PONTIFICIA UNIVERSIDAD CATOLICA DE CHILE
ESCUELA DE INGENIERIA

ADVANTAGES AND DISADVANTAGES OF ASPECT ORIENTED DESIGN IN AN ENTERPRISE ENVIRONMENT

PELAYO J. BESA

Thesis submitted to the Office of Research and Graduate Studies in partial fulfillment of the requirements for the Degree of Master of Science in Engineering.

Advisor:
YADRAN ETEROVIC

Santiago de Chile, (July, 2011)

PONTIFICIA UNIVERSIDAD CATOLICA DE CHILE

ESCUELA DE INGENIERIA

ADVANTAGES AND DISADVANTAGES OF ASPECT ORIENTED DESIGN IN AN ENTERPRISE ENVIRONMENT

PELAYO JOSÉ BESA VIAL

Thesis submitted to the Office of Research and Graduate Studies in partial fulfillment of the requirements for the Degree of Master of Science in Engineering.

Santiago de Chile, (July, 2011)

This thesis is dedicated to my parents.

ACKNOWLEDGEMENTS

I would like to thank Yadran Eterovic for his support and pacience during this endeavour.

INDEX

INDEX OF TABLES

INDEX OF FIGURES

RESUMEN

El diseño orientado a aspectos permite separar conceptos transversales que surgen en un diseño orientado a objetos y que no son adecuadamente modularizados por este. A pesar de su gran avance y sofisticación teórica, su aplicación en proyectos industriales sigue baja. En este trabajo se buscaron algunas razones de por qué esto no se ha dado.

Para ello, primero se diseñó un sistema en una empresa real, consistente en una plataforma distribuida de captura de datos. Luego, se extrajo y se determinó una serie de aspectos, rediseñándose la aplicación para incluirlos. Por último, se compararon los dos diseños, tanto desde el enfoque técnico (cómo mejoró el sistema en adquirir buenas caracteristicas de diseño), como desde la organización y el proceso de desarrollo de software.

Se concluyó que, a pesar de que la orientación a aspectos presenta grandes ventajas en términos de facilitar la reusabilidad y evolución de los módulos, también manifiesta algunos problemas, como posibles impedimentos para el trabajo en equipos y costos relacionados con entrenamiento.

Palabras Claves: orientación a aspectos, ingeniería de software, diseño de software, desarrollo de software

ABSTRACT

Aspect oriented design aims at separating crosscutting concerns that aren't adequately modularized by object oriented design. Despite the important advances made in its theory its application in an enterprise setting remains low. In this work searched for reasons why this hasn't occurred.

Towards this goal, we first designed a software system in a real company, a distributed data capture platform. We then extracted and determined a series of aspects and redesigned the application to include them. Finally, we compared these two designs from a technical standpoint (how the system was enhanced by obtaining better design characteristics) and from a organization and development process standpoint.

We concluded that, although aspect oriented design presents important advantages by facilitating reuse and evolution of modules, it also presents some problems, such as certain impediments to teamwork and costs related to training.

Keywords: aspect orientation, software engineering, software design, software development

1. INTRODUCTION

When designing a software system using a purely object oriented (OO) approach we are often left with concerns that appear throughout the system and cannot be encapsulated in a cohesive module. Aspect oriented design (AOD) is a design paradigm that aims at separating and modularizing these crosscutting concerns.

AOD achieves its purpose through the use of aspects. These are units of code composed of a join-point and an advice. The join-point defines a point in the OO system's execution in which the advice —a set of statements— should be executed. This allows the system to execute without knowledge of the aspect's existence, which facilitates more cohesive and less coupled modules. A more detailed description of AOD can be found in (Kiczales et al, 1997).

Aspect orientation is not new. Over the past 10 years, aspect-oriented programming and design, and aspect mining have made important advances as described in textbooks and papers, presented at conferences or published in journals. In spite of this (Weise et al, 2007) observed the fact that then, 2007, AOD hadn't disseminated as successfully as it should have. More recently, (Laddad, 2009) notes that AOD has followed a standard Gartner Hype Cycle as determined by (Fenn, 1995) for new technology. Laddad determines that AOD is currently on the slope of enlightenment. In this phase users begin to use and experiment with the technology. During this period the technology must grow in acceptance or die out.

It is in this light that we present this work. Its main objective is to show AOD in an actual industry setting. Although AOD has recently begun to penetrate the industry, few case studies have been presented. Most studies focus on theory and analyze simple examples such as logging and tracing instead of real-world applications. For project managers and developers to realize the benefits that AOD brings to a project, actual case studies that show its costs and benefits must be published.

In this paper we show the advantages and disadvantages of implementing an AOD solution in a real-world setting, both directly on the application but also the impact on the development process.

We start with an object-oriented design of a real application, in which we apply all applicable design patterns. This application strikes a balance in complexity to both allow us to apply AOD in non-trivial ways and determine the impact it has on the project as a whole, while being simple enough to isolate the costs and benefits to be assured that they are a product of AOD and not the complexity of the program. Then, we analyze the design to find design features that decrease the application classes' cohesion or that increase their coupling. Next, we redesign the application trying to separate those design features into aspects. Finally, we compare the resulting aspect-oriented design with the original object-oriented one and analyze the benefits that this brings.

2. RELATED WORK

The reasons behind why AOD hasn't been widely adopted are still being studied. In (Wiese et al, 2007) reviewed the steps that must be taken in order to convince the industry to use AOD. These steps included the availability of tools to facilitate the use of AOD, disseminating the virtues of AOD and presenting positive case studies and successful implementations of important applications. Since then, AOD tools have been implemented for a multitude of programming systems and papers have continued studying and refining AOD.

In spite of these advances AOP was still not widely adopted in 2009, when (Muñoz et al, 2009) analyzed 38 open source projects to determine how AOP was being utilized and concluded that AOP was implemented in the projects for basic scenarios only. They presented various possible reasons for this situation, such as a lack of knowledge and skill in extracting cross-cutting concerns, lack of trust in AOP because of security concerns, and the possibility that the tools (AspectJ in this study) were not flexible enough. Unfortunately, since the study focused on open source projects, their results can't be directly applied to the enterprise environment.

(Pohl et al, 2008), from SAP Research, present their findings when studying a SAP environment. They overview a few non-trivial aspects and determine difficulties they observed when developing and implementing AOD in their company. Our work confirms how some of the observed difficulties, such as the maintenance issues created by obliviousness and the cost of training developers under a new paradigm, present new challenges to promote the use of AOD in an industry setting and also reviews the advantages AOD delivers to a project.

The following studies focus on some of the specific issues that impede AOD adoption and try to design and develop solutions to address them. One task that these papers consider important towards the adoption of AOD in the industry is the presentation of

AOD success-stories, stories in which AOD has improved the software development project.

Throughout the years one of the areas in which papers have been published announcing successful implementations has been in software development frameworks. This appears to be one of the areas where AOD has been more widely employed in an enterprise setting. For example, (Yao et al, 2005), (Urban et al, 2010), and (Mortensen et al, 2006) present frameworks that implement aspects and also facilitate AOD development. They present interesting use case scenarios and success stories, but do not address the impacts of AOD on a project. (Yao et al, 2005) is a C++ AOD implementation taking advantage of C++ language constructs for said implementation, allowing the use of generic C++ compilers and systems. A similar work can be found in (Urban et al, 2010) which presents a case study on PUMA, a framework to develop applications that analyze and, optionally, transform C or C++ code. Although they present a successful case, they do not look into the impact of using AOD on a project’s team or company. When comparing designs to demonstrate an improvement on the side of AOD, they focus solely on empiric OO code based metrics and not on AOD ones. These same problems are present in (Mortensen et al, 2006), which investigates the use of AOP specifically in C++ frameworks. They report a reduction of code and an improvement in modularity although their analysis is highly code driven and focuses on code refactoring more than initial design. Although in this publication they briefly mention the obliviousness issue, the authors find that it does not apply to them and don’t go into more details about it.

Another interesting area is proving of the usefulness of AOP in terms of metrics. (Ortiz et al, 2008), (Cui et al, 2009), and (Narendra et al, 2007) present case studies to validate the impact of AOP on the software. To compare OOP and AOD they focus mostly on standard OO code metrics such as LOCs and NOCs. An analysis on the separation of concern or a design perspective, such as the one shown in this paper, is missing. Another missing analysis is the impact on business and team development, which we emphasize in this work. (Cottenier et al, 2007), a project in Motorola focusing mostly on design in a business context, reach the same conclusions by simple inspection, but unfortunately, an analysis on the impact on development processes is missing here as well.

How AOP reacts to changes in real-world projects is also a subject that requires investigation before the industry is convinced to adopt it. (Greenwood et al, 2007) and (Nunes et al, 2008) use a real test case and on it apply different scenarios that a project could face during a normal development cycle. These papers utilize metrics imported from OO analysis such as LOCs and NOCs and new metrics specifically developed to determine separation of concerns. In (Greenwood et al, 2007) the authors present the “ripple effect” of making changes on the code and note how this affects different modules. The issue of obliviousness appears but they do not recognize it as such. In (Nunes et al, 2008) the authors conclude that AOP has better modularization than OO design but a larger code size which, they determine, would increase the difficulty of understanding the module.

3. DESCRIPTION OF THE APPLICATION’S ORIGINAL OO DESIGN

In this section we describe the OO design of the application used as a basis for this work. First we describe its structure in general and then we focus on some of the components to which we will refer to in later sections. This application, a distributed data capture platform, is both complex enough to present non-trivial aspects and simple enough to allow us to associate arising issues that occur to vices caused by cross-cutting concerns and not to the complexity of the problem.

The platform must be able to connect to certain web sites on the Internet, download the corresponding web page, analyze it to extract specific information, and store the information for later retrieval and processing. This job must be performed thousands of times per day, which adds complexity in terms of bandwidth, processing, and interaction with the sites. These interactions pose the main architectural challenge for the platform. Since it can interact thousands of times a day with a single site, care must be taken so that the sites do not identify the platform as a non-human user; otherwise, the chances of the platform being blocked and denied access to the web pages are high. To solve this problem we can use an IP cool down: a waiting period between requests at the cost of fewer requests made by the platform to that website through the same IP. To counter this reduction in the number of requests per unit of time we can simply add more IPs. This is where the distributed nature of the platform comes into play.

The platform’s high-level design follows a standard producer/consumer architectural pattern, with a dispatcher producing batches of tasks, which the workers must perform. Once a worker performs a batch of tasks, it sends the results back to the dispatcher and requests more tasks.

