute a specific method when the $fire()$ is called. The method signature comprises of a list of arguments identical to the intercepted function but prepended with the *Aspect_Meta* object. The below class diagram illustrates two objects that we will use in the example: In this

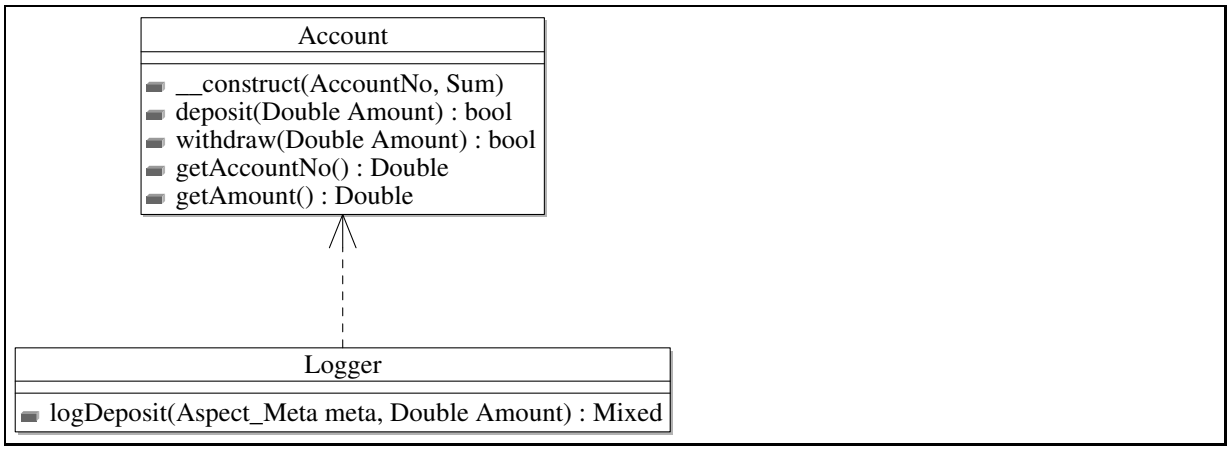

Figure 3.7: Object Delegator

example we will assume that the hook is set to match the class *Account* and the methods: *deposit*() and *withdraw*().

```
1  //Create an instance of an arbitrary object
2  $object = new Logger();
3  $aspect = new Aspect_Object(
4      //Specify Hook
5      $h,
6      //Specify the instance to which fire() is delegated
7      $object,
8      //Specify the function to which fire() is delegated
9      "logDeposit"
10   );
11
12
```

```
13 //Assume account is the object Account
14 //Calling:
15 $account->deposit(300);
16 //Will trigger:
17 $object->logDeposit($meta, 300);
```

Now whenever *Account.deposit*() or *Account.withdraw*() is called, *Logger.logDeposit*() will be called.

Aspect_Function Delegator

The *Aspect_Object* delegator works similarly to the *Aspect_Object* delegator excluding objects. It delegates the *fire*() to either a function or a static function. It has to follow the same signature as the method of the object described in the previous section.

```
1  //Create a function
2  function customLogDeposit($meta, $amount) {
3    //Function code
4  }
5  $aspect = new Aspect_Function( $h, //Hook
6      //Specify the function to which fire() is delegated
7      "customLogDeposit"
8    );
9  //For static Binding:
10 $aspect = new Aspect_Function($h, //Hook
11     //Specify the function to which fire() is delegated
12     "SomeClass::SomeFunc"
13   );
```

Aspect_BuiltinFunction Delegator

The *Aspect_BuiltinFunction* delegator works similarly to the *Aspect_Function* delegator excluding the meta object. It is meant to delegate the *fire*() function to a builtin PHP or a function that is not designed to handle cross-cutting concerns. Therefore it passes the same arguments in the same order.

Aspect_OneToOne Delegator

The *Aspect_OneToOne* delegator works similarly to the *Aspect_Object* delegator but does not specify a specific method. Instead it takes an optional prefix. Whenever an aspect matches a certain object's method, it forwards the call to the same function for the object prefixed by the optional prefix. This is useful to implement observers or one-to-one mapping between different layers of objects. An example could be an Account and an AccountModel, where the *AccountModel* get forwarded calls to update a certain database, keeping the Data Access separated from the Business model.

```
1  $object = new Logger();  //Arbitrary Object
2  $aspect = new Aspect_Object($h, //Hook
3      $object,  //Specify the instance to which fire() is delegated
4      //Specify the prefix
5      "log_");
6  //Calling:
7  $account->deposit(300);
8  //Will trigger:
9  $object->log_deposit($meta, 300);
10 //Calling:
11 $account->withdraw(300);
12 //Will trigger:
13 $object->log_withdraw($meta, 300);
```

3.3 Weaving

Now that the programmer's code can be attached to a specific segment code, this section deals with the API provided to the programmer to register the bind point aspect to the location, and illustrates how the framework internally deals with handling all this. At first the framework's $publish - subscribe$ API will be discussed, followed by the internal weave process of the framework.

