2010

University of North Carolina Wilmington

Master of Science in

Computer Science and Information Systems

Proceedings

https://csbapp.uncw.edu/mscsis

An Evaluation of Software Architectures – Using Aspects

Maziar Boddoohi

A Capstone Project Submitted to the
University of North Carolina Wilmington in Partial Fulfillment
of the Requirements for the Degree of
Master of Science

Department of Computer Science
Department of Information Systems and Operations Management

University of North Carolina Wilmington

2010

Approved by

Advisory Committee

______________________	______________________
Dr. Ron Vetter	Dr. Bryan Reinicke

____
Dr. Devon Simmonds, Chair

ABSTRACT

To meet market demands and stay competitive, many systems were traditionally pieced together without enough consideration given to quality attributes such as modifiability, scalability, security, and maintainability. This has since littered the computing landscape with brittle applications with high maintenance and complexity. Over time, these maligned systems have propagated in size and merged or integrated to become too complex and fragile to amend or administer. To realign software engineering with its productivity and proficiency developmental goals, the software architecture discipline was formed. Yet, to fully meet its potential, software architecture must be proven to enable stakeholders to judge the quality of the end product before committing to a potential long term financial burden. In support of this goal, this research presents a case study that quantified the benefits of software architectures by comparing an application developed using four different architectural styles. The performance, modifiability and extensibility quality attributes of the four programs are compared and inferences and limitations are discussed. The evaluation of each quality attribute is done as a separate experiment where the original programs are modified in a non-invasive manner using aspect oriented programming (AOP). Use of AOP ensures that the same treatment is applied to each program.

List of Tables

Table 1 - Quality Attributes by Styles 34

Table 2 - Baseline Architectural Evaluation 49

Table 3- Input File Sizes 56

Table 4-Performance of Application based on variable file sizes 57

Table 5- Performance of Application based on variable file sizes (in seconds) 57

Table 6 - Conceptual View of the Dictionary Class 90

Table 7- Dictionary Class Modification 93

Table 8 - OutputToXML Conceptual View 101

List of Figures

Figure 1- Software Architecture Process ____ 16

Figure 2- Meta Architecture ____ 17

Figure 3- Object Oriented Style ____ 18

Figure 4- Pipe-and-Filter Style ____ 19

Figure 5- Event-Driven Style ____ 20

Figure 6 - Shared Data Style ____ 21

Figure 7- Client Server Style ____ 22

Figure 8- Conceptual View Framework ____ 23

Figure 9- Conceptual View ____ 23

Figure 10- Logical View ____ 24

Figure 11- Process View ____ 25

Figure 12- Developmental View ____ 25

Figure 14- Scenarios ____ 27

Figure 13- Physical View ____ 26

Figure 15- Quality Attributes ____ 34

Figure 16 - Metric Definitions ____ 35

Figure 17- Quality Metrics ____ 37

Figure 18 - Eclipse AJDT Plug-in ____ 41

Figure 19- Eclipse Metrics Plug-in ____ 42

Figure 20- Experimental Progression ____ 46

Figure 21- Sample Data File ____ 56

Figure 22- AspectJ Sample Source ____ 56

Figure 23- Performance of Architectural Styles ____ 57

Figure 24 - Experiment - Performance - Metrics Evaluation ____ 59

Figure 25 - Dictionary Class 60

Figure 26 - Experiment - Modifiability - Metric Evaluation 61

Figure 27 - CreateXMLFile 62

Figure 28 - Experiment - Extensibility - Metric Evaluation 63

Figure 29 - Dictionary Sample Source 92

Contents

Chapter 1. Introduction 9

1.1 Problem Statement 10

1.2 Proposed Solution 11

1.3 Outline of Paper 12

Chapter 2. Background 13

2.1 Importance of Software Architecture 14

2.2 How to Approach Software Architecture 15

2.2.1 Architectural Requirements 16

2.2.2 Architectural Style 16

2.2.3 Meta-architecture 17

2.3 Architectural Analytical Techniques 29

2.3.1 Architecture Tradeoff Analysis Method (ATAM) 29

2.3.2 The Software Architecture Comparison Analysis Method (SACAM) 30

2.3.3 Scenario-Based Architecture Analysis Method (SAAM) 31

2.3.4 Cost Benefit Analysis method (CBAM) 31

2.4 Architectural Metrics 32

2.4.1 Quality Attributes 32

2.4.2 Metrics 34

2.5 Tools and Technologies 37

2.5.1 Visio 38

2.5.2 Rational Modeler 38

2.5.3 ArchJava 38

2.5.4 Omondo 39

2.5.5 Aspect Oriented Programming (AOP) using AspectJ 39

2.6 Summary 42

Chapter 3. Method 46

3.1 Key Word in Context (KWIC) 48

3.2 KWIC Architectures 49

Chapter 4. Experiments and Results 55

4.1 Experiment – 1 – Performance 55

4.2 Experiment – 2 – Modifiability 60

4.3 Experiment – 3 – Extensibility 62

Chapter 5. Discussion and Limitations 64

5.1 Limitations and Future Work 64

5.2 Experiment – 1 – Performance 65

5.3 Experiment – 2 – Modifiability 66

5.4 Experiment – 3 – Extensibility 67

Chapter 6. Conclusion 69

Chapter 7. Works Cited 72

Appendix A – Experiment – Performance 82

Appendix B – Experiment – Modifiability 90

Appendix C – Experiment – Reusability 101

Appendix D – Experiment Metric Evaluation 113

Appendix E –AspectJ Source-Code 117

Chapter 1. Introduction

In recent decades, Informational Technology (IT) has rapidly emerged as the primary subject that chief executive officers have resourced to address rising costs and promote productivity. The introduction and use of the World Wide Web has helped to facilitate the elevation of IT into an essential business and individual commodity [1] [2]. Information Technology is no longer a luxurious extension to a large corporation's logistics or inventory management system; rather IT has become a fundamental component for life in the modern world [3]. Today's IT systems can generate live information from clusters of data across the world, and can adapt to provide decision support systems to support organizational strategic goals. It is no wonder that this technology has prompted and promoted exponential growth and influence around the globe.

This expansion has meant more seeded reliance on and significance being given to the computational resources of organizations. Structurally, modern day companies are integrated and interconnected from the ground up [4] [2]. For example, while a clerk is searching the company's catalog on the sales floors, querying customer relations and inventory data, a production manager can schedule capacity by leveraging logistics and inventory systems [5] [6]. Whether the focus is a typical small business or large multi-national corporation, careful examination reveals multiple layers of cohesive systems operating in conjunction, commanding or administering the information needs of organizations. Clearly, IT is no longer simply an analytical tool for data, it has become a leverage point, and a competitive advantage that is used to trim costs, reach a broader audience, and forge new alliances [4].

1.1 Problem Statement

As with any emerging technology, IT suffers from its immaturity [7]. Computing sciences and informational systems only span few decades, and when compared with more traditional sciences, are still in their infancy. While the first computer company was created in the 1950's, it wasn't until the late 1970's that the technology became available for everyday users [8]. The 1990's saw an immense soar in the computing industry as IT became pervasive. Although we have learned a lot from our previous mistakes and experiences, today's standards are far from being optimal. To meet market demands and stay competitive, many software systems were pieced together without ample consideration given to overall software qualities such as modifiability, scalability, security, or maintainability [9] [10]. Lack of attention to these important quality attributes has led to a computing landscape littered with brittle applications having high maintenance costs and accidental complexity [11] [12]. As it turns out, that same IT innovation that aided companies to scale globally has since become an expensive and liable tumor. Over time, these misalignment systems have propagated in size and merged or integrated to become too complex and fragile to amend or administer [13].

To address these kinds of problems, the discipline of *software architecture (SA)* was forged [14] [15]. The term *software architecture* refers to a description of the overall structure of software as defined by the major software components, the dependencies between components, and the component connectors or communication mechanisms used by components [16] [11] [17]. From its conception SA has promised to answer and realign IT with its original goals of proficiency and efficiency [5]. Research has repeatedly shown that accurately designed systems have proven to not only dramatically reduce overall cost and customer satisfaction, but also ensure such important standards as availability, modifiability, security, testability, management, construction, usability, re-usability, comprehensibility, portability, scalability, time to market, and interoperability [18] [15]. As Shaw and Garlan [19] outline, software architecture can resolve wide organizational or global structures, protocols, synchronizations, and data accessibility, functional assignments, physical distribution, scaling and performance.

Many different software architectures are possible for a given application. These potential architectures are often classified into architectural subfamilies called architectural styles. Each architectural style defines a collection of architectures with a unique collection of subsystems and connectors. Typical architectural styles include shared repository, layered architectures and client server architectures [11].

SA brings many benefits to software development including serving as a communication medium among stakeholders. As a high level abstraction, SA can create a mutual understanding among stakeholders in formulating early design decisions, assembling system priorities and addressing competing concerns. Furthermore, by abstracting and modularizing a system, SA can facilitate a transfer of similar components and promote large-scale reuse [15]. Traditional development approaches have failed to properly decouple computation from communication within a system, thus reducing opportunities for re-configurability and reuse [20]. Therefore another benefit of SA is that SA allows developers to focus away from lines-of-code to more coarse-grained components and their overall interconnection structures.

1.2 Proposed Solution

While SA brings many intuitive benefits to software engineering yet, to fully meet its potential, SA must be transformed into a measurable and predictable discipline rather than being primarily intuitive. To accomplish this, architectural quality metrics must be computed for different architectural styles. This enables comparison of the costs and benefits of the different styles and allows informed decisions to be made about the architecture best suited for a specific application. To this end, this research:

1. Selected a target application and four different architectural styles.
2. Developed a program for each architectural style.
3. Selected a set of architectural quality metrics and evaluate each implementation using these quality metrics.

4. Compared the results of the evaluations and infer benefits and properties of the different architectural styles.

The research utilized the Key Word in Context (KWIC) application first introduced by Parnas in his 1972 paper *"On the Criteria to Be Used in Decomposing Systems into Modules"* [21]. KWIC provides a convenient search mechanism for information in a long list of lines, such as book titles or online documentation entries. Each KWIC implementation was evaluated for performance, modifiability and reusability. The source-code for each Architectural Style (Object-Oriented, Shared-Data, Pipe-and-Filet, and Implicit Invocation) was obtained from Dr. Denis Helic's course on software architecture at the Institute for Information Processing and Computer Supported New Media (IICM) at Graz University of Technology, in Austria [22].

Modifications and extensions to the various implementations is conducted by utilizing Aspect Oriented Programming [23] [24] [25] . AOP allows for non-invasive extension of all four applications while performing the exact transformation to each program. The overall goal of this research is to (1) identify the architectural style which provides the best realization of each quality metric, and (2) classify and rank potential code realizations based on the quality attributes represented in the associated architecture. Indeed, by analyzing the research results, we should gain measured insight into how architectural decisions made in the dawning hours of a system can directly affect its long term quality.

1.3 Outline of Paper

Chapter 2 offers an overview of software architectural styles and quality attributes. Chapter 3 reviews the current state of SA, by reviewing relevant and referenced sources for this research. The research method is presented in Chapter 4 along with an overview of the target applications. Chapter 5 describes the for the research performance, modifiability, and reusability experiments conducted. Finally, the timeline for completing the research are laid out in Chapter 6.

Chapter 2. Background

The term Architecture as described by Merriam-Webster dictionary refers to "*a unifying or coherent form or structure*" [26]. Recent research in the high-level blueprint of systems, has demonstrated the necessity of this overall *structural* design before actual implementation occurs. As a system is designed, developed, tested, and moved to be deployed, amendments or requirement failure discovery can be devastating to the timely delivery of a system. Furthermore, once a system is delivered and matures, if the system is not designed with expansion and evolution in mind, any additional change can be costly and potentially disable the existing interaction [19].

Software Architecture (SA) formal definition remains elusive due to the broad umbrella of the topics it represents, as demonstrated in the collection of definitions below:

> *"The software architecture of a program or computing system is the structure or structures of the systems, which comprise software elements, the externally visible properties of those elements, and the relationship among them."* [16]

> Software Architecture is *"the structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time."* [18]

> *"Abstractly, software architecture involves the description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns. In general, a particular system is defined in terms of a collection of components and interactions among those components."* [19]

While the label may be vague and indistinct, what is clear is the classification of its function to the computing discipline – to topographically illustrate and define conceptually the purpose of a system

accompanied with designations of decisions and external interactions with it, and its interaction between each component. David Garlan describes SA best by portraying it as a "*bridge between requirements and implementations*" [14]. This "*bridge*" is intended to decompose the complexity of systems by: (1) "*providing abstractions that hides unnecessary detail, providing unifying and simplifying concepts*", (2) making "*development of the system easier to manage by enhancing communications, providing better work partitioning with decreased and/or more manageable dependencies"* [27]. Finally we can illustrate the concept of SA by defining it as a structured solution that meets the entire technical and operational requirement, while optimizing common quality attributes such as performance, security, and manageability. It involves a series of decisions based on a wide range of factors, and each of these decisions can have considerable impact on the quality, performance, maintainability, and overall success of the application.

Now that we have defined SA, we can identify its importance and begin to recognize its components and framework. To do so, let's put ourselves in the architects place and begin to decompose the design process by walking in the shoes of the designer.

2.1 Importance of Software Architecture

Software architecture captures the gross partitioning of the system and expresses the fundamental structural organization of the system elements and the relationships between them. Again, this organization is essential for meeting the functional and quality attribute requirements on delivery day as well as throughout the life of the system [28] [29]. Systems built without a well-designed and documented architecture will exhibit unpredictable properties—the system *might* be modifiable, it *might* perform as required, and it *might* interoperate with other systems [28]. SA's goal is to bridge this uncertainty with a communication medium among stakeholders. SA proposes facilitation of a high level abstraction to create a mutual understanding in the early design decisions while assembling the systems priorities and

addressing competing concerns. Furthermore, by abstracting and modularizing a system, it can facilitate a transfer of similar components to promote large-scale reuse [15].

David Garlan defines six aspects that are imperative in the success or failure of the overall systems: understanding, reuse, construction, evolution, analysis, and management [14]. Incorrect design can significantly affect the systems design and deployment with respect to these aspects. The correct choice will affect how the system leverages past architectural designs and answer concerns as they arise [30]. Often, a poorly designed system behaves unpredictably and therefore cannot be scheduled or resourced properly. Once deployed, they require a higher level of technical support and often, a small change to an independent module will propagate failure across the system. Evolution or upgrades can cause future extension to fail and ultimately increase development, testing, and maintenance costs. Also, maintaining quality personnel and keeping development teams morale high can be difficult when system are constantly maligned and tangled [31].

As we can see, it is quite rational to design a software system much like a structure. Before grounds are cleared or buildings demolished, the architect of the structure gathers data to help identify the purpose. The architect then blue prints a model to present his/her concept to the stakeholder, and only then begins to codify the specs for a high level builder to develop. But a greater question looms: How would we approach this very broad and difficult problem?

2.2 How to Approach Software Architecture

As we discussed in the prior section, SA is crucial in the overall effectiveness and longevity of a system, and not surprisingly, it is the most difficult task to perform. Architects' knowledge and experience play a vital role in recognizing and identifying areas of interest and complexity [32]. One cannot design something without knowing its ***function and purpose***. Therefore, the architect begins by first identifying the requirements that impact the structure of the application. As shown in Figure 1, the architect utilizes those requirements as inputs into designing the system and will routinely evaluate the

patterns in the various architectural styles to strategize the strength and weakness of each with regards to those requirements. It is then that the architect can specify the individual components that make up the application, showing how they fit into the overall framework [32].

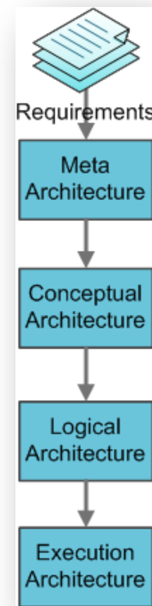


Figure 1- Software Architecture Process

2.2.1 Architectural Requirements

Architectural requirement can be described as how the system is designed and modeled to adhere with overall business goal of the project [33]. Gathering and verbalizing these needs is crucial in delivering a final product that meets the stakeholder's requirements in the system. This can be an extensive and iterative process as the stakeholder grasps and gets a handle on what the system should provide and how. Requirements aid in specifying the scope and the constraints on the application and lay out a priority of functional and nonfunctional features the stakeholder expects to be encompassed in the final product [31]. ***Functional Requirement*** is defined aspects of the system as requested by the stakeholders. These requirements can be graphed and itemized by using use cases to define their scope. ***Non-functional requirement*** are system-wide qualities that have architectural significance like maintainability, security, and expandability.

2.2.2 Architectural Style

Once the requirements of a system are verbalized and agreed upon, we begin by prioritize and identify how to decompose the proposed solution. To do so, we must first identify an ***Architectural style*** that will encompass the overall structure of the system. An Architectural style refers to *"a set of design rules that identify the kinds of components and connectors that may be used to compose a system or subsystem, together with local or global constraints on the way the composition is done"* [34]. Components refer to computational artifacts, while connectors refer to the interactions between them. Constraints are how the interactions between the connectors and components should behave [31].

By gauging the appropriateness and effectiveness of different alternatives, we can answer questions about how to differentiate between components, connectors, and constraints. We can then select the proper computational model to present the system while leveraging and referencing known solutions. For small applications, a single architecture pattern, like the client-server may suffice, while more complex applications require the architect to specify more than one pattern integrated to form the overall architecture. Once we have identified our styles we can begin to outline and classify each alternative and draw from quality metrics for selection analysis.

2.2.3 Meta-architecture

Bredemyer [31] describes Meta-architecture as a set of high-level decisions that guides the structure of a system by utilizing styles, interaction patterns, principles, and philosophies (refer to Figure 2). As the architect, we can define meta-architecture as the bridge between the business strategy and technical objectives. This concept enables the development of the system to weigh selections and allows for a trade-off analysis by documenting and strategizing the communication and coordination of repetitive components. This ensures a consistent and elegant architecture that is easy to understand and honor.

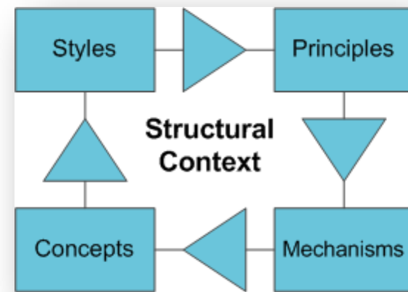


Figure 2- Meta Architecture

Meta-architecture first requires the architect to formulate a ***structural blue-print*** for the rest of the software development cycle. Research in documented styles, patterns, designs and models will typically illuminate a variety of sources, references, organization, partners and competitors that can be studied before diving into the production process [31].

Now that the architect has his high level structural blue-print, it is time to define methodologies and styles that can potentially encompass the structural context of the Meta-Architecture. It is here that

the architect must ponder over whether to select Data Abstraction, Data Flow, Data Centric, Client-Server, or Implicit Invocation.

2.2.3.1 Object Oriented or Data Abstraction

The Object-Oriented architectural style features abstract ***data types or objects that represent data and interact through functional or procedural invocations***. While objects are responsible for preserving the integrity of their representation, they also hide information from other objects. In the Data Abstraction architectural style, the system can be viewed as a collection of objects encapsulated to provide interfaces for other objects in the system to interact with (refer to Figure 3). The communications can be achieved by sending message or passing parameters from and to other objects [11]. An object is responsible for preserving its own integrity and representation from other objects in the application (*private* or *public*). Each module provides interface or variable parameter for other components access the object through.

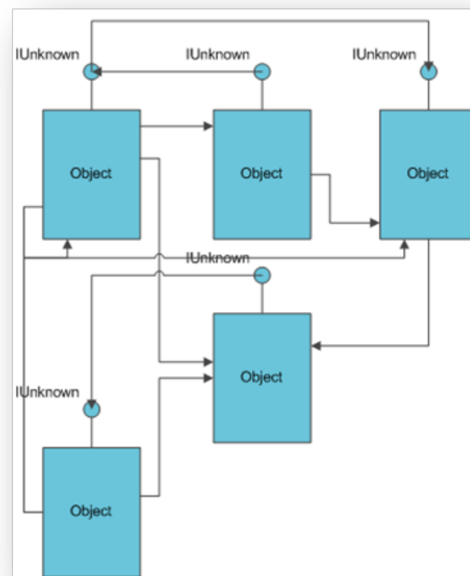


Figure 3- Object Oriented Style

Data Abstraction is the most widely used and taught architectural style and many examples exist in the software landscape. For example, Common Object Request Broker Architecture (CORBA) utilizes this architecture to instantiate and broker objects to and from clients/servers.

Reusability and modifiability are the strengths of this style, as they facilitate the modification of functionality while reusing other objects. Yet as a side effect, if one object interacts with another, it must know the identity of that other object. Therefore if the first object changes, the method that invokes that

object will be agnostic to those changes. Also, if a child object inherits functionality from a parent object and the parent object changes, then the functional integrity of the child object may be adversely affected.

2.2.3.2 Pipe and Filter or Data Flow

This style comprises of components that represent set of ***inputs and outputs***. Components read from a stream of data as its input and processes the data to deliver a stream of data for its output [7] (refer to Figure 4). Each component applies a local transformation and produces the output streams of data sequentially, because each output is a result set of another component's input stream of data,. These independent processes are refered to as filters. The communication or connection between the filters termed pipes, since they serve as delivery mechanism [35]. Pipes serve as the channel for the streams and transforms outputs from one filter to inputs of another. This model architects the system into a set of sequential components run in succession. Each component does a local transformation to the input stream of data and produces an output steam of data. Therefore, each process is intiated only after its predessessor has completed its transfromation.

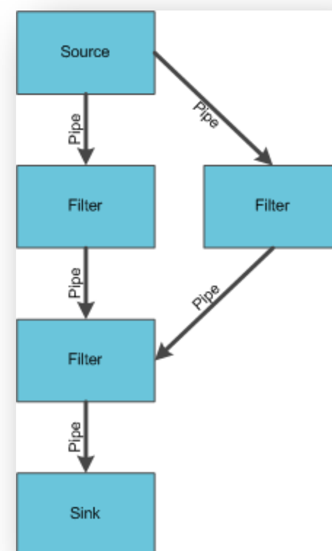


Figure 4- Pipe-and-Filter Style

An example of this architectural style is a message based system. In this popular style the clients send requests to the queue, where the message is stored until an application removes it. For example, a client may format an email, and place it onto the queue for processing. The server will, at some stage in the future, remove the message and send the email using a mail server. The client really doesn't need to know when the server processes the message, it only knows that the message has been sent.

This style's strengths lie in its ease of understanding and ability to reuse filters. It promotes loose coupling by indirectly binding each filter. Filters are oblivious to the state and condition of each other.

Yet, the weakness lies in the inability to process batch-type or threaded instructions. Since a process cannot start until its predessessor has ended, a thread cannot be instantiated before the actions is completed [35].

2.2.3.3 Implicit Invocation or Event-Driven

As shown in Figure 5, implicit invocation or an event-driven architectural style provides the system with the ability to ***announce one or more events while other agents associate a procedure with the event separately***. When an event is announced, the broadcasting system invokes all of the procedures that have been registered for the event itself (41). Garland and Shaw describe implicit invocation systems: "*The idea behind implicit invocation is that instead of invoking a procedure directly, a component can announce (or broadcast) one or more events. Other components in the system can register an interest in an event by associating a procedure with the event. When the event is announced the system itself invokes all of the procedures that have been registered for the event. Thus an event 'implicitly' causes the invocation of procedures in other modules* [11]."

Implicit Invocation utalizes components by enabling the integration of data abstractly and invoking computation on that data implicitly. The connection between components is achieved through an announcement (or broadcast) an event. Other components in the system can register an interest in that event by acting on that event [11]. When the event is announced, the applications invokes all of the procedures that have been registered for the event. Thus, an event announcement implicitly causes the invocation of procedures in other components. The announcers of events do

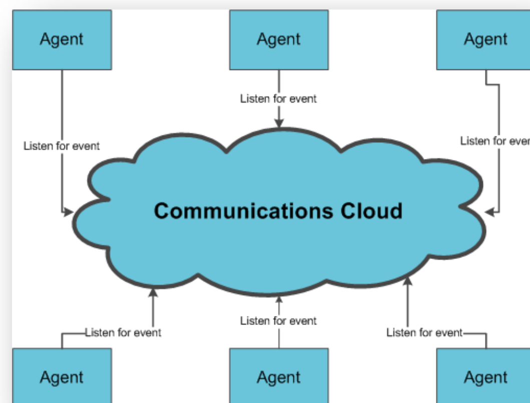


Figure 5- Event-Driven Style

not know which components will be affected by those events.

An advantage of event-based invocation is that it encourages reuse across the system. It's framework allows for agents suscribing to an event to act and function atuonomously to the state of other agents in the system. Furthremore other agents, objects, processes, and servers, can be introduced in a system simply by registering it for the events, thereby enabling scaling and expansion of the system without restracuting of the components. A drawback to this approach is that event-based systems become quite unpredictable because of a lack of a transactional broker to marshall actions. and perfrom porrly due its inability to support batched processes.

2.2.3.4 Data-Centric or Shared Data

Figure 6 portrays the shared repository or data-flow centric architecture, which consists of a ***central data structure (often a database) and a collection of independent components which operate on the central data structure***. This architectural style typically consists of these components running in parallel and communicating through a data channel [11]. Data is communicated between the components through shared storage or global array visible to all entities in the application. Many examples of repository architectures exist in IT. An example of such a system is the blackboard architectures [11], where a blackboard serves as communication center for a collection of knowledge sources, and database systems serving several applications.

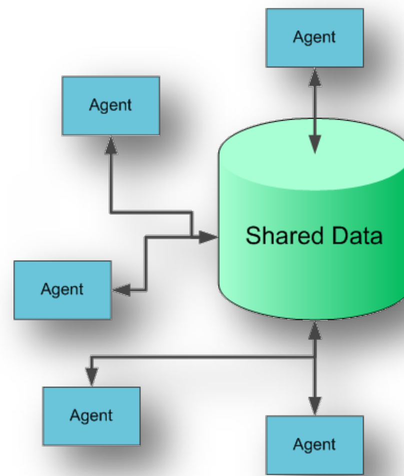


Figure 6 - Shared Data Style

This design constrained due to the necessity for data is communicated between the components through shared storage and greater effort is needed to be for data synchronization. This style also relies heavily on the health of the shared data structure – if the data container is affected so is the rest of the system and it agents. Furthermore, the lack of cohesion between members of the agents reduces the

reusability and extendibility of the underlying code. A major advantage to this style is the simplicity with which modifications can be made – only one source is upgraded or amended rather than the segregates

2.2.3.5 Client-Server

Figure 7 describes the Client-Server style which is commonly implemented when the application is distributed across the system. In the Client-Server style, there are two components ***server and the client***. The server is designed to provide services to multiple instances of clients connected within the network which provides data access and maintains data integrity.

The clients can only communicate with the server, and not with each other. The communication is typically a request initiated by the client, which results in a respond by the server. The server typically performs some computation and delivers the values back to the client.

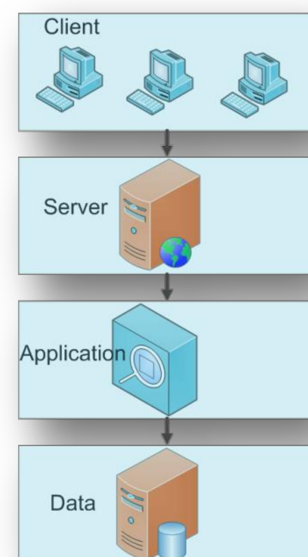


Figure 7- Client Server Style

This is a widely used style especially within the world-wide-web application. When the HTTP channel is opened, the client instantiate a session with the server and follows through several complementary interactions in order achieve its descent of information or computation.

The greatest advantage of this style lays in its promotion the separation of concerns into different tiers. This enables the system to be portioned onto different independent machines or platform, each operating autonomously, until an action is required. Also this style enables systems to communicate synchronously through request-reply methodology, ensuring the delivery/receipt of actions.

2.2.3.6 Conceptual Architecture

As the architect, now that we have identified and defined architectural styles, we must begin to lay the conceptual foundation. The conceptual phase is described by Bredemyer [27] as one in which the system is decomposed prior to expanding on the interface details (refer to Figure 8). This facilitates the overall description that can be used to portray the design model to the non-technical stakeholder. While this phase hinges on the description and more detailed definition of the systems' overall syntax, it is still very high level. This phase allows for system components to be identified by representing their responsibilities and inter-connections. Typically the design methodology is driven by the quality requirement preset by the user [31]. Conceptual phase also enables architectural constraints to be imposed early in design and provides the framework to enforce this structure as the systems evolves [14].

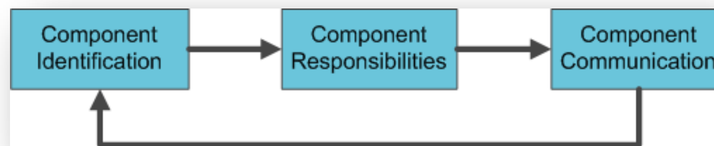


Figure 8- Conceptual View Framework

To achieve this, the architect assigns responsibilities and interactions of each component through the utilization of a number of ***views***. As figure 9 shows, a view is a representation of a coherent set of architectures and the relations among them. By abstracting functional portions of the stakeholders concerns, views enable the architect to determine the effect of change, while guiding and describing the development process [16] [36]. The architect is able to extract an architectural model to satisfy the major functionality and performance requirements of the system, as well as some other, non-functional

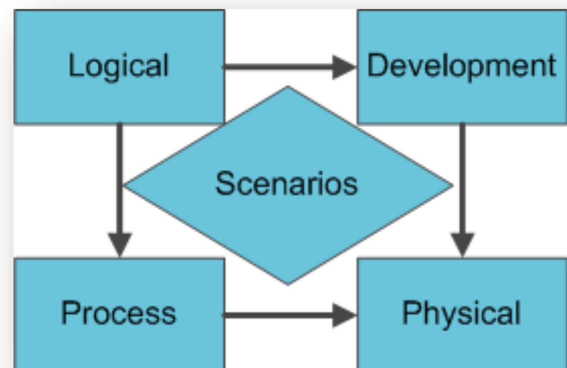


Figure 9- Conceptual View

requirements such as reliability, scalability, portability, and availability.

In order describe this abstract decomposition of systems, Philippe Kruchten proposes five main views to esthetically describe their composition. (1) Logical View – to represent the object model of the design, (2) Process View – to capture concurrency and synchronization, (3) Developmental – annotates the static organization of software in the development environment, (4) Physical View – to describe the mapping of software onto the hardware, and (5) Scenarios – which describe the decisions made around the architecture [37].

2.2.3.6.1 Logical or Functional View

The Logical or Functional View provides an abstraction of a system and its relationships with components (refer to Figure 10). These components represent such elements as functions, key ***system abstractions***, and domain elements that objectify system to address non-functional requirements. Users of this view are typically domain engineers, product-line designers, and end users. The most commonly used representation of this view is class diagrams. In the class diagrams a logical or functional view into object classes are drafted. Furthermore, class relationship, behavior, association, aggregation, and inheritance are depicted in order to maintain a single, coherent object model across the system. The connectors represent dependencies and data flow [15] [37].

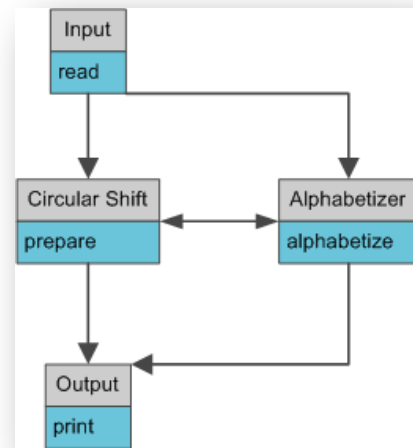


Figure 10- Logical View

2.2.3.6.2 Process or Concurrency View

The Process or Concurrency View enables processes such as performance, availability, and integrity to be structured and identified between elements. Processes are identified as a grouping of these tasks that form an executable unit. This process creates multiple ***levels of abstractions*** so that

programmers and tester can find a medium to address concerns with the development of data via flow, events, and synchronization [15]. Typically this view is depicted by state, sequence, collaboration, or activity diagrams, as shown in Figure 11.

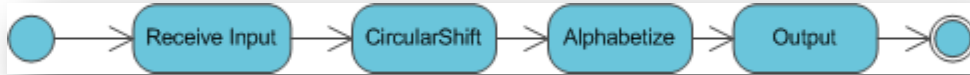


Figure 11- Process View

2.2.3.6.3 Development or Component View

Developmental or Component View is used for partitioning development and testing, this views aims to facilitate modifiability/maintainability by creating source code repository that various users create, modify, and manage [15]. This view describes the software architecture as an organization of modules and subsystems that can be concurrently developed and manipulated (refer to Figure 12). This view enables a layer of internal requirements like reuse, software management, and ease of development to be addressed. Most commonly used depiction of the process view is ***component diagram*** which describes software components – such as source, binary, or executables – and their dependencies to each other, representing the structure of the code itself.

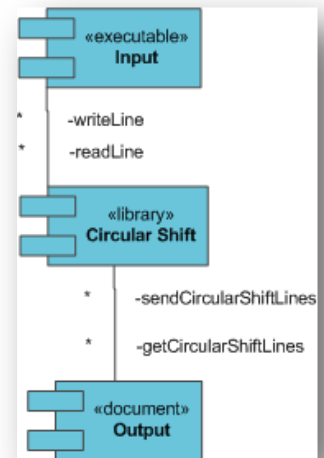


Figure 12- Developmental View

2.2.3.6.4 Physical View

In the Physical View CPUs, storage, and other peripherals that are connected through network communication devices that describe the system in hardware [15]. This view primarily accounts for the non-functional requirements of a system by mapping the ***physical hardware*** with the written code. For example figure 13 represents the communications between desktop computers and the server or mainframe. The diagram describes the physical interaction between the computers and the network layer.

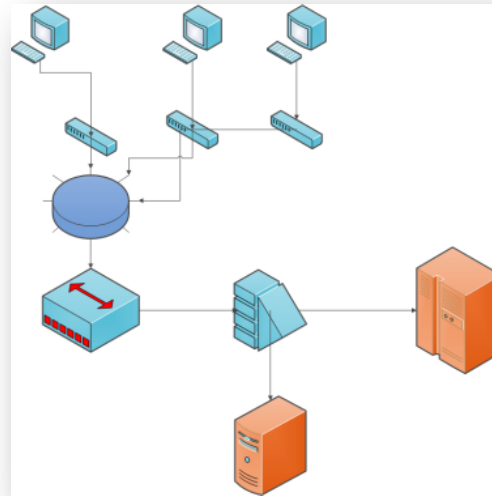


Figure 13- Physical View

2.2.3.6.5 Scenarios

Scenarios finalize the conceptual phase by allowing a walk through the architecture [15]. This view is the glue that enables all views to be work seamlessly by instantiating general ***use cases*** to describe the correspondence between each view (refer to Figure 14). This view provides the mechanism to portray the functionality of the system and how it is perceived by external actors. This view is used by the stakeholders, developers, and users to identify and specify the contents and elements of the systems. More importantly it provides the verification of the functional and non-functional requirements upon the system.