Following the flow in [Figure 1] and [Figure 2], we can see that the platform’s customer submits an order to the Order Handler, which splits it into several tasks, which are handed over to the Task Queue Manager. Any free worker will request tasks from the dispatcher, which will pack a batch of these and send them to the worker. The worker will capture and process the data, according to the tasks, and return the results to the Receiver. Finally, the Receiver will pack all the results for one customer in a report, and will make the package available for the customer to request.

In the following sections we will provide a more in-depth look at some of the components of the application.

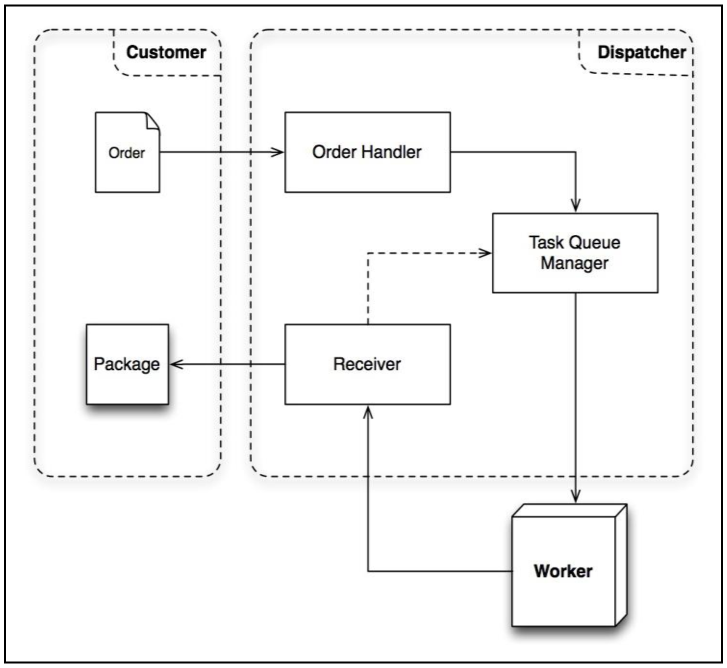

Figure 1: The dispatcher’s components and their relationships with the customer and worker.

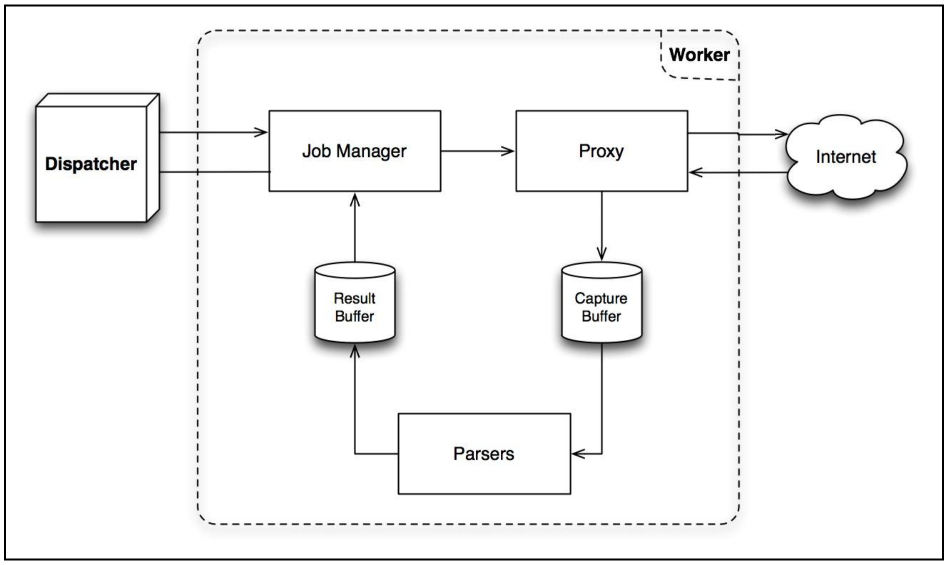

Figure 2: The worker's components and their relationships with the dispatcher and the internet.

The dispatcher's order handler component

The Order Handler is in charge of receiving an order and converting it into the appropriate tasks. The order is composed of a header, which contains the parameters necessary to process it, and a payload, which will be processed to create tasks.

[Figure 3] is a class diagram for the Order Handler. The Order Handler receives an order at the OrderReceiver, which passes it off to the OrderProcessor. This class splits the order into its header and its payload and sends both to the PluginHandler. This allows the PluginHandler to select the correct plug-in to process the payload. The PluginHandler is a factory that creates an instance of the correct plug-in, parses the payload with said plug-in, and returns a number of tasks. These tasks are returned to the OrderProcessor who finally forwards the tasks off to the TaskReceiverHandler in the Task Queue Manager.

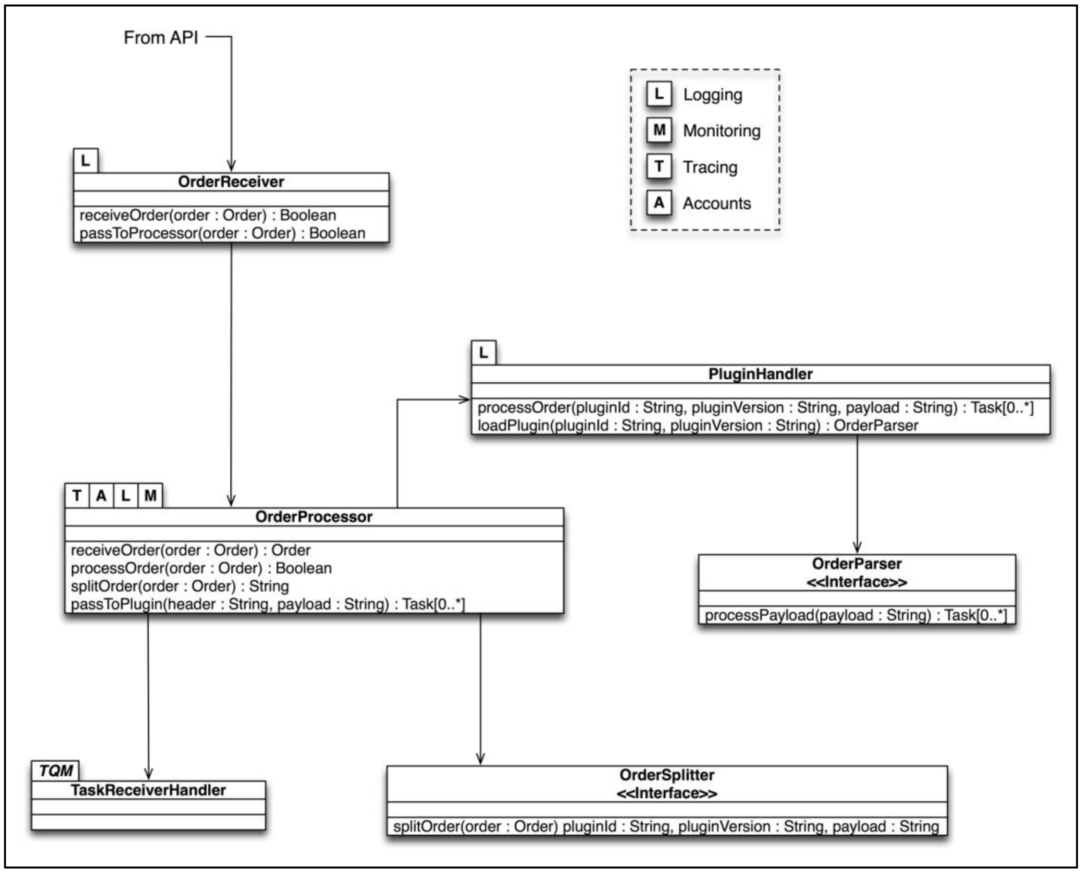
From API
L Logging
M Monitoring
T Tracing
A Accounts
L
OrderReceiver
receiveOrder(order : Order) : Boolean
passToProcessor(order : Order) : Boolean
L
PluginHandler
processOrder(pluginId : String, pluginVersion : String, payload : String) : Task[0..*]
loadPlugin(pluginId : String, pluginVersion : String) : OrderParser
T A L M
OrderProcessor
receiveOrder(order : Order) : Order
processOrder(order : Order) : Boolean
splitOrder(order : Order) : String
passToPlugin(header : String, payload : String) : Task[0..*]
OrderParser
<<Interface>>
processPayload(payload : String) : Task[0..*]
TQM
TaskReceiverHandler
OrderSplitter
<<Interface>>
splitOrder(order : Order) pluginId : String, pluginVersion : String, payload : String

Figure 3: Order Handler class diagram – responsible for processing orders that come in from the application's customer API and splitting them into individual tasks.

To keep these figures simple, we have excluded from the diagram the following modules: Logging, Monitoring, Tracing, and Accounting. We can see the specific classes where each of these mechanisms intervenes by taking notice of the letters representing each one. We will review the Accounting mechanism with greater depth ahead.

This design has beneficial properties. Firstly and on a more general note, having one tight module – isolated from the rest of the application, with only one "entry" point and one "return" – allows for separation and reusability. With very few modifications we may use this module in any other application where we might need to process different types of data and pass on that data to another module.

Secondly, the use of a plug-in system allows us to easily develop plug-ins for different types of orders without even stopping the application's execution. Another benefit is that having the splitter in its own class gives us the ability to redesign the order system by modifying only one specific class. Finally, having the receiver separated from other classes allows us to modify the reception mechanism and abstract it from any processing the may be done on the order. If we want to switch the reception mechanism or modify it in any way then we only need to modify one class.

All these design benefits give us lower coupling between classes and higher cohesion, which translates into more reusable and better evolving modules.

The worker's task capturing proxy

The Proxy [Figure 4] is in charge of receiving a task, capturing the web page indicated by the task's request through one of its many IPs, and storing the web page on a buffer for later processing.

Internally, the TaskReceiver receives a task, creates a TaskProcessorThread and hands off the task to it. The TaskProcessorThread will extract the request from the task and hands it off to the IPManager. The IPManager is a singleton that controls a pool of IPs, each one with a cool down period for each domain. To maintain the Cooldown class separated from the IP class we use a wrapper class that merges the two, leaving the IPManager to actually manage a pool of IPWrappers. The IPManager selects an IP that has already been cooled down for the specific domain in the request and makes the request, returning the captured page to the TaskProcessorThread that made it.

That TaskProcessorThread then passes the resulting capture on to the CaptureBuffer and ends its execution. The CaptureBuffer is a singleton in charge of receiving the captures, storing them and keeping track of which captures have already been processed by the parsers. To maintain this information and to separate the Capture class from the "processed/unprocessed" state we use a wrapper, joining the Capture with its boolean state.

