DESIGN OF AN OBJECT-ORIENTED FRAMEWORK FOR DATA FORMAT CLASSIFICATION AND TRANSFORMATION

BY

DUSTIN GRAVES

B.S. IN COMPUTER SCIENCE, MAY 1999, THE GEORGE WASHINGTON UNIVERSITY

A THESIS SUBMITTED TO

THE FACULTY OF

THE SCHOOL OF ENGINEERING AND APPLIED SCIENCE
OF THE GEORGE WASHINGTON UNIVERSITY
IN PARTIAL SATISFACTION OF THE REQUIREMENTS
FOR THE DEGREE OF MASTER OF SCIENCE

MAY 18, 2008

THESIS DIRECTED BY

RAHUL SIMHA
PROFESSOR OF ENGINEERING AND APPLIED SCIENCE
DEPARTMENT OF COMPUTER SCIENCE
THE GEORGE WASHINGTON UNIVERSITY

© COPYRIGHT 2008 DUSTIN GRAVES

ALL RIGHTS RESERVED

Abstract

Ensuring interoperability among software applications and devices which must exchange data can be a complex task when dealing with multiple data producers and consumers that operate on different data formats. Standard data formats can help to reduce the complexity associated with interoperability, but there are many situations where adherence to a single standard format is impractical or even impossible. Multiple standards may exist within the same application domain, standards may not exist for research and development projects utilizing specialized metrics defined for a specific subject area, and interaction with legacy systems which cannot be modified to support standard formats may be required.

This thesis presents an object-oriented framework for data format classification and transformation. The framework is designed to simplify the process of data interchange among applications and devices. Utilizing the flexibility of dynamic polymorphism and reusable design patterns, the framework defines a collection of extensible objects encapsulating common functional properties of the data format classification and transformation process. The framework may be integrated directly with application software, or may serve as the basis for independent data conversion applications providing data transformation gateways between two or more software applications or devices.

Designed to process data received simultaneously from one or more sources, the framework is capable of exploiting the parallelism inherent to independent streams of structured data while scaling to support implementation for systems with performance characteristics ranging from high-performance multi-core systems to simple embedded systems.

Table of Contents

Abstract iii

Table of Figures vii

Table of Tables viii

1. Introduction 1

  1.1 Problem statement 2

  1.2 Motivation 6

  1.3 Related Work 10

    1.3.1 Object Domains 11

    1.3.2 Frameworks 12

    1.3.3 Design Patterns 14

    1.3.4 Linear Types for Packet Processing 16

  1.4 Approach 17

    1.4.1 Addressing the Problem with the Framework 18

    1.4.2 Applying the Framework to the Problem 21

  1.5 Thesis Organization 22

2. Important Properties of Data Formats and Sources 22

  2.1 Properties of Data Formats 23

  2.2 Properties of Data Streams 24

  2.3 Exploiting Data Parallelism 26

3. Framework Design 27

3.1 Source Monitor 28

3.2 Pipeline Executor 29

3.3 Framework Component Specifications 30

3.3.1 Format 30

3.3.2 Format Group 30

3.3.3 Source 31

3.3.4 Pipeline 32

3.3.5 Classifier 34

3.3.6 Cache 34

3.3.7 Delegate 35

3.3.8 Decomposer 35

3.3.9 Monitor 36

3.3.10 Executor 36

3.4 Downscaling 37

4. Results 38

4.1 Implementation 38

4.1.1 Source Monitor Implementation 39

4.1.2 Pipeline Executor Implementation 39

4.2 Dynamic Creation of Format and Classifier Objects 41

4.3 Memory Management 42

4.4 Processing Simple Finite Data Streams 42

5. Usage 44

5.1 Data Converter Example 44

5.2 Data Consumer Example 46

5.3 Data Producer Example 48

6. Conclusion and Future Direction 49

7. Acknowledgements 51

Bibliography 52

Appendix A 55

Table of Figures

Figure 1 - Sensor Network with Multiple Data Formats ... 7

Figure 2 - Sensor Network with a Common Data Format ... 8

Figure 3 - Example Situational Awareness Display with Nominal and Actual Data ... 9