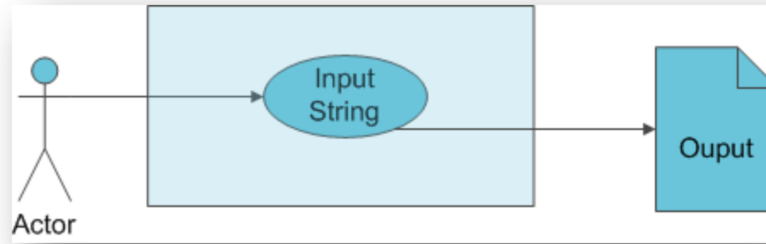


Figure 14- Scenarios

Another important concept in the development of the conceptual framework is the ***Architectural Description Language (ADL)***. ADL is important to structural development of a system, because it provides a high-level structure of the overall application rather than the implementation details of any specific source module. ADLs provide both a concrete syntax and a conceptual framework for modeling a software system's conceptual and logical architectures. The building blocks of an architectural description are described by:

- Components - units of computation or data stores
- Connectors - architectural building blocks used to model interactions among components and rules that govern those interactions.
- Architectural configurations - connected graphs of components and connectors that describe architectural structure [34].

Architectural models include elements such as software components, communication mechanisms, states, processes, threads, hosts, events, external systems, and source code modules. Relationships between these elements address such issues as message passing, data flow, resource usage, dependencies, state transitions, causality, and temporal orderings [38]. Uniform Markup Language is well suited for this because it provides a useful and extensible set of predefined constructs, it is semi-formally defined, it has substantial tool support, and it is based on experience with mainstream development methods [38].

2.2.3.7 Uniform Markup Language (UML)

A UML model of a software system consists of several partial models, each of which addresses a certain set of issues at a certain level of fidelity [39]. Fidelity refers to how close the model will be to the eventual implementation of the system, low-fidelity models tend to be used early in the life-cycle and be more problem-oriented and generic, whereas high-fidelity models tend to be used later and be more solution oriented and specific. Increasing fidelity demands effort and knowledge to build more detailed models, but results in more properties of the model holding true in the system [39].

The graphical nature of UML's ***well-defined syntax and semantics*** enables the mapping of graphical elements to elements of the underlying semantic constraint model. The constraints are expressed in the Object Constraint Language (OCL), which defines common operations on elements, and highlights the relationships between attributes and the elements [38].

2.2.3.8 Logical Architecture

As the architect, now that we have a well-defined conceptual framework for the system, we can now begin to erect the structure from the foundation. Bredemyer consulting defines this Logical phase as one in which detailed architectural specifications are appended to the conceptual model in order to address cross-cutting concerns with the definition of component interfaces/protocols and their dynamics with the system [31]. In this phase, the system's functionality is ***precisely defined*** and diagramed so that the designer and developers can collaborate in creating a static structure to build the system upon. Such mechanism as component interfaces, actionable procedures, and exception control mechanisms are clearly and completely defined in this phase.

2.2.3.9 Execution Architecture

Finally, we can conclude our architectural process by identifying our ultimate executive phase. The execution phase consists of the mapping of components onto the ***physical hardware*** in order to

address the overall throughput and scalability of a system [31]. This phase is especially important for distributed or concurrent systems due to the deployment representation of the system onto its environment. Also, in this phase, the programming language for the development of the project is selected.

2.3 Architectural Analytical Techniques

In the prior section we reviewed how the software architectural discipline can guide the decomposition of the design process so the results can meet the requirements. As we learned during the conceptual phase, the architect must compare and contrast the alternatives available in order to weigh the best solution. Comparing software architectures for any nontrivial system is a difficult task no matter how many organizations necessitate long-term candidate architectures. As a result, currently, there are many different heuristics that can guide architects through their selection process. In this section we overview a few analytical techniques that architect can use to enable superior comparative selections.

2.3.1 Architecture Tradeoff Analysis Method (ATAM)

Architecture Tradeoff Analysis Method (ATAM) is a thorough and comprehensive way to evaluate software architecture which is designed to elicit the business goals for the system as well as for the architecture [16]. ATAM structures the trade-off analysis by partitioning the participants of the systems and assigning each division clear and concise goals for evaluation of architecture. This process is not aimed to predict the outcome of the project, its goal is to identify and gauge risks involved with each approach. The Bass paper describes the process as stated below:

- **The evaluation team**. External group that consists of members who are assigned a number of specific roles. The evaluation team may work for the same organization as the development team whose architecture is on the table, or they may be outside consultants but must be recognized as competent, unbiased outsiders.

- **Project decision makers**. Decision makers are empowered to speak for the development project or have the authority to mandate changes to it. They usually include the project manager, or architect
- **Architecture stakeholders**. Stakeholders have a vested interest in the architecture performing as advertised. They are the ones whose ability to do their jobs hinges on the architecture promoting modifiability, security, high reliability, or the like. Stakeholders include developers, testers, integrators, maintainers, performance engineers, users, builders of systems interacting with the one under consideration, and others. Their job during an evaluation is to articulate the specific quality attribute goals that the architecture should meet in order for the system to be considered a success.

2.3.2 The Software Architecture Comparison Analysis Method (SACAM)

The Software Architecture Comparison Analysis Method (SACAM) compares architectures based on a set of criteria derived from the business goals of an organization. The SACAM was developed in a technical reuse context where an organization investigated architectural commonalities and differences to explore architectural designs for software product line architecture. The comparison is performed in a series of steps: (1) The architect gathers the requirements needed for the preparation of the analysis; (2) Comparison criteria are chosen, collated, and then refined into concrete quality attribute scenarios; (3) The scenarios and existing architectural documentation of the architecture candidates are used to identify the relevant information for the comparison. This information is extracted from the architectural documentation and analyzed to determine how well the required scenarios are supported. (4) The stakeholders score the architecture on a scenario basis, which leads to a recommendation for selection. The scores might reflect weights that are provided by the stakeholders for the criteria. The artifacts generated during the course of the method can be used for subsequent processes, such as an architectural commonality and variability analysis for a software product line migration [40].

2.3.3 Scenario-Based Architecture Analysis Method (SAAM)

Scenario-Based Architecture Analysis Method (SAAM) focuses on analyzing the architectural design of a software system by defining each in (at least) three perspectives—the functional partitioning of its domain of interest, its structure, and the allocation of domain function to that structure. [13] The perspectives can then be used to weigh and score each style:

- Functionality: The overall behavior that partitions such that the collection is simplified and easy to conceptualize.
- Structure: Reveals how the system is constructed by depicting the structure as a collection of components which represent computational processes or data. Also, the structure represents the connections between these by enabling communication and control relationships among them.
- Allocation: Refers to how the functionality is realized in the software structure in order to understand how the functionality is going to be achieved.

2.3.4 Cost Benefit Analysis method (CBAM)

Cost Benefit Analysis method (CBAM) aids in the analysis of the benefit and cost implications of the architectural design decisions being made [41]. Given the fact that there are large uncertainties—both technical as well as economic—during the architecture design stage of a software system, the architects faces risks with regards to the system's ability to meet its business goals. One of the advantages of our approach lies in the fact that it forces the stakeholders to quantify these risks, by explicitly quantifying the benefits of architectural decisions, as well their costs, quality attribute responses, and the uncertainty surrounding these estimates [41]. In this method the stakeholders individually rate each of the attributes so that the sum of their scores is 100. The insight gained from the real-option formulation is that we want to postpone a decision when the Net Present Value (NPV) is less than the value of the option to delay. We can estimate these values based upon the information that we elicit in the CBAM process, because we are already estimating the costs and expected benefits as part of the elicitation process.

2.4 Architectural Metrics

Architectural metric can be defined as attributes that we can use to gauge the viability of an architectural style versus other alternatives. Metrics are crucial in weighing architectural alternatives in order to better structure and deliver systems to the stakeholder. Metrics also provide an iterative assessment of the design and implementation and thereby enlighten areas that may affect the eligibility of the overall architecture. Analytical techniques that architects use to contrast styles often hinge on the fitness of these measures.

To understand metrics and their value, we must first define what quality signifies in Information Systems. This section defines quality attributes and classifies major metrics to value alternatives against.

2.4.1 Quality Attributes

Quality attributes are system properties that enable the architect to analyze the tradeoffs while differentiating between characteristics that a system must attain in order to assure specifications. *It is the degree to which a customer or user perceives that software meets his or her composite expectations. It is the composite characteristics of software that determine the degree to which the software in use, will meet the expectations of the stakeholder* [42]. The evaluation of these qualities provides the basis for specifying quality requirements and evaluating quality while describing a set of characteristics and relationships between them. This establishes a framework to perform some kind of measurement of the specific desirable features that are needed in the final system as perceived by the stakeholder.

The most important effects of metrics are the comparison and identification of strengths and weaknesses in different architectural alternatives during the design stage. Quality refers to the degree of which software possesses a desired combination of attributes [43]. Quality attributes can be broken into operational – artifacts that enable the users to better exercise the system– and developmental requirements – where the importance lies in the developer's ability to labor. Developmental requirements include such variables as maintainability, understandability, and flexibility, while operational requirements include

such priorities as performance and usability. Quality is described by the stakeholder in order to quantify what they require of the system [29].

The following excerpts are directly taken from [29] Institute of Electrical and Electronics Engineers (IEEE) which describes the leading quality attributes as:

Maintainability	Maintainability is a long-term requirement that can typically be d with readability, understandability, modifiability, and comprehensibility. *"The ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to changed environment."* To measure: Average Number of Operations: the more operations in a Software Architecture result in reduced ability to maintain code.
Performance	Performance is a measure that is evident immediately to the stakeholder and can be described by resolving such issues as availability, load-balancing, and usability. *"The degree to which a system or component accomplishes its designated functions within given constraints, such as speed, accuracy, or memory usage."*
Testability	Testability can be defined as the amount of effort it takes to verify the developed system against the functional requirements of the stakeholder. *"The degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met."*
Portability	Is a long-term goal that enables software migration and system upgrades and is best defined by: *"The ease with which a system or component can be transferred from one hardware or software environment to another."*
Functionality	ISO standards describe functionality as the set of attributes that bear on the existence

	of a set of functions and their specified properties.
Reusability	Reusability refers to the ability to reuse components or the system itself in a new or modified stage.
Extensibility	Allows adding, enhancing, or repairing functionality without undesired implications.

Figure 15- Quality Attributes

As stated earlier, we can predict some quality attributes of a style. For example, we know that the Abstract-Data Type model is reusable and functional, while Implicit Invocation lacks in performance. Yet, Implicit Invocation is easily modifiable because each module operates on its own, but it is difficult to understand and poor when the number of instructions increase. By intuition and trail-and-error, the below table can be deduced to differentiate each quality attribute from its source style.

Table 1 - Quality Attributes by Styles

Method	Advantage	Disadvantage
Data Abstraction	Functionality, reusability	Maintainability
Data Flow	Portability, usability	Functionality, maintainability
Implicit Invocation	Modifiability, reuse	Performance, understandability
Data-Centric	Understandability, performance	Maintainability, functionality

Yet, how do we – numerically – portray the above table so that future systems can take advantage of our prior knowledge? We know intuitively, but can we prove this empirically? To answer these questions we must establish a standard to measure and discover the above attributes. To do so, the next section defines Metrics, as it relates to SA.

2.4.2 Metrics

Now that we understand what quality in software architecture refers to, let us peer into how to measure for these quality-attributes [44]. By identifying and qualifying different metrics, we then begin to numerically identify these attributes more clearly. Roger Pressman defines *metrics* as "*a quantitative* ***measure of the degree to which a system, component, or process posses a given attribute***" [45]. In

software development there are numerous measures that the architect can reference to aid in the discovery of measure. First, we must define key terms that are universal to the identification of metrics. These terms are as follows [46]:

Attribute	Defines the structural properties of classes which may include objects methods.
Cohesion	The degree to which the methods within a class are related to one another.
Coupling	Object X is coupled to object Y if and only if X sends a message to Y.
Complexity	Structural characteristic of how components are interrelated to one another [45].
Inheritance	Object within a class that inherent characteristics from one or more other classes.
Message	A request that an object makes of another object to perform an operation.
Method	An operation upon on object, defined within as part of the declaration of a class.
Object	An instantiation of some class which is able to save a state (information) and which offers a number of operations to examine or affect this state.
Operation	An action performed by or on an object, available to all instances of class, need not be unique.

Figure 16 - Metric Definitions

Given these terms, we can begin to define metrics to evaluate our design against. The following are leading metrics used to evaluate software project as stated on *Source Forge*'s Metrics site [47]:

Cyclomatic Complexity	Computed considering the average, number of dependencies between Entities subtracted by the number of Entities which aids in finding complexity
Number of Entities	The Number of Entities (of an ISA) is computed by counting the number of Entities. Allows to gauge the Maintainability of an ISA tends to decrease with this metric increase.

Number of Relations	Computed by counting the number of relations between Entities. Allows finding the Maintainability of an ISA tends to decrease with this metric increase.
Lines of Code (LOC)	Total lines of code in the selected scope.
Method Lines of Code (MLOC)	Number of lines of code inside method bodies.
Number of Static Methods (NSM)	Total number of static methods in the selected scope.
Number of Classes (NOC)	Total number of classes in the selected scope.
Number of Attributes (NOF)	Total number of attributes in the selected scope.
Number of Packages (NOP)	Total number of packages in the selected scope.
Number of Overridden Methods (NORM)	Total number of methods in the selected scope that are overridden from an ancestor class.
Number of Static Attributes (NSF)	Total number of static attributes in the selected scope.
Number of Methods (NOM)	Total number of methods defined in the selected scope.
Number of Parameters (PAR)	Total number of parameters in the selected scope.
Number of Interfaces (NOI)	Total number of interfaces in the selected scope.
Number of Children (NSC)	Total number of direct subclasses of a class.
Depth of Inheritance Tree (DIT)	Is the maximum length from a node to the root (base class) where lower level subclasses inherit a number of methods making behavior harder to predict.

Lack of Cohesion of Methods (LCOM)	Calculated with the Henderson-Sellers method. A low value indicates a cohesive class and a value close to 1 indicates a lack of cohesion and suggests the class might better be split into a number of (sub) classes.
McCabe Cyclomatic Complexity	Counts the number of flows through a piece of code. Each time a branch occurs this metric is incremented by one.
Method Lines of Code (MLOC)	Total number of lines of code inside method bodies, excluding blank lines and comments.
Nested Block Depth	The depth of nested blocks of code.
Weighted Methods per Class (WMC):	Sum of the McCabe Cyclomatic Complexity for all methods in a class.

Figure 17- Quality Metrics

Now that we have laid the foundation and boundaries for SA, we can begin to analyze how to achieve our final goal through experimentation. The next section introduces tools used to bridge concepts with products.

2.5 Tools and Technologies

This section provides an overview the tools used in conducting this research. While all tools were equally influential in the development and process analysis of this project, IBM's Eclipse IDE was the primary source for development and testing. Source Forge's Metrics, AJDT (refer to Section 2.5.5), and Omondo's UML plug-ins also played a crucial role in the analysis and design of the source inside the Eclipse environment. While AJDT proved the environment to create and compile AspectJ source, the metrics plug-in supplied the assessment. Omondo's UML created the logical views of the source, while most conceptual drawing and figures were drafted using Microsoft's Visio.

2.5.1 Visio

Microsoft Visio is a diagramming program for Microsoft Windows that uses vector graphics to create diagrams. Software application developers can model the application's design and functionality with Visio and Unified Model Language (UML) 2.0 and even perform reverse engineering on an implemented system and transform existing code into a UML model. (http://office.microsoft.com/en-us/visio/)

2.5.2 Rational Modeler

The IBM Rational Software Architect family is an integrated design and development suite that leverages the Rational Software Modeler to enable the creation of robust, scalable solutions for requirements elaboration, design, and general modeling. The Modeler suite allows for the evaluation of UML profiles and the generation of tooling for them such as palettes, context menus, property sheets, and model templates. Modeler also uses UML2.0 modeling including Use Case, Class, Sequence, Activity, Composite Structure, State Machine, Communication, and Component diagrams. (http://www-01.ibm.com/software/awdtools/modeler/swmodeler/)

2.5.3 ArchJava

Carnegie-Melon has played a very influential role in the development of SA. As the ArchJava website explains, Arch Java's ability to approach SA by decoupling implementation code from architecture. ArchJava attempts to extend Java in order to seamlessly unify software architecture with implementation, using a type system to ensure that the implementation conforms to architectural constraints. ArchJava is a standard language features such as objects/function pointers and shared mutable references that verifies a program's ability to conform to architecture. ArchJava enforces communication integrity for control flow: a component may only invoke the method of another component if it is connected to the other component in the architecture. ArchJava is also the only extension to a mainstream

language that explicitly describes a very general class of software architectures. (http://archjava.fluid.cs.cmu.edu/)

2.5.4 Omondo

Omondo Eclipse UML is a visual modeling tool that natively integrates with Eclipse to provide data modeling, UML, J2ee, and business process modeling. Omondo also implements Use Case, Class, and Sequence diagrams to enable the developer to quickly model and design the desired system. The advantage of Omondo lies in its utilization from within Eclipse IDE. This enables the developer to access and respond to changes in architecture within a centralized resource. (http://www.uml2.org/)

2.5.5 Aspect Oriented Programming (AOP) using AspectJ

Aspect-oriented modeling (AOM) is an approach to design that focuses on conceptualizing, describing, and analyzing features that realize software dependability goals [48] [49]. Development of modern software systems is complicated due to the need to balance multiple competing dependability goals. These goals seek to (1) reduce the complexity when replacing alternatives features, (2) increase understandability, and (3) aid in the evolution of the system through its life-cycle. The advantages of AOM are that(1) the approach allows developers to conceptualize, describe, and communicate crosscutting dependability features as conceptual design units, (2) it allows changes to a dependability that feature can be made in one place, and can be elected by composing the changed aspect model with a primary model, and (3) isolating dependability features in aspects can lead to generalized solutions that are potentially reusable across different systems in an organization [25].

Aspects are modular units of crosscutting implementation that can be defined by aspect declarations. Declarations have a form similar to that of class declarations, in that they may include point cut declarations, advice declarations, as well as all other kinds of declarations permitted in class declarations.

AspectJ is a simple and practical aspect-oriented extension to Java. AspectJ extends Java with support for two kinds of crosscutting implementation [50] [23]. This makes it possible to define additional implementation to run at certain well-defined points in the execution of the program.

Below defines three main concepts of aspects, which are Join Points, Weaves, and Cross-Cutting Concerns [24].

- ***Join points***: are those elements of the component language semantics that the aspect program coordinates. Like nodes in the dataflow graph & runtime method invocations they are clear, but perhaps implicit, elements of the component program's semantics. The join point representation can be generated at runtime using a reflective runtime for the component language. In this approach, the aspect language is implemented as a meta-program, called at each method invocation, which uses the join point in formation and the aspect program to know how to appropriately marshal the arguments [24].
- Aspect ***weavers*** work by generating a join point representation of the component program, and then executing (or compiling) the aspect program with respect to it.
- ***Crosscutting concern*** can be described as behavior that cuts across the typical divisions of responsibility, such as logging or debugging a problem which a program tries to solve. It also refers to the aspects of a program that do not relate to the core concerns directly, but which proper program execution nevertheless requires [24].

2.5.5.1 IBM's Eclipse IDE

As stated on the Eclipse web site, Eclipse is a "*universal tool platform – an open extensible IDE for anything and nothing particular*" [51]. Eclipse is freeware that is obtainable as an open source software development platform that provides users with the necessary functionality to develop a wide range of applications as well utilize third-party plug-ins. (http://www.eclipse.org)

2.5.5.1.1 Eclipse ADJT Plug-in (2.0.2)

AspectJ Development Tools (AJDT) is the system implementation of AOSD that enables the Eclipse user to access and apply such functionalities as logging, error handling, and standards enforcement all while developing compliable code inside the Eclipse IDE. AJDT enables the user to create and step through aspect code, while enabling the user to graphically display and model aspects within the code.

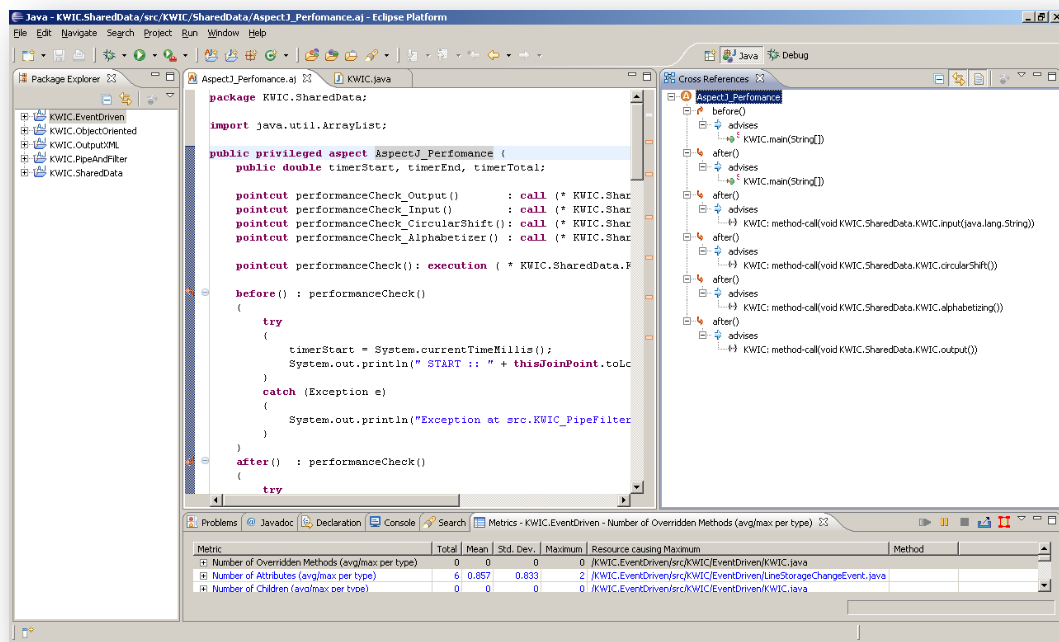


Figure 18 - Eclipse AJDT Plug-in

(http://www.eclipse.org/ajdt/)

2.5.5.1.2 Eclipse Metrics Plug-in (1.3.6)

As the Source-Forge website indicates, the metric plug-in "*provide[s] metrics calculation and dependency analyzer plug-in for the Eclipse platform. Measure various metrics with average and standard deviation and detect cycles in package and type dependencies and graph them*" [47]. This tool aided in the profiling of the projects for all metric calculations.

Metric	Total	Mean	Std. Dev.	Maximum	Resource causing Maximum	Method
Number of Overridden Methods (avg/max per type)	0	0	0	0	/KWIC.EventDriven/src/KWIC/EventDriven/KWIC.java	
Number of Attributes (avg/max per type)	6	0.857	0.833	2	/KWIC.EventDriven/src/KWIC/EventDriven/LineStorageChangeEvent.java	
Number of Children (avg/max per type)	0	0	0	0	/KWIC.EventDriven/src/KWIC/EventDriven/KWIC.java	
Number of Classes (avg/max per packageFragment)	7	3.5	2.5	6	/KWIC.EventDriven/src/KWIC/EventDriven	
Method Lines of Code (avg/max per method)	191	4.064	4.822	23	/KWIC.EventDriven/src/outputXML/CreateXMLFile.java	createFile
Number of Methods (avg/max per type)	46	6.571	7.89	25	/KWIC.EventDriven/src/KWIC/EventDriven/LineStorage.java	
Nested Block Depth (avg/max per method)		1.213	0.617	4	/KWIC.EventDriven/src/KWIC/EventDriven/LineStorage.java	writeToFile
Depth of Inheritance Tree (avg/max per type)		1.143	0.35	2	/KWIC.EventDriven/src/KWIC/EventDriven/LineStorageWrapper.java	
Number of Packages	2					
Afferent Coupling (avg/max per packageFragment)		0.5	0.5	1	/KWIC.EventDriven/src/outputXML	
Number of Interfaces (avg/max per packageFragment)	0	0	0	0	/KWIC.EventDriven/src/KWIC/EventDriven	
McCabe Cyclomatic Complexity (avg/max per method)		1.489	0.896	4	/KWIC.EventDriven/src/KWIC/EventDriven/LineStorage.java	getLineAsString
Total Lines of Code	346					
Instability (avg/max per packageFragment)		0.75	0.25	1	/KWIC.EventDriven/src/KWIC/EventDriven	
Number of Parameters (avg/max per method)		1.426	0.94	4	/KWIC.EventDriven/src/KWIC/EventDriven/LineStorage.java	setChar
Lack of Cohesion of Methods (avg/max per type)		0.143	0.226	0.5	/KWIC.EventDriven/src/KWIC/EventDriven/LineStorageChangeEvent.java	
Efferent Coupling (avg/max per packageFragment)		1.5	0.5	2	/KWIC.EventDriven/src/KWIC/EventDriven	
Number of Static Methods (avg/max per type)	1	0.143	0.35	1	/KWIC.EventDriven/src/KWIC/EventDriven/KWIC.java	
Normalized Distance (avg/max per packageFragment)		0.25	0.25	0.5	/KWIC.EventDriven/src/outputXML	
Abstractness (avg/max per packageFragment)		0	0	0	/KWIC.EventDriven/src/KWIC/EventDriven	
Specialization Index (avg/max per type)		0	0	0	/KWIC.EventDriven/src/KWIC/EventDriven/KWIC.java	
Weighted methods per Class (avg/max per type)	70	10	11.674	38	/KWIC.EventDriven/src/KWIC/EventDriven/LineStorage.java	
Number of Static Attributes (avg/max per type)	3	0.429	1.05	3	/KWIC.EventDriven/src/KWIC/EventDriven/LineStorageChangeEvent.java	

Figure 19- Eclipse Metrics Plug-in (http://metrics.sourceforge.net/)

2.6 Summary

Although Software Architecture may be newly recognized discipline in computing, many people have set precedence for its development and advancement. D. Harel's *On Visual Formalism* [52] first introduced and justified the utilization of graphical diagrams in clarifying design propositions. P. Kruchten expands on visual depiction of architecture in his paper, *Architectural Blue Prints – The "4+1" View Model of Software Architecture* [37] where "views" are used to identify concepts and communicate them to stakeholders by addressing functional and non-functional requirements. R. Kazman , Mary Shaw, and David Garlan can be credited for forging ground and seeding the concept of SA with their papers *Software Architecture* [15], *The Coming-of-Age of Software Architecture Research* [7], *an Introduction to Software Architecture* [11], and *Software Architecture: A Roadmap* [14]. These papers were instrumental

in developing the architectural development process and classifying architectural views and perspectives in the computing discipline.

For the purposes of this paper, Bredemyer Consulting has been a very important and reliable source for clarifying the subject matter. Bredemyer Consulting provides professional architectural education and consulting to prepare and educate developers for the architectural frameworks. This is demonstrated in their paper, *Architectural Resources – for Enterprise Advantage* [33], which provides the definition and recognition of software architecture from a high level perspective. It formulates and organizes SA into layers that can be communicated to the stake-holder and developers by citing and implementing an architectural decision framework. This paper also aids in the classification and indication of such properties as priority settings, system properties and cross-cutting concerns, system to context, and system integrity. Bredemyer consulting also identifies Meta-Architectural frameworks and defines the conceptual, logical and executive architecture [27] [33]. This concept is echoed by Microsoft's *Application Architecture Guide* [32], D.E. Perry, A.L. Wolf's *Foundation of the Study of Software Architecture* [17], and L. Chung, B.A. Nixon, E. Yu's *Using Non-Functional Requirements to Systematically Select among Alternatives in Architectural Design* [53].

Software prediction and analytics play a very large role in the solidification and justification of this discipline, and can be seen in C.H. Lung and K. Kalaichelvan's paper *An Approach to Qualitative Architecture Sensitivity Analysis* [54]. This paper researches quantitative and measurable evaluations as opposed to qualitative assessments and presents an empirical case study of metrics. While the paper focuses on the robustness of software, it also provides a comparative approach to analyzing and contrasting quality selections. This comparative approach is reverberated by A. Vasconcelos, P. Sousa, and J. Tribolet [44] where they define popular metrics used to determine quality in software. C. Stoermer, F. Bachmann, C. Verhoef [40], and A.J. Lattanze [28] pioneer the utilization of the software architecture comparison analysis methodology (SACAM) and the architectural centric development methodology (ACDM) for software selection. Software analysis is then broadened by J. Asundi and R. Kazman's [41]

introduction of the cost benefit analysis method (CBAM) and R. Kazman, L. Bass, G. Abowed, and M. Webb's [13] software architecture analysis method (SAAM). H. Leung and Z. Fan further define the effort required to develop software by introducing their *Software Cost Estimation* [55] process. These analytical methodologies are then compared and contrasted by L. Dobrica and E. Niemela's paper, *A Survey on Software Architecture Analysis Methods*, defining the differences in each.

The topic of Software Quality is imperative to the analysis methodology, and characterized by A. Sharma, R. Kumar, P.S. Grover paper *Estimation of Quality for Software Components – an Empirical Approach* [42] and expanded by V.S. Sharma and K.S. Trivedi's *Architectural Based Analysis of Performance, Reliability, and Security of Software Systems* [9] . Quality is an important topic in SA because it identifies and provides a balancing standard to measure and weigh solutions against. The concept of software quality can be spotlighted by the following authors: J.K. Blundell, M.L Hines, and J. Stach's paper *The Measurement of Software Design Quality* [56], M. Mattsson, H. Grahn, and F. Martensson *Software Architecture Evaluation Methods for Performance, Maintainability, Testability, and Portability* [29]; C. Hu, F. Jiao, and C. Zhao's *An Architectural Quality Assessment for Domain-Specific Software* [57]*;* M. Al Sharif, W.P. Bond, T. Al-Otaiby *Assessing the Complexity of Software Architecture* [10], A. Avritzer and E. Weyuker's *Metrics to Assess the Likelihood of Project Success Based on Architecture Reviews* [58]; and L.H. Rosenberg's *Applying and Interpreting Object Oriented Metrics* [59].

Another cornerstone on this paper revolves around the utilization of Aspect-Oriented Programming (AOP). AOP has gained considerable traction in the last decade due to its provision and clarity in evaluating and extending source at runtime. The practice of AOP in the analytics, as used by this paper, can be reviewed in such papers as R. Reddy, R. France, and G. Georg's *An Aspect Oriented Approach to Analyzing Dependability* [60]; K. Hoffman and P. Eugster *Towards Reusable Components with Aspects: an Empirical Study on Modularity and Obliviousness* [48]; G. Kiczales and E. Hilsdale's

Aspect-oriented programming [23]; and finally R. Pawlak, L. Seinturier, J.P. Retaille's *Foundation of AOP for J2EE Development* [24].

Finally, as the provider for the source-code, Dr. Denis Helic's course on Software Architecture at the Institute for Information Processing and Computer Supported New Media (IICM) at Graz University of Technology, in Austria [22] is instrumental in the production of the experiments for this research project. Dr. Helic's course material also provided both the concise disciplinary structure as to how SA is organized. The course's practical project assignments were used to draw out the concepts for the extension and modifiability of the experiments.

As we can see, although SA is a researched quarter of computing, academia lacks the proper empirical study of how and why to use SA when weaving an application or software suite. Research studies are yet to evaluate and identify how to thread the concept of SA into analytics and measure with quality attributes. Furthermore, academia lacks practical studies that strand and portray how to compute quality from source and therefore provide a known framework for evaluation. It is in this area that this research hopes to contribute to the computing discipline.

Chapter 3. Method

This research is motivated by several research questions including: (1) Which architectural style provides or promotes the best realization of each quality attribute in an implemented system?, and (2) Given a specific architectural specification, can we classify potential code realizations and rank them in terms of how they reflect the quality attributes represented in the architecture?

Since, intuitively we cannot gauge the quality of software until it is deliverable, there is a need to analyze source written to implement the same requirements, and then compare and measure the quality of each (refer to Figure 20). This paper presents an answer to the above questions by comparing and evaluating final implementations to gain perspectives ***on how the development of the system can aid in the advancement of future architectural projects.*** To realize our goal, this paper utilizes Java implementations written by Institute for Information Processing and Computer Supported New Media (IICM) at Graz University of Technology, Austria [22]. The source code implements the ***KWIC*** – Keyword in Context – that facilitates any line of text to be circularly shifted by repeatedly removing the first word and appending it at the end of line. The output is a listing of all circular shifts of all lines in alphabetical order providing a mechanism to index and thereby search for a context in a file.

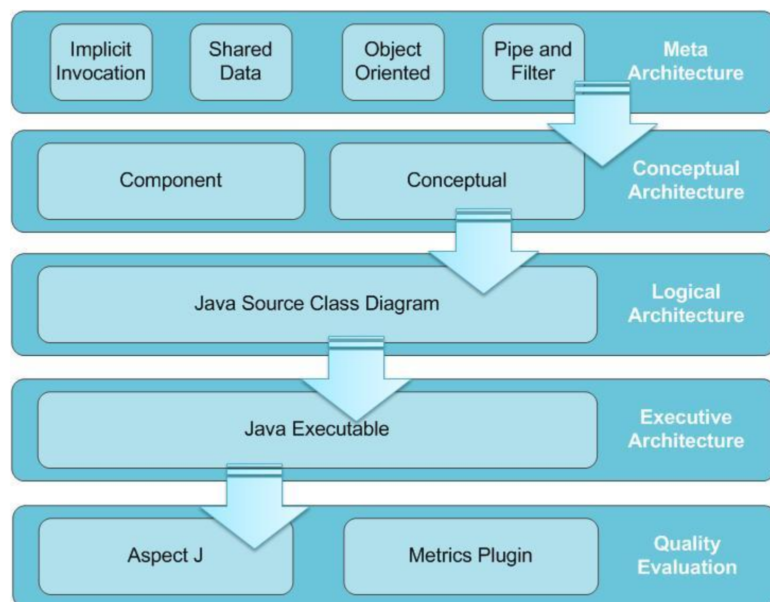


Figure 20- Experimental Progression

KWIC presents a sufficiently complex experimental source, written for the same purpose, but in different styles, and thereby aids in the understanding of how the selection of a style affects the final product.

To achieve this goal, the source is reversed engineered to create conceptual and logical architectural views as to understand in identifying point-cuts and cross-cutting concerns. Graphical presentation of the source enables the high-level view of the architecture before any of the source-code is evaluated.