This design persues a series of benefits. Requests on the proxy are parallelized. This allows the proxy to maximize the amount of requests and not be hindered by connection issues. If one request is slow the impact on the rest of the captures should be minimal. This same reason explains the use of a buffer, to separate the capture and the processing process from each other. If the module that processes the captured web page is slow for any reason, this should not impact the capture process.

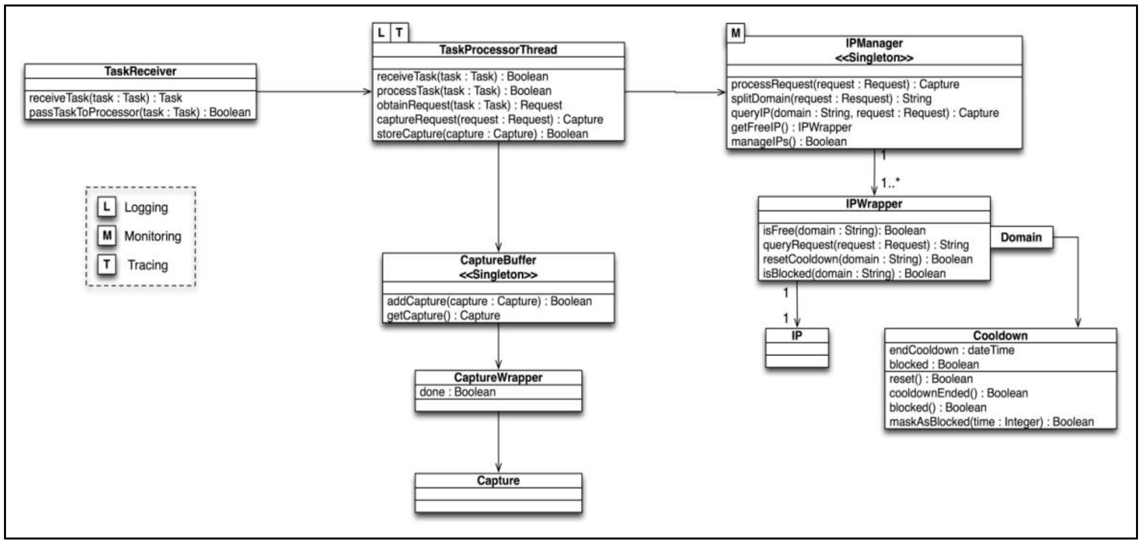
TaskReceiver
receiveTask(task : Task) : Task
passTaskToProcessor(task : Task) : Boolean
L
T
TaskProcessorThread
receiveTask(task : Task) : Boolean
processTask(task : Task) : Boolean
obtainRequest(task : Task) : Request
captureRequest(request : Request) : Capture
storeCapture(capture : Capture) : Boolean
M
IPManager
<<Singleton>>
processRequest(request : Request) : Capture
splitDomain(request : Resquest) : String
queryIP(domain : String, request : Request) : Capture
getFreeIP() : IPWrapper
manageIPs() : Boolean
1
1..*
L Logging
M Monitoring
T Tracing
IPWrapper
isFree(domain : String): Boolean
queryRequest(request : Request) : String
resetCooldown(domain : String) : Boolean
isBlocked(domain : String) : Boolean
Domain
1
1
IP
Cooldown
endCooldown : dateTime
blocked : Boolean
reset() : Boolean
cooldownEnded() : Boolean
blocked() : Boolean
maskAsBlocked(time : Integer) : Boolean
CaptureBuffer
<<Singleton>>
addCapture(capture : Capture) : Boolean
getCapture() : Capture
CaptureWrapper
done : Boolean
Capture

Figure 4: Proxy class diagram – responsible for receiving tasks, capturing their data through the IPManager and storing the captured data on the capture buffer.

Taking a closer look at this design we can detect a series of key features that fulfill the objectives that developers look for in a good OO design. Firstly, just as we observed in the Order Handler, the reception of the tasks is separated from the processing. This allows us to separate components, for example by placing the proxy on its own machine or reused on a different project altogether. Secondly, the use of wrappers, both for IPs and Captures, ensure that they can be reused in other projects. They also allow us to modify behaviors without editing the core components. For example, if in a future requirement there is the need for another condition for an IP to be marked as available to be used, developers would not need to modify the IP or the Cooldown classes. The only necessary modification would be to add the requirement to the IPWrapper. Finally the IP and Capture classes obtain a greater cohesion since they are now separated from their states, making them more easily reusable and easier to evolve.

The worker's processing parser pool

The Parser Pool [Figure 5] is in charge of getting a capture from the CaptureBuffer, processing it with the appropriate plug-in and writing the result onto the ResultsBuffer. It does this through the use of multiple, independently threaded RequestHandlers.

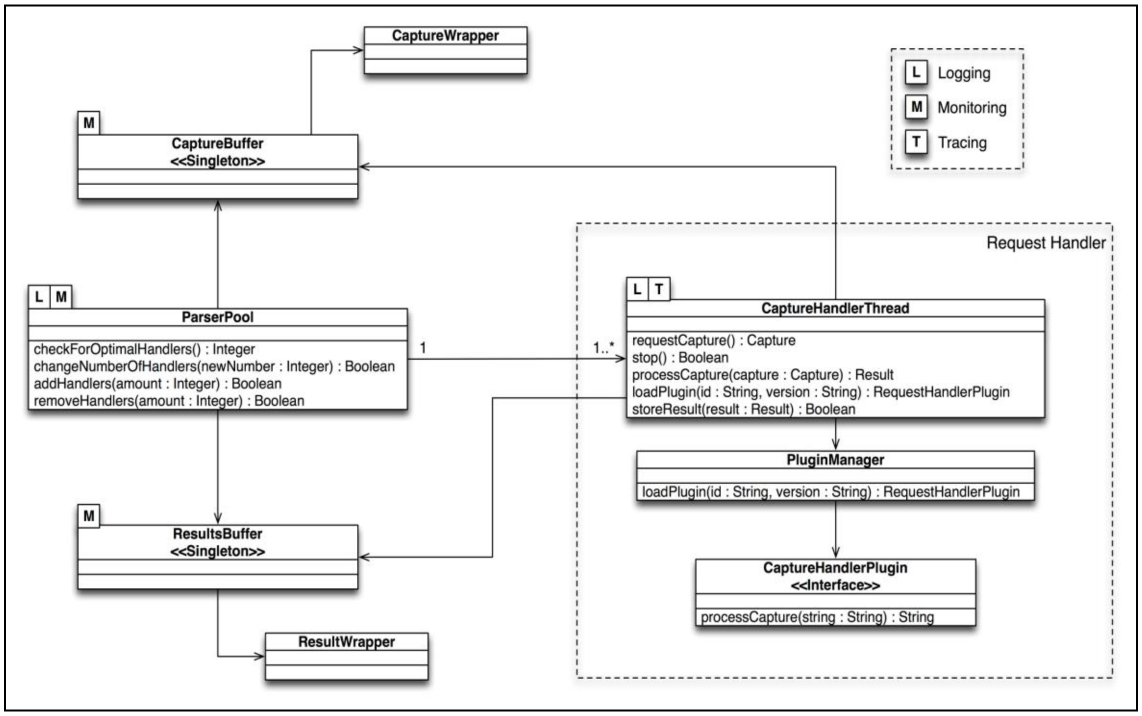
CaptureWrapper
M
CaptureBuffer
<<Singleton>>
L Logging
M Monitoring
T Tracing
Request Handler
L M
ParserPool
checkForOptimalHandlers() : Integer
changeNumberOfHandlers(newNumber : Integer) : Boolean
addHandlers(amount : Integer) : Boolean
removeHandlers(amount : Integer) : Boolean
1
1..*
L T
CaptureHandlerThread
requestCapture() : Capture
stop() : Boolean
processCapture(capture : Capture) : Result
loadPlugin(id : String, version : String) : RequestHandlerPlugin
storeResult(result : Result) : Boolean
PluginManager
loadPlugin(id : String, version : String) : RequestHandlerPlugin
M
ResultsBuffer
<<Singleton>>
CaptureHandlerPlugin
<<Interface>>
processCapture(string : String) : String
ResultWrapper

Figure 5: Parser Pool class diagram – responsible for parsing the data in the capture buffer and storing the resulting information in the results buffer.

Each RequestHandler operates on its own thread. The amount of threads is constantly being optimized according to the amount of unprocessed captures, CPU workload and ParserPool trends.

This design provides various benefits. Firstly, the use of plug-ins allows modifying the parsing process without stopping the system. Similar to the Order Handler, if a new requirement appears, for example the need to parse a page differently, writing a new plug-in and, using the integrated versioning system, deploying it without stopping the capture and parsing process improves the application’s speed. Secondly, the pool of CaptureHandlerThreads permits fast and independent processing. Using independent threads allows the captures to be processed without being delayed if one process is slowed down. Using a thread pool allows the ParserPool to manage the threads according to environmental and external variables. Through this, the ParserPool can add and remove threads as needed to maximize the speed with which the captures are parsed. This leads to another benefit: the use of a ResultBuffer.

The ResultBuffer separates the capture parsing with any subsequent jobs that need to be executed on the result. It manages a pool of ResultWrappers that separate the Result from any state variables that need to be associated to it.

4. CRITERIA FOR COMPARING THE OO DESIGN WITH THE AOD (RE)DESIGN

For a long time, software engineers —researchers and practitioners alike— have been studying what characterizes a well-designed application. (Eder et al, 1994) stress the importance of achieving two objectives, currently considered design principles: low coupling and high cohesion. Designs that obey these principles result in modules, components and classes that are easier to understand, change, correct and maintain.

For many years object orientation has been the predominant paradigm for designing and developing software, and studies, such as (Chidamber et al, 1994), have been made to measure the extent to which a particular object-oriented design follows the principles.

For AOD things are not as developed. Studies such as (Wampler et al, 2007), (Zhao et al, 2004) and, (Zakaria et al, 2003) focus on developing metrics for AOD, but most of these studies focus on code analysis, leaving design analysis to a secondary position.

Since AOD design is based on OO design and aims to solve problems such as crosscutting concerns and classes with high coupling or low cohesion it is safe to say that the main objectives we pursue with our AOD code are also to strive for low coupling and high cohesion while keeping a high level of understandability.

Using the Goal/Question/Metric approach by (Basili et al, 1994), (Sant' Anna et al, 2003) propose asking a series of questions to predict the maintainability, reusability and understandability of systems (and system components). In their most general form, these questions may be stated as follows:

1. How easy is it for the system to evolve?
2. How easy is it to reuse the system elements?

For both of these questions Sant' Anna et al. propose the same sub questions:

1. How easy is it to understand the system?

With this question we are asking:

- How concise is the system (number of components, attributes and methods)?
- How well are concerns localized (analysis of scattering and entanglement)?
- How high is the coupling in the system?
- How high is the cohesion in the system?