3.3.1 The Publish-Subscribe Pattern

The $Publish - Subscribe$ pattern is a messaging pattern where we have two basic entities: Publishers and Subscribers. Publishers generally define messages to be published to whoever is interested to listen for those messages. Subscribers are entities that *register* to a given message to receive data, regardless of how many publishers there exist.

The $Publish - Subscribe$ pattern allows for *Loose Coupling* in the sense that $Publishers$ are loosely coupled to the $Subscribers$, the $Publishers$ are not even aware of the existence of the $Subscribers$. This is intentional because we need to *isolate* the bound code from the matching segment, after all *cross-cutting concerns* are of different domain than the actual code in the object and need be separate.

Namely in our own system, the publishers would be the objects and methods we would like to intercept and the subscribers are the aspects, where they match a message criteria and execute their bound code.

3.3.2 Subscribe using *Aspect_Handler*

Once we have our aspect defined, by compositing from a hook and a bound code, we want to specify the *location* in which the aspect will be *subscribed*, locations are the main message channels for the subscribe method. To subscribe an aspect with the *Aspect_Handler* we use one of four functions (similar to locations):

1. $wrapBefore()$: Registers aspect for the $Before$ location.
2. $wrapAroundBefore()$: Registers aspect for the $AroundBefore$ location.
3. $wrapAroundAfter()$: Registers aspect for the $AroundAfter$ location.
4. $wrapAfter()$: Registers aspect for the $After$ location.

Upon successful subscription, the above functions return a boolean true.

Note that the *Aspect_Handler* class is a class simply constructed with static functions, it is the main library class for our system.

3.3.3 Unsubscribe

Aspects may be removed by calling the equivalent *unwrap* functions: $unwrapBefore()$, $unwrapAroundBefore()$, $unwrapAroundAfter()$, $unwrapAfter()$. These function take as parameter the $AspectID$ which can be obtained by querying the *Aspect_Abstract* object using the $getAspectId()$ method. This removes the aspect from the binding.

3.3.4 Publish using *Aspect_Wrapper*

For programmers to specify that their objects need to use our system, they must explicitly declare them using the *Aspect_Wrapper* class. This classes using two primary static functions to construct our publishers: *load*() and *wrap*(). In this part we will discuss the *Transparent Wrapping* property of the *Aspect_Wrapper* class and the two functions that produce an instance of *Aspect_Wrapper*.

Transparent Wrapping

The *Aspect_Wrapper* object is the main object the programmer will be using while interacting with our system. And understanding this property is crucial to understanding how a programmer can use our system. To be able to better address the problem, we will be regarding the objects as entities that receive messages, and can execute the message if they have a defined message name. This is contrary to the typical *typed Object-Oriented* programmer where an object's methods are clearly defined in the object itself. So in short if an arbitrary object *obj* receives a message *message*() then if the *obj* has a declaration in its code for *message*() it will execute it, otherwise it will error. This kind of object message passing is implemented in certain languages (e.g: *PHP*, *Ruby*, *Smalltalk*) and this is the scheme we will need for our system to work.

What the *Aspect_Wrapper* accomplishes is that it takes any arbitrary object, called the *wrapped object* and forwards the messages it receives to the *wrapped object* while publishing to the system that the *object* called the *message* which is the *function*. Using the *Aspect_Wrapper* therefore instead of the *object* provide an identical result with respect to *public methods*. Consider an object *obj* of class *cl* with the public methods: $f1()$, $f2()$, $f3()$, and an object *wrapper* of type *Aspect_Wrapper* with an wrapped object of class *cl*, then *wrapper* will support calls to $f1()$, $f2()$, $f3()$.

Using wrap()

The *wrap*() static function of the *Aspect_Wrapper* class takes an arbitrary object and returns an *Aspect_Wrapper* with the wrapped object the parameter passed. The *wrap*() function is useful to register an exiting instance of an object as publisher.