The performance and reusability quality attributes are then measured by directly modifying the source and obliviously interjecting code using AspectJ. AspectJ is an aspect-oriented programming language that facilitates the identification, separation and representation of crosscutting software concerns [23]. AspectJ is a great fit for evaluating architectural styles, because it is a simple and practical aspect-oriented extension to Java. Implementing each quality variable using AspectJ code atop of the working architectural style enables noninvasive evaluation using the same controls for each system. Use of AspectJ also enables the reusability of the test cases themselves so that other systems and requirements can be numerically compared and contrasted.

Therefore, our experimentation pursues the following progression. First the performance of each baseline style is recorded by using AspectJ to record the performance of each style against different file sizes. Then, we can begin to modify and extend the source in order to measure the correlation of each quality attribute. Second, to do this, this research modifies the underlying code by implementing a Dictionary class and recording the amount of time required to test and deliver. The class verifies the file's text against a known (indexed) list of English words. If a word is not contained in the dictionary, then the word is truncated from being alphabetized and excluded from the result set altogether. The purpose of the Dictionary class is to simulate a feature change or amendment. Third, once we are able to gauge the affect of the amendment to source, we can begin to compute its reusability by extending each architectural style with a new functionality to the application. To judge the affect of source supplement, a XML output adds

to the process. The XML output functionality will simulate an addition to the existing source code. Data can then be collected on the time/effort and quality as a result of the amendment.

In the next section defines the source of research shedding light into its purpose and how it functions.

3.1 Key Word in Context (KWIC)

KWIC (Key Word in Context) index system provides a convenient search mechanism for information in a long list of lines, such as book titles, dictionaries, or online documentation entries. It is widely used for a web search engine and the permuted index for the Unix Man (help) pages [7]. The system accepts an ordered set of lines, each line is an ordered set of word, and each word is an ordered set of characters. Any line may be circularly shifted by repeatedly removing the first word and appending it at the end of line. The output is a listing of all circular shifts of all lines in alphabetical order [22].

For example, if the original text says "*Gone with the Wind*"

Circular shifts (key words underlined)

Gone with the Wind
with the Wind Gone
the Wind Gone with
Wind Gone with the

The input function is called to read and parse the lines from an input file. The input function then reads and processes these lines and represents them as characters and line. Then Shift function is called to circularly shift each particular line and store it in the circular shifts object. This is followed by the alphabetize function, which sorts circular shift object alphabetically. The result is alphabetically stored and finally outputted by printing to the terminal's screen.

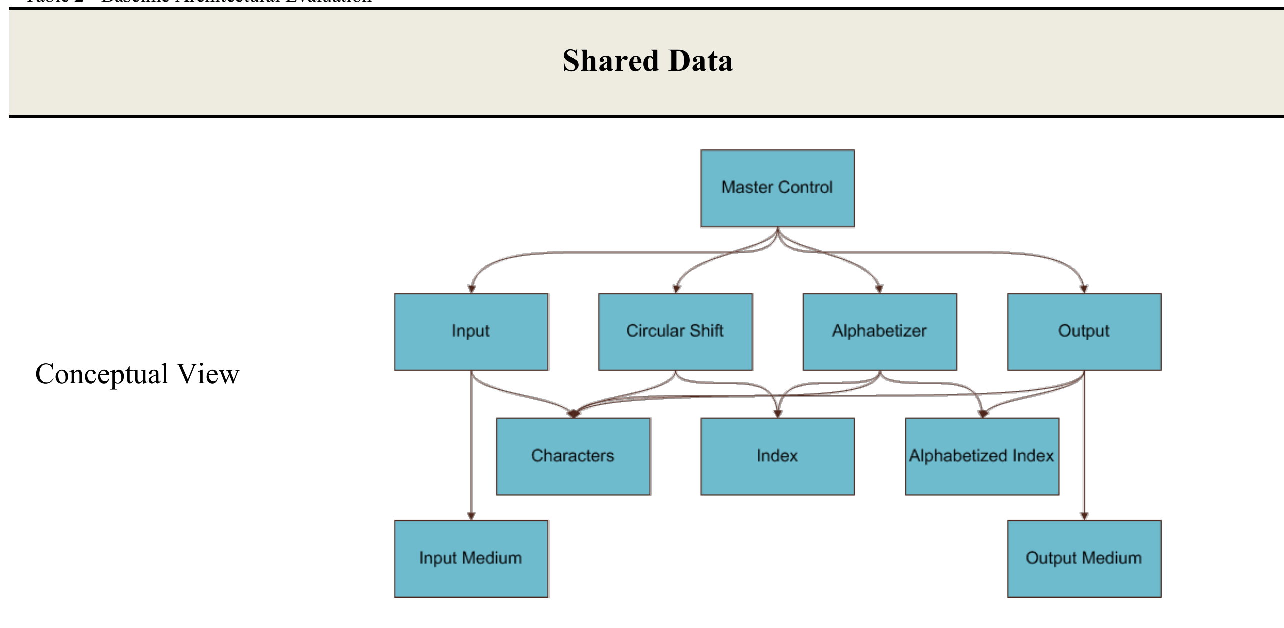
3.2 KWIC Architectures

In this section, four alternative architectural styles are presented and justified: shared data, object-oriented design, implicit invocation, and pipe and filter. Since our implementation did not include a client-server style, that model is excluded from this research. Conceptual and logical layout of these models is portrayed, and in doing so, assist in the theoretical organization and arrangement of KWIC within the framework of SA. The goal of this section is to ***layout the structure of each style within the framework of Software architecture*** as to identify and disparage similarities and contrasts. While each style differs, the basic components are similar according to the functions – including input, circular shift, alphabetizer and output. Yet, each implementation encompasses a unique architectural design methodology.

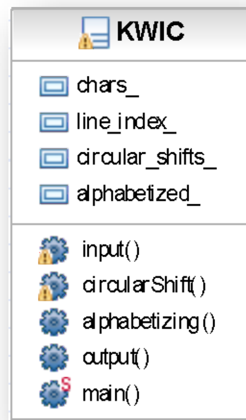
Table 2 – below – portrays each style (Shared-Data, Abstract-Data Type, Implicit Invocation, and Pipe-and-Filter) by visually exhibiting their conceptual and logical structures, and numerically classifying their source.

Source-code used in experimentation of this project can be found at: http://coronet.iicm.tugraz.at/sa/

Table 2 - Baseline Architectural Evaluation



Logical View

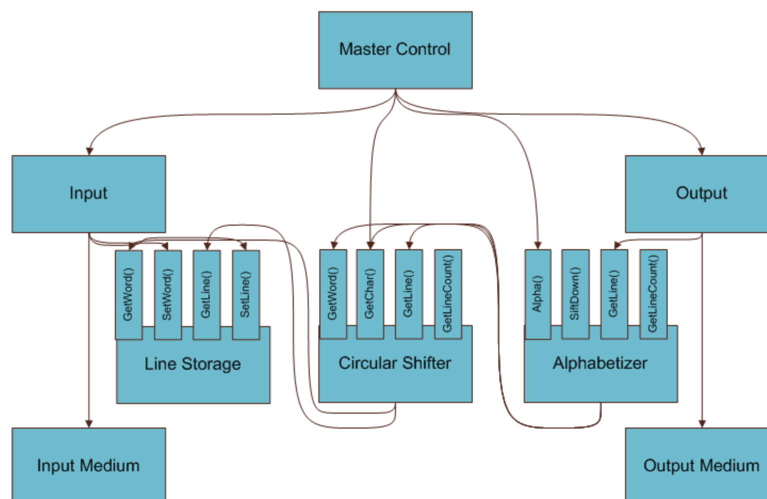


Metrics

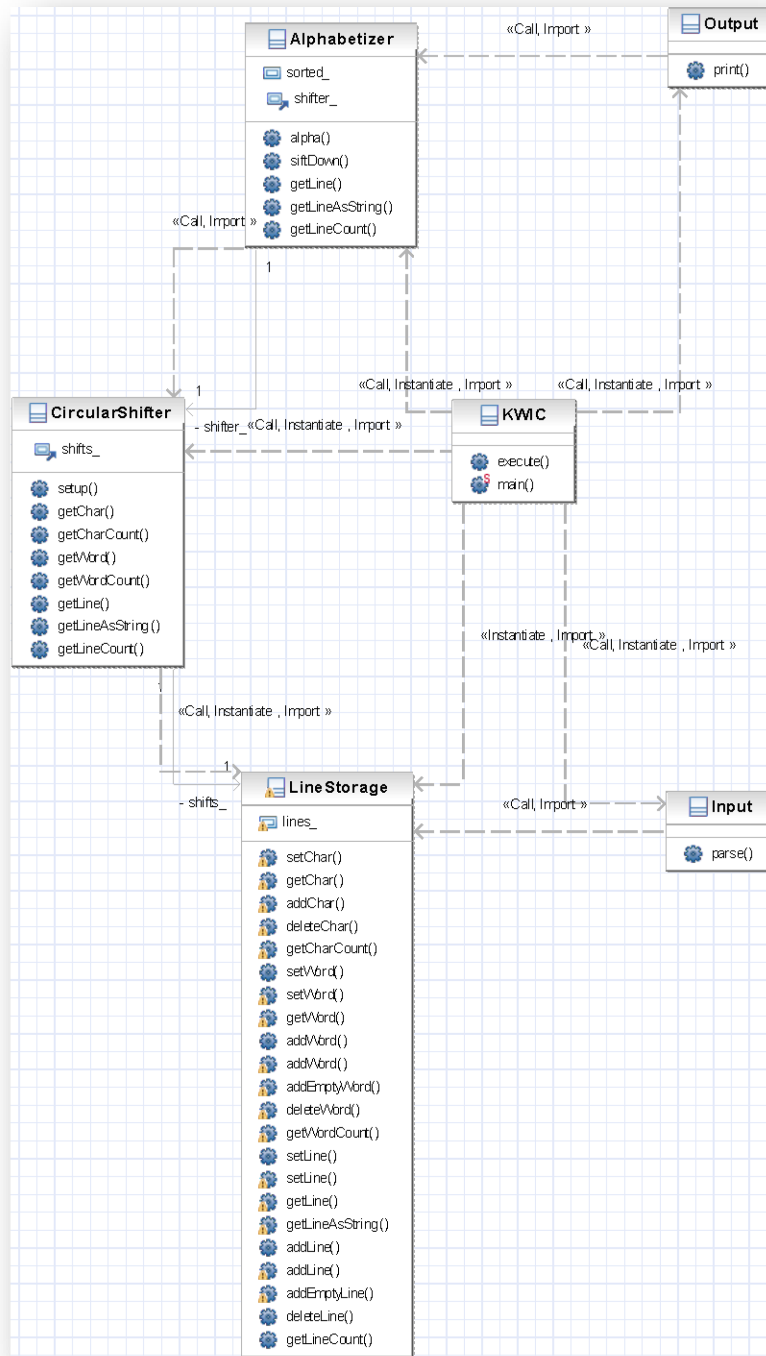
	Values
Number of Attributes	6
Number of Children	0
Number of Classes	1
Number of Interfaces	0
Number of Methods	5
Number of Overridden Methods	0
Number of Packages	1
Number of Parameters	5
Number of Static Attributes	0
Number of Static Methods	1

Abstract Data Type Design

Conceptual View



Logical View

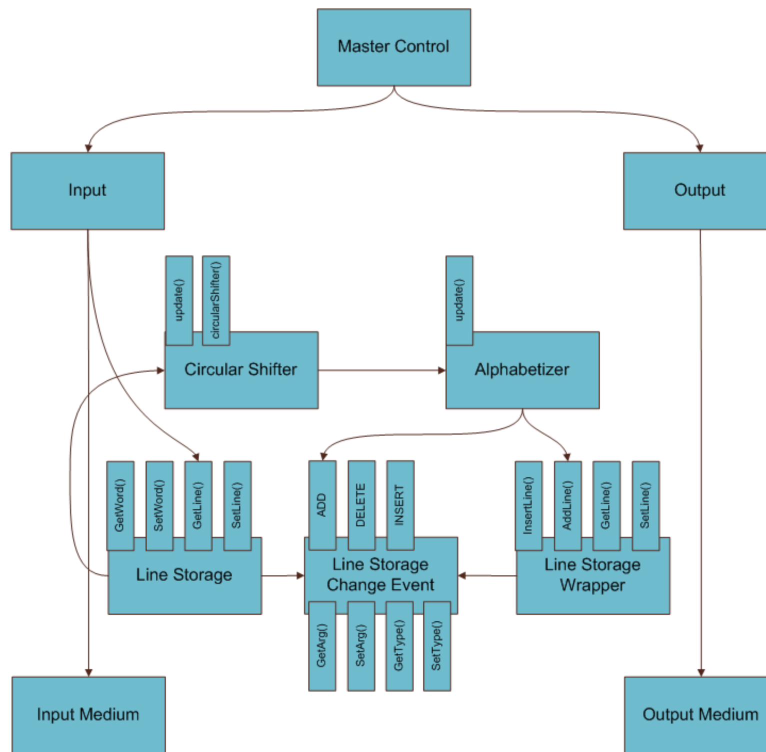


Metrics

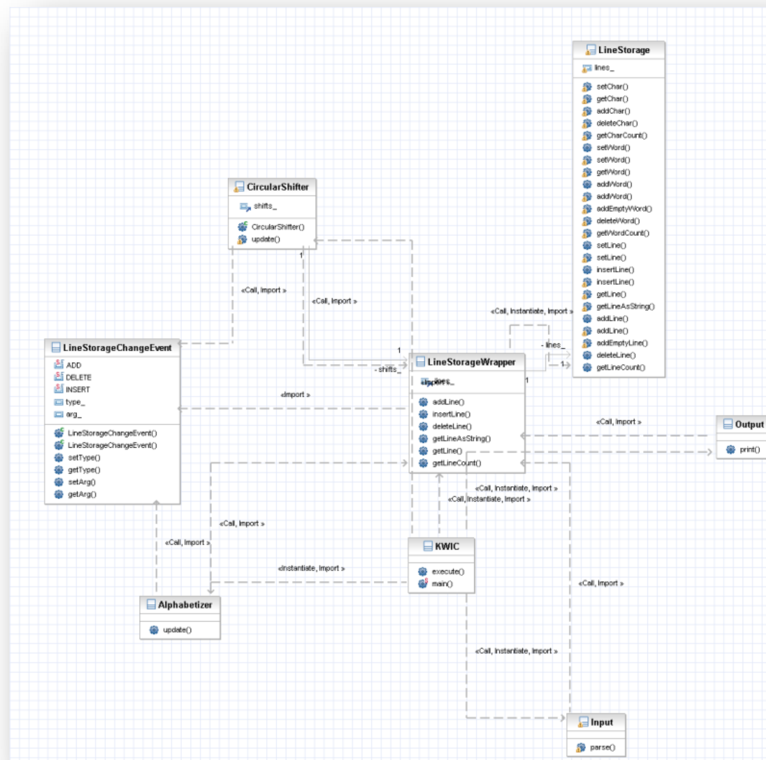
	Values
Number of Attributes	4
Number of Children	0
Number of Classes	6
Number of Interfaces	0
Number of Methods	38
Number of Overridden Methods	0
Number of Packages	1
Number of Parameters	61
Number of Static Attributes	0
Number of Static Methods	1

Implicit Invocation

Conceptual View



Logical Design

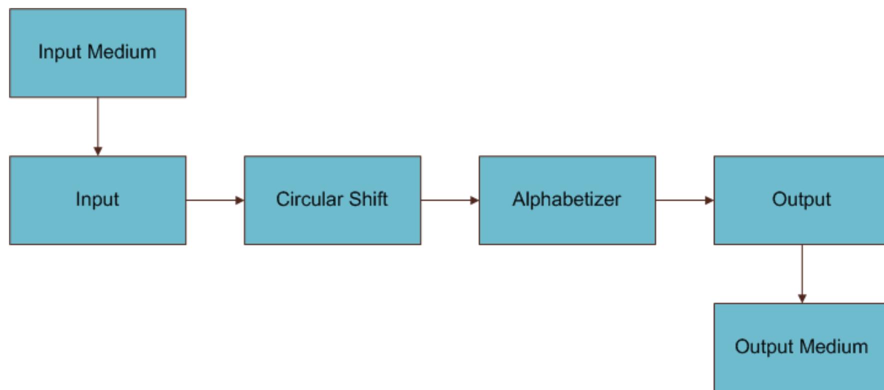


Metrics

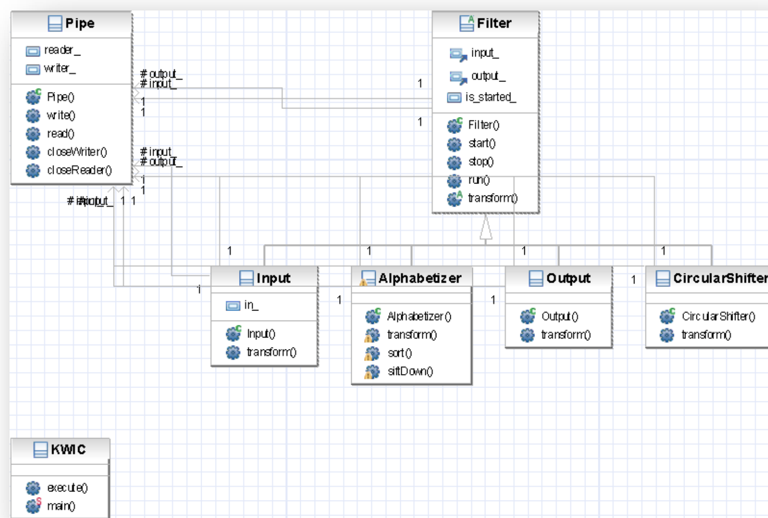
	Values
Number of Attributes	5
Number of Children	0
Number of Classes	8
Number of Interfaces	0
Number of Methods	42
Number of Overridden Methods	0
Number of Packages	1
Number of Parameters	65
Number of Static Attributes	3
Number of Static Methods	1

Pipe-and-Filter

Conceptual View



Logical Design



Metrics

	Values
Number of Attributes	6
Number of Children	5
Number of Classes	8
Number of Interfaces	0
Number of Methods	24
Number of Overridden Methods	0
Number of Packages	1
Number of Parameters	19
Number of Static Attributes	0
Number of Static Methods	1

Chapter 4. Experiments and Results

As we have stated in this paper, the purpose of this research is to (1) identify architectural styles which provides the best realization of each quality, and (2) classify and rank potential code realizations based on the quality attributes represented in the architecture. We proposed to answer these questions by (1) defining conceptual and logical architecture of each style, and (2) introducing quality metrics to weigh them against. Using this basic approach, three experiments were designed to address performance, modification, and extensibility. Experiment 1 measures the performance by timing the amount of time it takes to process a set of file sizes. Experiment 2 analyzes the ability to modify each style by changing the source-code to include a dictionary lookup on each word before they are used as indexes. Finally, Experiment 3 extends the source-code by adding aspects written in AspectJ, such that the outputted indexes are saved into an XML file.

4.1 Experiment – 1 – Performance

To evaluate performance, the execution of each architectural style was timed. The code to perform this timing was introduced into the program using an AspectJ aspect (see Figure 22 and Appendix A). However, before any modifications or extensions are applied to the source, metrics were collected for each program. Each approach's total time of execution was then recorded against each inputted file size. The file sizes are shown in Table 3. The sample file was then multiplied in size, by copying-and-pasting its content in order to simulate a workload amplification. A sample of the content is as follows – Figure 21:

```
Gone With The Wind

Brave heart

The Godfather

Avatar
```

Figure 21- Sample Data File

Below, table 3 describes the sample file sizes that is used and one such case of AspectJ interjection:

Table 3- Input File Sizes

	A Input	B Input	C input	E input	F input	G input	X Input	Y input
Input File (kb)	6 kb	33 kb	66 kb	132 kb	394 kb	788 kb	1707 kb	9449 kb

Figure 22 portrays an example of AspectJ code used to gather performance measures from the source:

```
public privileged aspect AspectJ_Performance {

        public double timerStart, timerEnd, timerTotal;

        pointcut performanceCheck(): execution ( * KWIC.EventDriven.KWIC.main(..) );

        before() : performanceCheck()
        {
                try
                {
                   timerStart = System.currentTimeMillis();
                   System.out.println(" START :: " +
               thisJoinPoint.toLongString() + Double.toString(timerStart));
                }
                catch (Exception e)
                {
                   System.out.println("Exception at "+  e.toString());
                }
        }
        after()  : performanceCheck()
        {
                try
                {
                   timerEnd = System.currentTimeMillis();
                   timerTotal = timerEnd - timerStart;
                   System.out.println(" END :: " + thisJoinPoint.toLongString() + " :: " +
                                         Double.toString(timerStart) + " :: TOTAL TIME :: (" +
                                         Double.toString(timerTotal) + ")");
                   System.exit(1);
                }
                catch (Exception e)
                {
                   System.out.println("Exception at "+  e.toString());
                }
        }
```

Figure 22- AspectJ Sample Source

Table 4 and 5 describe the time recorded by running the above aspects against each architectural style (milliseconds & in seconds):

Table 4-Performance of Application based on variable file sizes

In Milliseconds								
Event Driven	366.4	8616.5	32086	136962	1297883	25500000	9999999999	9999999999
Pipe-and-Filter	339.5	858.3	1308.9	2354.4	3304.8	3336.9	3257.6	*-1*
Object-Oriented	119.3	433.6	797.2	1768.6	5661	9955.3	21760.3	*-1*
Shared-Data	142.2	260.6	448.5	920.2	4676.5	14132.3	82385.5	*-1*

Table 5- Performance of Application based on variable file sizes (in seconds)

In seconds							
0.3664 Sec	8.6165 Sec	32.086 Sec	136.96 Sec	1297.88 Sec	25500 Sec	1E+07 Sec	1E+07 Sec
0.3395 Sec	0.8583 Sec	1.3089 Sec	2.3544 Sec	3.3048 Sec	3.3369 Sec	3.2576 Sec	*-0.001 Sec*
0.1193 Sec	0.4336 Sec	0.7972 Sec	1.7686 Sec	5.661 Sec	9.9553 Sec	21.76 Sec	*-0.001 Sec*
0.1422 Sec	0.2606 Sec	0.4485 Sec	0.9202 Sec	4.6765 Sec	14.1323 Sec	82.386 Sec	*-0.001 Sec*

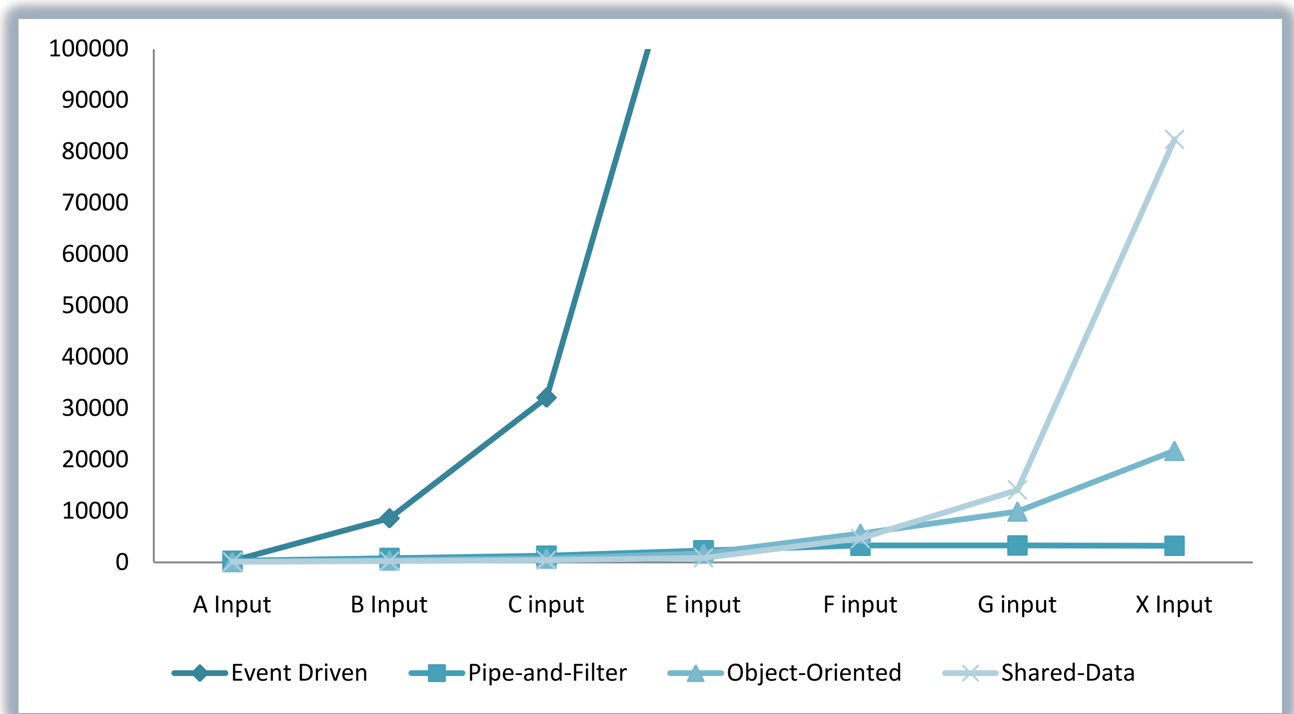


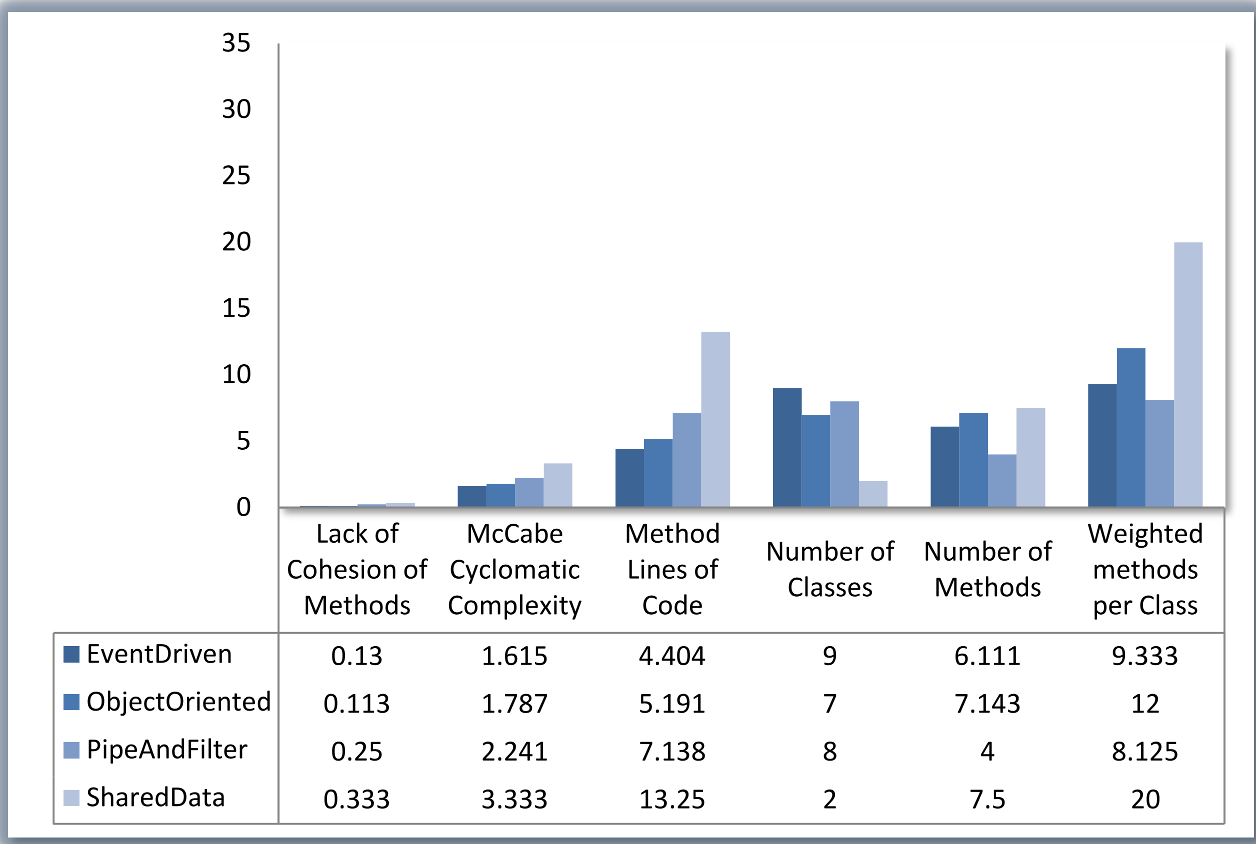
Figure 23- Performance of Architectural Styles

All approaches fail to instantiate after the buffer grows past 1707kb due to the restraints of the array buffer. The application's alphabetizer or circular shifter steps simply overflow and the steps are

prematurely existed therefore we can designate the maximum file size that can be inputted into the application at 1.66MB.

Figure 23 shows execution time as a function of file size for the four architectural styles. As we can see, while Pipe-and-Filter and Object Oriented approaches grows fairly evenly as the input grows, Event-based and Shared-data approaches bloat exponentially toward infinity. The Shared-Data approach performs better when the file size is relatively small, but quickly skyrockets as the instructions increase. The Event-Driven approach performs the worst, quickly reaching infinity. This clearly shows a deficiency in performance for this architectural style due to its lack of support for process batching, and therefore must be threaded. This demonstrates – numerically – the logical inference that event based systems should not architected for performance heavy requirements. Furthermore, the results reveal the accuracy of the test methodology and AspectJ usage.

In order to identify how and why one approach performs better than the other, we must now decompose the style into its components. We can infer that the Event-Based architecture lacks performance because of batches, but how can we translate that to a measurable value? To do so, we can hypothesize the utilization of metrics defined earlier (refer to Section 2.4.2) to compare the underlying code. Is there a lack of cohesion, or coupling? Or, is one source simply more complex, and therefore subjected to latency?



	Lack of Cohesion of Methods	McCabe Cyclomatic Complexity	Method Lines of Code	Number of Classes	Number of Methods	Weighted methods per Class
EventDriven	0.13	1.615	4.404	9	6.111	9.333
ObjectOriented	0.113	1.787	5.191	7	7.143	12
PipeAndFilter	0.25	2.241	7.138	8	4	8.125
SharedData	0.333	3.333	13.25	2	7.5	20

Figure 24 - Experiment - Performance - Metrics Evaluation

Figure 24 graphically conveys and compares each style's quality average metric scores. As we can see, although the OO and Pipe-and-Filter approaches contain a high number of parameters, methods, and classes they are able to sustain a linear relationship with the size of input. Pipe-and-Filter approach faired the best, only slightly increasing in latency as the file size was enlarged. The Shared-Data seems to be implemented in a simpler fashion (less parameters and classes), yet it lacks the abstractions needed to sustain the approaches when the number of instructions and processes increase, and is evident in the LCOM measurement – Lack of cohesion. With the cohesive mark near 1, the methodology shows that its methods are less related to one another. All the while, OO and Pipe-and-Filter approaches contain higher Weighted and Cyclomatic complexity, with more method lines of code and nested depth.

Now that we understand the behavior of the base-line code, what if we were to modify the code? How easily can we append a class or functionality? How easily can we modify an existing functionality?

4.2 Experiment – 2 – Modifiability

Style	Hours
Shared Data	43.5
Object Oriented	4.73
Event-Based	5.25
Pipe-and-Filter	7.25

As stated earlier, we now commence in the gathering of information to measure the effectiveness and modifiability of the architectural styles by adjoining a dictionary (refer to Figure 25) lookup to each style. The appending of the Dictionary class represented a modification, which is a routine process in software development – refer to Appendix C. During the implementation of the dictionary, the prior metrics that pointed to the lack of cohesion and elegance of the underlying code became evident. While the OO and Event-Based styles could be easily modified due to their cohesion and inheritance, the Shared Data and Pipe-and-Filter implementations were difficult to decode and test. This was especially true for the Shared Data implementation in which two arrays of data (one for the actual data and one for the end-of-line index) had to be wrapped in *for* loops every time any data needed to be extracted or reordered. This causes the implementation to be prone to more errors and null pointers, and therefore took longer to test and debug. As a result the modification of the source took exponentially longer to put in place, as explained in the table above.