2. How flexible is the system?

With this question we are asking:

- How well are the concerns localized (analysis of scattering and entanglement)?
- How high is the coupling in the system?
- How high is the cohesion in the system?

By responding these questions we should be able to see the answers to the more global queries.

These queries are essential for motivating the use of AOD. If the system is easy to understand then its maintenance costs in time and training for new developers will be lower. If the evolution and reuse of the system and its elements is improved that effort and money can be saved in other company projects.

AOD allows for these improvements by modularizing crosscutting concerns.

If we analyze our AOD design in light of these objectives and, in the case of our questions we leave code analysis to a secondary, more speculative, position we can compare an OO design with an AOD one and determine the benefits that AOD imparts on a projects.

5. ANALYSIS OF AN ASPECT ORIENTED (RE)DESIGN OF THE APPLICATION

Our AOD approach is based on finding aspects in the original OO design. These aspects are obtained by searching for crosscutting concerns and classes with low cohesion or high coupling at the design level. In the following sections, we examine several points where AOD could be included in the original OO design. The objective in this section is to show how the AOD design fares against the OO design keeping in mind the objectives previously mentioned and commenting the additional efforts that would be necessary to maintain an AOD project over an OO design project in a "real-world" company.

Accounting in the dispatcher

When a customer sends an order, certain checks must be made; for example: Is the number of tasks sent by the customer within his/her quota? Does the customer have permission to execute the plug-ins he/she requires? Furthermore, the requests must also be logged, so the customer can be charged later. This situation repeats itself when the customer requests a package with results: Is it his/her package? Does the request comply with the customer's quota?

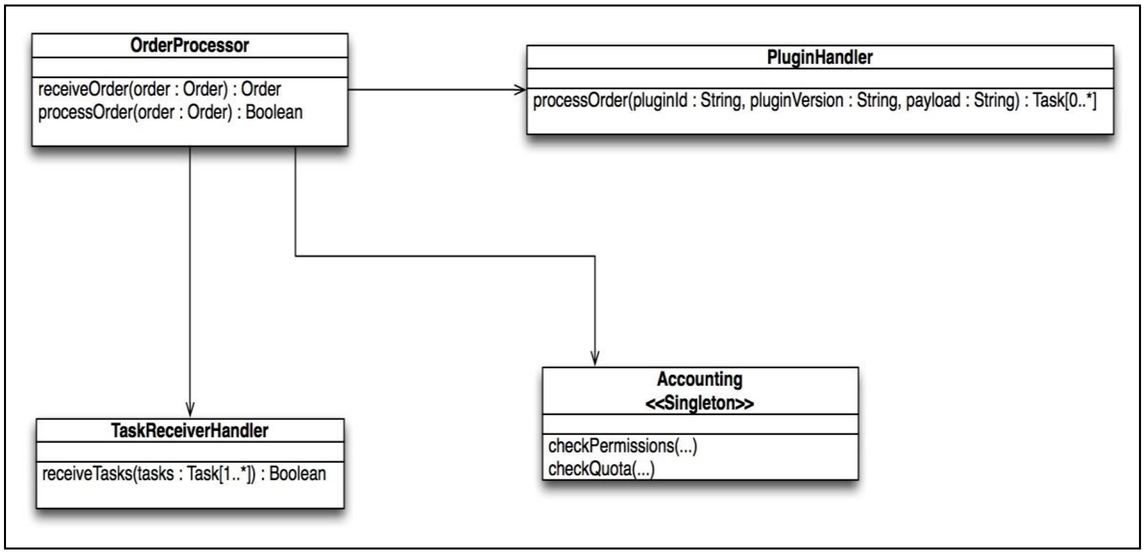
OrderProcessor
receiveOrder(order : Order) : Order
processOrder(order : Order) : Boolean
PluginHandler
processOrder(pluginId : String, pluginVersion : String, payload : String) : Task[0..*]
Accounting
<<Singleton>>
checkPermissions(...)
checkQuota(...)
TaskReceiverHandler
receiveTasks(tasks : Task[1..*]) : Boolean

Figure 6: Accounting OO design – responsible for checking permissions and quotas on receiving and processing orders through an accounting singleton.

In the OO design [Figure 6], the OrderProcessor in the dispatcher invokes the Accounting module, passing the required parameters to perform the checks. This creates two problems. First, it couples the OrderProcessor module with the Accounting module. Second, the OrderProcessor is concerned with some accounting responsibilities: calculating accounting parameters.

These problems can be solved via an Accounting aspect [Figure 7]. This aspect is in charge of obtaining all the necessary data required by the Accounting module and then invoking said module. This decouples the OrderProcessor from the Accounting module, and increases its cohesion.

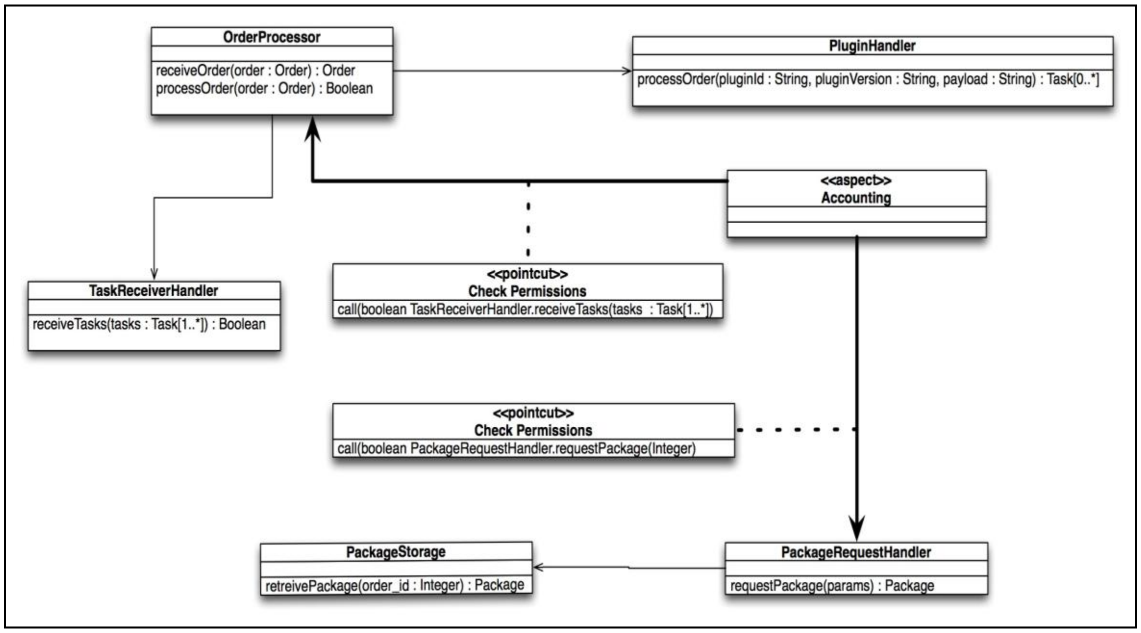
OrderProcessor
receiveOrder(order : Order) : Order
processOrder(order : Order) : Boolean
PluginHandler
processOrder(pluginId : String, pluginVersion : String, payload : String) : Task[0..*]
<<aspect>>
Accounting
TaskReceiverHandler
receiveTasks(tasks : Task[1..*]) : Boolean
<<pointcut>>
Check Permissions
call(boolean TaskReceiverHandler.receiveTasks(tasks : Task[1..*])
<<pointcut>>
Check Permissions
call(boolean PackageRequestHandler.requestPackage(Integer)
PackageStorage
retreivePackage(order_id : Integer) : Package
PackageRequestHandler
requestPackage(params) : Package

Figure 7: Accounting AOD (For AOD diagrams we will use the notation proposed in (Bustos et al, 2007)) – allows checking permissions and quotas without the order processor and storage being intervened, creating a separation between the two concerns.

When comparing the AOD and the OO design we can observe that the AOD has the benefit of having lower coupling and higher cohesion. In it, the accounting concern also presents less scattering and entanglement than in the OO design. In the latter the accounting concern is entangled in both the Accounting module and in the OrderProcessor while in the AOD the concern is localized solely in the Accounting module.

Following the criteria presented in Chapter 4, we compare reusability and evolution in both designs. From this comparison we observe that reusability [Table 1] improves in the AOD with respect to the OO design. The former benefits from the lack of entangled and scattered concerns as well as from the existence of a clear separation of the modules.

Evolution [Table 2] presents a similar situation. In the AOD, evolution of the Accounting module also benefits from a clear separation of the modules making it easy to modify and understand. Flexibility presents issues on both systems. Evolving the modules in the OO design requires dealing with the entangled and scattered code. The AOD design is free of this issue but presents a problem when modifying the OrderHandler. Because of obliviousness any modification on the OrderHandler may potentially modify the join points that the Accounting aspect uses causing the aspect's advice not to execute or to execute improperly.

Table 1 – Accounting reusability comparison: AOD provides an improvement when compared to the OO design.

	OO design	**AOD**
Understandability	Impaired by entanglement and scattering in both modules.	Elements are clearly separated and understandable.
Flexibility	OrderHandler must be cleaned up prior to reuse.	Both modules can be easily reused.

Table 2 – Accounting evolution comparison: although AOD provides separation of concerns it also introduces the problem of obliviousness.

	OO design	**AOD**
Understandability	Impaired by entanglement and scattering.	Elements are clearly separated and understandable.
Flexibility	Extensions to the modules require working around the entangled design.	Obliviousness impairs easy OrderHandler modifications.

Backing up the captured data

As we mentioned in section 3.3, whenever the proxy captures a web page's HTML, this is added to a buffer for parsing. If this parsing fails in any way and this is not noticed immediately, then a backup of that day's HTML capture is necessary for reprocessing.

Two possible ways to solve this problem in an OO environment are:

1. Have another thread/program observing the buffer periodically copying the HTML captures that have not been backed up [Figure 8]
2. Have the buffer or proxy write to a different backup buffer when it receives a captured HTML [Figure 9]

Both of these methods have drawbacks. OO design alternative 1 involves implementing a completely new application or thread in the current program. This new application will have to keep track of which HTML files have been backed up, either in the backup system or in the buffer, while coordinating with the buffer to avoid concurrency problems. OO design alternative 2 lowers cohesion and raises coupling.