Using load()

The *load*() static function takes as first parameter a class name, then arguments to be passed to the constructor, in the form of: *load*(*Class*, arg_1, arg_2, .. arg_n). This function will construct a new object of type *Class*, *wrap* it and then return it as an *Aspect_Wrapper* instance. This differs from the *wrap*() method because it triggers a specific method *newInstance*, which is a function with the signature of the *constructor* this method supports subscribers.

The *newInstance* is treated like any other function but differs for subscribers in the following locations:

1. In $Before$ and $AroundBefore$ the actual object instance is not yet constructed, the *instance* provided to the subscribers is a class of type $ReflectionClass$ of the *object*'s class. The $ReflectionClass$ is part of the PHP $ReflectionAPI$ and consists of the metadata of the *object*'s real class. This is only to be intercepted by people who are familiar with the $ReflectionAPI$ specific to PHP.

2. In $AroundBefore$ if a replace is provided, then the parameters passed for the constructor of the object will be changed to their replace value. This can be used by Data Abstraction layers where the first parameter could be an ID, and the rest are populated from database.

3. In any *Read-Write Context* stopping and returning, will return an object to be wrapped later on. On $false$ the call to create a new object will return null.

Once the *newInstance* is executed an additional message will be fired which is *wrapInstance* which provides the object to be wrapped as the instance of the method. The *wrapInstance* supports subscribers. The *wrapInstance* call itself is a keyword for the system which will tell it to execute a wrap on the object and will not call $object- > wrapInstance()$ therefore stopping before that call is made requires the programmer to return an $Aspect\_Wrapper$ object. Generally these methods are provided to provide additional control over the weaving process and should not in general be subscribed to.

Publishing

When a call to method $func()$ is executed on an $Aspect\_Wrapper$ object with a wrapped object $\$obj$, it triggers the $Aspect\_Handler$'s $fire()$ method which publishes the call to the registered aspects compacting the arguments in list (shown as $\$args$ in the diagram), the return value of the function is indicated by $retVal$. The sequence is pretty straightforward and shown below:

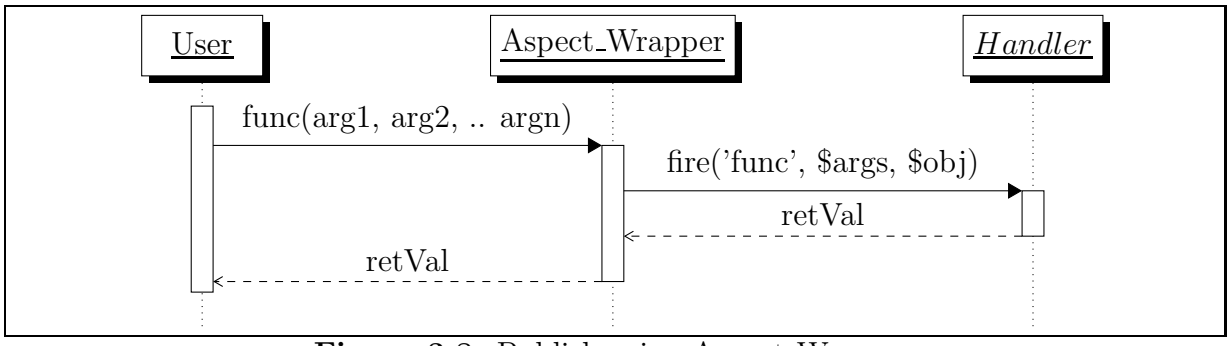

Figure 3.8: Publish using Aspect_Wrapper

3.3.5 Internal Weave using $Aspect\_Handler$

The $Aspect\_Handler$ main library handles the subscription of the new aspects using $wrap$ functions, the unsubscription using $unwrap$ and the publishing using $fire$ forwarded from an $Aspect\_Wrapper$ object. In this part we explain the internal functioning of each of the three aspects with respect to the $Aspect\_Handler$.

Wrapping

The $wrap$ functions operate by defining for each of the 4 locations two hash tables: one for classes and one for functions. The hash-tables work as an optimization technique for fast search for exact matches. When a $wrap$ is called the $fire$ function queries the $Aspect\_Abstract$ object for its $classes$ and $functions$ defined in the hook for simple-matching.

In which case we have two cases:

1. **Classes or Functions lists aren't empty**: which means that the aspect implements simple matching. In this case the Handler looks up the Class or Function in its hash-table using class name or function name as key, and the value is a list containing the aspects attached to that. It adds the aspect to that list.