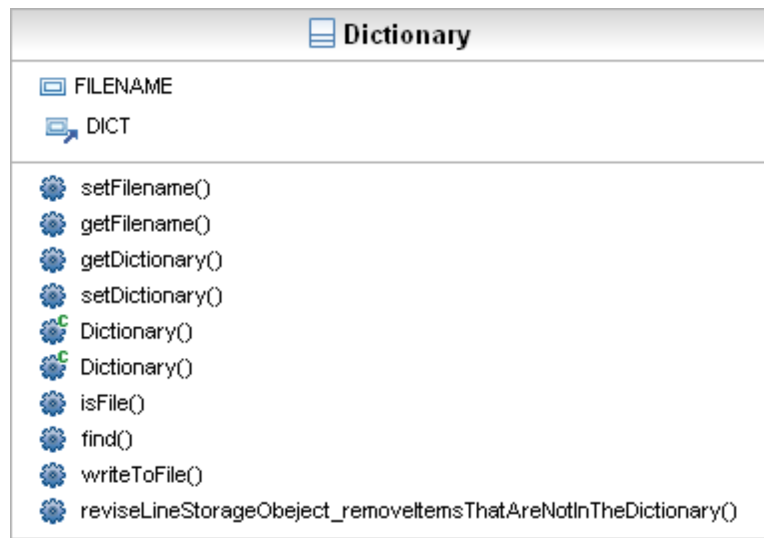


Figure 25 - Dictionary Class

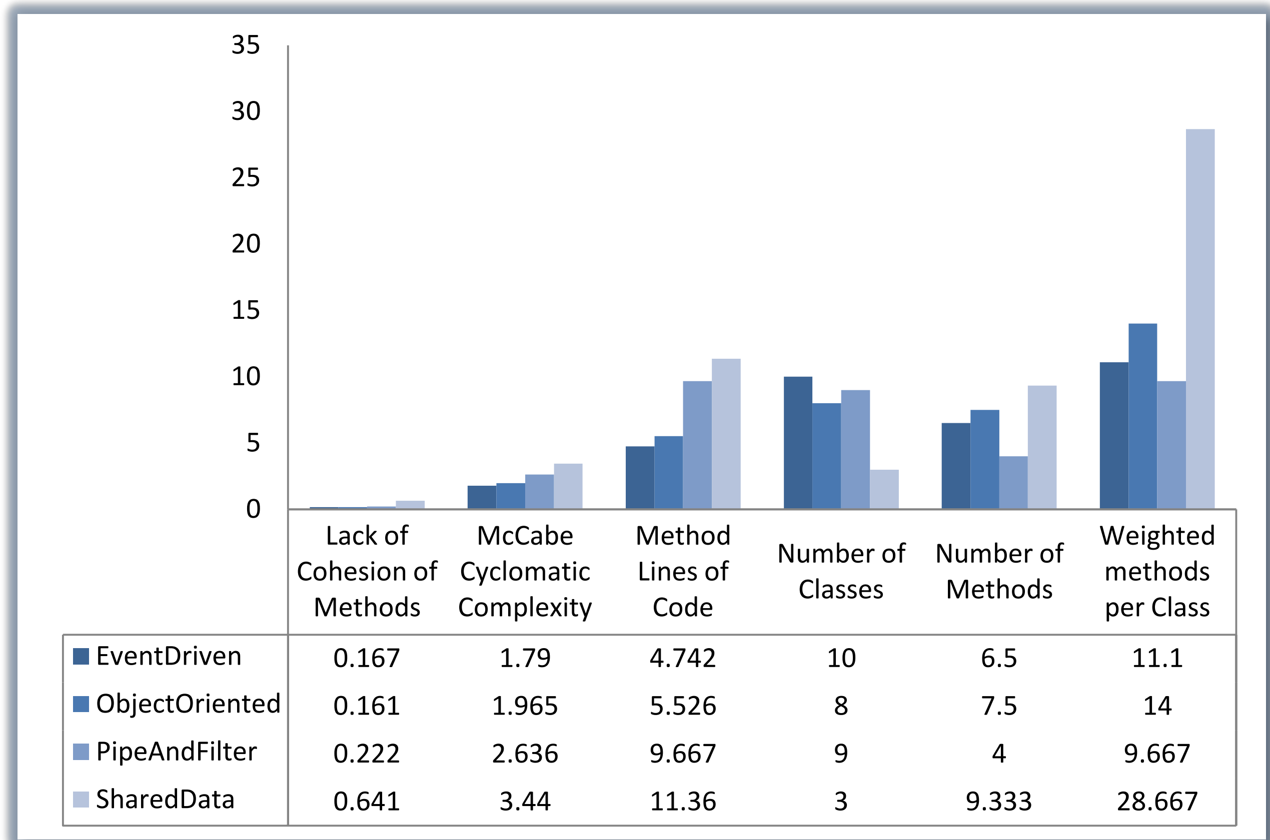


Figure 26 - Experiment - Modifiability - Metric Evaluation

As figure 26 shows, by modifying the source, all average metrics rose significantly. Most notably the average WMC complexity of the Shared-Data climbed from 20 to 28. We can assert that as a result of the modification, the source actually rose in complexity and decreased in cohesion. As a result, we can draw a correlation between the effort to alter source and the complexity and cohesion of the style. The higher the complexity, and the lower the cohesion, the longer it will take to modify. Shared-Data and Pipe-and-Filter register the highest WMC, McCabe Complexity, and LCOM, followed by Object-Oriented and Event-Based.

4.3 Experiment – 3 – Extensibility

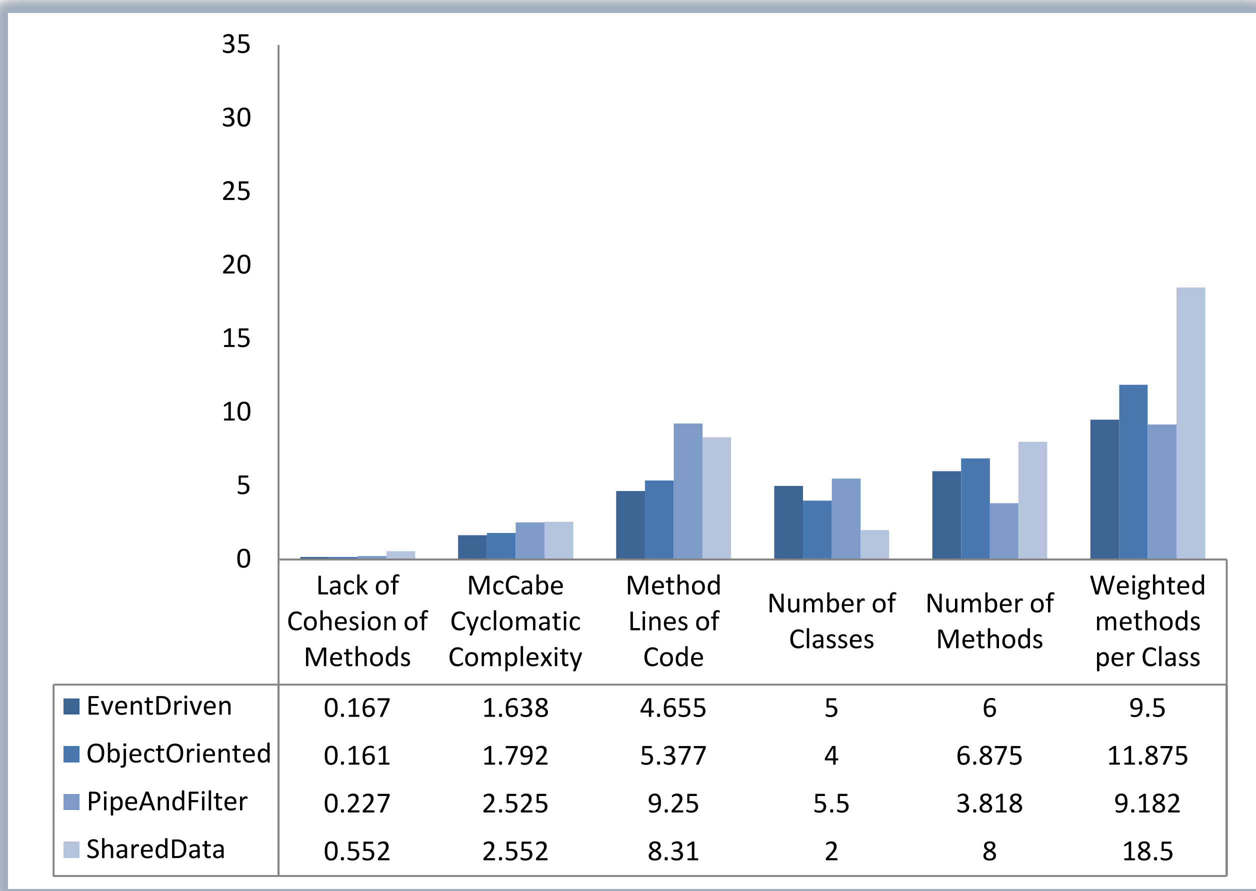
Style	Hours
Shared Data	.50
Object Oriented	.75
Event-Based	1.25
Pipe-and-Filter	2.5

In this experiment the extensibility of the programs were evaluated by appending a new functionality to the application. The specific functionality chosen was that of changing from console output to output to a XML file (refer to Appendix D and Figure 27). This is typical in any software development environment, where additional functionalities are constantly requested and implemented. Yet in this experiment, rather than modifying the already established source – as we demonstrated in experiment 2 – this experiment simply appends or extends to the already functional module. This provides a clear boundary for separation of concerns and does so noninvasively. Data can then be collected on the time/effort and quality as a result of the amendment.

+ CreateXMLFile
FILENAME OUTPUT
+ setFilename() + getFilename() + CreateXMLFile() + CreateXMLFile() - createFile()

Figure 27 - CreateXMLFile

As experiment 2 portrayed, the amendment of source by modification, is a formidable and time consuming process – the shared-Data modification took almost 43 hours to complete. But the simple appending of functionality using aspect proved to be much less expensive. Once the AspectJ source was established, weaving of the code on top of the existing source took considerably less time. Testing and debugging the code was also as trivial, since the the functionality of the source was not modified to achieve the results. Due to simplicity of the Shared-Data style (lack of inheritance) the injection of XML output at the end of the routine took only a halve hour, while the complexity and modularity of the Implicit Invocation style took longer to implement.



	Lack of Cohesion of Methods	McCabe Cyclomatic Complexity	Method Lines of Code	Number of Classes	Number of Methods	Weighted methods per Class
EventDriven	0.167	1.638	4.655	5	6	9.5
ObjectOriented	0.161	1.792	5.377	4	6.875	11.875
PipeAndFilter	0.227	2.525	9.25	5.5	3.818	9.182
SharedData	0.552	2.552	8.31	2	8	18.5

Figure 28 - Experiment - Extensibility - Metric Evaluation

As figure 28 portrays the effect of the AspectJ to the baseline code is minimal as compared to Figure 24. The LCOM, McCabe, MLOC, NOC, NOM, and WMC remain near the original values of the baseline source (refer to Section 4.1). This shows the convenience and prevalence of AOP when weaving two packages together. By utilizing AOP the source was able to adapt to a new function without adversely affecting the underlying source.

Chapter 5. Discussion and Limitations

This section compliments the results and inferences presented in Chapter 4 with lessons learned and limitations of the experiments conducted in this research. In doing so, we will show how performance is both a variable of the architectural selection and developmental precision; identify architectural style most difficult to modify; and, show how extensibility benefits from understandability and uniformity of the underlying source-code.

5.1 Limitations and Future Work

Although the goal of this project is desirable and the results are encouraging, the research is limited in its capacity to be generalizing across the computing discipline. Since only one application was evaluated, we cannot assert the finding as true across all software systems. In order to establish a standard for future design and development, both small and large scale packages should be evaluated using the framework discussed in this paper. To facilitate generalization of the results across a broader range of software systems, several research avenues need to be pursued.

While this research evaluated quality attributes for one application designed in four different architectural styles, future research should evaluate alternative code realizations for each architectural style. For example, one experiment could select the Event-Driven architectural style, but develop four different source programs for this style. This would simulate the many different ways we can design the same application under one framework. Secondly, future work should expand the experiments to encompass other quality attributes such as reusability, security, testability, functionality, and usability. By evaluating other quality attributes under the same scrutiny as the ones discussed in this paper, we may be able to better quantify early design decisions. More in depth analysis and correlation of the quality metrics should also be done to understand the effect of source development on the overall structure of an application. Finally, other varied experiments for each quality attribute should be postulated and tested.

For example, in this paper we chose to measure modification by adding a dictionary lookup, while future experiments can verify source by examining the effect of rewriting the *Alphabetize* function all together. The experiment can explore such modifications as reversing or randomizing the queue rather than alphabetizing. By having different competing actions for the same experiment we can normalize very difficult modifications or extensions from trivial ones.

5.2 Experiment – 1 – Performance

Experiment 1 exposed the limitation of the event-based architectural style to accommodate performance when the input increases above a few kilobytes (from 6kb to 66kb). The program used exhibited the style's inherent vulnerability in processing large number of instructions or processes. Each alphabetize or circular-shift method invocation triggers an independent event that is queued until its predecessor is completed. As the number of events grows, so does the queue, causing concurrent transactions to overlap and block processes from completing. In Chapter 2, Table 1, we predicted Event-based systems to be the least suitable when efficiency is required, and our experiment appears to be in line with that literature review.

A surprising result of Experiment 1 is the lack of performance when utilizing the shared data style. Theoretically, data-centric architectural style is well suited for volume processing by virtue of the data being stored "close" to the processes accessing or acting upon it. Data selection and amendment occurs within the same container, allowing for faster seek and update. Temporary object do not have to be instantiated and disposed when computing or interacting with the data. Yet, when the results of Experiment 1 are analyzed, we discover that the Shared-Data approach optimally executes when the input content is relatively small (from 6kb to788kb). As the input file size grows, the performance of the style decreases exponentially. As a result a further inspection of the source code is necessary to indentify factors that may cause this delay.

In order discover the root to the unexpected results of Experiment-1 we gathered quality metrics on the baseline source. By evaluating the metrics we are able to infer the unexpected latency of the data-centric style to be the result of complexity and lack of cohesion of the source-code. While the data-centric methodology may generally lead to better performance, the implementation of the KWIC application was filled with complex and nested *for* loops which raise the McCabe Cyclomatic Complexity to over 20. As a result, when the file size is relatively small, the complexity is not apparent. Yet, as the input content grows in volume, so does the complexity, and therefore the sharp decrease in performance. Furthermore, Pipe-and-Filter and Object-Oriented styles were not recognized as efficient in literature, yet Pipe-and-Filter performs the best, while Object-Oriented style grew proportionally to the size of input.

Given these results, we can assert that performance of a style is both a variable of the architectural selection and developmental precision. We can clearly identify Implicit Invocation as structurally inept in high-volume processing, yet distinguishing behavioral attributes across the other styles is unclear. In order to distinguish the remaining styles more granularly, the actual implementation of the source becomes an important variable in the overall performance of end result.

5.3 Experiment – 2 – Modifiability

In Experiment 1 we learned that while architectural selection coupled with developmental approach may resolve the problem at hand, the lack of cohesion between classes and methods that result, can cause significant challenges when the application is faced with large inputs or processes. Experiment 2 supplements the argument for complexity by presenting the effort required to modify the underlying style. Experiment 2 aims to gauge the modifiability of a style given a common software request for a feature change. Feature changes and modifications are frequent demands of the stakeholder. Experiment 2 shows that while design and implementation may have initially taken less time to develop, the modification of the incoherent and complex source, takes a significantly higher amount of time to

accomplish. The Shared-Data approach took approximately 40% longer to amend than all approaches combined.

In Chapter 2, Table 1, we predicted that the Shared-Data, Pipe-and-Filter, and Object-Oriented architectural styles would be less maintainable, but we did not recognize the scale of the maintenance. Experiment 2 demonstrates the ease of changing Event-Based or Object-Oriented architectural styles due to their understandability and separation of concerns. Experiment 2 also portrays the considerable difficulty in modifying the Pipe-and-Filter and Shared-Data style because of the adverse affect of any modification on the rest of application. When one unit of the application is modified, it can have an unintended effect on all other processes of the system. The quality metrics corroborate this assertion by displaying the highest McCabe Cyclomatic Complexity and Lack of Cohesion for the Shared-Data and Pipe-and-Filter styles.

As a result, we can ascertain that data-centric architectural design is the most difficult to modify, followed by Pipe-and-Filter, Implicit Invocation, and finally Object-Oriented. The understandability and cohesion of the source has a clear affect on the modification, testing, and debugging of application.

5.4 Experiment – 3 – Extensibility

While Experiment 2 portrays the challenges in the modification of source, Experiment 3 aims to extend and adapt the existing code obliviously. By using AOP, Experiment 3 shows the ease of adapting existing code by appending AspectJ atop of the “working” source. This yielded a considerable advantage in modification to encompass a new feature implementation. First, by obliviously weaving the two packages together, we could ensure the quality of each package was not inadvertently affected in the process. This reduces testing and debugging processes, and thereby increasing efficiency and isolating packages in separation of concerns. Second, the implementations follow almost the same “glue” source-code with less lines of source, ensuring consistent evaluation of source across all styles. Finally, and most

importantly, the clear advantage of injecting oblivious source extension is the substantial overall reduction in development effort.

In Chapter 2, Table 1, we predicted that the Object-Oriented and Implicit Invocation styles would award the least effort in reuse. We expected the same understandability and separation of concern characteristics to yield ease of reuse or extension, yet the results of Experiment 3 show otherwise. When utilizing AOP methodology, the elaborate structure of the Event-Driven and Pipe-and-Filter implementations prompt more research in identification of join-points, while Object-Oriented and Shared-Data style's join-points can be easily classified from the *main* function. The OO and Shared-Data methods – *Alphabetize* and *Shift* – can be easily identified, and instantiating storage variable – *Char[][]* or *ArrayList<String>* - can be effortlessly read from and written to using aspects. Pipe-and-Filter implementation necessitated the utilization of the customized *Pipe* and *Filter* objects when interacting with the package, therefore adding to the effort required in extension.

As a result, we can imply that when utilizing aspects to weave packages, extensibility benefits from understandability and uniformity of source**.** Furthermore, we can rank style selection in the following order: Shared-Data, Object-Oriented, Event-Based, and Pipe-and-Filter.

Chapter 6. Conclusion

To meet market demands and stay competitive, systems are routinely pieced together without consideration of the overall software quality. Often, performance, modifiability, or maintainability are sidelined for an expidiate time to shelf. As a result, today's computing landscape is littered with brittle applications with high maintenance costs and inadvertent complexity. In this paper we introduced software architecture as a promising discipline aimed to answer these concerns. To understand how it can become a viable answer, we decomposed software architecture into its unique styles and quality attributes. Alternative styles were then compared by utilizing the KWIC – Keyword in Context – application written in different architectural styles. These styles were then assessed and compared to (1) analyze the performance of each style, (2) compare the effects of source modification to each style, and (3) gauge the effort and outcome of extension of functionality.

In doing so, we are able to ascertain which architectural style provides the best realization of performance, maintainability, and extensibility quality attributes. In Experiment 1, we showed how performance of a style is both a variable of the architectural selection and developmental precision. We – numerically – identified Implicit Invocation as structurally unable to handle high-volume processing, and showed how the actual implementation of the application affects the overall performance. Experiment 2 validated the association of understandability and cohesion of source with the effort required to modify, test, and debug – Data-Centric architectural ranks as the most difficult to modify, followed by Pipe-and-Filter, Implicit Invocation, and finally Object-Oriented. Experiment 3, exhibited the advantage of understandability and uniformity of source in achieving an extension to the application – Shared-Data ranking as the most extensible followed by Object-Oriented, Event-Based, and Pipe-and-Filter. The summation of these experiments has enabled the classification of potential alternative designs.

Most importantly, this research has established the foundation and framework for capturing metrics and analyzing source-code. Prior to this research, academia lacked the stepwise process to evaluate and compare architectural styles. Although theoretical axioms guided the evaluation of software architecture, they did not explain how or why to choose one path over another. Through experience, developers and architects had maintained a general rule for selection and evaluation of opposing styles, but no real numerical data was available to corroborate the experience. As a result, although many developers understand the importance of design, they can not quantify the consequence of early decisions on the final product. This project enables researchers to travel down a known path to gauge and compare various architectural styles, and thereby in the future accumulate enough knowledge to fortify and justify a standard approach to alternative selection. By developing and demonstrating a systematic approach to decomposing and evaluating each style, we can now begin to capture and collect data on a wide variety of systems so that collectively we can prevent brittle and costly implementations.

In addition to constitution appraisal methodology, this project explored the emergence of Aspect Oriented Programming and established its significance in the maturity of software architecture as a discipline. This research has shown its potential for noninvasive extension of source and standard measurement of performance. By utilizing AOP to gather data on source, we were both able to evaluate performance uniformly across all styles, and opened the opportunity for leveraging this framework for other computational statistics. Furthermore, this project identified the simplicity of utilizing AOP to mend isolated packages without inadvertently harming the underlying source. This is an important factor in combating complex and wide-breadth systems which are routinely stacked and merged onto one another. By manipulating AOP we were able to maintain the boundaries between packages while achieving our primary objectives.

As we have seen, the software architecture has the potential to extend the computing industry into a mature discipline. To do so, SA must be proven to be a viable solution to the growing complexity of the computing landscape. This research has shown how to approach this difficult problem by introducing and

demonstrating a framework for comparison of design alternatives. As more and more systems are decomposed and assed using this road-map, we can cultivate a verifiable association with system architectures and there long term quality.

Chapter 7. Works Cited

[1]. *Information Techonology and Innovation at Shinsei.* **Upton, David and Fuller, Virginia.** Boston, Ma : Harvard Business Press, Oct 4, 2007. 9-607-010.

[2]. *Information Systems at First Caribbean: Choosing a Standard Operating Environment.* **Beaubien, Louis and Mahon, Sonia.** Ontario, Canada : Ivey Managment Services, 2004, Vol. 9B04E032.

[3]. *Business Intelligence Software at SYSCO.* **Mcafee, Andrew and Wagonfeld, Alison Berkley.** Boston, Ma : Harvard Business School, Sept 11, 2006. 9-604-080.

[4]. *Amazon.com: The Brink of Bankruptcy.* **Applegate, Lynda M.** Boston, Ma : Harvard Business Press, April 2, 2009. 9-808-014.

[5]. *Andina Bottling Co.* **Narayanan, V G.** Boston, Ma : Harvard Busniess Press, Oct 1, 2002. 9-102-040.

[6]. *Information Techonology at COSCO.* **McFarlan, Warren, Guoqing, Checn and Lane, David.** Boston, Ma : Harvard Business School, Nov 21, 2005. 9-305-080.

[7]. *The Coming-of-Age of Software Architecture Research.* **Shaw, Mary.** 2001, IEEE , p. 656664.

[8]. **Cringely, Bob.** A History of a Computer. *The Nerds.* [Online] PBS Broadcasting. [Cited: 02 022, 2010.] http://www.pbs.org/nerds/timeline/.

[9]. *Architecture based analysis of performance, reliability and security of software systems.* **Sharma, Vibhu Saujanya and Trivedi, Kishor S.** Palma, Spain : ACM, Proceedings of the 5th international workshop on Software and performance, 2005\. 1-59593-087-6 .

[10]. *Assessing the complexity of software architecture.* **AlSharif, Mohsen, Bond, Walter P and Al-Otaiby, Turky.** s.l. : ACM Southeast Regional Conference Proceedings of the 42nd annual Southeast regional conference , 2004. 1-58113-870-9 .

[11]. **Garlan, David and Shaw, Mary.** *An Introduction to Software Architecture.* [http://www.cs.cmu.edu/afs/cs/project/vit/ftp/pdf/intro_softarch.pdf] Pittsburgh, PA : Carnegie Mellon University, 1994. CMU-CS-94-166.

[12]. *No Silver Bullet Essence and Accidents of Software Engineering.* **Frederick P. Brooks, Jr.** Los Alamitos, CA, USA : IEEE Computer Society Press , 1987. 0018-9162 .

[13]. *SAAM: A Method for Analyzing the Properties of Software Architectures.* **Kazman, Rick, et al.** Sorrento, Italy : IEEE Computer Society Press , 1994, Vol. International Conference on Software Engineering. Pages: 81 - 90.

[14]. *Software Architecture: a Roadmap.* **Garlan, David.** 2000, ACM, pp. 91-101.

[15]. *Software Architecture.* **Kazan, Rick.** 2000, Handbook of Software Engineering and Knowledge Engineering.

[16]. **Bass, Len, Clements, Paul and Kazman, Rick.** *Software Architecture in Practice (Second Edition).* s.l. : Addison-Wesley, 2008.

[17]. *Foundations for the Study of Software Architecture.* **Perry, Dewayne E and Wolf, Alexander L.** 1992, ACM SIGSOFT - Software Engineering Notes, pp. 40-52.

[18]. *Introduction to the special issue on software architecture.* **Perry, D., Garlan, D.** 1995, IEEE Trans. on Software Engineering, pp. 269-274.

[19]. **Shaw, M., Garlan, D.** *Software Architecture - Perspective on an Emerging Dicipline.* s.l. : Prentice-Hall Inc., 1996.

[20]. *An Architecture-Based Approach to Software Evolution.* **Medvidovic, Nenad, Taylor, Richard N. and Rosenblum, David S.** Kyoto, Japan : Proceedings of the International Workshop on the Principles of Software Evolution, 1998, Vols. Pages 11-15.

[21]. **Parnas, D. L.** On the criteria to be used in decomposing systems into modules. [DOI= http://0-doi.acm.org.uncclc.coast.uncwil.edu/10.1145/361598.361623]. s.l. : Commun. ACM, Dec 1972. Vol. 15, 12. 1053-1058.

[22]. **Helic, Denis.** Software Architecture VO/KU . *Software Architecture (707.023/707.024).* [Online] Graz University of Technology. [Cited: 10 13, 2009.] http://coronet.iicm.tugraz.at/denis/homepage/.

[23]. *Aspect-oriented programming.* **Kiczales, Gregor and Hilsdale, Erik.** Vienna, Austria : ACM Special Interest Group on Software Engineering: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering, 2001. 1-58113-390-1.

[24]. **Pawlak, Renaud, Seinturier, Lionel and Retaille, Jean-Philippe.** *Foundation of AOP for J2EE Development.* New York, NY : APress, 2005. 1-59059-507-6.

[25]. **Gardecki, Joseph and Lesiecki, Nicholas.** *Mastering AspectJ.* Indianapolis, Indiana : Wiley Publishing, 2003. 0-471-43104-4.

[26]. Merriam-Webster Dictionary. *Merriam-Webster Dictionary.* [Online] 03 05, 2010. [Cited: 03 05, 2010.] http://www.merriam-webster.com/.

[27]. **Malan, Ruth and Bredemeyer, Dana.** Software Architecture: Centeral Concerns, Key Desicions. *Software Architecture Action Guide.* s.l. : Bredemeyer Consulting, 2002.

[28]. **Lattanze, Anthony J.** *The Architecture Centric Development Method.* Pittsburgh, PA : School of Computer Science, Carnegie Mellon University, 2005. CMU-ISRI-05-103.

[29]. **Mattsson, Michael, Grahn, Hakan and Mattsson, Frans.** *Software Architecture Evalutation Methods for Performance, Maintainability, Testability, and Portability.* Blekinge, Sweden : Blekinge - Engineering Software Qualities.

[30]. *Emergent (Mis)behavior vs. Complex Software Systems.* **Mogul, Jeffrey C.** Palo Alto, CA : Hewlett-Packard Development Company, L.P., December 22, 2005.

[31]. **Malan, Ruth and Bredemeyer, Dana.** *The Visual Architecting Process.* s.l. : Bredemeyer Consulting, 2005.

[32]. **J.D. Meier, Alex Homer, David Hill, Jason Taylor, Prashant Bansode, Lonnie Wall, Rob Boucher Jr, Akshay Bogawat.** *Application Architecture Guide 2.0.* s.l. : Microsoft Corporation, 2008.

[33]. **Malan, Ruth and Bredemeyer, Dana.** Architecture Resources - For Enterprise Advantage. *Architecture Resources - For Enterprise Advantage.* [Online] 11 29, 2009. http://www.bredemeyer.com/.

[34]. *Domains of Concern in Software Architectures and Architecture Description Languages.* **Medvidovic, Nenad and Rosenblum, David S.** Santa Barbara, California : USENIX Proceedings of the 1997 USENIX Conference on Domain-Specific Languages, 1997.

[35]. *Towards a Taxonomy of Software Connectors.* **Mehta, R Nikunj, Medvidovic, Nenad and Phadke, Sandeep.** 1-58113-206, Limerick, Ireland : ACM, 2006, Vol. 2300.

[36]. **Lassing, Nico, Rijsenbrij, Daan and van Vliet, Hans.** Using UML in Architecture-Level Modifiability Analysis. [Online] 2001. [Cited: 10 22, 2009.] www.cs.vu.nl/~hans/publications/y2001/UML.pdf.

[37]. *Architectual Blueprints -- The "4+1" View Model of Software Architecture.* **Kruchten, Philippe.** 1995, IEEE SOftware 12, pp. 42-50.

[38]. *Integrating Architecture Description Languages with a Standard Design Method.* **Robbins, Jason C, et al.** Kyoto, Japan : IEEE Computer Society, 1998, Vols. Proceedings of the 20th international conference on Software engineering Pg (209 - 218).

[39]. **Medvidovic, Nenad and Rosenblum, David S.** Assessing the Suitability of a Standard Design Method for Modeling Software Architectures. [Online] 1999. [Cited: 11 1, 2009.] http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.82.603&rep=rep1&type=pdf. 0-7923-8453-9.

[40]. **Stoermer, Christoph, Bachman, Felix and Verhoef, Chris.** *SACAM: The Software Architecture Comparison Analysis Method.* Pittsburgh, PA : Carnegi Melon Software Engineering Institute, 2003.

[41]. **Asundi, Jai and Kazman, Rick.** A Foundation for the Economic Analysis of Software Architectures. [Online] [Cited: 10 20, 2009.] www.cs.virginia.edu/~sullivan/edser3/asundi.pdf.

[42]. *Estimation of Quality for Software Componets - an Empirical Approach.* **Sharma, Arun, Kumar, Rajesh and Grover, P S.** November 2008 Volumne 33 Number 6, SIGSOFT Software Engineering Notes, pp. 1-10.

[43]. **MSDN.** Learn The ABCs Of Programming Windows Communication Foundation. [Online] http://msdn.microsoft.com/en-us/magazine/cc163647.aspx.

[44]. *Information System Architecture Metrics: an Enterprise Engineering Evaluation Approach.* **Vasconcelos, Andre, Sousa, Pedro and Tribolet, Jose.** 1, s.l. : The Electronic Journal Information Systems Evaluation, 2007, Vol. 10.

[45]. **Pressman, Roger S.** *Software Engineering: A Practitioner's Approach.* s.l. : McGraw-Hill Higher Education, 2001. 0072496681 .

[46]. **Kaur, Amandeep, Singh, Satwinder and Kahlon, K. S.** A Metric Framework for Analysis of Quality of Object Oriented Design. *A Metric Framework for Analysis of Quality of Object Oriented Design.* [Online] 6 2009. [Cited: 3 1, 2010.] www.waset.org/journals/waset/v60/v60-79.pdf.

[47]. Metrics 1.3.6. *Metrics 1.3.6.* [Online] www.sourceforge.net. http://metrics.sourceforge.net/.

[48]. *Towards Reusable Components with Aspects: An Emporicial Study on Modularity and Obliviosness.* **Hoffman, Kevin and Eugster, Patrick.** Leipzig, Germany : Association for Computing Machinery Special Interest Group on Software Engineering , 2008. 978-1-60558-079-1 .

[49]. **Colyer, Adrian, et al.** *Eclipse AspectJ.* Upper Saddle River, NJ : Addison-Wiley Publishing, December 2004. 0-32-124587-3.

[50]. *Towards reusable components with aspects: an empirical study on modularity and obliviousness.* **Hoffman, Kevin and Eugster, Patrick.** Leipzig, Germany : ACM, Proceedings of the 30th international conference on Software engineering , 2008. 978-1-60558-079-1 .

[51]. **Ryman, Arthur.** Eclipse: The Story of Web Tools Platform 0.7. *Eclipse Developer Journal.* [Online] SYS-CON Media, Inc, 10 19, 2005. http://eclipse.sys-con.com/node/111212.

[52]. *On visual formalisms.* **Harel, David.** Issue 5, New York, NY : ACM, 1988 , Vol. Volume 31. 0001-0782.

[53]. *Using Non-Functional Requirements to Systematically Select Among Alternatives in Architectual Design.* **Chung, Lawrence and Nixon, Brian A.**

[54]. *An Approach to Quantitative Software Architecture Sensitivity Analysis.* **Lung, Chung-Horng and Kalaichelvan, Kalai.** 1, s.l. : International Journal of Software Engineering and Knowledge Engineering, pg (97-114), 2000, Vol. 10.

[55]. *Software Cost Estimation.* **Leung, Hareton and Fan, Zhang.** Las Vegas, NV : Handbook of Software Engineering & Knowledge Engineering , 2006 , Vol. 2. 0-7695-2611-X .

[56]. *The measurement of software design quality.* **Blundell, James Kenneth, Hines, Mary Lou and Stach, Jerold.** 1997, Annals of Software Engineering 4, pp. 235-255.

[57]. *An Architectural Quality Assessment for Domain-Specific Software.* **Hu, Changjun, Jiao, Feng and Chongchong, Zhao.** 2008, 2008 Internal Conference on Computer Science & Software Engineering, pp. 143-146.

[58]. *Metrics to Assess the Likelihood of Project Success Based on Architecture Reviews .* **Avritzer, Alberto and Weyuker, Elaine J.** 3, Middletown, NJ : Empirical Software Engineering, Pg 199-215, 2004, Vol. 4.

[59]. *Applying and interpreting object oriented metrics.* **Rosenberg, Linda H.** s.l. : In Software Technology Conference '98, 1998.

[60]. **Reddy, Raghu, France, Robert and Georg, Geri.** *An Aspect-based Approach to Modeling and Analyzing Dependability Features.* s.l. : Colorado State University, November 2004. Technical Report CS04-109.

[61]. *On Architecture.* **Booch, Grady.** 2006, IEEE Computer Society.

[62]. **Gerber, A J, Barnard, A and Van Der Merwe, A J.** *Design and Evaluation Criteria for Layered Architectures.* Guateng, South Africa : Meraka Institute .

[63]. *A Survey on SOftware Architecture Analysis Methods.* **Dobrica, Liliana and Niemela, Elia.** 7, s.l. : IEEE Transations of Software Engineering, 2002, Vol. 28.

[64]. *Architecture-Based Runtime Software Evolution.* **Oreizy, Peyman, Medvidovic, Nenad and Taylor, Richard N.** Kyoto, Japan : International Conference on Software Engineering, 1998. ICSE98.

[65]. *A Language and Environment for Architecture-Based Software Development and Evolution.* **Medvidovic, Nenad, Rosenblum, David S and Taylor, Richard N.** s.l. : Defense Advanced Research Projects, 2001. National Science Foundation.

[66]. *A Framework for Classifying and Comparing Architecture Description Languages.* **Medvidovic, Nenad and Taylor, Richard N.** Irvine, California : Department of Information and Computer Science University of California, Irvine, Vols. contract number F30602-94-C-0218.

[67]. *ArchJava: Connecting Software Architecture to Implementation.* **Aldrich, Jonathan, Chambers, Craig and Notkin, David.** Orlando, FL : ICSE, 2002.

[68]. *Integrating architecture description languages with a standard design method.* **Robbins, Jason E, et al.** Proceedings of the 20th international conference on Software engineering , Kyoto, Japan : IEEE Computer Society, 1998, pp. 209-218.

[69]. **Dong, Jing, Chen, Shanguo and Jeng, Jun-Jang.** Event-Based Blackboard Architecture for Multi-Agent Systems. *University of Dallas.* [Online] IEEE, April 4, 2005. [Cited: 11 28, 2009.] http://www.utdallas.edu/~jdong/papers/agent05.pdf. On page(s): 379- 384 Vol. 2.

[70]. **Edwards, Benjamin.** An Introduction to Implicit Invocation Architectures. *www.mach-ii.com.* [Online] [Cited: 11 28, 2009.] http://www.mach-ii.com/resources/intro_to_implicit_invocation.pdf.

[71]. **Booch, Grady.** *Best of Booch.* New York, New York : SIGS Books & Multimedia, 1996. 1-884842-71-2.

[72]. *Software Testing Research: Achievements, Challanges, Dreams.* **Bertolino, Antonia.** Pisa, Italy : IEEE Computer Society, 2007. 0-7695-2829-05/07.

[73]. **Kemerer, Chidamber and.** *A Metrics Suite for Object Oriented Design.* s.l. : IEEE Trans. Software Eng, 1994. Pg 476-493, vol 20, no 6.

[74]. *Component-based software engineering.* **Hasselbring, Wilhelm.** Handbook of Software Engineering and Knowledge Engineering, pages 289--305, River Edge, NJ : World Scientific Publishing, 2002, Vol. Volume 2.