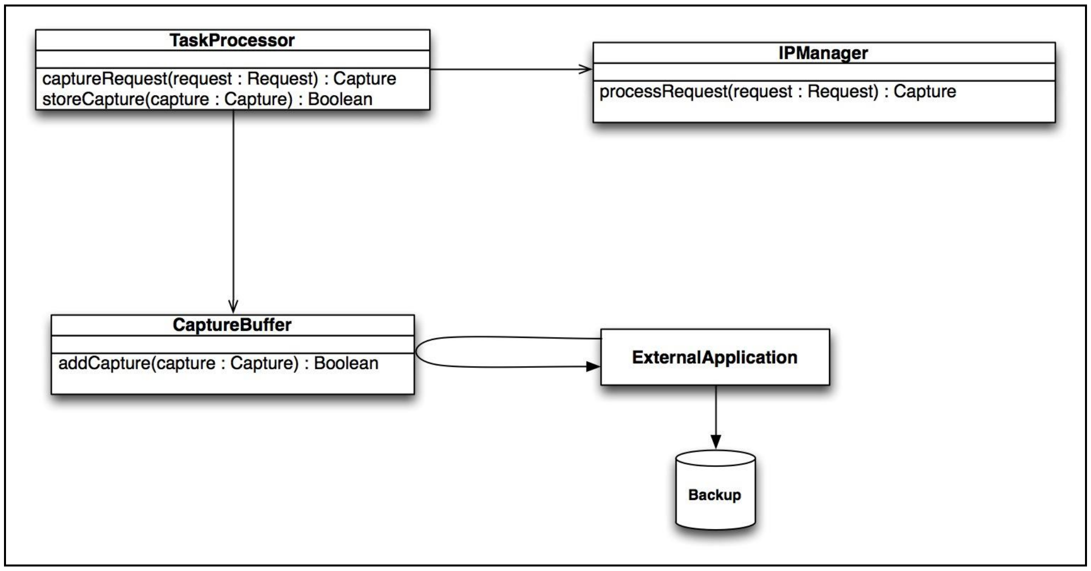
TaskProcessor
captureRequest(request : Request) : Capture
storeCapture(capture : Capture) : Boolean
IPManager
processRequest(request : Request) : Capture
CaptureBuffer
addCapture(capture : Capture) : Boolean
ExternalApplication
Backup

Figure 8: Backup OO design alternative 1 – although it requires developing an external application to backup the capture buffer it does create a separation of concerns.

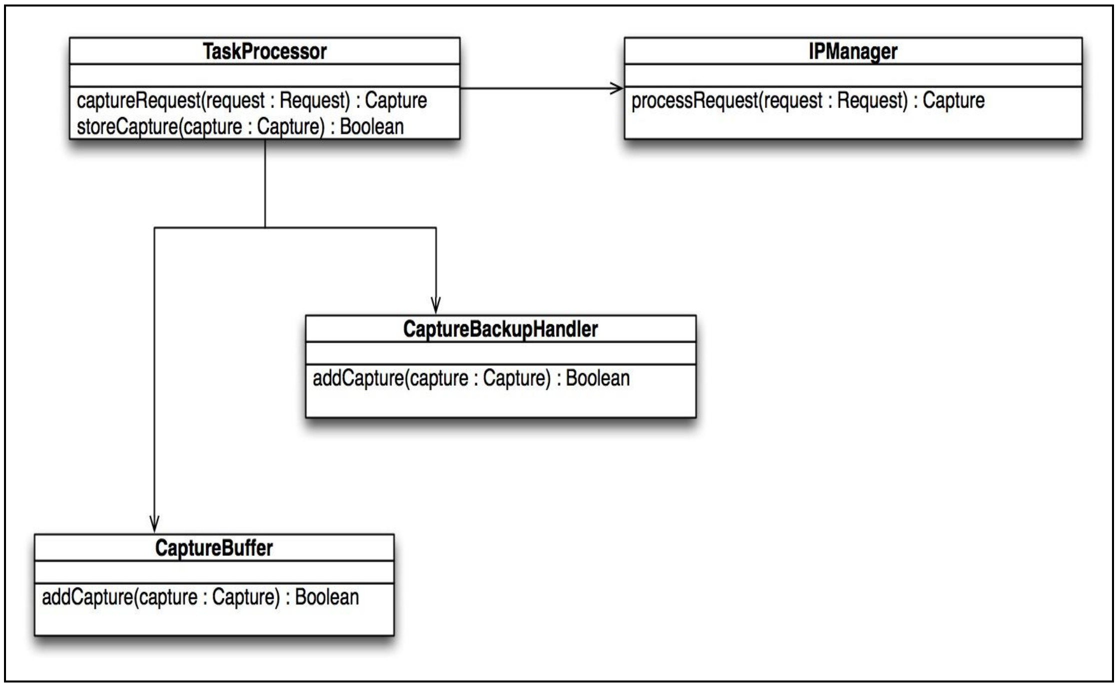
TaskProcessor
captureRequest(request : Request) : Capture
storeCapture(capture : Capture) : Boolean
IPManager
processRequest(request : Request) : Capture
CaptureBackupHandler
addCapture(capture : Capture) : Boolean
CaptureBuffer
addCapture(capture : Capture) : Boolean

Figure 9: Backup OO design alternative 2 – although it does not require the development of an external application, the backup is now entangles with the task processor's concerns.

Using AOD we can avoid these issues altogether by creating a point cut such that every time a capture is made an aspect creates a thread that backs up the HTML on the backup buffer. This eliminates the need to keep track of which captures have already been backed up potentially saving a substantial amount of time and processing power [Figure 10].

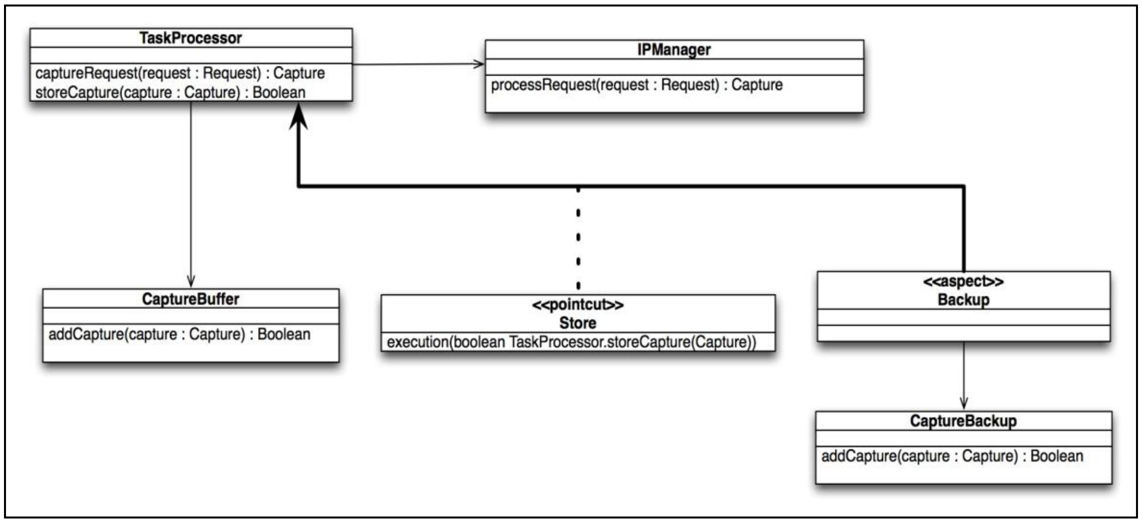
TaskProcessor
captureRequest(request : Request) : Capture
storeCapture(capture : Capture) : Boolean
IPManager
processRequest(request : Request) : Capture
CaptureBuffer
addCapture(capture : Capture) : Boolean
<<pointcut>>
Store
execution(boolean TaskProcessor.storeCapture(Capture))
<<aspect>>
Backup
CaptureBackup
addCapture(capture : Capture) : Boolean

Figure 10: Backup AOD design – provides the best of both OO designs, allowing for separated concerns without the disadvantages of a separate application.

Using the criteria from Chapter 4 we compared the reusability of all three designs [Table 3]. Contrasting the understandability and flexibility of the three designs we reach the conclusion that the AOD is easier to reuse. OO design alternative 1 has a higher complexity and the backup program is easy to reuse only if the buffers are similar. OO design alternative 2 is impaired by entanglement and scattering, suffering both in understandability and flexibility because of the fact. The AOD suffers from none of these issues since its elements are clearly separated and understandable.

Table 3 – Backup reusability comparison: the AOD design separates concerns making it easier to reuse when compared to the OO designs.

	OO design alternative 1	**OO design alternative 2**	**AOD**
Understandability	Increased complexity.	Impaired by entanglement and scattering.	Elements are clearly separated and understandable.
Flexibility	Backup program is easy to reuse only if the buffer is similar.	Requires removing the entangled and scattered elements.	Modules can be easily reused.

Comparing the evolution of the designs [Table 4] has similar results. OO design alternative 1 suffers from increased complexity, although expanding everything but the CaptureBuffer is easy and straightforward. OO design alternative 2 suffers from entanglement and scattering. Finally, the AOD's elements are clearly separated and understandable but evolution of the TaskProcessor could potentially break the aspects' point cuts.

Table 4 – Backup evolution comparison: although evolution is improved in the AOD, obliviousness can create an issue when making modifications.

	OO design alternative 1	OO design alternative 2	AOD
Understandability	Increased Complexity.	Impaired by entanglement and scattering.	Elements are clearly separated and understandable.
Flexibility	Easy to expand, except on the Capture Buffer.	Reuse of the TaskProcessor requires working around the entangled and scattered elements.	Backup only requires rewriting the point cuts. Obliviousness impairs TaskProcessor evolution.

Persistent storage throughout the application

As in most applications there is a need for persistent storage. Two of options to handle this task in the application are to use an ORM or custom code. Given that the application will need to handle millions of objects, using an ORM is not an option given its overhead. Custom code will allow us to tweak and optimize the handling of objects in ways that are simply not possible otherwise.

Also, because of the variable size of some objects such as orders and captures which can range from a few kilobytes to several megabytes, storing some of these objects completely in a relational database is impractical. To solve this, a module in charge of persistence might choose to distribute object storage throughout various systems. For example, it may store the order information in a relational database but the payload in a file system.

Another necessary feature is to control when and how much information is obtained from the database. Certain attributes of an object might be necessary at different moments of the application’s execution. At these times we only want to load necessary data. For example, in the case of the Order, we might only want to know which customer owns it. Loading the full Order, payload included, would waste resources. By obtaining and handling only the pertinent information the objects are smaller and easier to handle and there is less traffic with the storage system. An extreme example of this is “lazy-loading”, where data is not loaded until it is necessary which lowers memory usage and, if not abused, I/O or network traffic by loading less information.