2. **Classes or Functions lists are empty**: which means that the aspect does not implement simple matching. In this case the Handler adds the aspect to the *global* list of aspects that run for that location.

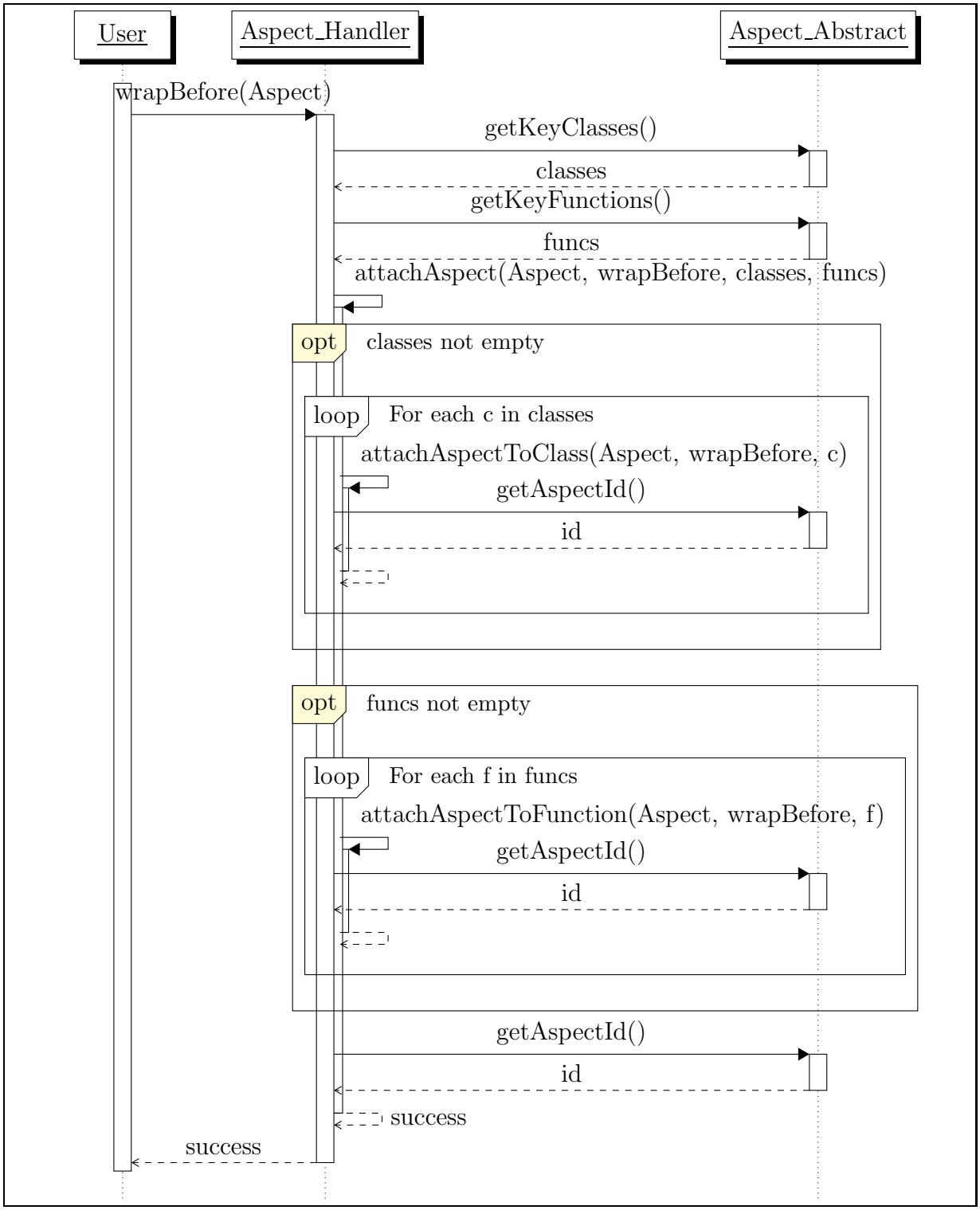

Figure 3.9: Aspect_Handler Handling a *wrapBefore* Request

Unwrapping

The *unwrap* function operate similarly to the *wrap* functionality, however the $Aspect\_Handler$ removes the *aspect* from its hash-table's lists using the $AspectID$.