[75]. An Introduction to Object-Oriented Metrics. *An Introduction to Object-Oriented Metrics.* [Online] [Cited: 03 01, 2010.] http://agile.csc.ncsu.edu/SEMaterials/OOMetrics.htm.

[76]. *The Measurement of Software Sesign Quality.* **Blundell, James Kenneth, Hines, Mary Lou and Stach, Jerold.** 1997, Annals of Software Engineering 4, pp. 235-255.

[77]. *A survey on Software Architecture Analysis methods.* **Dobrica, Liliana and Niemela, Eila.** JULY 2002, IEEE Computer Society, pp. 638-653.

[78]. *A Comparative Evaluation of Tests Generated from Different UML Diagrams.* **Kansomkeat, Supaporn, et al.** s.l. : IEEE Computer Society , Vols. Proceedings of the 2008 Ninth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing . 978-0-7695-3263-9 .

[79]. *A join point for loops in AspectJ.* **Harbulot, Bruno and Gurd, John R.** Bonn, Germany : AOSD-Europe , Proceedings of the 5th international conference on Aspect-oriented software development, 2006. 1-59593-300-X.

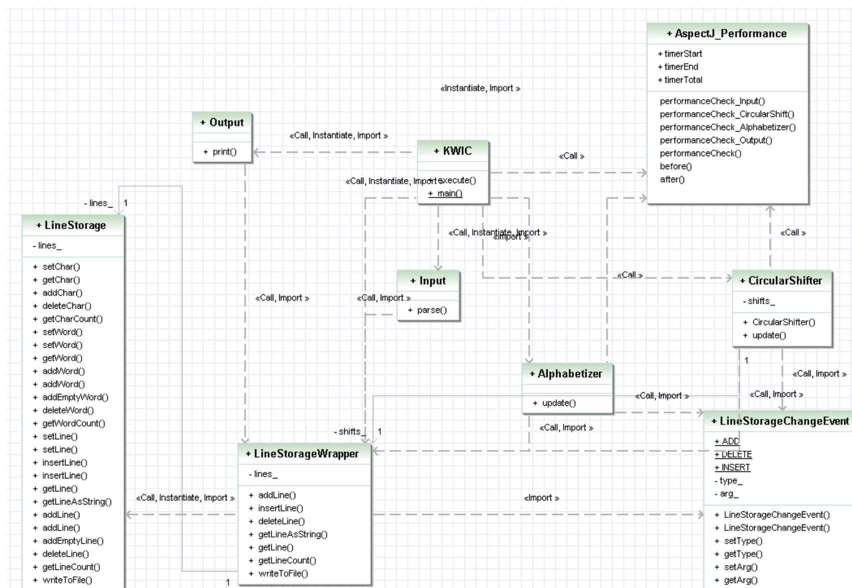
[80]. Software Architecture VO/KU . [Online] Graz University of Technology. http://coronet.iicm.tugraz.at/denis/homepage/.

Appendix A – Experiment – Performance

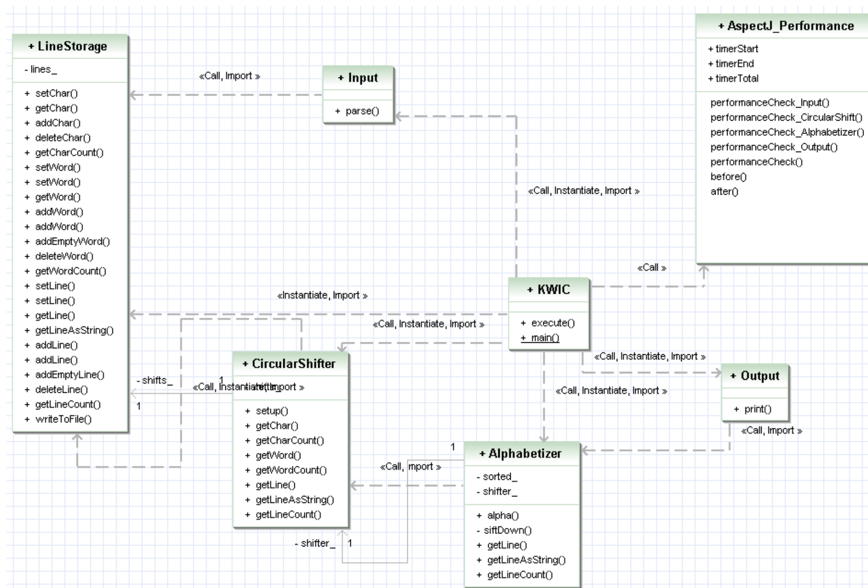
Logical View of source

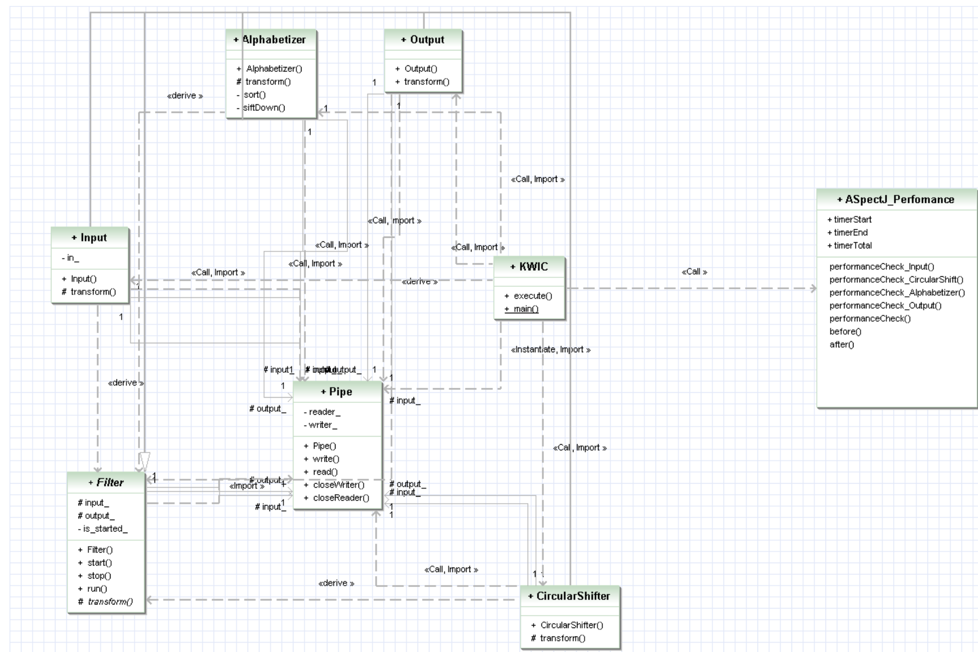
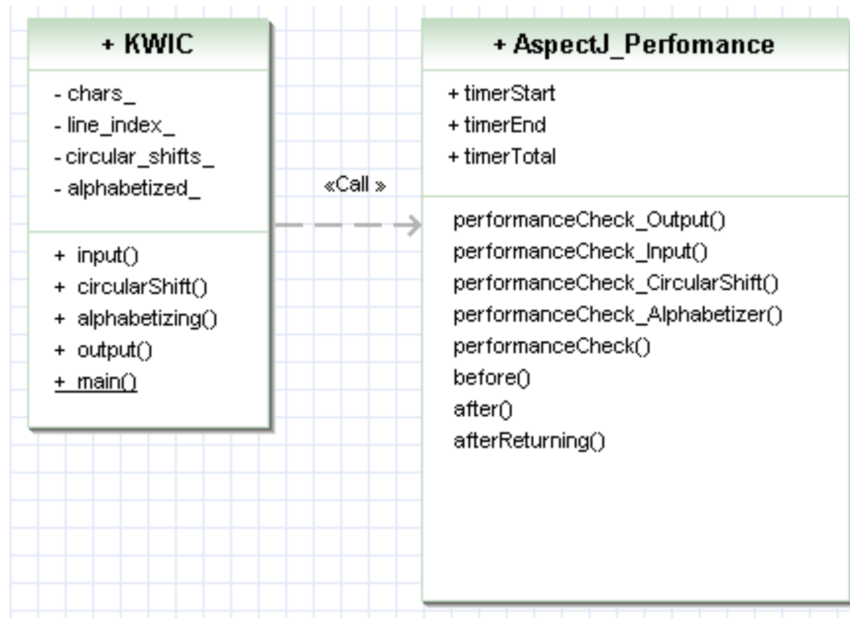
The below table describes the affect of the AspectJ source on the underlying architectural styles:

Event-Driven



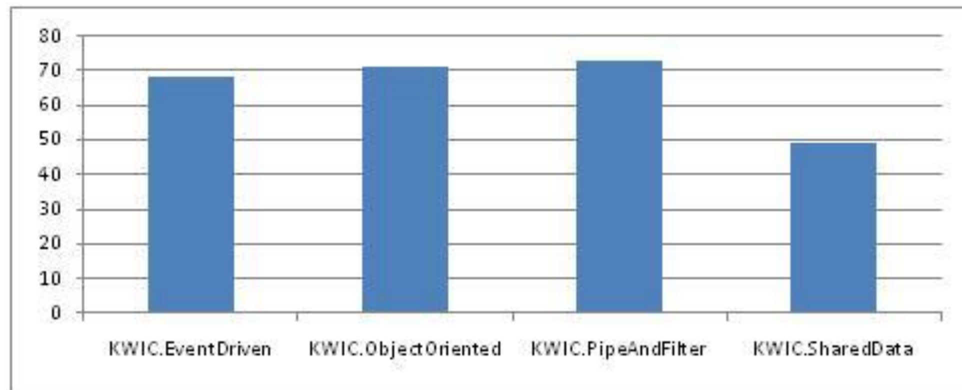
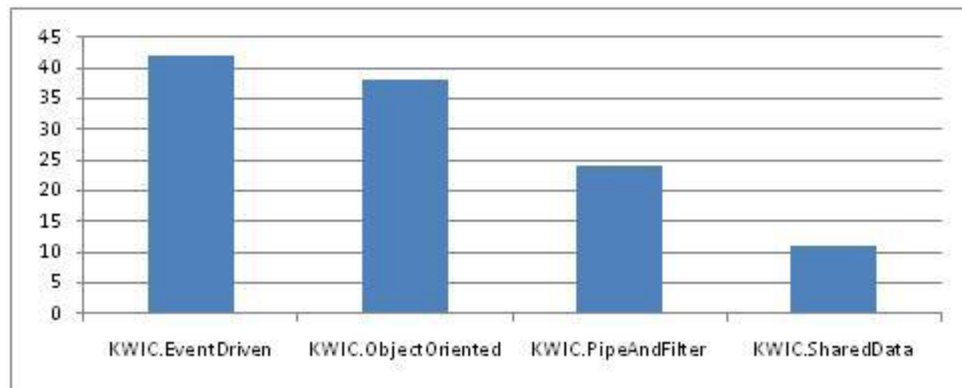
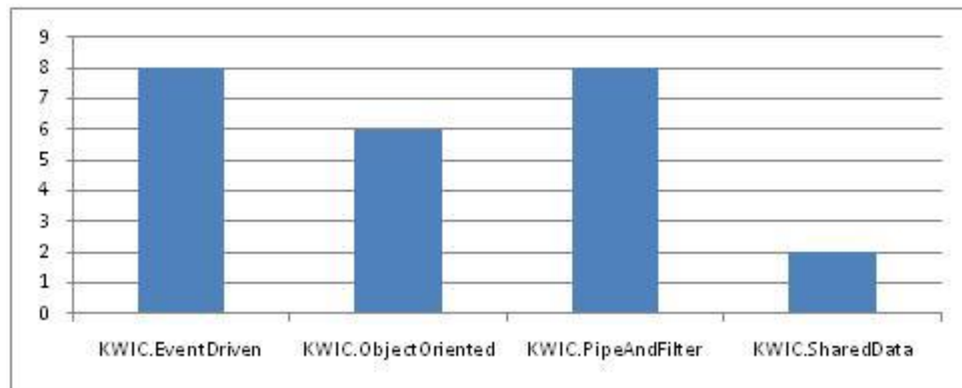
Object-Oriented

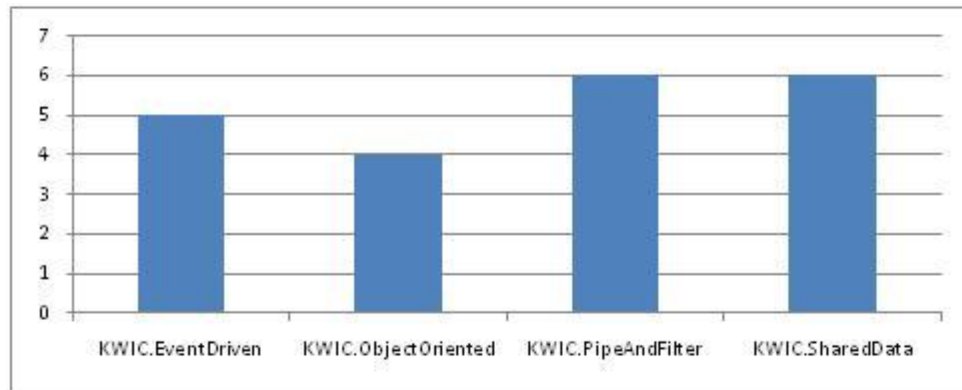
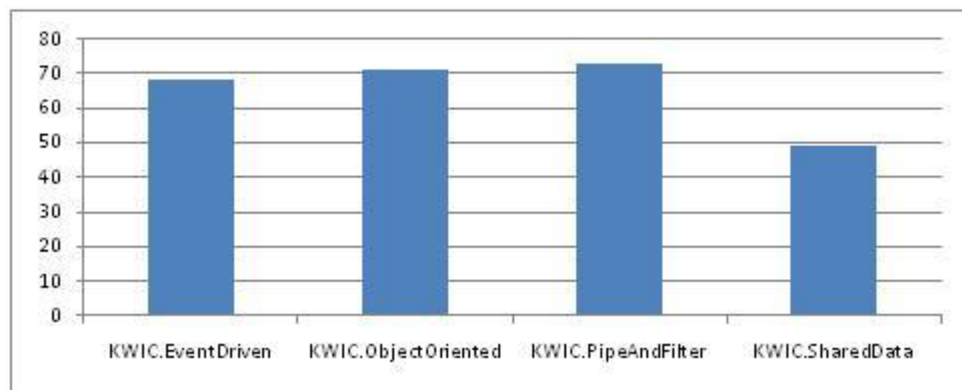
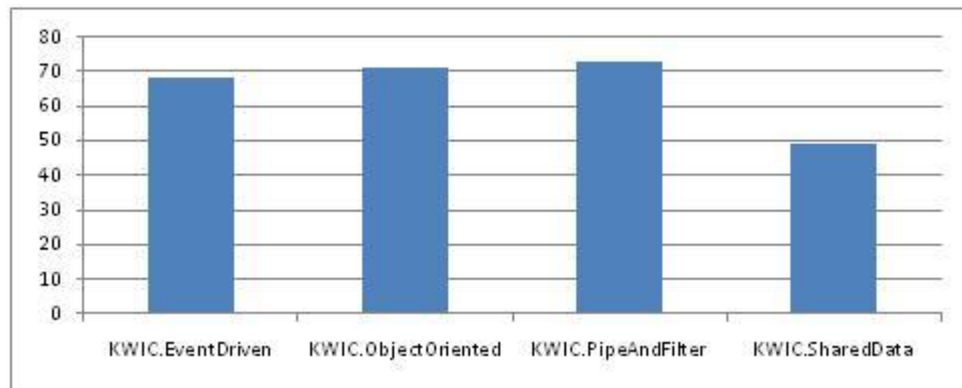


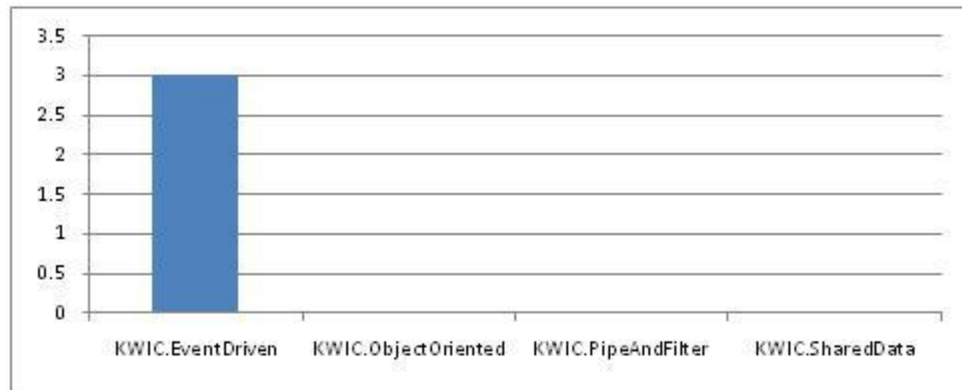
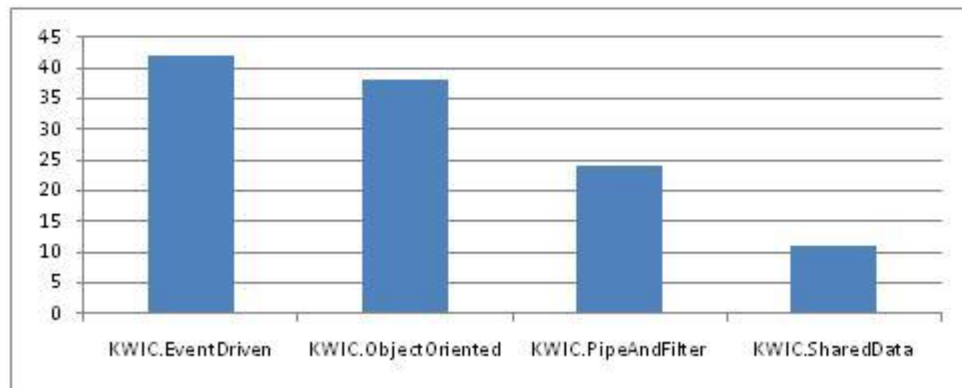
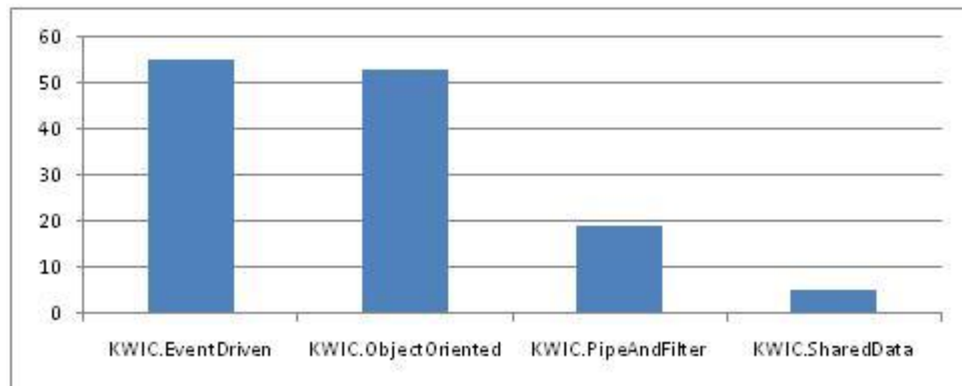
Pipe-and-Filter	
Shared-Data	

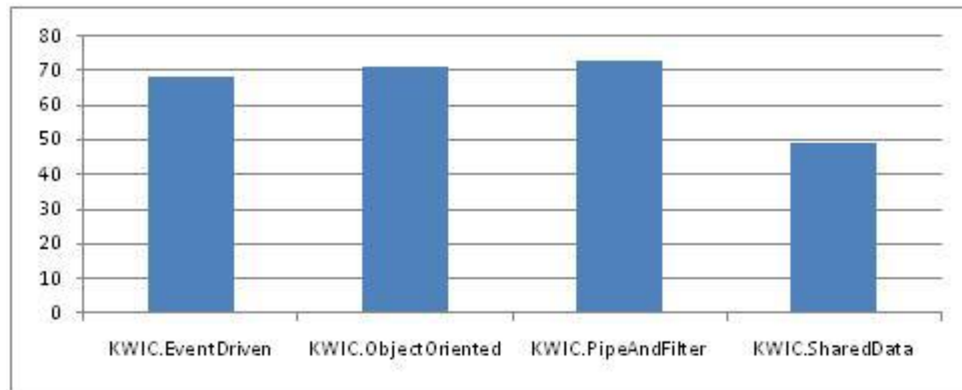
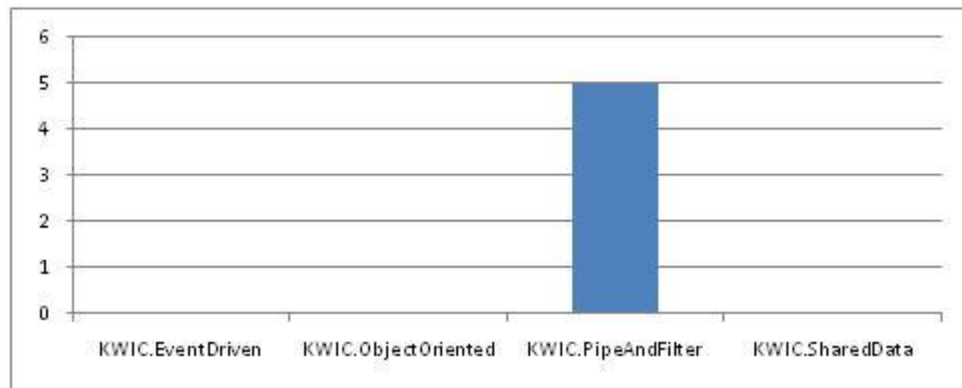
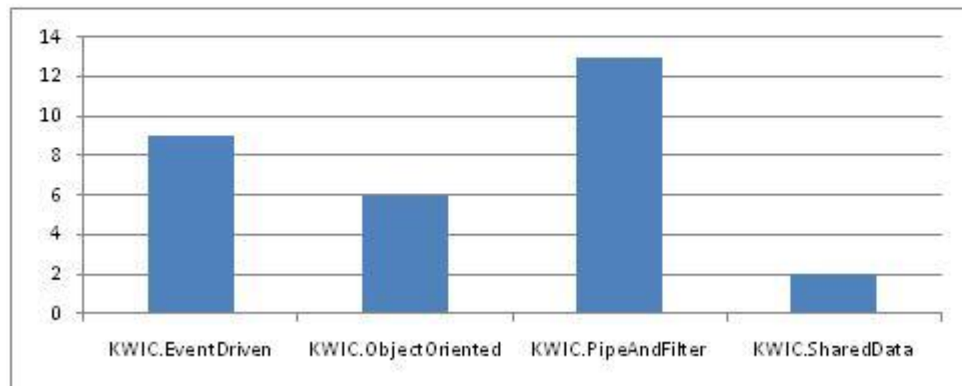
Metric Evaluation

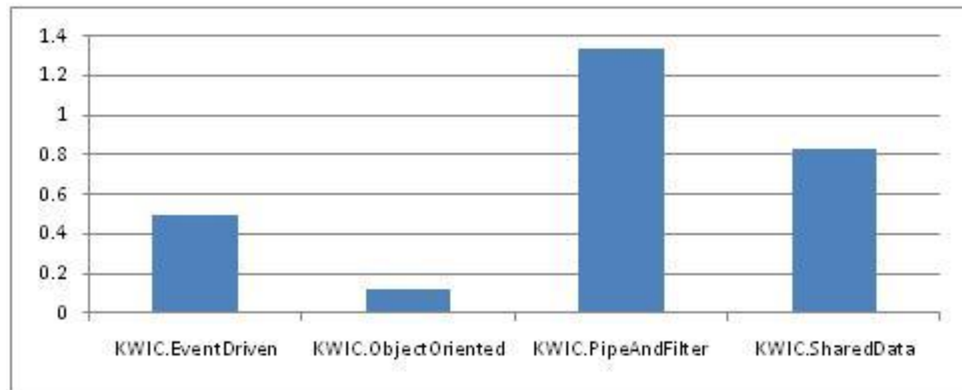
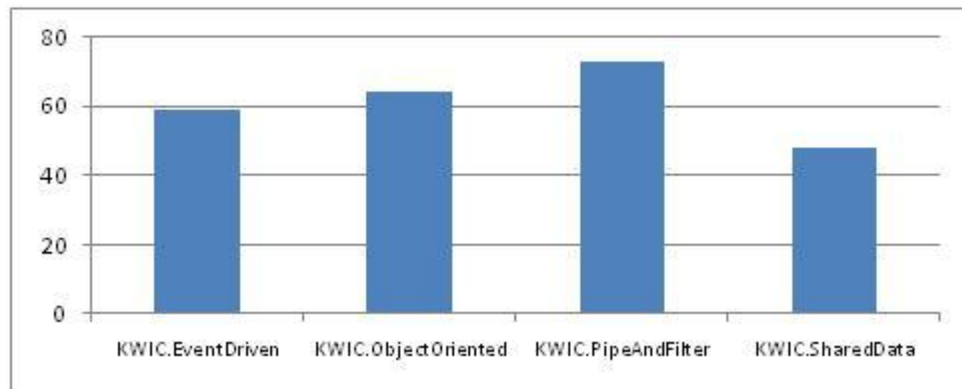
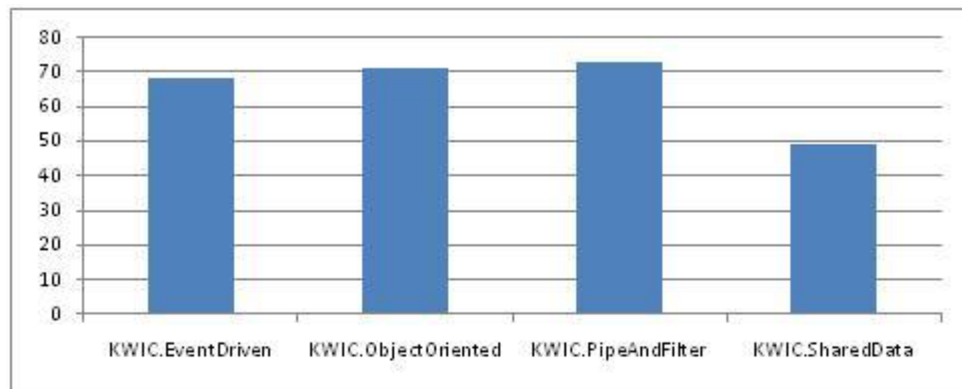
Below table breaks down each metrics evaluation by the architectural style:

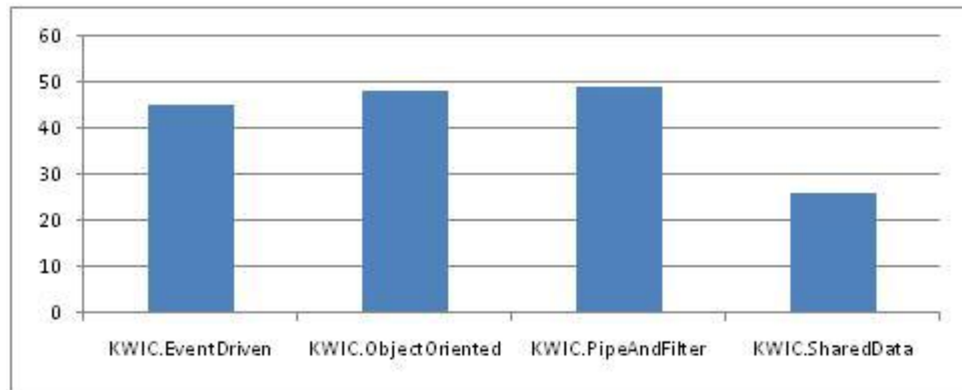
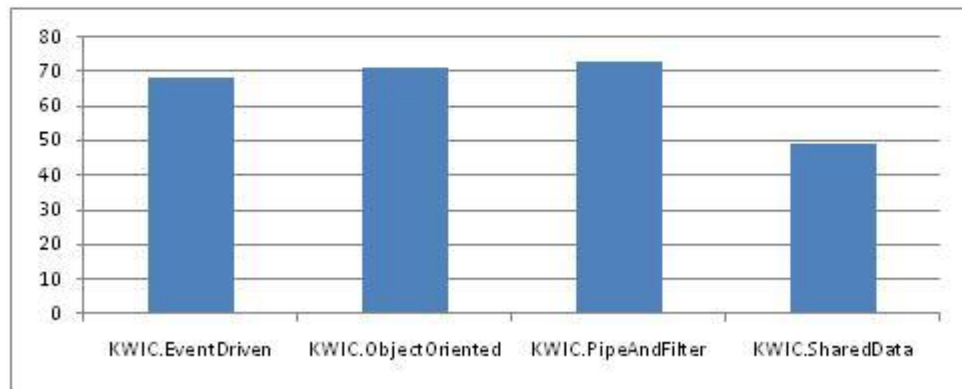
Lines of Code (LOC)	
Number of Static Methods (NSM)	
Number of Classes (NOC)	

Number of Attributes (NOF)	
Number of Packages (NOP)	
Number of Overridden Methods (NORM)	

Number of Static Attributes (NSF)	
Number of Methods (NOM)	
Number of Parameter (PAR)	

Number of Interfaces (NOI)	
Number of Children (NSC)	
Depth of Inheritance Tree (DIT)	

Lack of Cohesion of Methods (LCOM)	
McCabe Cyclomatic Complexity	
Method Lines of Code (MLOC)	

Nested Block Depth	
Weighted Methods per Class (WMC)	

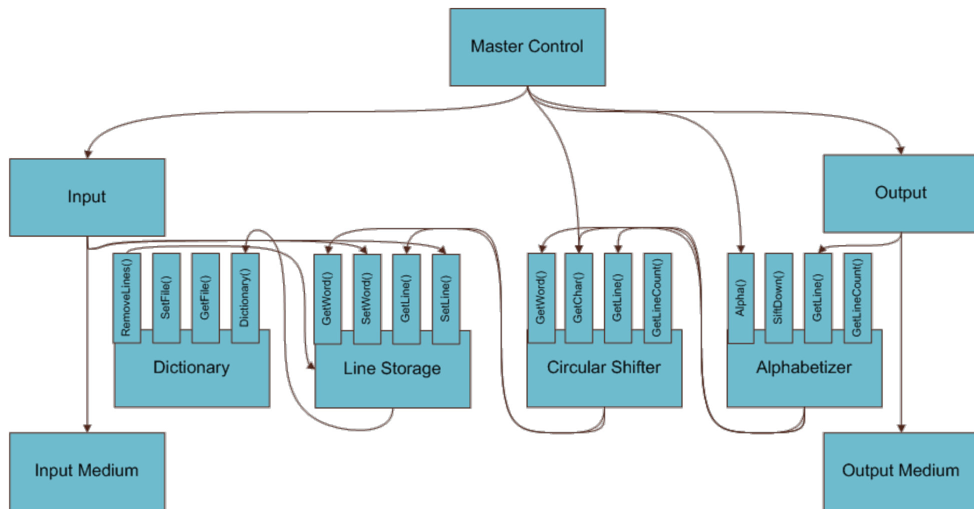
Appendix B – Experiment – Modifiability

Conceptual View

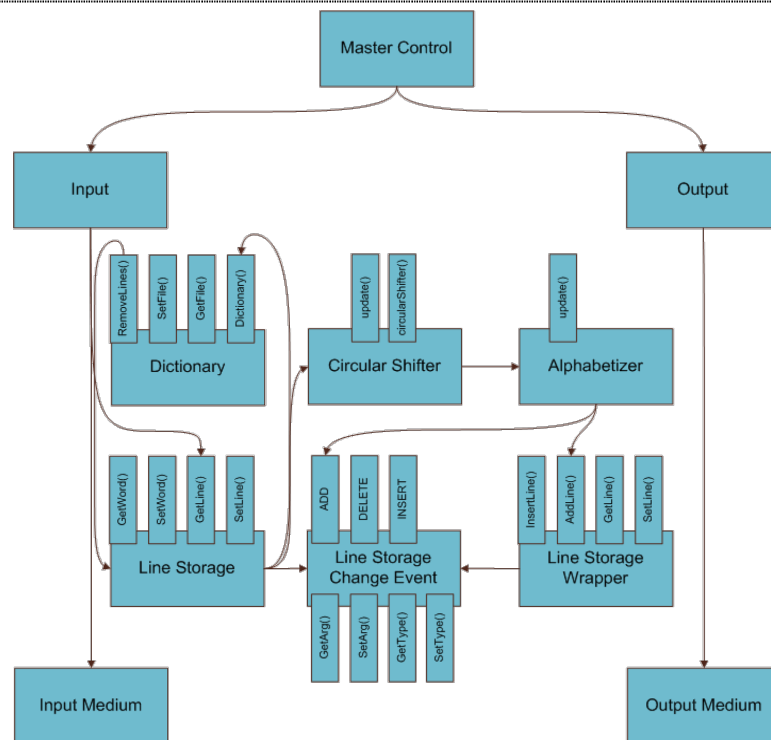
The conceptual view of the modification can be seen in the table below:

Table 6 - Conceptual View of the Dictionary Class

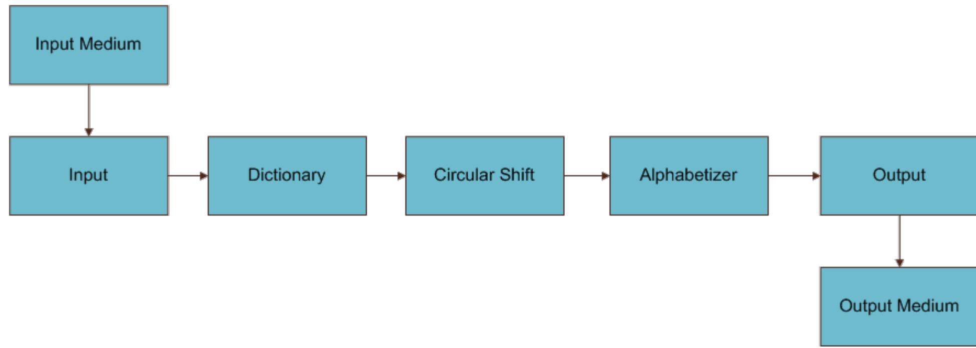
Abstract Data Type



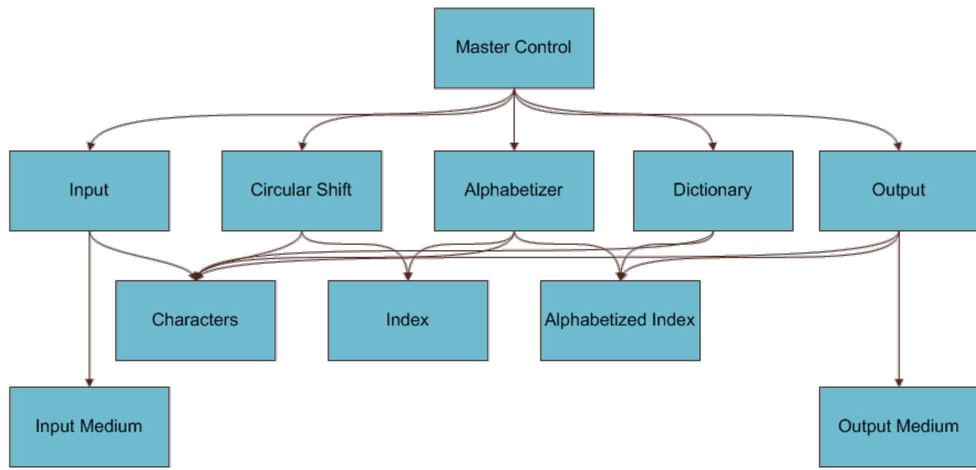
Implicit Invocation



Pipe-and-Filter



Shared Data



Dictionary Class source

The class' source is described below:

```
import java.io.BufferedInputStream;
import java.io.DataInputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.ArrayList;
import java.util.StringTokenizer;

import javax.swing.JOptionPane;

public class Dictionary {

        String FILENAME = "";
        ArrayList<String> DICT = new ArrayList<String>();

        public void setFilename(String filename)
        {
                this.FILENAME = filename;
        }
        public String getFilename()
```

```
{
		return this.FILENAME;
}

public ArrayList<String> getDictionary()
{
		if (this.DICT.size() > 0) return this.DICT;
		else return null;
}
@SuppressWarnings({ "deprecation" })
private void setDictionary(String filename)
{
		if (this.FILENAME.equalsIgnoreCase(filename)) this.setFilename(filename);

		try
		{
				if ( this.isFile() )
				{
						File _file = new File(this.FILENAME);
				FileInputStream _fileInputStream = new FileInputStream (_file);
				BufferedInputStream _bis = new BufferedInputStream(_fileInputStream);

				DataInputStream _dis = new DataInputStream(_bis);
				String line = _dis.readLine();

				while(line != null)
				{
					StringTokenizer tokenizer = new StringTokenizer(line);
					this.DICT.add(line);

					while(tokenizer.hasMoreTokens()) tokenizer.nextToken();

					line = _dis.readLine();
			}
				}
		}
		catch (FileNotFoundException ex)    {ex.printStackTrace();}
   catch (IllegalArgumentException ex) {ex.printStackTrace();}
   catch (IOException ex)                              {ex.printStackTrace();}
   catch (NullPointerException ex)             {ex.printStackTrace();}
}

public Dictionary() {}

public Dictionary(String filename)
{
		this.setFilename(filename);
		this.setDictionary(this.getFilename());
}

private boolean isFile() throws FileNotFoundException, IOException
{
		java.io.File _file = new java.io.File(this.FILENAME);
		if (_file.exists() && _file.canRead()) return true;
		else return false;
}
```

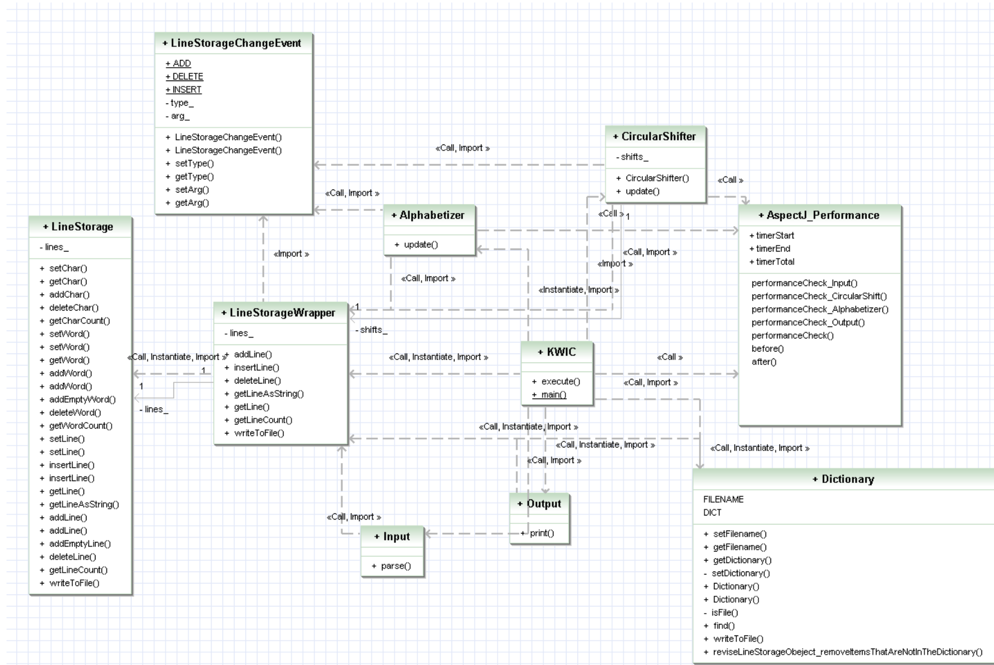
Figure 29 - Dictionary Sample Source

Logical View of source

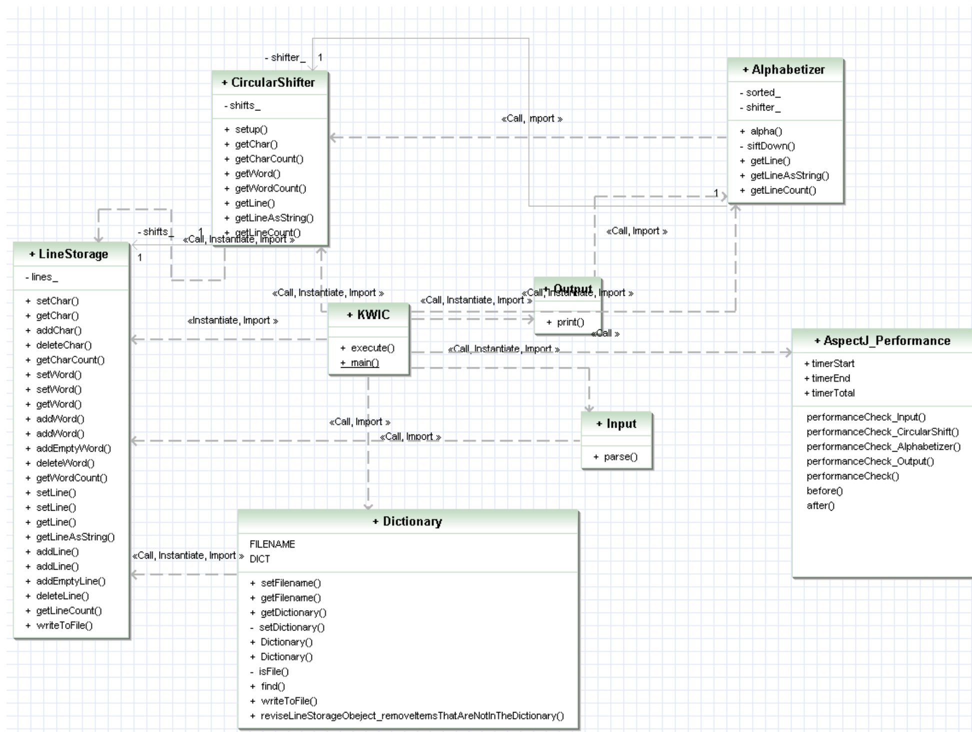
The logical view of the styles after the modification and insertion of the Dictionary class can be review below:

Table 7- Dictionary Class Modification

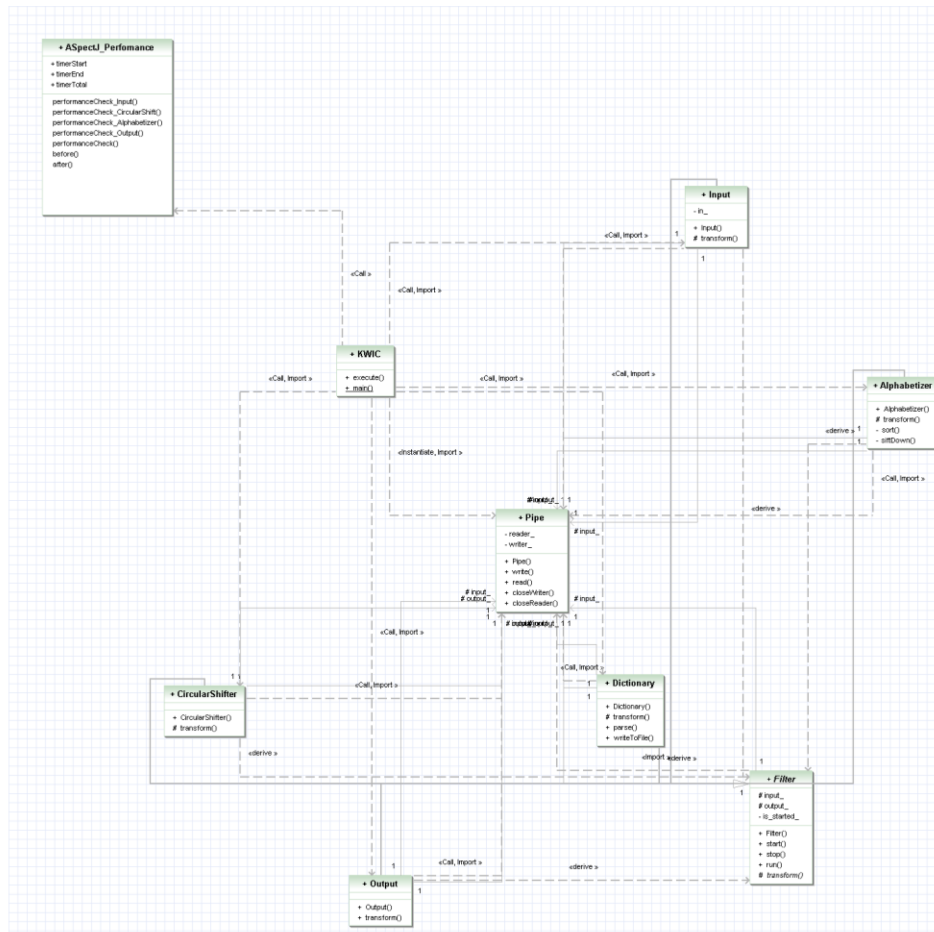
Event-Driven



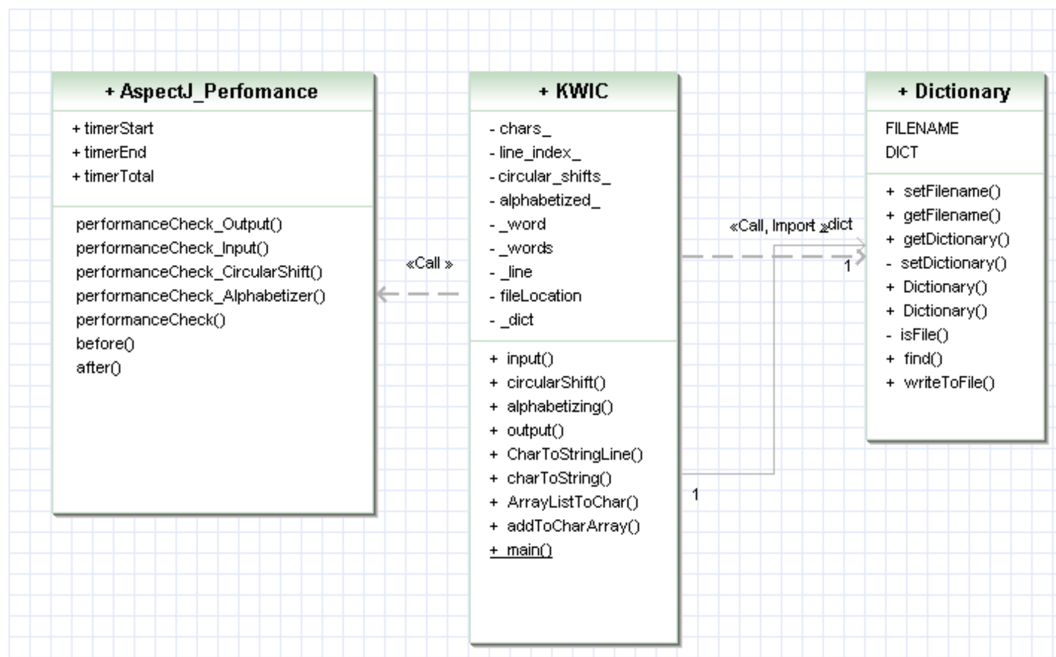
Object-Oriented



Pipe-and-Filter

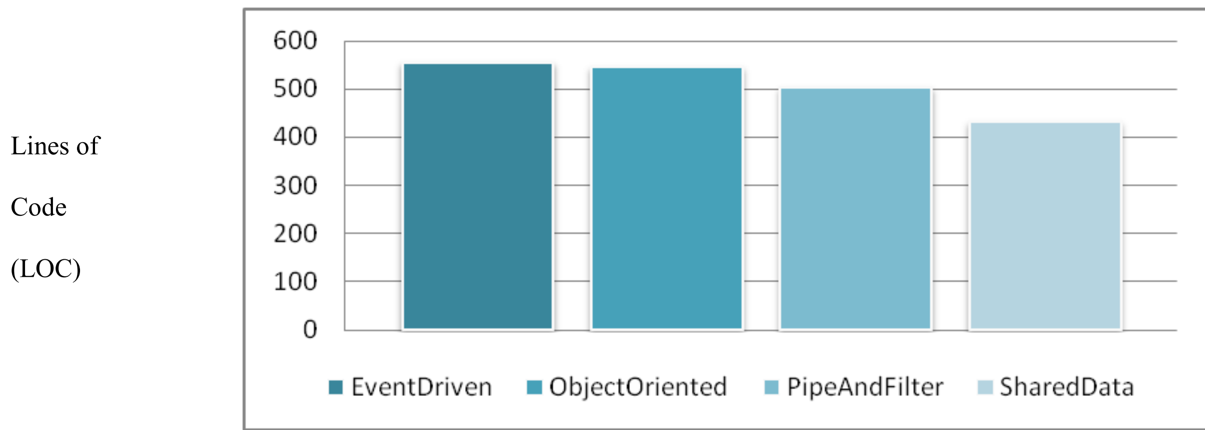
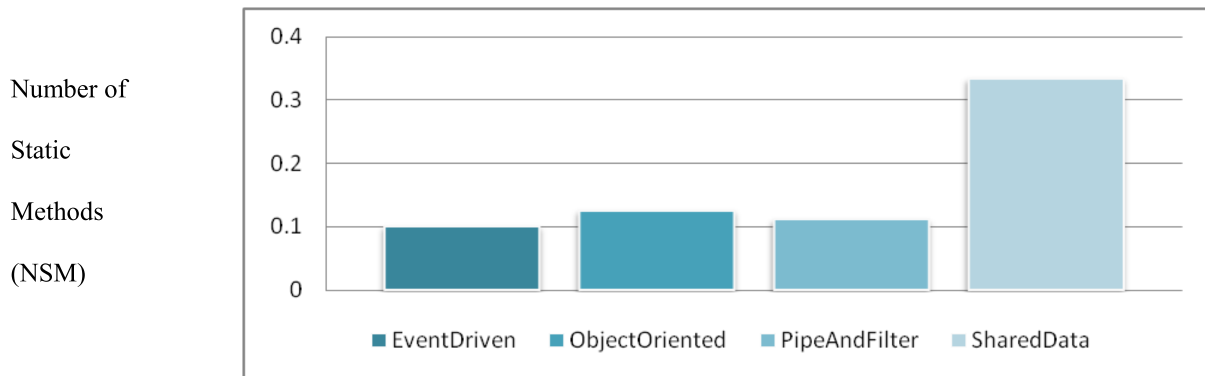
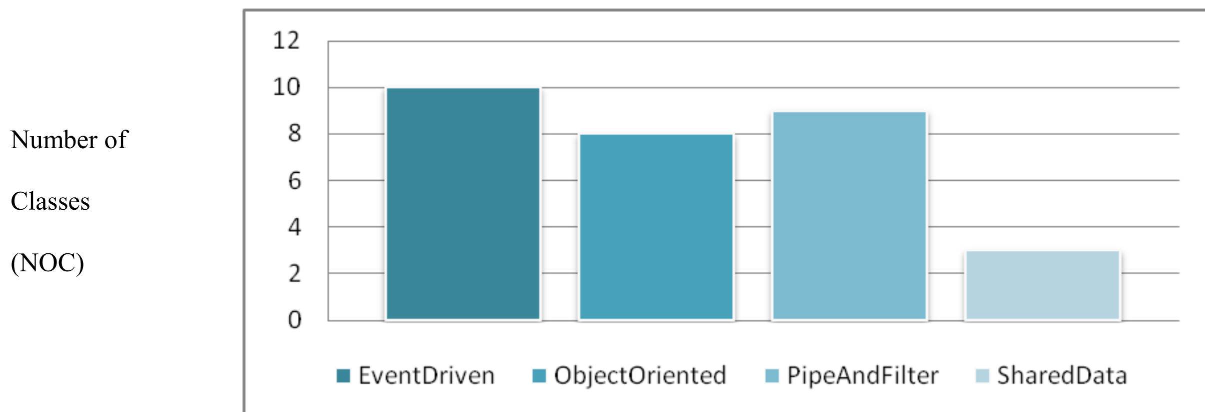


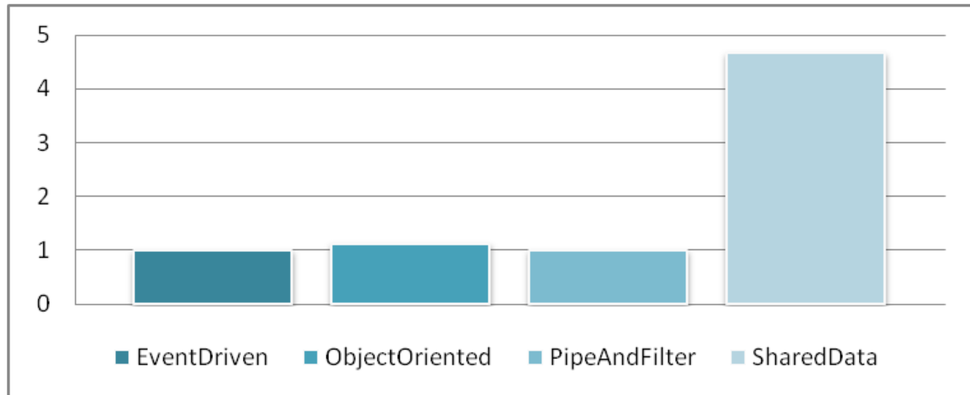
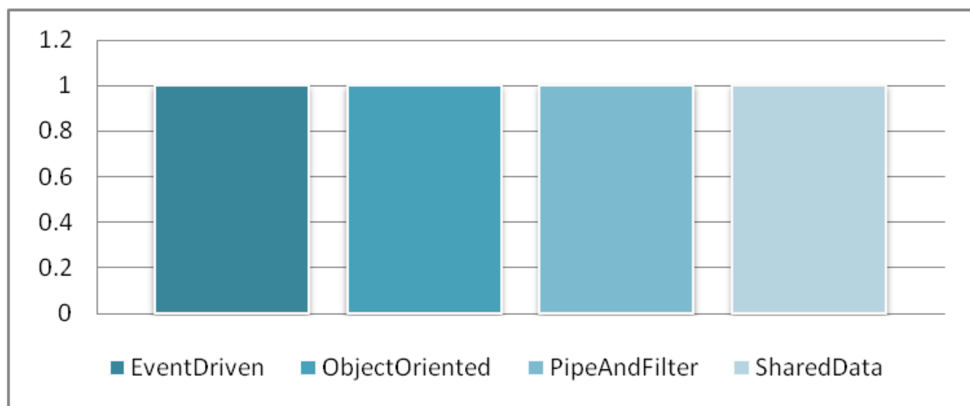
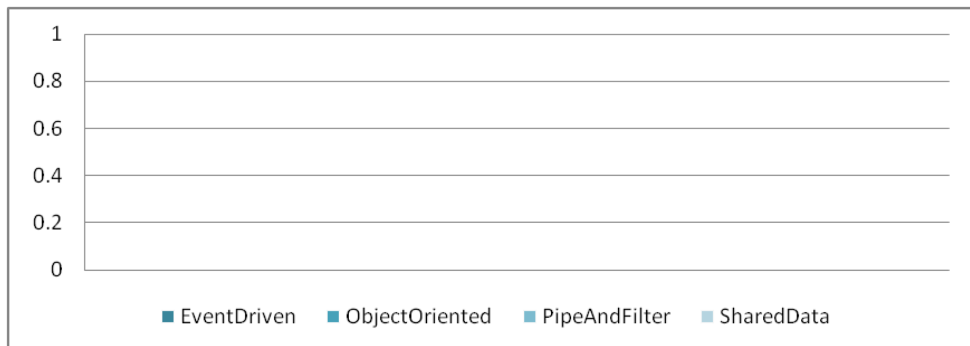
Shared-Data

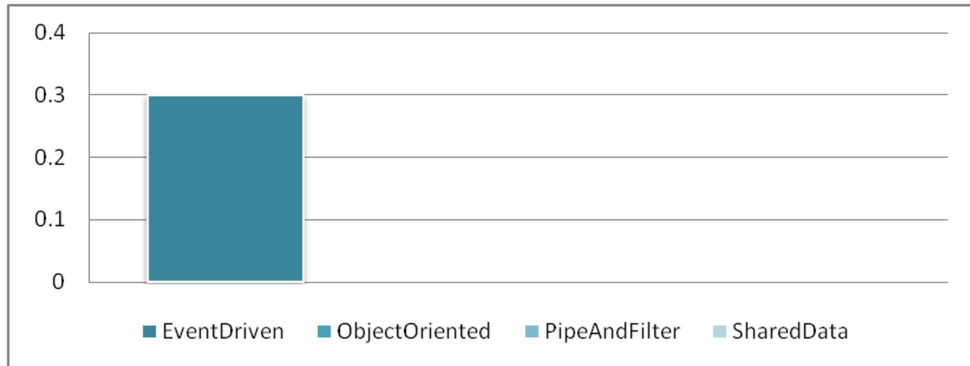
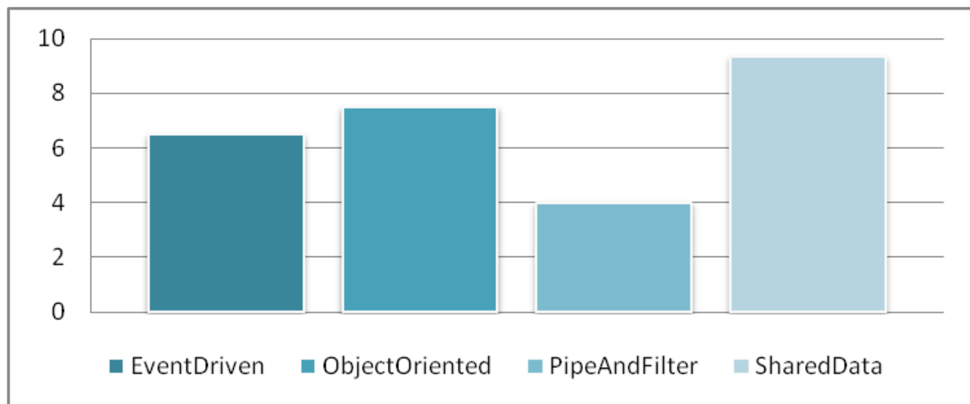
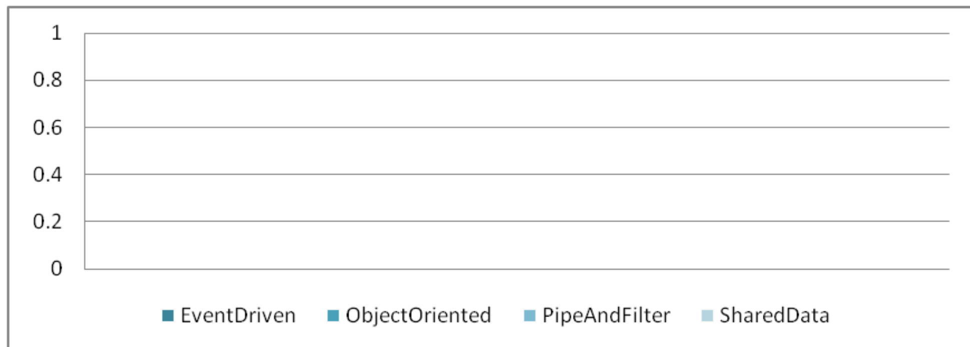


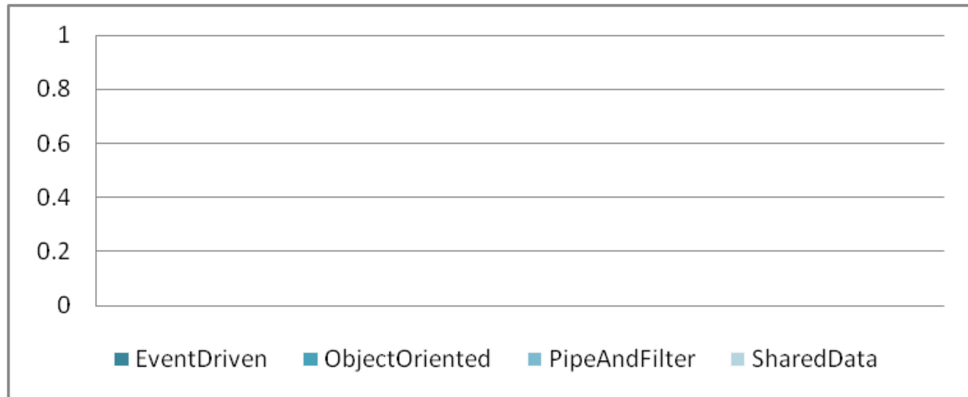
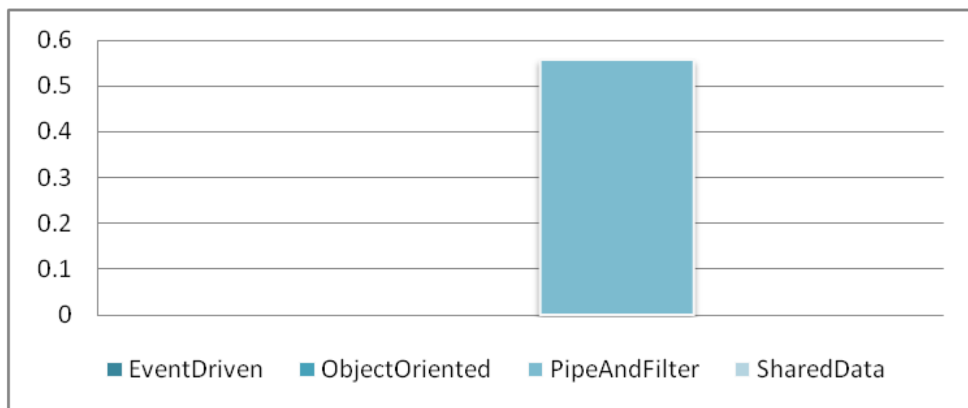
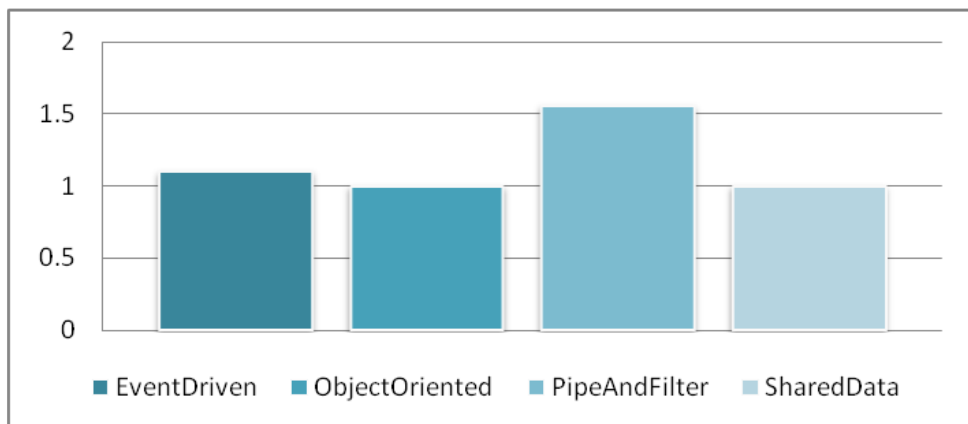
Metrics Evaluation

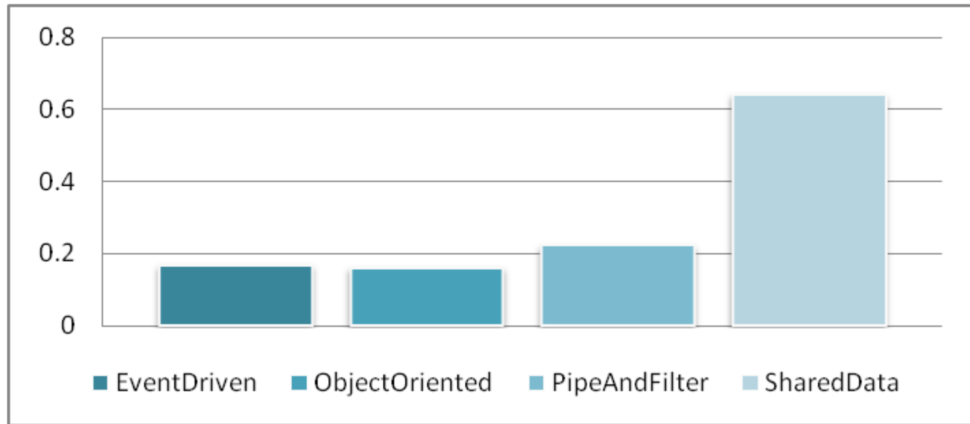
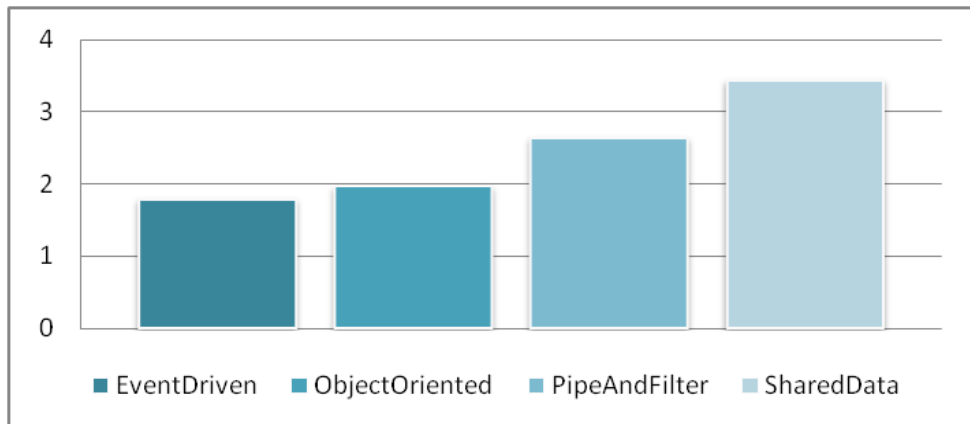
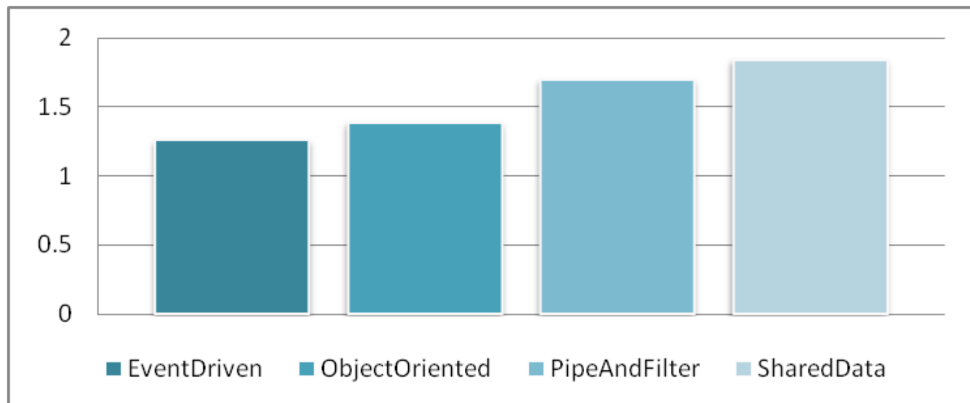
After the modification, the below metrics were collected for analysis:

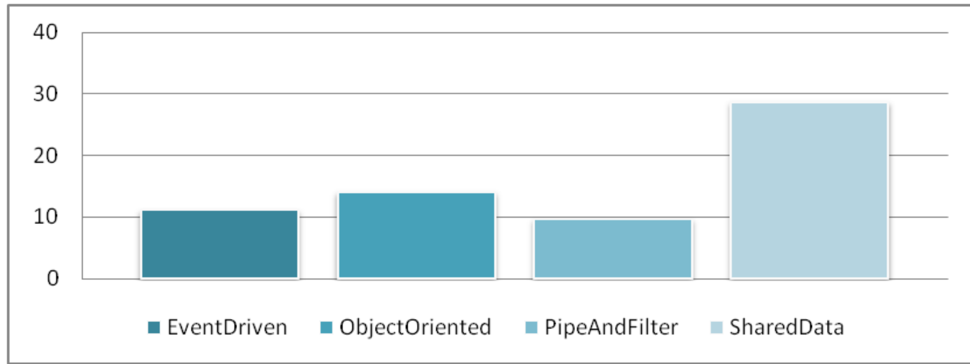
Lines of Code (LOC)	
Number of Static Methods (NSM)	
Number of Classes (NOC)	

Number of Attributes (NOA)	
Number of Packages (NOP)	
Number of Overridden Methods (NORM)	

Number of Static Attributes (NSF)	
Number of Methods (NOM)	
Number of Parameter (PAR)	

Number of Interfaces (NOI)	
Number of Children (NSC)	
Depth of Inheritance Tree (DIT)	

Lack of Cohesion of Methods (LCOM)	
McCabe Cyclomatic Complexity	
Nested Block Depth	