Since most “non-trivial” storage situations will be similar, we will analyze the Order and OrderStorage classes to compare their designs.

In the OO design [Figure 11] the Order class calls methods from OrderStorage to obtain and save data from storage. Implementing lazy-loading in the Order class requires modifying its getters to obtain the information from storage in case they haven’t been already obtained. One problem is if the order has many attributes to obtain it will make many requests thus lowering performance. To solve this issue the application needs to load different subsets of attributes at different moments in time. This requires that the attributes or subsets of attributes be available through specific methods. Finally, the Order class needs to implement a method to save the object into storage. Lazy storage could be implemented in some classes, by intervening the class’s setter methods, but in most cases it is desirable to store the objects completely and not partially.

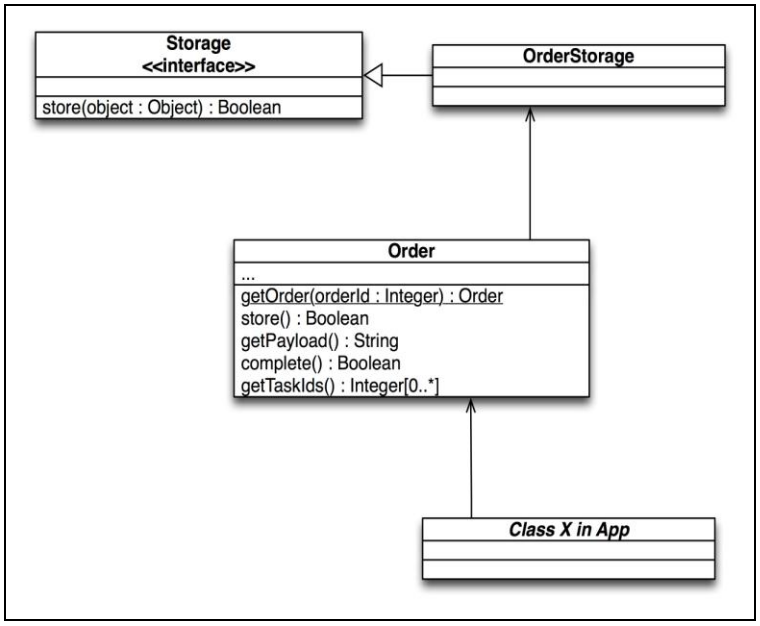

Figure 11: Order storage OO design – responsible for providing persistent storage for an Order in the application.

One big problem that this design has is that the storage concern is now entangled throughout the application. Any class that needs the order or needs to save an order must call the storage mechanism through the Order class. Because of this, changes in the saving mechanism might require extensive modifications in other classes.

Using AOD [Figure 12] we can solve some of these issues. By weaving an aspect between the Order and the storage classes we can cleanly separate the Order from the storage allowing "lazy-loading" without having the Order even know it. This allows the application developers to focus on the main flow of the program and not worry about the loading portion of the objects.

Saving an object presents a problem, though. What happens if a developer requires an object to be saved at a specific moment? For example, if the developer designing a part of the application needs an Order to be saved a certain time, how can he ensure this to happen? One solution is to set a note somewhere, external from the application, reminding the OrderStorage aspect developer that he needs the Order to be saved at said point. The obvious problem with this solution is that this is a soft requirement. The compiler will not enforce this requirement and the application may be compiled without the need being fulfilled.

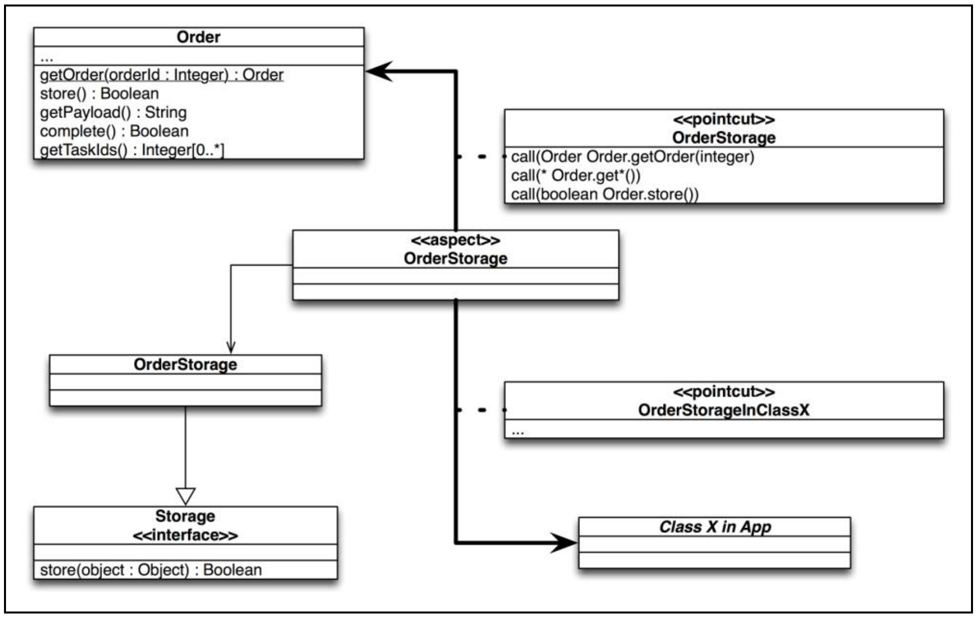

Figure 12: Order storage AOD design – allows for lazy loading and saving and also for more fine grain storage control without intervening in the application.

A hard requirement would be a method call to the Order's "save" method. In this case the OrderStorage aspect could have a pointcut on the method call and execute a "save advice". The drawback to this solution is the entanglement and scattering problems it produces. Though less than in the OO design the same negative side effects are present.

Yet another solution to this problem would be to create an "Aspect Contract" which includes all the necessary point-cuts that the aspect, in this case the OrderStorage aspect, must implement. This would allow the point-cut requirements to be defined externally from the class while permitting them to be enforced.

When comparing the designs in terms of reuse [Table 5] the AOD presents advantages except when saving objects. Each of these solutions to the storing problem (method call, “aspect contract”, etc.) present disadvantages either in entanglement or understandability.

Evolution of the modules [Table 6] also demonstrates the advantages of AOD. Although saving can be difficult to understand, the separation of concerns allows for easy development of the design.

Table 5 – Order storage reusability comparison: although AOD creates a separation of concerns it can be harder to understand.

	OO design	**AOD**
Understandability	Impeded by entanglement and scattering.	Elements are clearly separated and understandable. Saving can be difficult to understand.
Flexibility	Order needs to be cleaned up. OrderStorage might can be too dependant on the application.	Although the OrderStorage could be reused, the aspect that links it to the application would need to be rewritten.

Table 6 – Order storage ease of evolution comparison: AOD adds complexity but separates concerns, making the modules more flexible.

	OO design	AOD
Understandability	Impaired by entanglement and scattering in both modules.	Elements are clearly separated and understandable. Saving can be difficult to understand.
Flexibility	In the Order requires working around the storage code.	Both modules are flexible.

The problems observed in extracting storage into its own aspect shine a light on an important problem: when should a concern be extracted into an aspect? In this case, the extraction of Order Storage into its own aspect presents a series of difficulties. The main one is ensuring that the object is stored when necessary. Unless a solution can be devised and implemented an aspect may not be the best choice The small number of use cases where implementing it is helpful, may be in itself an important shortcoming of AOD. Since AOD requires a specific training and preparation, management and developers might see that the cost-benefit ratio is not beneficial to their company.

More possibilities for an AOD (re)design

An important feature in an industry setting is the set of Service Level Agreements. These help determining if the platform is capturing all the data it must and help in ensuring that it is being properly processed. They have a broad reaching impact both internally, it can be used for alarms and monitoring, and externally, it assures the customer that his data is being captured and handled correctly for accounting and reporting purposes.

Throughout the platform many metrics should be calculated. For example, the "correct captures" metric must be calculated in the Receiver. Or in the OrderHandler, parser data should be gathered to know how many tasks should be captured.

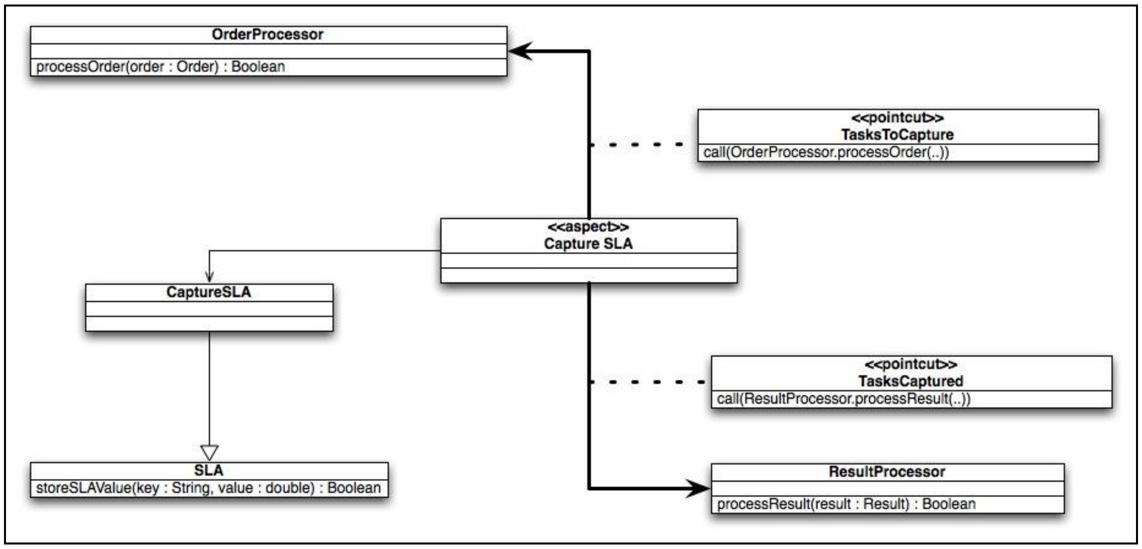
OrderProcessor
processOrder(order : Order) : Boolean
<<pointcut>>
TasksToCapture
call(OrderProcessor.processOrder(..))
<<aspect>>
Capture SLA
CaptureSLA
<<pointcut>>
TasksCaptured
call(ResultProcessor.processResult(..))
SLA
storeSLAValue(key : String, value : double) : Boolean
ResultProcessor
processResult(result : Result) : Boolean

Figure 13: AOD design of SLA capturing – this aspect allow the application to capture SLA results without the application knowing it is being intervened, although obliviousness can cause problems when intervening certain classes.