Publishing

The $fire()$ function is the main function that publishes the events. We have seen in the previous section that the $Aspect\_Wrapper$ forwards the $fire$ call to the $Aspect\_Handler$ with the method, object instance and the arguments. The $fire$ function handles all the logic specified for binding and executing code. To execute the original method on the object it uses PHP's internal $call\_user\_func\_array()$ that calls a method of an object on an instance with an array of arguments (equivalent to sending a message to the object). The following illustrates its work:

```
1 $obj = new Account(); //Assume it has a deposit method
2 call_user_func_array(
3   array($obj, 'deposit'),
4   array(200)
5 );
6 //Is equivalent to:
7 $obj->deposit(200);
```

The $fire$ calls another method *execChain* which matches and fires all aspects in a given location (or Chain). The sequence diagrams of both methods are displayed next. The $stopAndMerge$ function merges the new arguments with the request of a *Replace* and returns true if a stop is requested.

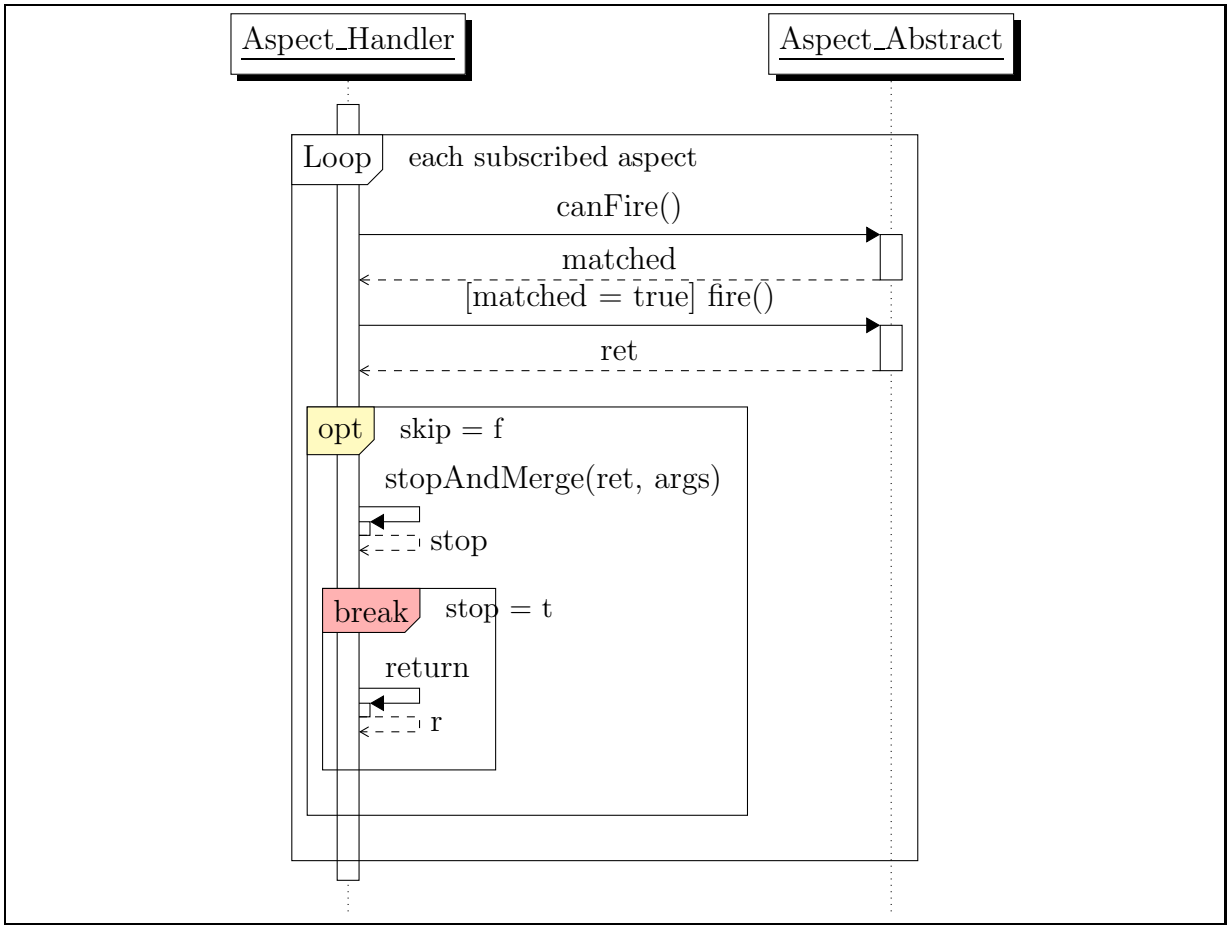

Figure 3.10: Execute Chain Loop Sequence Diagram

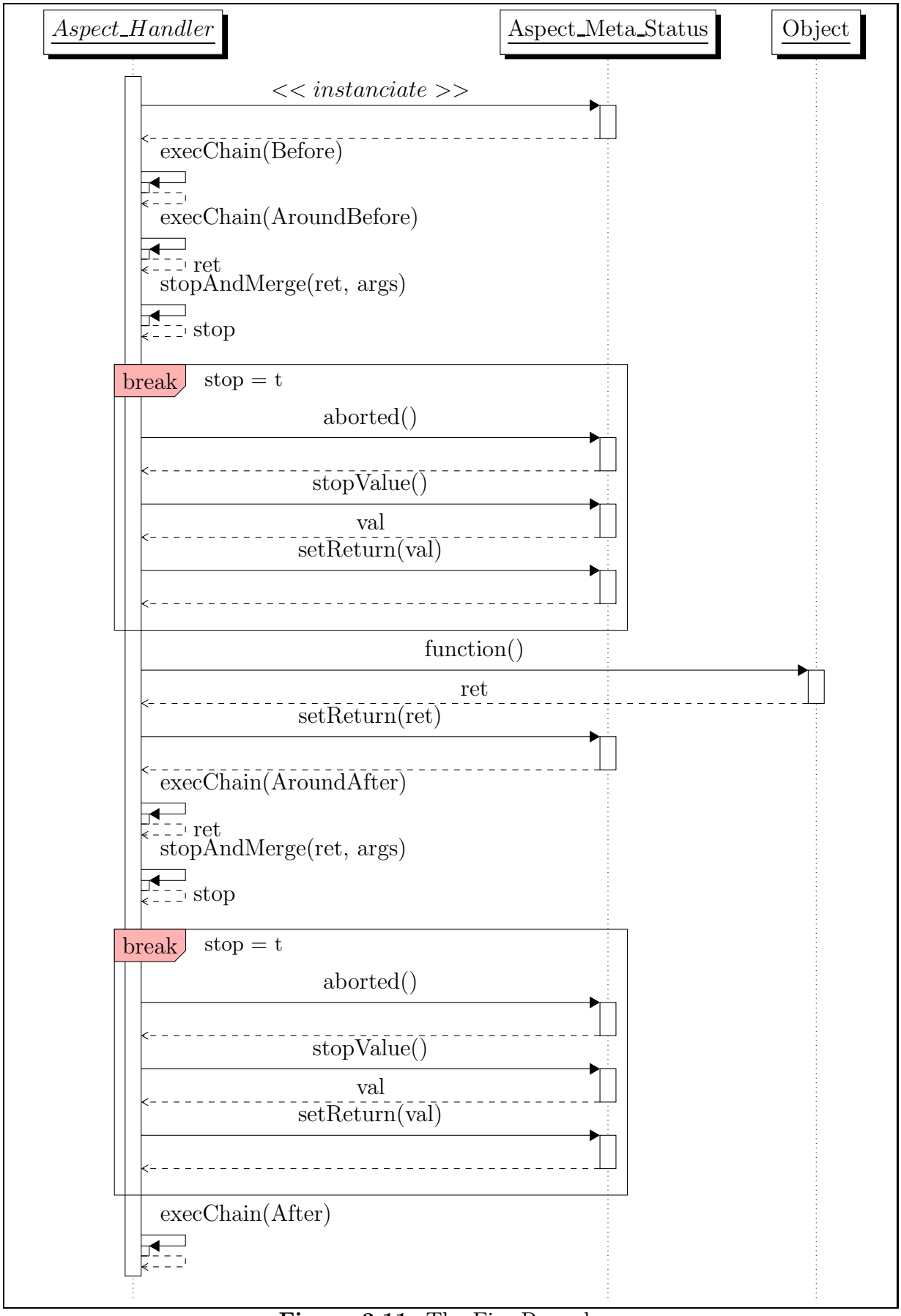

Figure 3.11: The Fire Procedure

Chapter 4

Conclusion and Future Work

We have proposed a design to support Cross-Cutting concerns in PHP using an Object-Oriented approach. The design supports matching of cross-cutting segments using a Join-Point model by using a list of conditions or a simple-matching strategy. We have next introduced the binding strategy of an additional code to be run using delegation. We also defined the two types of contexts proposed, read-only and read-write. We also illustrates the four possible locations for the code to be bound and their order of priority. We then illustrates how the framework implements the weaving of the aspects at run-time mainly through publish-subscribe methodology. And how the programmer can define the objects to be published.