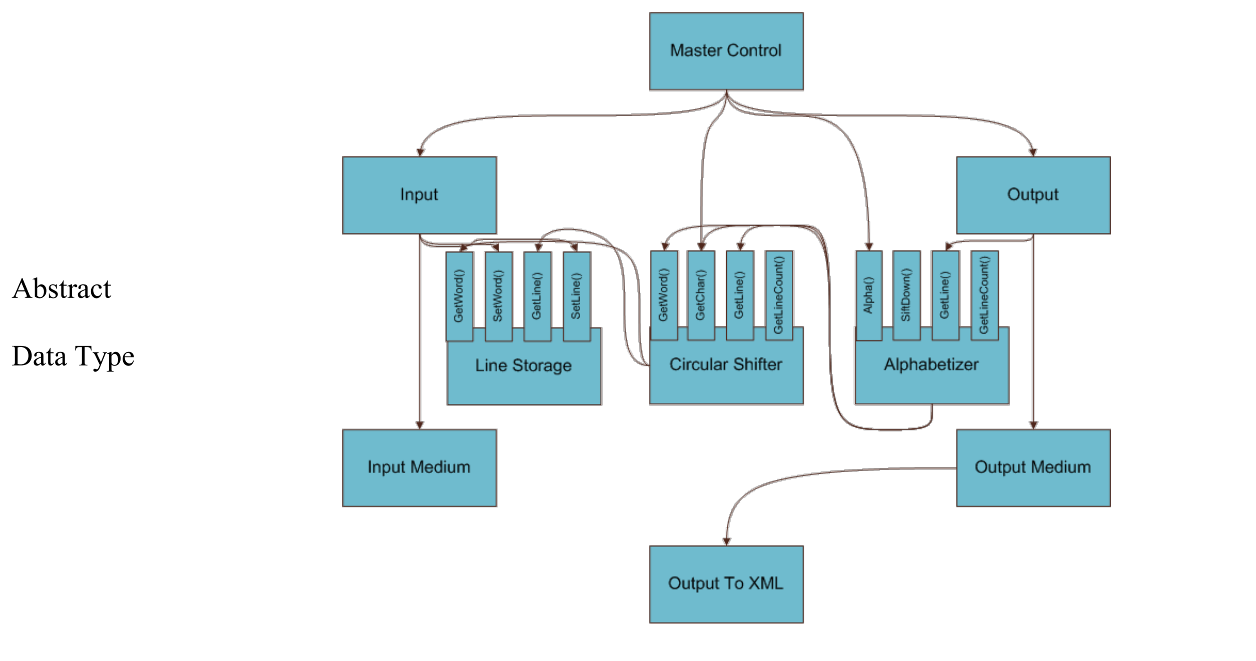
Weighted Methods per Class (WMC)	

Appendix C – Experiment – Reusability

Logical View

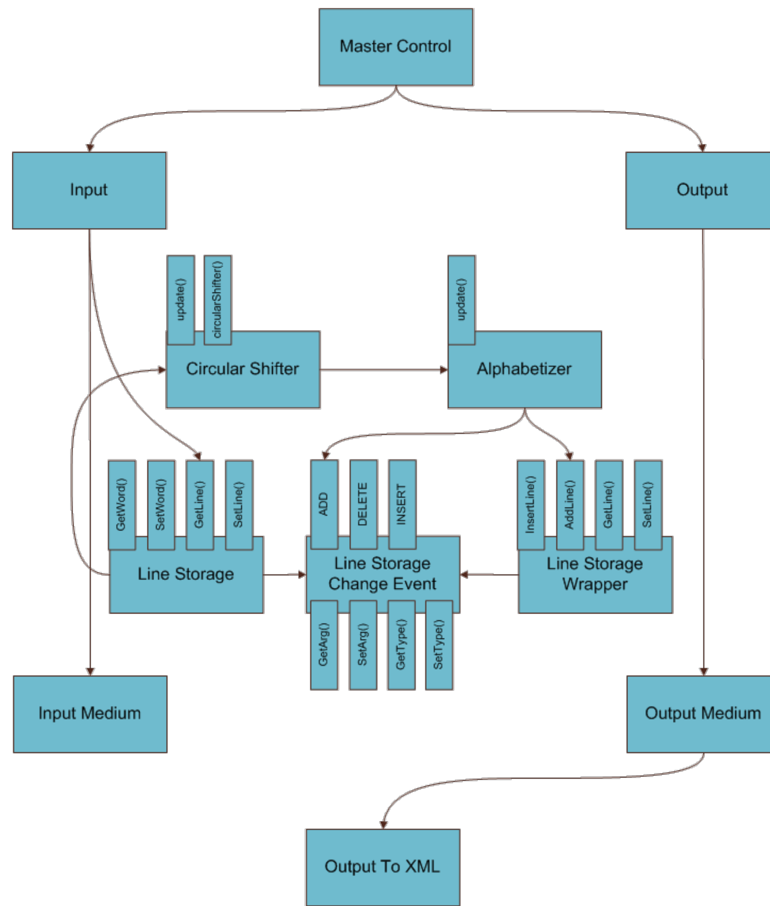
Following table conceptually describes the affect of the Out to XML package to KWIC architecture:

Table 8 - OutputToXML Conceptual View

Abstract Data Type	

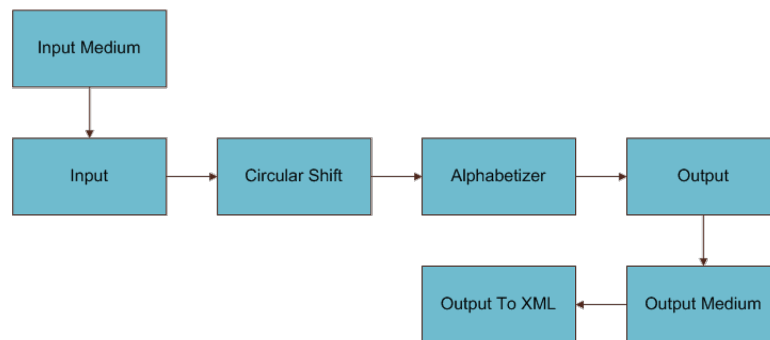
Implicit

Invocation

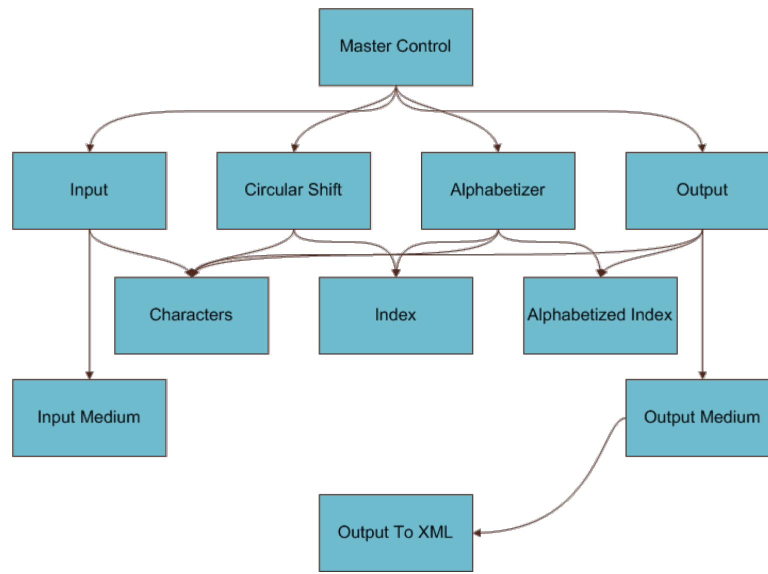


Pipe-and-

Filter



Shared Data



OutputToXML Package Source

```
package outputXML;

import java.util.ArrayList;

import javax.xml.parsers.*;
import javax.xml.transform.*;
import javax.xml.transform.dom.*;
import javax.xml.transform.stream.*;
import org.w3c.dom.*;

public class CreateXMLFile
{
        String FILENAME = "";
        ArrayList<String> OUTPUT = new ArrayList<String>();

        public void setFilename(String filename)
        {
                this.FILENAME = filename;
        }
        public String getFilename()
        {
                return this.FILENAME;
        }

        public CreateXMLFile () {}

        public CreateXMLFile (String filename, ArrayList<String> output)
        {
                this.FILENAME = filename;
                this.OUTPUT       = output;
                createFile(this.FILENAME, this.OUTPUT);
        }

        private void createFile(String filename, ArrayList<String> output)
```

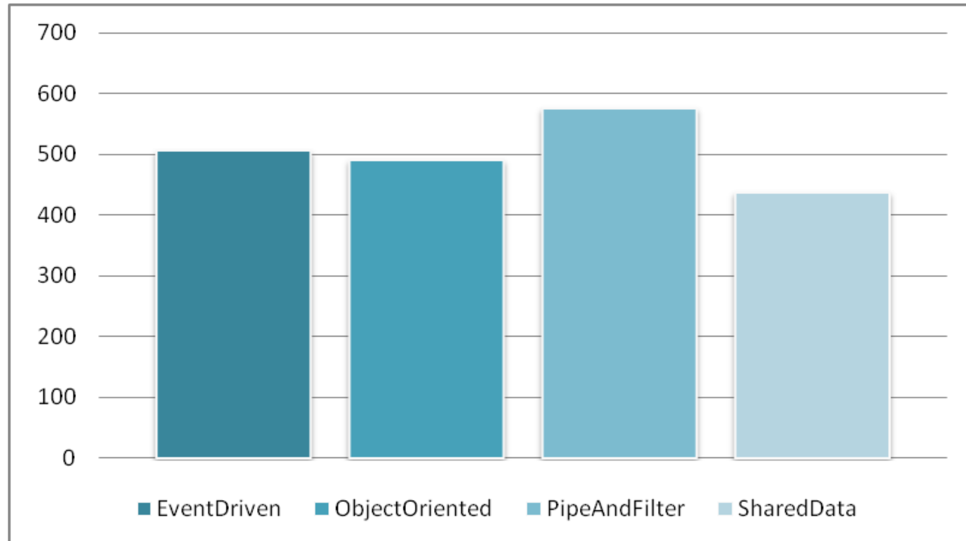
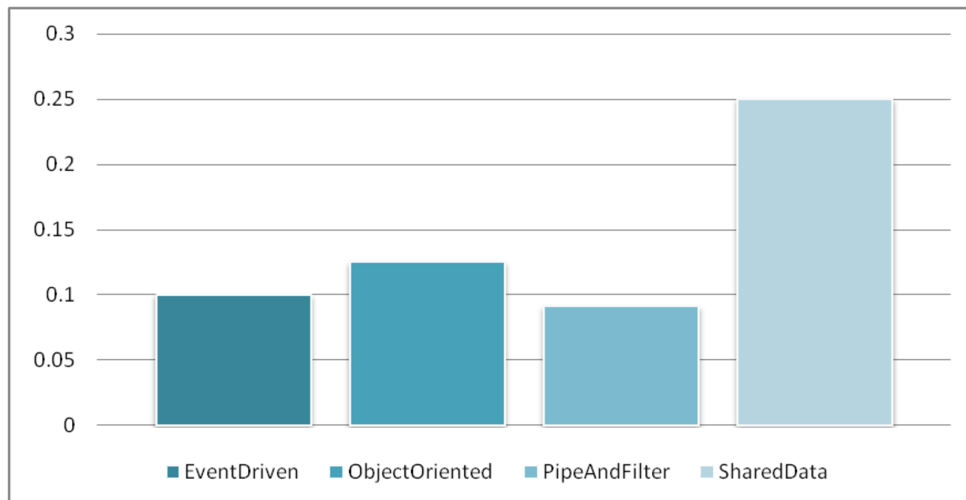
```
		{
			try
			{
			   DocumentBuilderFactory DBF = DocumentBuilderFactory.newInstance();
			   DocumentBuilder docBuilder = DBF.newDocumentBuilder();
			   Document doc = docBuilder.newDocument();

			   Element rootElement = doc.createElement("OutputToXML");
			   doc.appendChild(rootElement);

			   for (int i = 0; i < output.size(); i++)
			   {
				String element = "Element";
				String data = output.get(i);
				Element em = doc.createElement(element);
				em.appendChild(doc.createTextNode(data));
				rootElement.appendChild(em);
			   }
			   TransformerFactory transformerFactory = TransformerFactory.newInstance();
			   Transformer transformer = transformerFactory.newTransformer();
			   DOMSource source = new DOMSource(doc);

			   StreamResult result =  new StreamResult(this.FILENAME);
			   transformer.transform(source, result);
			}
			catch (TransformerException ex) {}
			catch (ParserConfigurationException ex) {}
		}
}
```

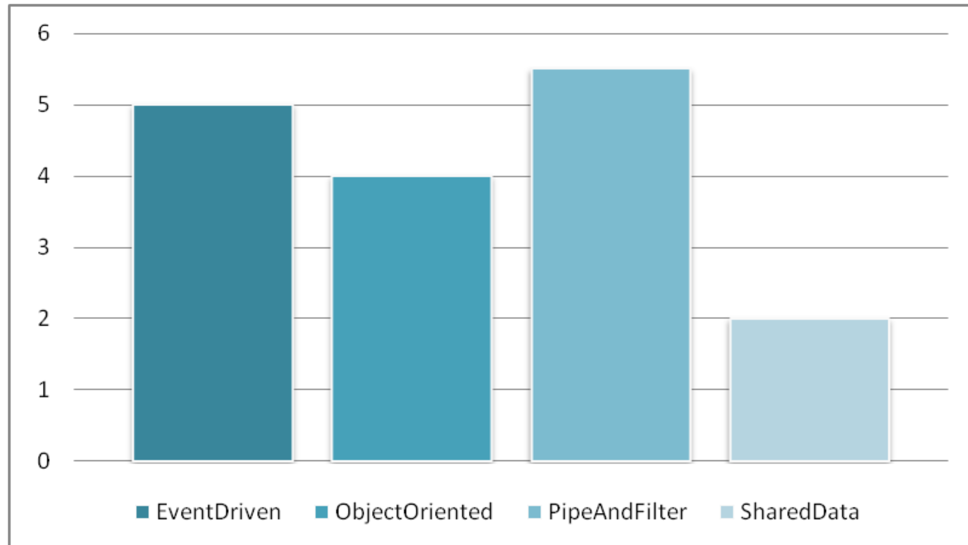
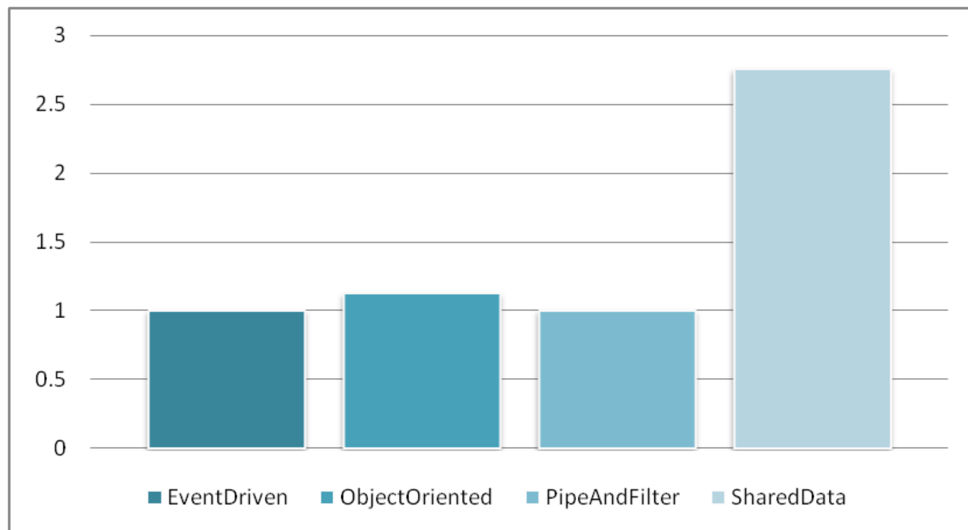
AspectJ Source (Sample)

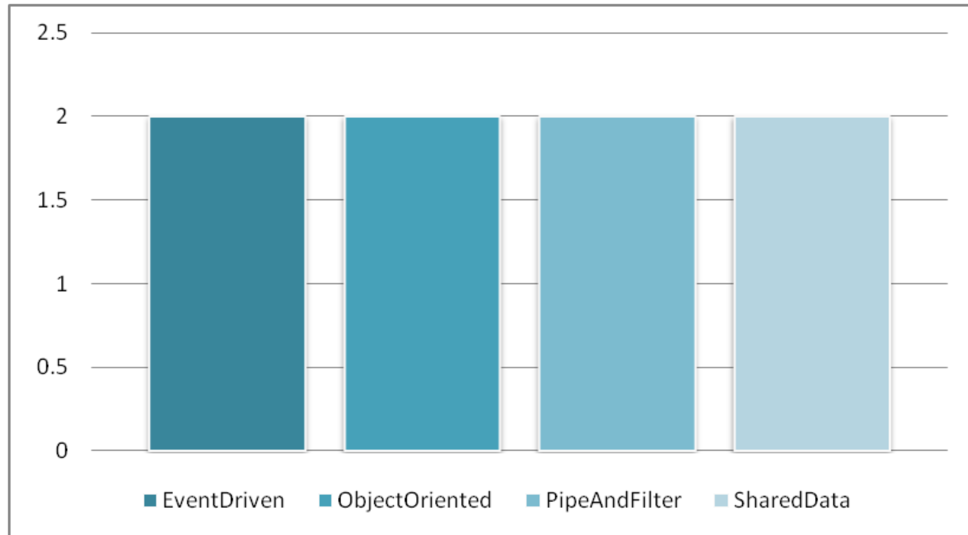
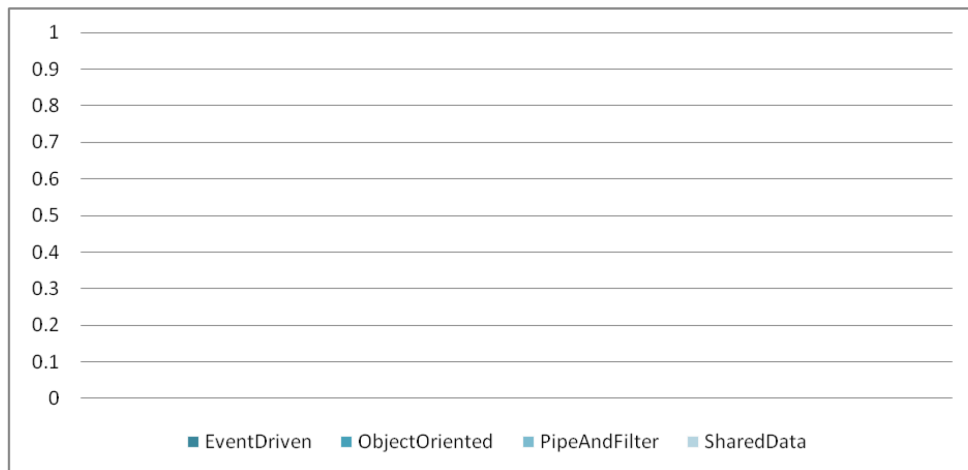
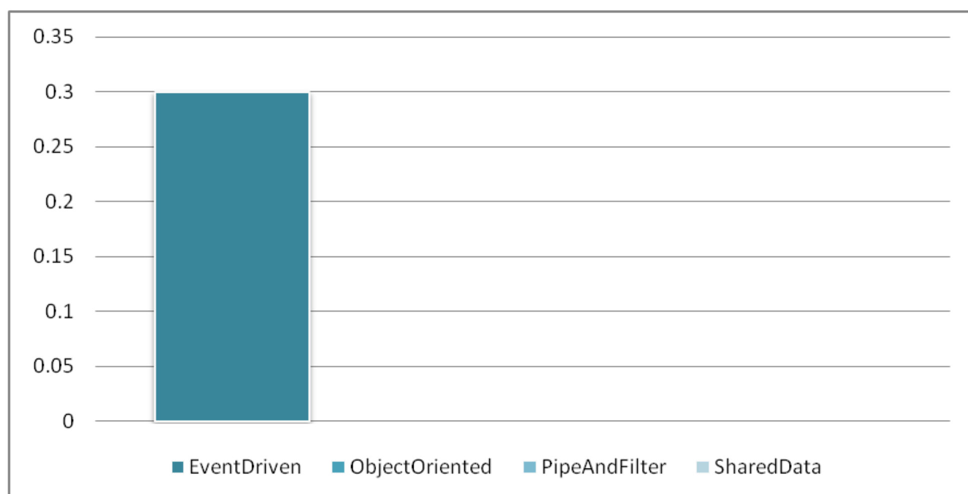
```
public privileged aspect AspectJ_OutputToXML {

		pointcut performanceCheck_Output(LineStorageWrapper shift_storage):
				call (* KWIC.EventDriven.Output.print(LineStorageWrapper))
				&& args(shift_storage);

		after(LineStorageWrapper shift_storage) : performanceCheck_Output(shift_storage)
		{
			try
			{
				ArrayList<String> _lines = new ArrayList<String>();
				String _filename = "C:\\temp\\EventDriven.XML";

				for(int i = 0; i < shift_storage.getLineCount(); i++)
					_lines.add(shift_storage.getLineAsString(i));

				outputXML.CreateXMLFile xml
		   = new outputXML.CreateXMLFile(_filename, _lines);
				System.out.println("Created XML successfully at: " + xml.getFilename());

			}
			catch (Exception e)
			{
				System.out.println(
		 "Exception at src.KWIC_PipeFilter.Aspect_KWIC_PipeFilter"
		   + e.toString());
			}
```

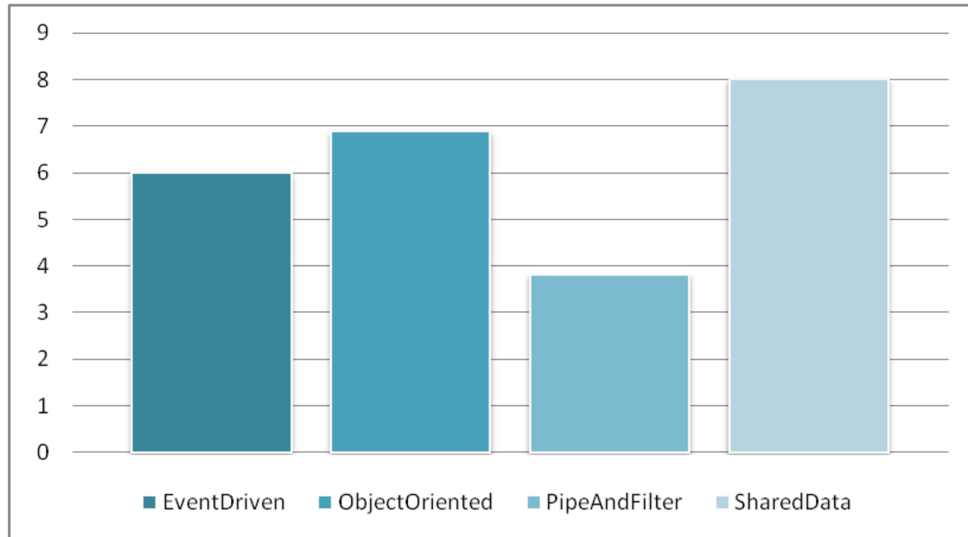
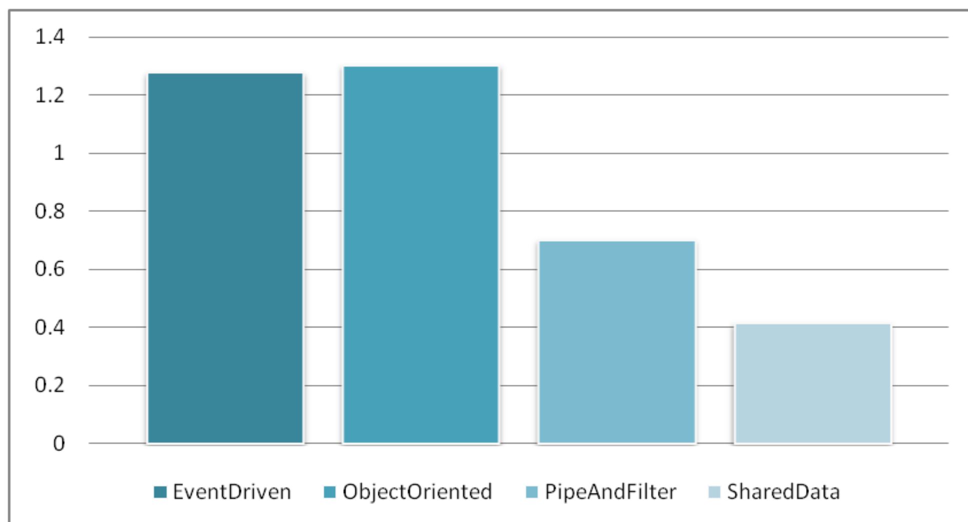
```
}
```

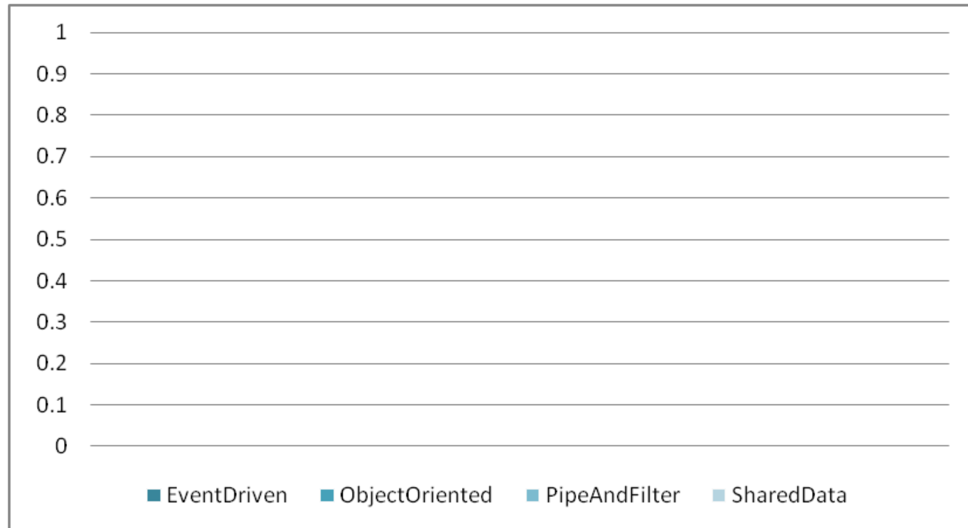
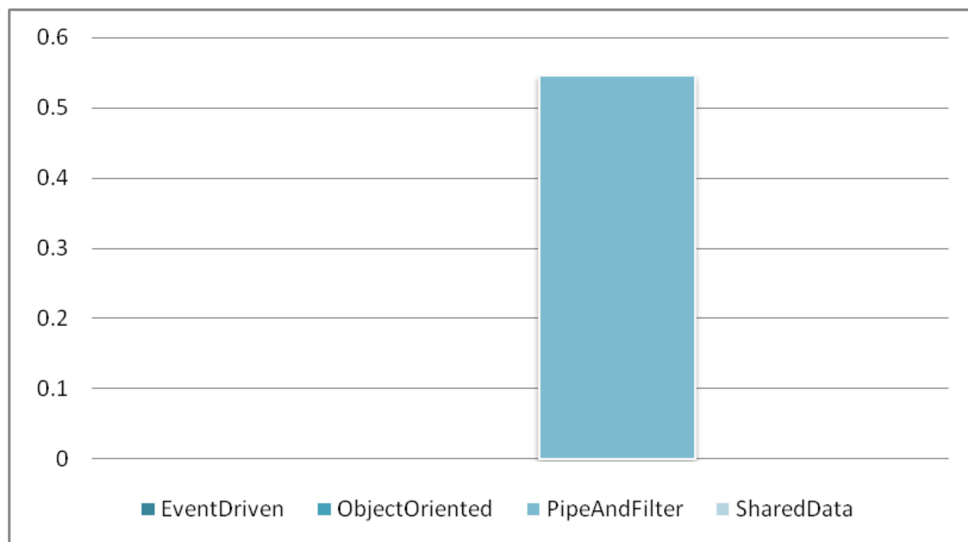
Metric Evaluation

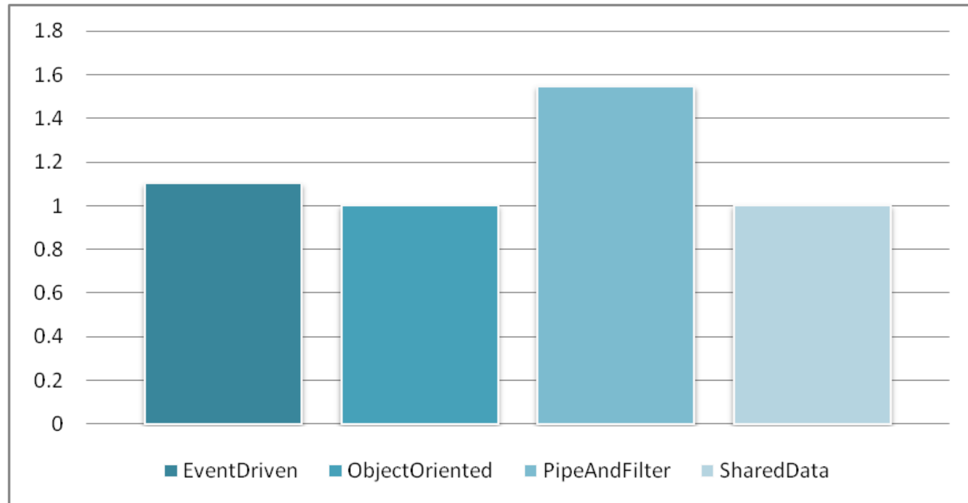
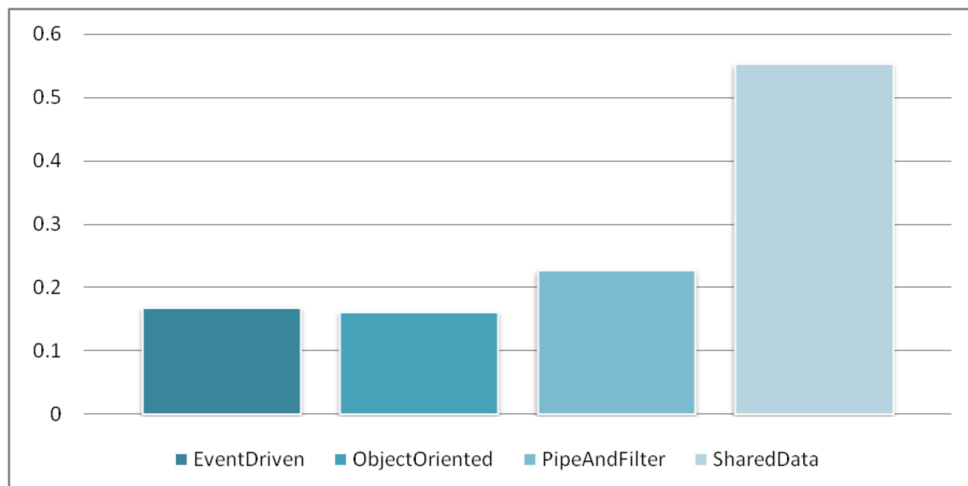
Lines of Code (LOC)	
Number of Static Methods (NSM)	

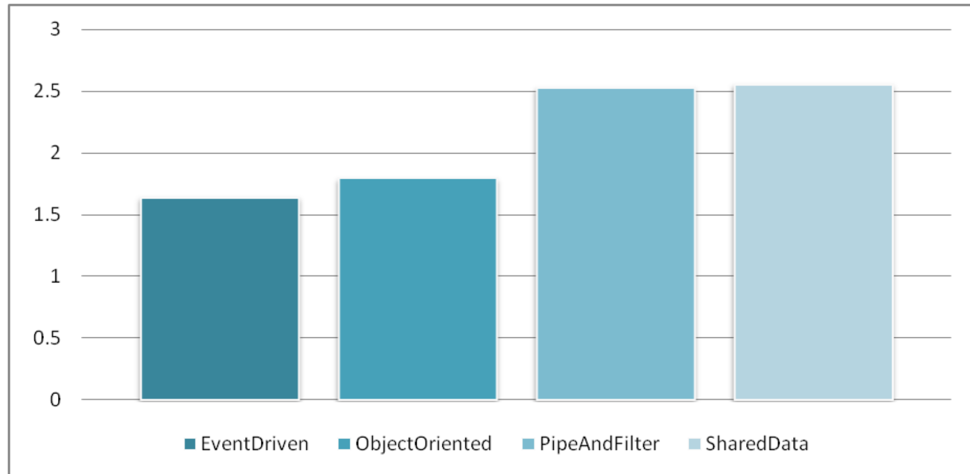
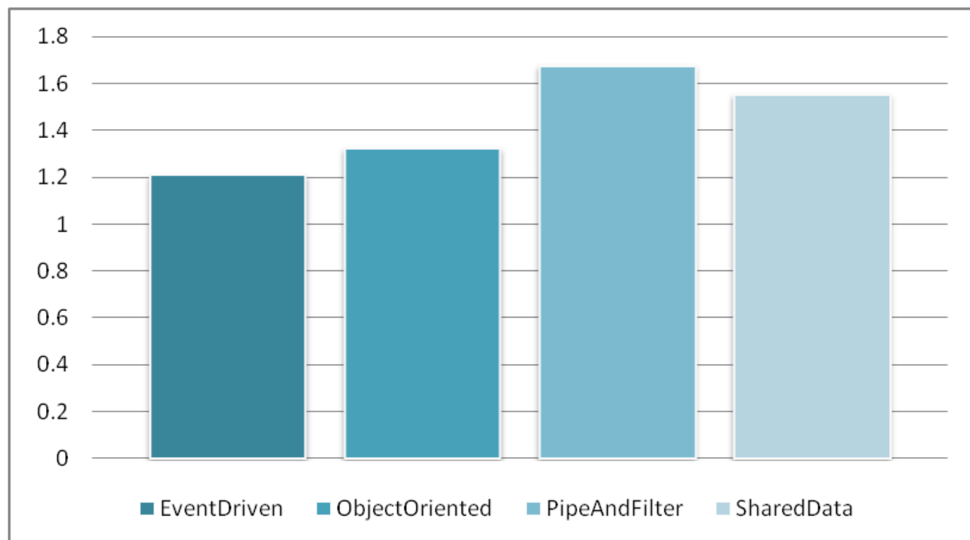
Number of Classes (NOC)	
Number of Attributes (NOA)	

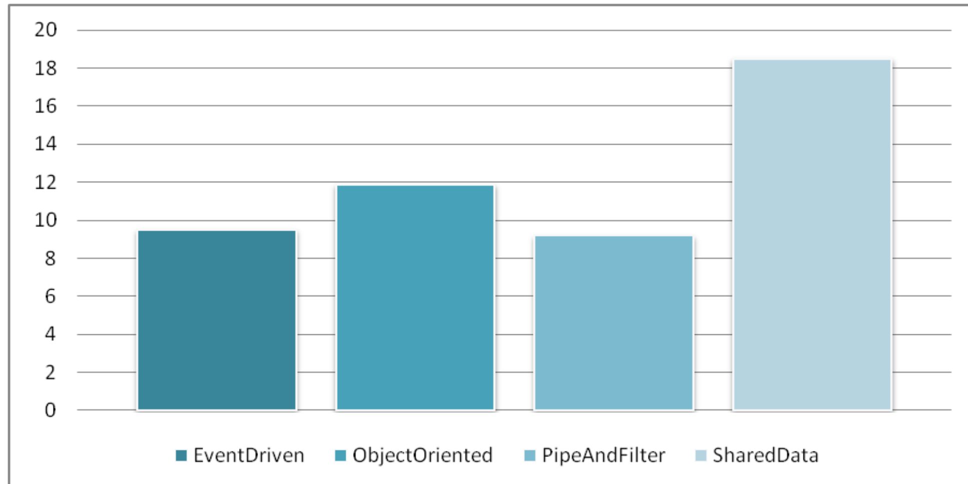
Number of Packages (NOP)	
Number of Overridden Methods (NORM)	
Number of Static Attributes (NSF)	