In an OO design setting these metrics could be created in an external module and called from the application lowering cohesion and raising coupling. Using AOD [Figure 13], these SLA metrics can be condensed into a single, well-defined module instead of dispersed throughout it, allowing for a more simple application with less dependencies and highly focused and reusable classes.

In this work we are taking a look at an industry perspective of AOD, where subjects such as documentation are important as they imply extra costs. In this light, AOD lowers the need of a large amount of documentation. In the OO design for SLAs for example, each metric must be carefully recorded, not just to only know what is being calculated, but also where specifically in the program. Having the SLA calls distributed throughout

many modules requires larger amounts of documentation. By having all SLA code in one place the need to specify exact call points in the documentation is eliminated. This advantage of not having concerns spread throughout the design is replicated in a large number of modules.
Another module that benefits from AOD is loading of configuration settings. Most of the modules in the platform require external setup that must be specifically loaded, be it from a flat file, a database, etc. In an OO design environment we would require a configuration module to load these configurations and in many cases this configuration module will need to be called from other classes, creating more entanglement. Using aspects we can implement a lazy loading method as mentioned in 5.3 when analyzing storage and keep the configuration module separated from the rest of the code.

This approach allows the design of modules without having the modules dealing with specific loading mechanisms. These modules may take for granted that the variables exist and are loaded, lowering coupling and raising cohesion. This facilitates evolution and reuse at the expense of a slightly lower understandability.

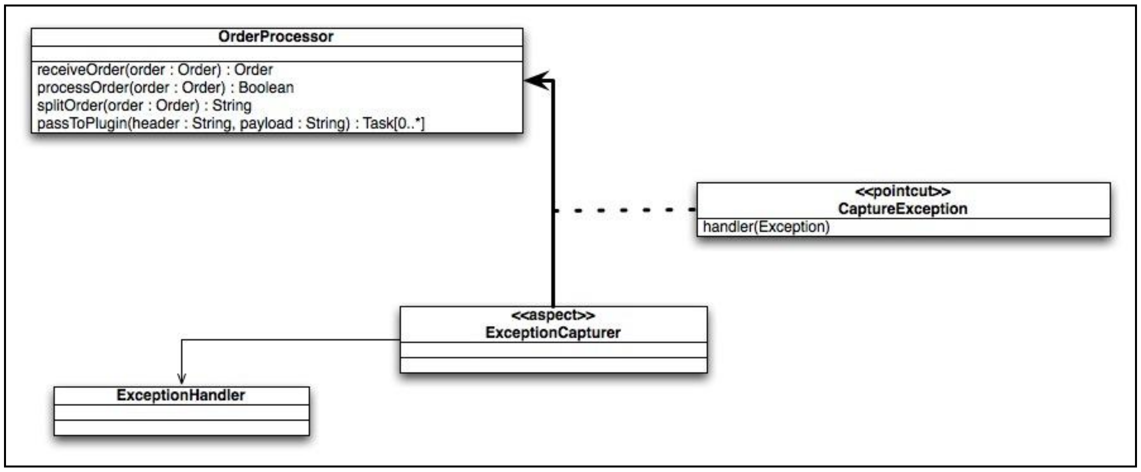
OrderProcessor
receiveOrder(order : Order) : Order
processOrder(order : Order) : Boolean
splitOrder(order : Order) : String
passToPlugin(header : String, payload : String) : Task[0..*]
<<pointcut>>
CaptureException
handler(Exception)
<<aspect>>
ExceptionCapturer
ExceptionHandler

Figure 14: AOD of exception handling in OrderProcessor – this aspect allows for varying degrees of exception handling, from broad application-wide issues to specific case-by-case control.

Another situation in which the use AOD is particularly beneficial is exception handling [Figure 14]. Every time that an unexpected error occurs, the error must be caught and appropriately handled. In an OO design solution the way to handle this is with a "try/catch" block. This may cause a series of issues. First it clutters the code. Many times the developer is faced with a large number of try/catch blocks that obscure the code and paving the way for more errors. By reacting to exceptions through the use of aspects, the code is greatly cleaned up and the main flow of the program is not entangled with what are mostly seldom called pieces of code. A second advantage is that errors are avoided. By using an appropriate aspect the developer can treat all exceptions of the same kind in one, well-defined block. A final advantage is a drastic reduction in the amount of necessary code. Since similar exception handlers can be grouped into a single aspect the developer now needs to write, maintain and document less code.

6. AOD STRENGTHS AND WEAKNESSES FOR ITS ADOPTION IN AN ENTERPRISE

In this section we will look at the problem from a software company's viewpoint. To do this we will take into account the work of (Cho et al, 2001), which presents a series of attributes that companies should take into account when deciding to adopt a technology. These variables are separated into two groups: Innovation attributes [Table 7] and Organization characteristics [Table 8].

Innovation attributes are attributes directly related to the technology. They focus on its probability of future successful adoption in the market, its current maturity, its relative benefits with the existing technology, its compatibility with the current technology and its perceived complexity.

Table 7 – Innovation attributes that are analyzed by companies when deciding on adopting a new technology: these attributes focus on the technology's future, its benefits and its complexity.

Variable	Operational definition
Expectation for market trend	The degree of expectation that a technology will be pervasively adopted in the future.
Maturity of technology	The degree to which a technology is perceived as mature in the IT industry.
Technology compatibility	The degree to which a technology is perceived as being consistent with the existing way of thinking, procedure, experiences, skill, and needs of the receivers.
Relative benefits	The degree to which a technology is perceived as being better than the current technology.
Complexity	The degree to which a technology is perceived as relatively difficult to understand and use.

Organization characteristics are related to the company that is evaluating the technology. This includes an analysis of management's receptiveness to new technologies, the companies attitude towards training its staff on new technologies, the companies satisfaction with the current technology and how engrained the current technology is within the company as represented by the developer's average years using it.

Table 8 – Organization characteristics that are analyzed by companies when deciding on adopting a new technology: these characteristics focus on the company's current status and how it faces new technologies in general.

Variable	Operational definition
Managerial innovativeness	The degree of managerial receptivity of technological change.
Intensity of new technology education	The degree of importance put on educating staff on new technology.
Satisfaction with existing technology	The degree of satisfaction with the development performance and product quality using current technology.
Average years of IT professional experience	Average years of IT professional experience of IS staff.

AOD strengths

AOD technology has been developing for already more than fifteen years. During this time it has matured both on a theoretical and practical level. This maturity and academic interest in the study of AOD apparently hasn't translated into a rapid adoption rate. According to (Laddad, 2009) AOD is on the slope of enlightenment in the Gartner Hype Cycle (Fenn, 1995). This prediction indicates that, if it is adopted as a technology, its usage will grow steadily.

AOD also benefits from a partial compatibility with the existing paradigm, OO design. Since AOD works as a “layer” that extends OO design, its barrier to entry is lowered. Regardless of this ease, AOD still forces developers to be able to analyze and extract concerns, which requires training and practice. Because of AOD’s short life, older generations of developers, who are currently working in the industry, have not been trained to use it. New developers recently entering the industry from a college or university background may also not have been exposed to AOD. This forces a company that actively seeks to implement AOD solutions to train its developers in designing and programming with this new paradigm in mind.

When analyzing the relative benefits of AOD over the current technology, OO design, we must first examine the problems in OO design that AOD looks to resolve. In the following, we will review the lessons that could be obtained from the previous section [Table 9].

Table 9 – Costs and benefits of using AOD in an industry as obtained from our analysis in section 5: when comparing OO design to AOD, developers obtain a series of benefits when using AOD but there are still issues such obliviousness that need to be resolved.

Costs	Benefits
Obliviousness issue impacts ease of evolution and requires more communication.	Easier to understand modules.
Requires training	Better reusability.
Lack of best practices/industry standards	Better evolution in many cases.
	Less errors.
	Less development time.

Two elements that introduced costs to the project are scattered and entangled concerns. These create several problems. First, they make both the design and the implementation of the application less understandable. For example, in section 5.1 we saw how the Accounting module was related to the OrderProcessor. In an OO setting, the developers of the OrderProcessor module would have to understand sections of the Accounting module and how it relates to them. Instead of focusing on just the concern of processing the order to extract tasks, the developers will have to understand the variables and calls required by the accounting concern.

Evolution and reusability suffer the same problem. To reuse modules they must be cleaned up to eliminate unnecessary concerns. When evolving a scattered or entangled concern problems may appear in unexpected sections. These two tasks can be difficult to do and might require large amounts of time and effort. For example, in section 5.2 we examined backing up the information captured in the TaskProcessor. In the second proposed design, to reuse the TaskProcessor in another project, developers must now clean up calls, references and variables that are required by the Backup module. These may include error handling, data preparation, among others. When evolving the TaskProcessor the entangled backup concern interferes. The code that required clean up to be able to reuse the module interferes and must be worked around. Evolving the backup module also might create problems. For example, if the backup team changes an error that the backup module throws when failing a connection the TaskProcessor team must now update their module to correctly handle the modified error. These modifications may involve many different modules and may be difficult and lengthy to implement. All of these problems create a burden on developers which impact on development times and costs.

These problems can be countered with more documentation, which should explain module interaction with greater detail. For example, understandability in the OrderProcessor may be improved if the Accounting module contains exactly what it requires from the OrderProcessor and what behaviors might be expected. The same occurs with the evolution and reusability. If the backup module is correctly documented the developers in charge of the TaskProcessor will know how to clean it up and how to use the backup module. Documenting a simple, compact concern with low variability may be a simple task, but if this concern is scattered across multiple modules the time taken documenting changes can be important.

In spite of this, this type of static documentation has an important shortcoming: it has a slow reaction to change. To counter a constantly changing project channels of communication and alerts are required. When the backup team changes the error definition in the Backup module the TaskProcessor team, and any other team that depends on it, must be alerted in some fashion. This raises the cost of the project, adding an entire structure to support interactions on teams that should be dealing with separate concerns.

OO design separates and simplifies large portions of a project but some cross-cutting concerns still remain, increasing complexity. AOD provides a series of benefits by allowing developers to simplify the design by extracting crosscutting concerns into its own modules.

One such benefit is that AOD increased understandability by reducing crosscutting concerns and isolating, for the most part, these concerns into separate modules. For example, the Accounting aspect extracts the task of reviewing permissions and quotas from the OrderProcessor, simplifying both modules by condensing their concerns. This results in lowering the amount of documentation necessary and also the required communication between teams, if separate teams develop these modules. This ultimately lowers design, development and debugging time and therefore costs.