Figure 4 - Interaction of Framework Modules ... 27

Figure 5 - Data Flow through the Source Monitor Module ... 28

Figure 6 - Data Flow through the Pipeline Executor Module ... 29

Figure 7- Comparison of Filter and Filter Group Execution ... 34

Table of Tables

Table 1 - Format Interface ... 30

Table 2 - Source Interface ... 31

Table 3 - Pipeline Interface ... 32

Table 4 - Pipeline Function Interfaces ... 33

Table 5 - Classifier Interface ... 34

Table 6 - Cache Interface ... 35

Table 7 - Delegate Interface ... 35

Table 8 - Monitor Interface ... 36

Table 9 - Executor Interface ... 37

1. Introduction

Comprehensive data exchange among applications and devices is dependent upon the type and organization of the data streams through which they communicate. Applications and devices exchanging data within a well-structured environment may do so through a common data format, while those exchanging data in a less structured environment may make no assertions regarding data format. The existence of proprietary application formats, legacy systems, and devices with limited computational resources within a domain may make adherence to a single comprehensive data format difficult. The problem is further complicated by the existence of multiple formats describing similar data types within the same domain, such as image and video coding (Sayood, 2006), and application standards which allow multiple formats, such as the HD DVD standard (DVD Forum, 2006), requiring applications and devices to support multiple formats or define a protocol for negotiating which specific format to use.

Exchange of data among applications and devices supporting disparate formats is a problem that has long faced the software development industry. Attempts to address the problem have led to the definition of standard data interchange formats such as IFF (Morrison, 1985), SDF (Hall, 1993), and EDI (NIST, 1996). Such standards are meant to be integrated with applications and devices to facilitate effective application-to-application data exchange. To be effective, application code integrating standard data format support must be written for applications and devices desiring to share data, or written for stand-alone data format conversion applications.

Standard data interchange formats work by identifying and capturing the common elements of the information contained by the data to be exchanged among applications and devices within specific domains. The same concept may be applied to the application code required to add support for data interchange to applications and devices, allowing common tasks performed by application code supporting data interchange to be identified. These tasks are common to application code for interchange of data of any type, and are not specific to format standards. A reusable framework embodying these

common tasks can reduce the amount of effort, and application code, required to support the interchange of data among applications and devices.

This thesis presents an object-oriented framework for classifying and transforming the formatting of data exchanged among applications and devices. The framework relies on the object-oriented principles of encapsulation and dynamic polymorphism, object-oriented design patterns, and generic cooperative multitasking algorithms to provide a scalable and flexible collection of software components for receiving, transmitting, and transforming data in a manner which abstracts the underlying platform and processing details from the application developer.

1.1 Problem statement

A general purpose object-oriented framework for classifying and transforming data formats for effective application-to-application data exchange must address some very specific problems related to data format classification, transformation, and communication while maintaining the flexibility to adapt to variations in the application domain. The problems facing the framework can be definitively expressed through a sequence of independent problem statements. Each statement identifies a specific problem to be systematically addressed by elements of the framework, from which an associated requirement statement may be generated. The following problem statements are organized relative to the order of information flow through an application receiving data as part of the interchange process.

Before data may be processed by an application, it must be introduced to the application's process address space. Data is commonly communicated through basic system I/O resources such as files, sockets, and serial ports. Data may also be communicated through high-level communication resources, or middleware, such as CORBA (OMG, 2008), entered by a user through a graphical user interface (GUI), or generated algorithmically from within the application. Because data received through a GUI or generated algorithmically will typically be represented by a supported application format, the framework

focuses on low-level and high-level communication resources. Types of communication resources can be both numerous and platform specific, requiring support for multiple diverse resource types.

- **Requirement 1:** The framework must support multiple communication resource types for data exchange.

Data to be processed by an application must be physically retrieved from a communication resource. Attempting to receive data from a communication resource that has no data available for receipt may cause the application to suspend execution, or block, while it waits for the communication operation to complete. Such an action may create a perceptible disruption of application execution. Communication resources traditionally provide a method for querying resource state, to determine when communication operations may be safely performed without blocking. Communication operations must be performed only when pending I/O requests are available.

- **Requirement 2**: The framework must observe communication resource state, performing I/O operations such that normal application operation is not disrupted.

To be processed by an application, data must be arranged in a formal manner that is recognized by the application. The format of the data represents the information that it contains. Data received from different resources may contain different types of information that is organized with different formats. These formats may be specific to the application or the application domain, requiring support for the definition of new data format types by the application.