Number of Methods (NOM)	
Number of Parameter (PAR)	

Number of Interfaces (NOI)	
Number of Children (NSC)	

Depth of Inheritance Tree (DIT)	
Lack of Cohesion of Methods (LCOM)	

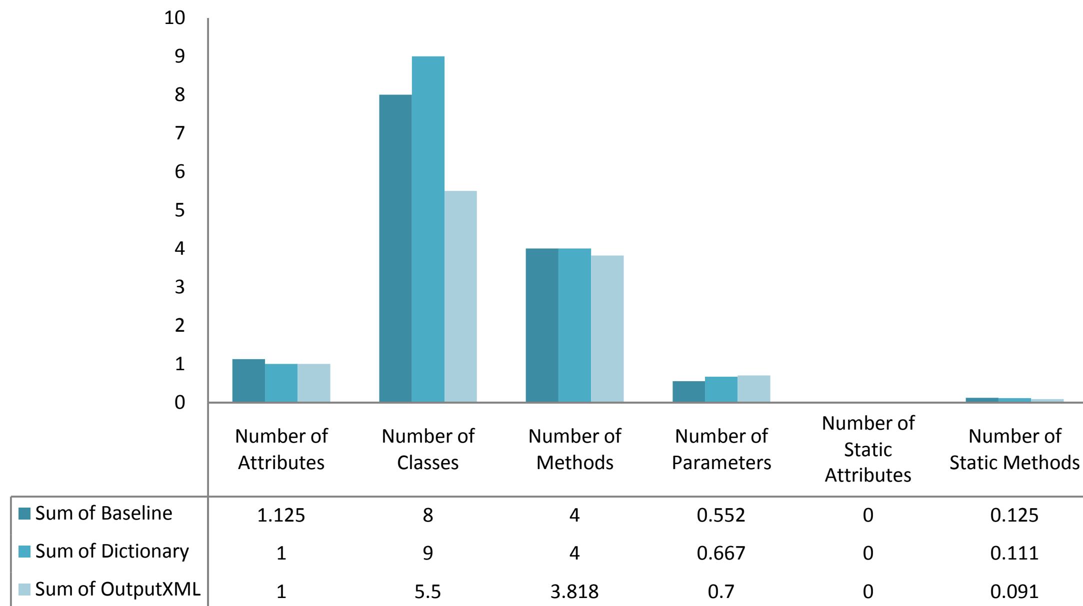
McCabe Cyclomatic Complexity	
Nested Block Depth	

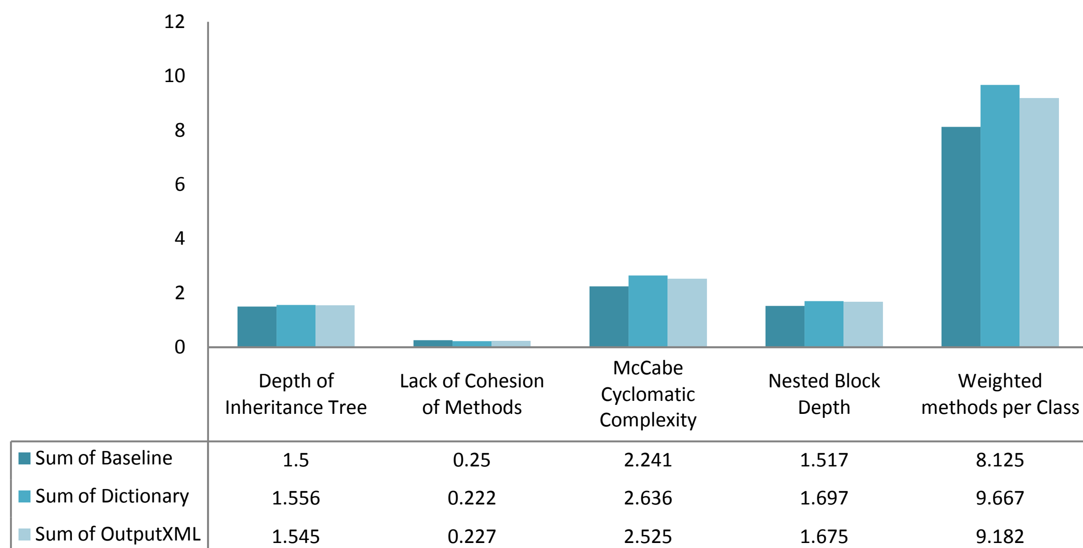
Weighted Methods per Class (WMC):	

Appendix D – Experiment Metric Evaluation

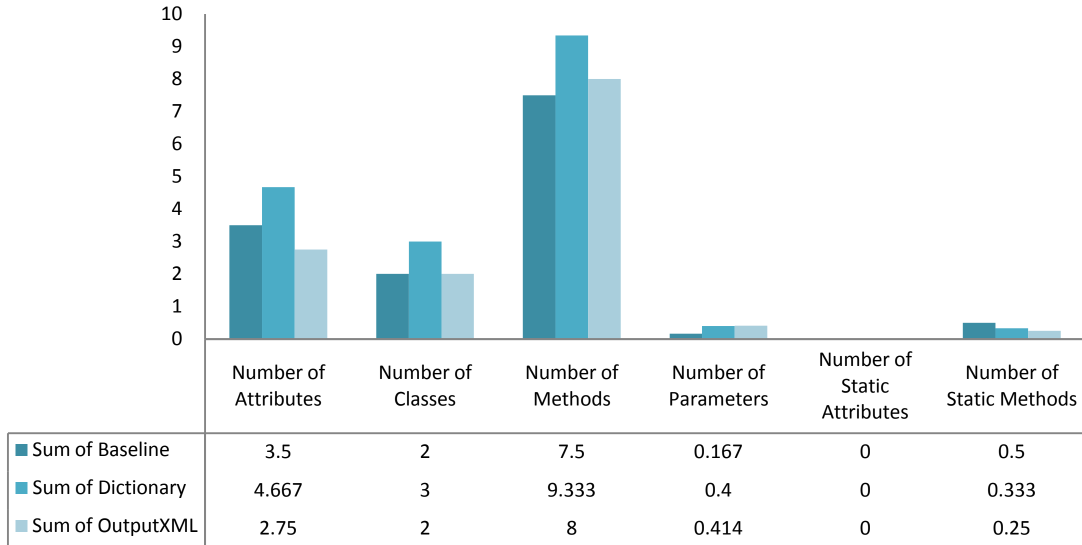
This section compares the effect of the three experiments on the source, by examining each Architectural Style.

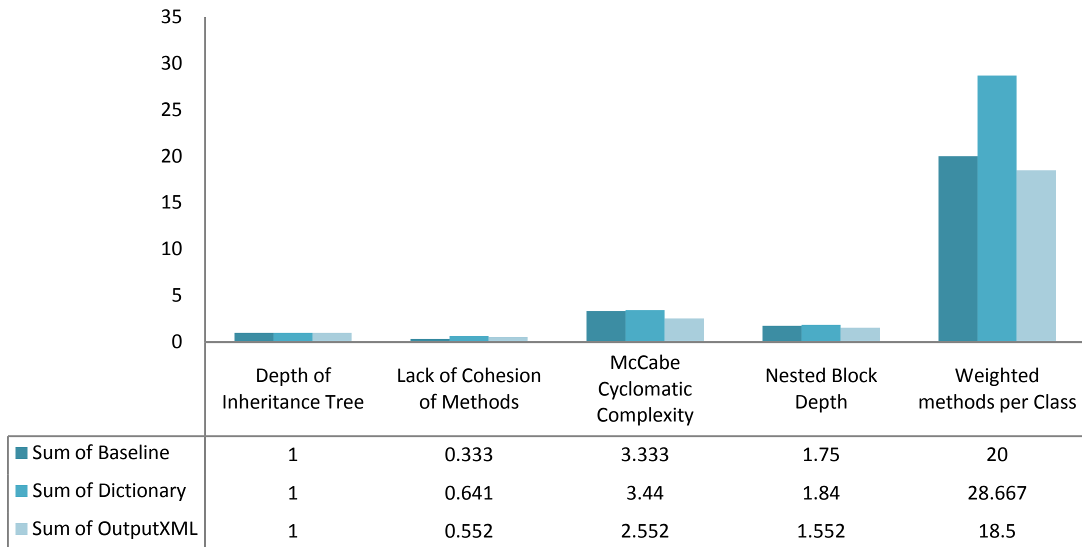
Pipe-and-Filter



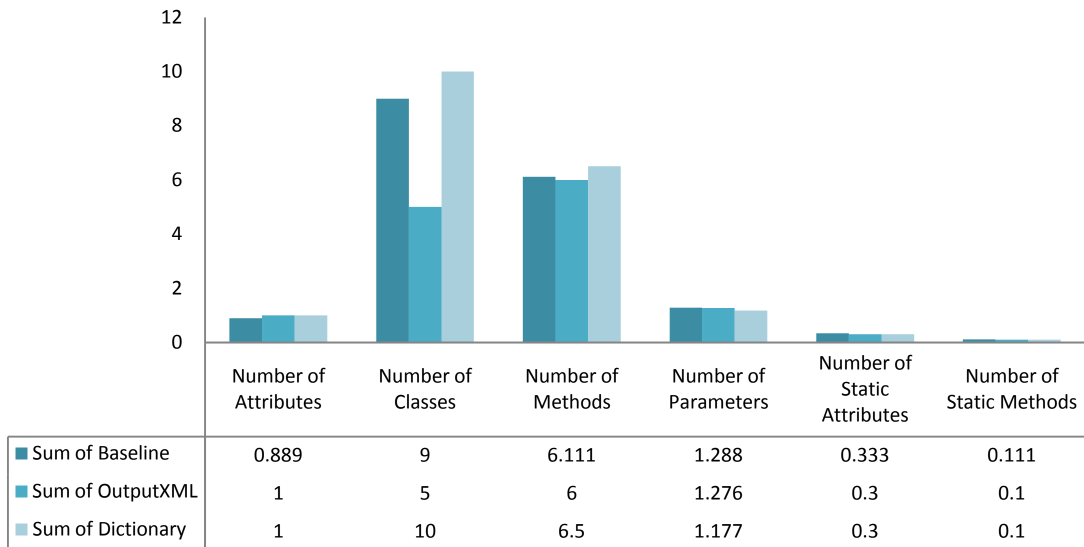


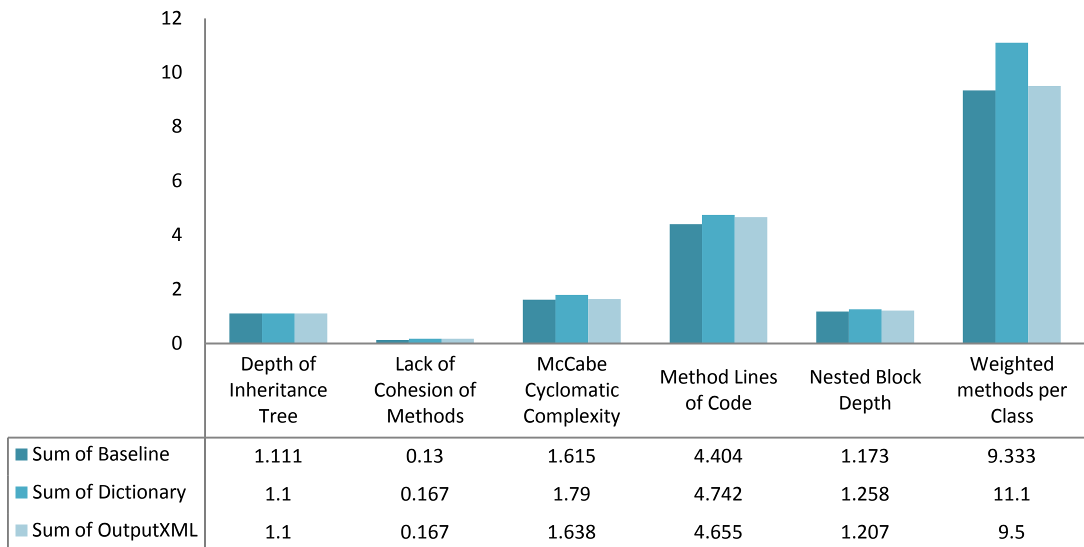
Shared-Data





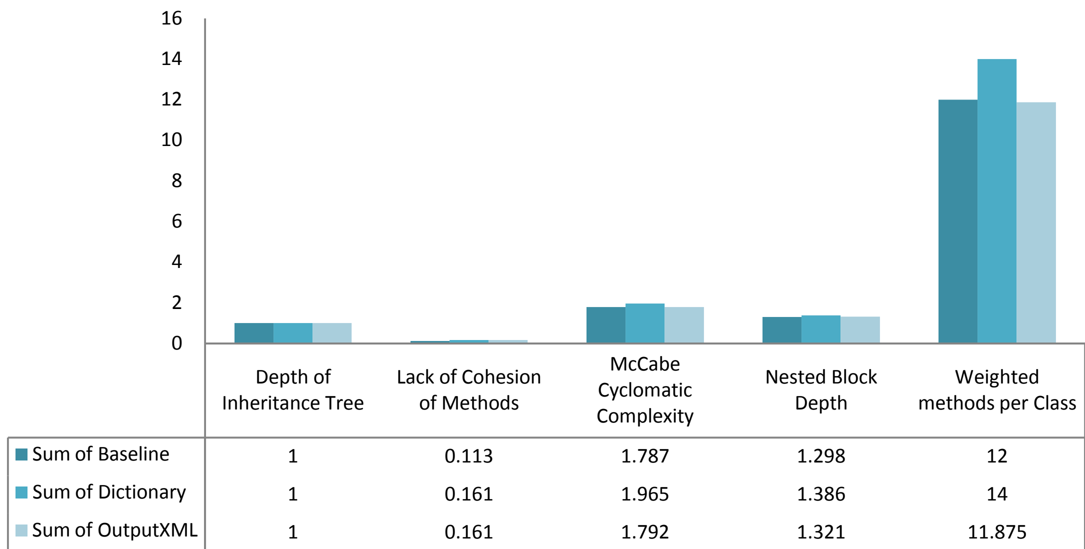
Event-Based





Object-Oriented

9
8
7
6
5
4
3
2
1
0
Number of Attributes
Number of Classes
Number of Methods
Number of Parameters
Number of Static Attributes
Number of Static Methods
Sum of Baseline 1 7 7.143 1.319 0 0.143
Sum of Dictionary 1.125 8 7.5 1.193 0 0.125
Sum of OutputXML 1.125 4 6.875 1.302 0 0.125


16
14
12
10
8
6
4
2
0
Depth of Inheritance Tree
Lack of Cohesion of Methods
McCabe Cyclomatic Complexity
Nested Block Depth
Weighted methods per Class
Sum of Baseline 1 0.113 1.787 1.298 12
Sum of Dictionary 1 0.161 1.965 1.386 14
Sum of OutputXML 1 0.161 1.792 1.321 11.875

Appendix E –AspectJ Source-Code

Event-Driven – Performance

```
package KWIC_EventDriven_Baseline;

public privileged aspect AspectJ_Performance {

	public double timerStart, timerEnd, timerTotal;

	pointcut performanceCheck_Input()	: call (* KWIC_EventDriven_Baseline.Input.parse(..));
	pointcut performanceCheck_CircularShift(): execution (* KWIC_EventDriven_Baseline.CircularShifter.update(..,..));
	pointcut performanceCheck_Alphabetizer() : execution (* KWIC_EventDriven_Baseline.Alphabetizer.update(.., ..));
	pointcut performanceCheck_Output()	: call (* KWIC_EventDriven_Baseline.Output.print(..));
	pointcut performanceCheck()		: execution ( * KWIC_EventDriven_Baseline.KWIC.main(..) );

	before() : performanceCheck()
	{
		try
		{
			timerStart = System.currentTimeMillis();
		   System.out.println(" START :: " + thisJoinPoint.toLongString() + Double.toString(timerStart));
		}
		catch (Exception e)
		{
			System.out.println("Exception at src.KWIC_PipeFilter.Aspect_KWIC_PipeFilter" + e.toString());
		}
	}
	after()  : performanceCheck()
	{
		try
		{
			timerEnd = System.currentTimeMillis();
			timerTotal = timerEnd - timerStart;
			System.out.println(" END :: " + thisJoinPoint.toLongString() + " :: " +
								Double.toString(timerStart) + " :: TOTAL TIME
:: (" +
								Double.toString(timerTotal) + ")");
			System.exit(1);
		}
		catch (Exception e)
		{
			System.out.println("Exception at src.KWIC_PipeFilter.Aspect_KWIC_PipeFilter" + e.toString());
		}
	}

	after()  : performanceCheck_Input()
	{
		try
		{
			timerEnd = System.currentTimeMillis();
			timerTotal = timerEnd - timerStart;
			System.out.println(" END :: " + thisJoinPoint.toLongString() + " :: " +
								Double.toString(timerStart) + " :: END TIME ::
(" +
```

```
									Double.toString(timerEnd) + ")");
		}
		catch (Exception e)
		{
			System.out.println("Exception at src.KWIC_PipeFilter.Aspect_KWIC_PipeFilter" + e.toString());
		}
	}
	after()  : performanceCheck_CircularShift()
	{
		try
		{
			timerEnd = System.currentTimeMillis();
			timerTotal = timerEnd - timerStart;
			System.out.println(" END :: " + thisJoinPoint.toLongString() + " :: " +
									Double.toString(timerStart) + " :: END TIME ::
(" +
									Double.toString(timerEnd) + ")");
		}
		catch (Exception e)
		{
			System.out.println("Exception at src.KWIC_PipeFilter.Aspect_KWIC_PipeFilter" + e.toString());
		}
	}
	after()  : performanceCheck_Alphabetizer()
	{
		try
		{
			timerEnd = System.currentTimeMillis();
			timerTotal = timerEnd - timerStart;
			System.out.println(" END :: " + thisJoinPoint.toLongString() + " :: " +
									Double.toString(timerStart) + " :: END TIME ::
(" +
									Double.toString(timerEnd) + ")");
		}
		catch (Exception e)
		{
			System.out.println("Exception at src.KWIC_PipeFilter.Aspect_KWIC_PipeFilter" + e.toString());
		}
	}
	after()  : performanceCheck_Output()
	{
		try
		{
			timerEnd = System.currentTimeMillis();
			timerTotal = timerEnd - timerStart;
			System.out.println(" END :: " + thisJoinPoint.toLongString() + " :: " +
									Double.toString(timerStart) + " :: END TIME ::
(" +
									Double.toString(timerEnd) + ")");
		}
		catch (Exception e)
		{
			System.out.println("Exception at src.KWIC_PipeFilter.Aspect_KWIC_PipeFilter" + e.toString());
		}
	}
}
```

Object-Oriented – Performance

```
package KWIC_ObjectOriented_Baseline;

public privileged aspect AspectJ_Performance {
```

```
	public double timerStart, timerEnd, timerTotal;

	pointcut performanceCheck_Input()	: call (* KWIC_ObjectOriented_Baseline.Input.parse(String, LineStorage));
	pointcut performanceCheck_CircularShift(): call (* KWIC_ObjectOriented_Baseline.CircularShifter.setup(..));
	pointcut performanceCheck_Alphabetizer() : call (* KWIC_ObjectOriented_Baseline.Alphabetizer.alpha(..));
	pointcut performanceCheck_Output()	: call (* KWIC_ObjectOriented_Baseline.Output.print(..));
	pointcut performanceCheck()		: execution ( * KWIC_ObjectOriented_Baseline.KWIC.execute(..) );

	before() : performanceCheck()
	{
		try
		{
			timerStart = System.currentTimeMillis();
		   System.out.println(" START :: " + thisJoinPoint.toLongString() + Double.toString(timerStart));
		}
		catch (Exception e)
		{
			System.out.println("Exception at " + e.toString());
		}
	}
	after()  : performanceCheck()
	{
		try
		{
			timerEnd = System.currentTimeMillis();
			timerTotal = timerEnd - timerStart;
			System.out.println(" END :: " + thisJoinPoint.toLongString() + " :: " +
								Double.toString(timerStart) + " \n TOTAL TIME
:: (" +
								Double.toString(timerTotal) + ")");
			System.exit(1);
		}
		catch (Exception e)
		{
			System.out.println("Exception at " + e.toString());
		}
	}

	after()  : performanceCheck_Input()
	{
		try
		{
			timerEnd = System.currentTimeMillis();
			timerTotal = timerEnd - timerStart;
			System.out.println(" END :: " + thisJoinPoint.toLongString() + " :: " +
								Double.toString(timerStart) + " :: END TIME ::
(" +
								Double.toString(timerEnd) + ")");
		}
		catch (Exception e)
		{
			System.out.println("Exception at " + e.toString());
		}
	}
	after()  : performanceCheck_CircularShift()
	{
		try
		{
			timerEnd = System.currentTimeMillis();
			timerTotal = timerEnd - timerStart;
			System.out.println(" END :: " + thisJoinPoint.toLongString() + " :: " +
```

```
										Double.toString(timerStart) + " :: END TIME ::
(" +
										Double.toString(timerEnd) + ")");
			}
			catch (Exception e)
			{
				System.out.println("Exception at " + e.toString());
			}
		}
		after()  : performanceCheck_Alphabetizer()
		{
			try
			{
				timerEnd = System.currentTimeMillis();
				timerTotal = timerEnd - timerStart;
				System.out.println(" END :: " + thisJoinPoint.toLongString() + " :: " +
										Double.toString(timerStart) + " :: END TIME ::
(" +
										Double.toString(timerEnd) + ")");
			}
			catch (Exception e)
			{
				System.out.println("Exception at " + e.toString());
			}
		}
		after()  : performanceCheck_Output()
		{
			try
			{
				timerEnd = System.currentTimeMillis();
				timerTotal = timerEnd - timerStart;
				System.out.println(" END :: " + thisJoinPoint.toLongString() + " :: " +
										Double.toString(timerStart) + " :: END TIME ::
(" +
										Double.toString(timerEnd) + ")");
			}
			catch (Exception e)
			{
				System.out.println("Exception at " + e.toString());
			}
		}

}
```

Pipe-and-Filter – Performance

```
package KWIC_PipeAndFilter_Baseline;


public privileged aspect ASpectJ_Perfomance {
		public double timerStart, timerEnd, timerTotal;

		pointcut performanceCheck_Input(): call (* KWIC_PipeAndFilter_Baseline.Input.start(..));

		pointcut performanceCheck_CircularShift(): call (* KWIC_PipeAndFilter_Baseline.CircularShifter.start(..));
		pointcut performanceCheck_Alphabetizer() : call (* KWIC_PipeAndFilter_Baseline.Alphabetizer.start(..));
		pointcut performanceCheck_Output()	: call (* KWIC_PipeAndFilter_Baseline.Output.start(..));

		pointcut performanceCheck(): execution ( * KWIC_PipeAndFilter_Baseline.KWIC.main(..) );
```

```
	before() : performanceCheck()
	{
		try
		{
			timerStart = System.currentTimeMillis();
		   System.out.println(" START :: " + thisJoinPoint.toLongString() + Double.toString(timerStart));
		}
		catch (Exception e)
		{
			System.out.println("Exception at src.KWIC_PipeFilter.Aspect_KWIC_PipeFilter" + e.toString());
		}
	}
	after()  : performanceCheck()
	{
		try
		{
			timerEnd = System.currentTimeMillis();
			timerTotal = timerEnd - timerStart;
			System.out.println(" END :: " + thisJoinPoint.toLongString() + " :: " +
									Double.toString(timerStart) + " :: TOTAL TIME
:: (" +
									Double.toString(timerTotal) + ")");
			System.exit(1);
		}
		catch (Exception e)
		{
			System.out.println("Exception at src.KWIC_PipeFilter.Aspect_KWIC_PipeFilter" + e.toString());
		}
	}

	after()  : performanceCheck_Input()
	{
		try
		{
			timerEnd = System.currentTimeMillis();
			timerTotal = timerEnd - timerStart;
			System.out.println(" END :: " + thisJoinPoint.toLongString() + " :: " +
									Double.toString(timerStart) + " :: END TIME ::
(" +
									Double.toString(timerEnd) + ")");
		}
		catch (Exception e)
		{
			System.out.println("Exception at src.KWIC_PipeFilter.Aspect_KWIC_PipeFilter" + e.toString());
		}
	}
	after()  : performanceCheck_CircularShift()
	{
		try
		{
			timerEnd = System.currentTimeMillis();
			timerTotal = timerEnd - timerStart;
			System.out.println(" END :: " + thisJoinPoint.toLongString() + " :: " +
									Double.toString(timerStart) + " :: END TIME ::
(" +
									Double.toString(timerEnd) + ")");
		}
		catch (Exception e)
		{
			System.out.println("Exception at src.KWIC_PipeFilter.Aspect_KWIC_PipeFilter" + e.toString());
		}
	}
```

```
	after()  : performanceCheck_Alphabetizer()
	{
		try
		{
			timerEnd = System.currentTimeMillis();
			timerTotal = timerEnd - timerStart;
			System.out.println(" END :: " + thisJoinPoint.toLongString() + " :: " +
								Double.toString(timerStart) + " :: END TIME ::
(" +
								Double.toString(timerEnd) + ")");
		}
		catch (Exception e)
		{
			System.out.println("Exception at src.KWIC_PipeFilter.Aspect_KWIC_PipeFilter" + e.toString());
		}
	}
	after()  : performanceCheck_Output()
	{
		try
		{
			timerEnd = System.currentTimeMillis();
			timerTotal = timerEnd - timerStart;
			System.out.println(" END :: " + thisJoinPoint.toLongString() + " :: " +
								Double.toString(timerStart) + " :: END TIME ::
(" +
								Double.toString(timerEnd) + ")");
		}
		catch (Exception e)
		{
			System.out.println("Exception at src.KWIC_PipeFilter.Aspect_KWIC_PipeFilter" + e.toString());
		}
	}

}
```

Shared-Data – Performance

```
package KWIC_SharedData_Baseline;

public privileged aspect AspectJ_Perfomance {
	public double timerStart, timerEnd, timerTotal;

	pointcut performanceCheck_Output()       : execution (* KWIC_SharedData_Baseline.KWIC.output() );
	pointcut performanceCheck_Input()        : call      (* KWIC_SharedData_Baseline.KWIC.input(..));
	pointcut performanceCheck_CircularShift(): call      (* KWIC_SharedData_Baseline.KWIC.circularShift());
	pointcut performanceCheck_Alphabetizer() : call      (* KWIC_SharedData_Baseline.KWIC.alphabetizing());
	pointcut performanceCheck()              : execution (* KWIC_SharedData_Baseline.KWIC.main(..) );

	before() : performanceCheck()
	{
		try
		{
			timerStart = System.currentTimeMillis();
		    System.out.println(" START :: " + thisJoinPoint.toLongString() + Double.toString(timerStart));
		}
		catch (Exception e)
		{
			System.out.println("Exception at src.KWIC_PipeFilter.Aspect_KWIC_PipeFilter" + e.toString());
		}
	}
	after()  : performanceCheck()
	{
```

```
		try
		{
			timerEnd = System.currentTimeMillis();
			timerTotal = timerEnd - timerStart;
			System.out.println(" END :: " + thisJoinPoint.toLongString() + " :: " +
								Double.toString(timerStart) + " :: TOTAL TIME
:: (" +
								Double.toString(timerTotal) + ")");
			System.exit(1);
		}
		catch (Exception e)
		{
			System.out.println("Exception at src.KWIC_PipeFilter.Aspect_KWIC_PipeFilter" + e.toString());
		}
	}


	after() returning : performanceCheck_Input()
	{
		try
		{
			timerEnd = System.currentTimeMillis();
			timerTotal = timerEnd - timerStart;
			System.out.println(" END :: " + thisJoinPoint.toLongString() + " :: " +
								Double.toString(timerStart) + " :: END TIME ::
(" +
								Double.toString(timerEnd) + ")");
		}
		catch (Exception e)
		{
			System.out.println("Exception at src.KWIC_PipeFilter.Aspect_KWIC_PipeFilter" + e.toString());
		}
	}
	after()  : performanceCheck_CircularShift()
	{
		try
		{
			timerEnd = System.currentTimeMillis();
			timerTotal = timerEnd - timerStart;
			System.out.println(" END :: " + thisJoinPoint.toLongString() + " :: " +
								Double.toString(timerStart) + " :: END TIME ::
(" +
								Double.toString(timerEnd) + ")");
		}
		catch (Exception e)
		{
			System.out.println("Exception at src.KWIC_PipeFilter.Aspect_KWIC_PipeFilter" + e.toString());
		}
	}
	after()  : performanceCheck_Alphabetizer()
	{
		try
		{
			timerEnd = System.currentTimeMillis();
			timerTotal = timerEnd - timerStart;
			System.out.println(" END :: " + thisJoinPoint.toLongString() + " :: " +
								Double.toString(timerStart) + " :: END TIME ::
(" +
								Double.toString(timerEnd) + ")");
		}
		catch (Exception e)
		{
```

```
				System.out.println("Exception at src.KWIC_PipeFilter.Aspect_KWIC_PipeFilter" + e.toString());
			}
		}
		after()  : performanceCheck_Output()
		{
			try
			{
				timerEnd = System.currentTimeMillis();
				timerTotal = timerEnd - timerStart;
				System.out.println(" END :: " + thisJoinPoint.toLongString() + " :: " +
									Double.toString(timerStart) + " :: END TIME ::
(" +
									Double.toString(timerEnd) + ")");
			}
			catch (Exception e)
			{
				System.out.println("Exception at src.KWIC_PipeFilter.Aspect_KWIC_PipeFilter" + e.toString());
			}
		}
}
```

Event-Based – Output To XML

```
package KWIC_EventDriven_OutputToXML;

import java.util.ArrayList;

public privileged aspect AspectJ_Performance {

		pointcut performanceCheck_Output(LineStorageWrapper shift_storage):
				call (* KWIC_EventDriven_OutputToXML.Output.print(LineStorageWrapper))
				&& args(shift_storage);

		after(LineStorageWrapper shift_storage) : performanceCheck_Output(shift_storage)
		{
			try
			{
				ArrayList<String> _lines = new ArrayList<String>();
				String _filename = "C:\\temp\\EventDriven.XML";

				for(int i = 0; i < shift_storage.getLineCount(); i++)
					_lines.add(shift_storage.getLineAsString(i));

				outputXML.CreateXMLFile xml = new outputXML.CreateXMLFile(_filename, _lines);
				System.out.println("Created XML successfully at: " + xml.getFilename());

			}
			catch (Exception e)
			{
				System.out.println("Exception at src.KWIC_PipeFilter.Aspect_KWIC_PipeFilter" + e.toString());
			}
		}
}
```

Object-Oriented – Output To XML

```
package KWIC_ObjectOriented_OutputToXML;

import java.util.ArrayList;
```

```
public privileged aspect AspectJ_Performance {
	pointcut performanceCheck_Output(Alphabetizer alphabetizer): call (*
KWIC_ObjectOriented_OutputToXML.Output.print(Alphabetizer))   && args(alphabetizer);

	before(Alphabetizer alphabetizer) : performanceCheck_Output(alphabetizer)
	{
		try
		{
			ArrayList<String> _lines = new ArrayList<String>();
			String _filename = "C:\\temp\\Test2.XML";

			for(int i = 0; i < alphabetizer.getLineCount(); i++)
				_lines.add(alphabetizer.getLineAsString(i));

			outputXML.CreateXMLFile xml = new outputXML.CreateXMLFile(_filename, _lines);
			System.out.println("Created XML successfully at: " + xml.getFilename());

		}
		catch (Exception e)
		{
			System.out.println("Exception at src.KWIC_PipeFilter.Aspect_KWIC_PipeFilter" + e.toString());
		}
	}
}
```

Pipe-and-Filter – Output to XML

```
package KWIC_PipeAndFilter_OutputToXML;

public privileged aspect ASpectJ_Perfomance {
	pointcut performanceCheck_Output() : call (* KWIC_PipeAndFilter_OutputToXML.Output.start(..));

	after ()  : performanceCheck_Output()
	{
		try
		{
			Output _target = (Output) thisJoinPoint.getTarget();
			outputXML.OutputToXML _output = new outputXML.OutputToXML(_target.input_);
			_output.start();
			_output.transform();

		}
		catch (Exception e)
		{
			System.out.println("Exception at src.KWIC_PipeFilter.Aspect_KWIC_PipeFilter" + e.toString());
		}
	}
}
```

Shared-Data – Output to XML

```
package KWIC_SharedData_OutputToXML;

import java.util.ArrayList;

public privileged aspect AspectJ_Perfomance {

	pointcut performanceCheck_Output()	: call (* KWIC_SharedData_OutputToXML.KWIC.output() );

	after()  : performanceCheck_Output()
```

```
	{
		try
		{
			KWIC _target = (KWIC)thisJoinPoint.getTarget();

			char[]  _chars          = _target.getChars_();
			int[]   _line_index     = _target.getline_index();
			int[][] _alphabetized   = _target.getAlphabetized();

			ArrayList<String> _lines = new ArrayList<String>();
			String _filename        = "C:\\temp\\SharedData.XML";

			String _str             = "";

			for(int i = 0; i < _alphabetized[0].length; i++)
			{
			   int line_count = _alphabetized[0][i];
			   int shift_begin = _alphabetized[1][i];
			   int line_begin = _line_index[line_count];
			   int line_end = 0;
			   if(line_count == (_line_index.length - 1))
			     line_end = _chars.length;
			   else
			     line_end = _line_index[line_count + 1];

			   if(line_begin != shift_begin){
			     for(int j = shift_begin; j < line_end; j++)
			       _str += _chars[j];
			       _str += ' ';
			     for(int j = line_begin; j < (shift_begin - 1); j++)
				_str += _chars[j];
			   }else
			     for(int j = line_begin; j < line_end; j++)
				_str += _chars[j];
			   _lines.add(_str);
			  }

			outputXML.CreateXMLFile xml = new outputXML.CreateXMLFile(_filename, _lines);
			System.out.println("Created XML successfully at: " + xml.getFilename());

		}
		catch (Exception e)
		{
			System.out.println("Exception at src.KWIC_PipeFilter.Aspect_KWIC_PipeFilter" + e.toString());
		}
	}
}
```