The system is open to extension using basic Object-Oriented Techniques and practices and provides seamless API for programmers to execute code that spans across multiple classes. The possible future work includes the deployment of new conditions for matching by creating specific implementation to the *Aspect_JointPoint_Abstract*. Possible delegation could be extended to provide more functionalities for the delegation of the firing code, an example could be a Filter delegator which reads the arguments passed to the method and replaces them with a filtered version using a built-in function (such as *html_entities*() or *strip_tags*(). Furthermore, the read-only contexts could be improved in a language supporting concurrent programming by making them run in parallel since they are independent and read only.

Appendix A

A Basic Complete Example

In this appendix, an example will be provided which features a basic account class. Although cross-cutting concerns span multiple classes we will be focusing on one class for simplicity but we will show separation of domain logic from business logic. We will create three aspects:

1. *Logger*: A logger aspect will log *All* function calls. (To illustrate the Meta Object and Status)

2. *Security*: A security aspect that will inject a new policy to now allow deposits and withdrawals above 5000 USD to be allowed. (To illustrate Stop)

3. *Promotion*: A promotion aspect that will give an additional 10% free gift for deposits when the promotion is active.

```
1  class Demo_Account
2  {
3    private $_balance = 0;
4    private $_no    = 0;
5    public function __construct($no, $balance)
6    {
7      $this->_balance = $balance;
8      $this->_no  = $no;
9    }
10   public function debit($amount)
11   {
12     if($amount > 0)
13     {
14       $this->_balance += $amount;
15       return true;
16     }
17     else   return false;
18   }
19   public function credit($amount)
20   {
21     if($amount > 0)
22     {
23       $this->_balance -= $amount;
24       return true;
25     }
26     else   return false;
27   }
28   public function printInfo()
29   {
30     echo "Account #{$this->_no}: \${$this->_balance}\n";
31   }
32   public function getBalance() { return $this->_balance; }
33   public function getNumber()  { return $this->_no;      }
34
35
36 }
```

```
class Demo_Security
{
    private static $_securityAspect = null;
    public static function disableLargeMoneyOperations()
    {
        //Match Credit and Debit functions of Demo_Account
        $h = new Aspect_Hook(array(
                new Aspect_JoinPoint_Regex_Function('/debit/'),
                new Aspect_JoinPoint_Regex_Function('/credit/')
        ), array('Demo_Account'));
        self::$_securityAspect = new Aspect_Function($h,
        'Demo_Security::blockMoneyTransfer');
        Aspect_Handler::wrapAroundBefore(self::$_securityAspect);
    }
    public static function enableLargeMoneyOperations()
    {
        if(self::$_securityAspect === null) return;
        Aspect_Handler::unwrapAroundBefore (self::$_securityAspect);
        self::$_securityAspect = null;
    }
    public static function blockMoneyTransfer($meta, $amount)
    {
        if($amount > 5000)
            return new Aspect_Result_Stop (false);
    }
}
```

```php
class Demo_Promotions
{
    private static $_promoAspect = null;

    public static function enablePromotion()
    {
        //Bind to function debit of Demo_Account
        $h = new Aspect_Hook(array(
                new Aspect_JoinPoint_Regex_Function('/debit/'),
            ), array('Demo_Account'));

        self::$_promoAspect = new Aspect_Function($h, 'Demo_Promotions::debitPromotion');
        Aspect_Handler::wrapAroundBefore(self::$_promoAspect);
    }

    public static function disablePromotion()
    {
        if(self::$_promoAspect === null) return;

        Aspect_Handler::unwrapAroundBefore (self::$_promoAspect);
        self::$_promoAspect = null;
    }
    // 10% More Cash for debit users
    public static function debitPromotion($meta, $amount)
    {
        //Replace amount by amount * 1.10
        return new Aspect_Result_Replace(array($amount * 1.10));
    }
}
```

```
1  class Demo_Logging
2  {
3    private static $_logAspect = array(null, null);
4    //Dynamically Enable Logging
5    public static function enableLogging()
6    {
7      //Create hook on everything
8      $h = new Aspect_Hook(array(
9              new Aspect_JoinPoint_True
10           ));
11
12     self::$_logAspect[0] = new Aspect_Function($h, 'Demo_Logging::logBefore');
13     self::$_logAspect[1] = new Aspect_Function($h, 'Demo_Logging::logAfter');
14     Aspect_Handler::wrapBefore(self::$_logAspect[0]);
15     Aspect_Handler::wrapAfter(self::$_logAspect[1]);
16   }
17   //Dynamically Disable Logging
18   public static function disableLogging()
19   {
20     if(self::$_logAspect === array(null, null)) return;
21     Aspect_Handler::unwrapBefore (self::$_logAspect[0]);
22     Aspect_Handler::unwrapAfter(self::$_logAspect[1]);
23     self::$_logAspect = array(null, null);
24   }
25   //Dynamically Enable Logging
26   public static function logBefore($meta) { self::_log($meta, "[Before] > ", func_get_args()); }
27   public static function logAfter($meta)  { self::_log($meta, "[After] > ", func_get_args()); }
28
29   //Our log Formatting function
30   private static function _log($meta, $prefix, $args2)
31   {
32     //Data provided by the Meta
33     $func   = $meta->getFunction();
34     $args   = $meta->getArgs();
35     $class  = $meta->getClass();
36     $return  = $meta->getReturn();
```

```
37      //Remove Meta from args
38      array_shift($args2);
39
40      echo "$prefix $class.$func(".implode(", ", $args2).") orig-args: (".implode(", ", $args).") ->
41      if(is_object($return))
42        if($return instanceof Aspect_Wrapper)
43          echo "{Wrapped} ". $return->__get_class();
44        else
45          echo get_class($return);
46      else
47      {
48        if($return === null) $return = "Null";
49        echo $return;
50      }
51
52      echo "\n";
53    }
54  }
```