The separation and decoupling of modules, which provides better understandability, also provide better reusability and easier evolution. With higher reusability companies can create a library of modules that can be easily reused in different projects. In an OO setting many of the modules would require extensive cleaning up to be reused in different and unrelated projects. This cleanup takes time, both from understanding the module and removing the unnecessary code, and has the potential to introduce errors. For example, in the Backup AOD, the only change necessary to reuse the Backup module is rewriting the Backup point cut. Even this modification could be unnecessary if proper company standards are in set, for example, using the same method name for all transactions that require backup.

Evolution also benefits from AOD. The extraction of crosscutting concerns simplifies evolution. In an OO setting developers must work around concerns that shouldn’t be in an object’s scope. This forces developers to spend more time studying the module before intervening and increases the potential for error. AOD extracts the concerns into separate modules, isolating the locations where intervention is necessary. One such scenario is in the realm of error handling. In our AOD solution error handling is extracted into an aspect, which reacts appropriately; logging the error and sending warnings if necessary. On adding a feature that throws an error this error must be added, if previous error handling point-cuts don’t already handle the exception, to the aspect. This module is the only point where the application must be modified to handle this error, focusing the implementation and removing the error handling concern from the added feature’s code.

Another scenario in which evolution is improved is in measuring SLAs. Suppose we need to add a new metric to the SLA. In an AOD design this SLA metric can be added directly into the SLA module without modifying any other components. This aspect can intervene in the appropriate join points without the intervened classes even knowing that the aspect had intervened. The changes were focused on one module, simplifying development and reducing the chance of errors.

AOD weaknesses

As mentioned in the previous section, AOD still has unsolved issues. One problem is obliviousness, which increases the possibility of error and also raises costs by requiring more communication and care. By obliviousness we refer to the issue we observed when trying to evolve the classes on which the aspects intervene, the base classes. Developers working on the base classes may invalidate join points that an intervening aspect requires. This can cause the aspect to execute incorrectly or even not to execute at all. For example, in the AOD backup design, if the developers in charge of the TaskProcessor change the method name for “storeCapture” or even change the method signature, then the Store point cut becomes invalid and the Backup aspect is never executed. If such a change is made then the team in charge of the aspect must somehow be alerted. As such, this problem requires a solid communication structure between teams, which translates into more costs. In a project with one small team in charge of maintaining both the aspect and the intervened classes then this is a small issue. In a large company, where teams are separated to work on different components this issue can have an important negative impact. This becomes even more patent in companies where personal rotation is high.

Another attribute that can create problems for AOD is that it allows for a complete modification of an applications flow. There are no limits on what an advice can do and what it can modify. In a project this could lead to confusion and errors. For example, writing an aspect that modifies a widely used method in a project. Since the original developers expected a certain behavior this modification might break many, if not all, the components that depend on it.

OO design is a mature and proven technology. Companies are satisfied with its performance are confident when determining time and costs on projects that use it. This entrenchment and security forms a barrier against new ideas and concepts that could introduce uncertainty. OO design has been an industry standard for decades. According to Cho et al., developers that have been using a technology for more time distrust new technologies since they have, potentially, more to sacrifice. Although AOD does not completely replace OO design it does require a new way of thinking and designing an application, replacing certain techniques that more senior developers might dislike.

Another factor in a company's culture that influences in AOD adoption is if a company's managers embrace new technologies. Placing emphasis on educating developers and implementing these new technologies in projects will lower the difficulty of trying to adopt AOD. If managers are not receptive to technological change or do not invest in education of new technologies in the company then they will have a harder time and require other arguments to finally embrace AOD.

All these points need to be taken into consideration for each company that's thinking of adopting AOD. From a technological standpoint AOD appears to be a technology worth investing. From an organizational standpoint the risk of becoming and early adopter depends on corporate culture with the enterprise.

7. CONCLUSIONS

AOD presents distinct advantages in lowering coupling and raising cohesion in a real world project and is gaining acceptance in the industry (is on the “slope of enlightenment” in the Gartner Hype Cycle). Although it has important advantages in reusability and separation of concerns, it also has various shortcomings that need to be addressed before it can be widely adopted, especially in large projects.

In particular we recognized the problems caused by obliviousness in classes intervened by aspects. Since a developer can’t know exactly where an aspect is intervening and what information it will need, the modification of a join point used by the aspect may cause unwanted results. We observed various occasions where this impeded evolution and reuse.

Another is the difficulty of convincing management to adopt AOD. This comes from organizations recognizing voids created by the current paradigm, OO design, and of the organization’s attitude towards change and new technologies. Is the high coupling and low cohesion that is sometimes inherent in an OO design a source of concern to the company or is it a minor issue? Are the possible applications for AOD abundant in the company’s projects? Is the company open to new technologies? Do they educate their developers in the latest techniques? These are a few of the questions that a manager evaluating AOD must think of.

This work also presents an initial stepping-stone for future work. Firstly, can the conclusions reached here be generalized to other projects and applications? Another possible track is the determining the real costs of AOD in a real-world project. Can an actual cost-benefit analysis be determined?

Another proposal is an actual study on AOD industry penetration. Although AOD is touted as beginning to be widely adopted in the industry, no such studies have been made to determine this claim. Is the industry adopting AOD? For what use cases is it regularly used? Are colleges and universities teaching AOD? A study to answer these questions would allow for better direction when promoting AOD and its benefits. From this study success stories could be discovered which would also impulse AOD adoption.

Another possible study and important study could be into AOD best practices. When is it appropriate to extract a concern into an aspect? As seen in this work, not all scenarios are suitable for aspect extraction. Answering this question would allow valuable insight on how far AOD should penetrate a software project. Also, when is a point-cut beneficial and what practices should be avoided? AOD allows ample flexibility to modify an application, but many modifications of the application’s flow may have negative side effects. A set of principles to avoid these scenarios would facilitate development and drive AOD further.

REFERENCES

Basili, V., Caldiera, G., Rombach, H. The Goal Question Metric Approach. Encyclopedia of Soft. Eng. Vol. 2, pp. 528-532, 1994

Bustos, A., & Eterovic, Y. Modeling aspects with UML's class, sequence and state diagrams in an industrial setting, Proceedings of the 11th IASTED International Conference on Software Engineering and Applications, pp. 403-410, 2007

Chidamber, S., & Kemerer, C. (1994). Metrics suite for object oriented design. IEEE Transactions on Software Engineering, pp. 476-493, 1994

Cottenier, T., van den Berg, A., & Elrad, T. The Motorola WEAVR: Model weaving in a large industrial context. In Proc. of the 6th Int. Conf. on Aspect-Oriented Software Development (AOSD'07), 2007

Cui, Z. and Wang, L. and Liu, H. and Li, X. Computational error handling as aspects: a case study. Proceedings of the 1st workshop on Linking aspect technology and evolution. pp. 7-11. 2009.

Eder, J., Kappel, G., & Schrefl, M. Coupling and cohesion in object-oriented systems. Technical Reprot (University of Klagenfurt), 1994

Fenn, J. When to leap on the Hype Cycle, Gartner Group Vol. 1, 1995

Greenwood, P. and Bartolomei, T. and Figueiredo et al. On the impact of aspectual decompositions on design stability: An empirical study. ECOOP, pp. 176-200, 2007

I.Cho, Y. Kim Journal of Management Information Systems / Winter 2001–2002, Vol. 18, No. 3, pp. 125–156.

Kiczales, G., Irwin, J., Lamping, J., Lopes, C., Maeda, C., Mendhekar, A., et al. Aspect-oriented programming. Proceedings of the 1997 11th European Conference on Object-Oriented Programming, ECOOP, pp. 220, 1997

Laddad, R. AspectJ in Action: Practical Aspect-Oriented Programming, Manning Publications Co., Greenwich, CT, pp. xxxvii-xl, 2009

M. Mortensen, S. Ghosh. Using aspects with object-oriented frameworks. AOSD '06: 5th International Conference on Aspect- Oriented Software Development, 2006.

Muñoz, F. and Baudry, B. and Delamare, R. and Le Traon, Y. Inquiring the usage of aspect-oriented programming: An empirical study, IEEE International Conference on Software Maintenance, 2009. pp. 137-146, 2009

N. Narendra, K. Ponnalagu, J. Krishnamurthy and R. Ramkumar, Run-Time Adaptation of Non-functional Properties of Composite Web Services Using Aspect-Oriented Programming, Service-Oriented Computing (ICSOC 2007), Volume 4749, pp. 546-557, 2007

Nunes, C. and Kulesza, U. and Sant'Anna, C. and Nunes, I. and Lucena, C. On the modularity assessment of aspect-oriented multi-agent systems product lines: a quantitative study, SBCARS, pp. 122–135, 2008

Ortiz, G. and Bordbar, B. and Hernandez, J. Evaluating the use of AOP and MDA in web service development, Third International Conference on Internet and Web Applications and Services, 2008. pp. 78-83, 2008

Pohl, C., Charfi, A., Gilani, W., Göbel, S., Grammel, B., Lochmann, H., et al. Adopting Aspect-Oriented Software Development in Business Application Engineering, Proceedings of the Aspect-Oriented Software Development (AOSD'08), pp. 1-10, 2008

Sant'Anna, C., Garcia, A., Chavez, C., Lucena, C., & Von Staa, A. On the reuse and maintenance of aspect-oriented software: An assessment framework. Proceedings of Brazilian Symposium on Software Engineering, pp. 19-34, 2003

Urban, M. and Lohmann, D. and Spinczyk, O. The aspect-oriented design of the PUMA C/C++ parser framework Proceedings of the Eighth International Conference on Aspect-Oriented Software Development, pp. 217-221. 2010

Wampler, D. Aspect-oriented design principles: Lessons from object-oriented design. Sixth International Conference on Aspect-Oriented Software Development (AOSD'07), 2007

Wiese, D., Hohenstein, U., & Meunier, R. How to convince industry of AOP. Proceedings of the 6th International Conference on Aspect-Oriented Software Development (AOSD'07), 2007

Yao, Z., Zheng, Q., Chen, G. AOP++: A generic aspect oriented programming framework in C++. Generative Programming and Component Engineering Conference (GPCE). Lecture Notes in Computer Science, pp. 94–108, 2005

Zakaria, A. A., & Hosny, H. (2003). Metrics for aspect-oriented software design. Proc. Third International Workshop on Aspect-Oriented Modeling, AOSD, 2003

Zhao, J., & Xu, B. Measuring aspect cohesion. Fundamental Approaches to Software Engineering, pp. 54-68, 2004

Zhao, J. Measuring coupling in aspect-oriented systems. Proc. of the 10th International Software Metrics Symposium (METRICS), 2004