If we use this file to run everything:

```
1  //Enable loading of classes under Folder/File.php if class-name is Folder_File
2  function __autoload($class)
3  {
4      require_once str_replace("_", "/", $class) . ".php";
5  }
6
7  //Enable the Aspects (subscribe)
8  Demo_Logging::enableLogging();
9  Demo_Promotions::enablePromotion();
10 Demo_Security::disableLargeMoneyOperations();
11
12 //Load a Demo_Account, the logger should match newInstance and wrapInstance
13 $a = Aspect_Wrapper::load('Demo_Account', 32, 200);
14 $a->debit(100);
15
16 //Big Money Operations should return false
17 var_dump($a->debit(10000));
18 var_dump($a->credit(6000));
19
20 echo "\n";
21 $a->printInfo();  //No Changes in Account for big
22 echo "\n";
23
24 //Disable one aspect (unsubscribe)
25 Demo_Security::enableLargeMoneyOperations();
26 var_dump($a->debit(10000));
27 var_dump($a->credit(6000));
28
29 echo "\n";
30 $a->printInfo();  //Changes in account for big
31 echo "\n";
```

We get the following output:

```
[Before] >  Demo_Account.newInstance(32, 200) orig-args: (32, 200) ->  Null
[After] >  Demo_Account.newInstance(32, 200) orig-args: (32, 200) ->  Demo_Account
[Before] >  Demo_Account.wrapInstance(32, 200) orig-args: (32, 200) ->  Null
[After] >  Demo_Account.wrapInstance(32, 200) orig-args: (32, 200) ->  {Wrapped} Demo_Account
[Before] >  Demo_Account.debit(100) orig-args: (100) ->  Null
[After] >  Demo_Account.debit(110) orig-args: (100) ->  1
[Before] >  Demo_Account.debit(10000) orig-args: (10000) ->  Null
[After] >  Demo_Account.debit(11000) orig-args: (10000) ->
bool(false)
[Before] >  Demo_Account.credit(6000) orig-args: (6000) ->  Null
[After] >  Demo_Account.credit(6000) orig-args: (6000) ->
bool(false)

[Before] >  Demo_Account.printInfo() orig-args: () ->  Null
Account #32: $310
[After] >  Demo_Account.printInfo() orig-args: () ->  Null

[Before] >  Demo_Account.debit(10000) orig-args: (10000) ->  Null
[After] >  Demo_Account.debit(11000) orig-args: (10000) ->  1
bool(true)
[Before] >  Demo_Account.credit(6000) orig-args: (6000) ->  Null
[After] >  Demo_Account.credit(6000) orig-args: (6000) ->  1
bool(true)

[Before] >  Demo_Account.printInfo() orig-args: () ->  Null
Account #32: $5310
[After] >  Demo_Account.printInfo() orig-args: () ->  Null
```