- **Requirement 3:** The framework must support the definition of new data format types at the application level.

Basic system I/O resources operate on unstructured arrays of bytes. The type of information contained by an array of bytes received from a resource must be identified and mapped to a structured object for processing by an application. Identification can be performed by checking for specific byte

patterns within the byte array. Binary data received from a resource may be represented with a different byte ordering than that employed by the host system, or may be otherwise encoded, requiring de-serialization of the data when mapped from an unstructured byte array to a structured format. Continuous streams of binary data may also require message framing to separate the data stream into distinct objects. High-level communication mechanisms typically operate on structured objects, performing any necessary serialization and message framing, and are not subject to this problem.

- **Requirement 4:** The framework must be able to identify and map unstructured data received from system I/O resources to a structured format, performing serialization and message framing as necessary.

After the data has been received and classified, format transformation may be required if the format of the data received is not the format expected by the application. Transformations may be applied to the data at two different structural levels. The organization of the fields of the data format may need to be reorganized relative to each other and the individual fields of the data may need to be modified. Filtering may also be required to prevent invalid data from being consumed by the application.

- **Requirement 5:** The framework must perform any transformations necessary to prepare data for consumption by an application.

Once the data is ready for consumption it must be delivered to the consumer. Methods of delivery and consumption will vary from application to application. Generally, the data will be submitted to an outgoing communication resource for transmission, re-submitted to the framework for further transformation, or submitted to the application for processing. It is necessary that the data delivery process be generic enough that each general submission category be addressed through a single interface for use with application-specific code.

- **Requirement 6:** The framework must provide generic support for application-specific methods of data consumption.

Writing application code to interact with the framework assumes the availability of a software engineer. However, this assertion may not always be true. Consider the case of an analyst who has received a data set as a file containing ASCII numbers, the most universal format used by many applications (Hall, 1993). The ASCII numbers contained by the file are organized into rows of columns, with one record per row and one record field per column. The metric units used to represent the ASCII numbers, and the ordering of the columns containing them, are different than the metric units and ordering expected by the software used by the Analyst for data analysis.

Before processing the data contained by the file with the analysis software, the analyst must pre-process it for transformation to the expected format. Pre-processing requires tools such as sed and awk (Dougherty & Robbins, 1997) to restructure the file, assuming the analyst has the appropriate programming skills, or a spreadsheet application capable of reordering the columns and applying a unit conversion scalar to each number. An application allowing the specification of data formats and transformations, with no requirement for the creation of application code, would greatly reduce the amount of pre-processing and programming skill required of the analyst.

- **Requirement 7**: The end-user, with limited knowledge of software engineering, should be able to define new data formats and transformations.

Each problem identified corresponds to a common functional characteristic shared by applications which must perform data interchange. Concrete requirements have been derived from each individual problem statement, as part of a requirements specification for design and development of a general purpose data interchange framework. A framework addressing each problem will provide a strong foundation for the rapid development of reusable, modular application components for data interchange.

1.2 Motivation

The initial motivation for the presented framework was derived from the need to integrate the SIMDIS 3D Display and Analysis Toolset (U.S. Naval Research Laboratory, 2008) with existing applications at various research facilities and military test and training ranges. Diversity among the applications and devices located at military test and training ranges around the world presents many interoperability challenges for the integration of third-party applications. The most notable challenge involves the sensor networks employed to track range activities.

Ranges contain networks of sensors reporting information describing range activities. There are multiple sensor categories, including radar for tracking objects above ground, hydrophones for tracking objects underwater, and internal measurement units (IMU) reporting exact object positions. Each category of sensor may use a different format when reporting data. Radar may report the range, azimuth, and elevation of an object relative to the radar position while an IMU may report an object position as Cartesian coordinates relative to the center of the Earth. Even sensors within the same category may produce data with different formats. Sensors from one vendor may produce data using different units and a different structural organization than sensors from another vendor. While one vendor's sensor reports Earth Centered, Earth Fixed (ECEF) Cartesian coordinates (Dana, 1999) with distance measured as meters, another vendor's sensor may report geodetic latitude, longitude, and altitude (Dana, 1999) measured with degrees and feet.

Differences among the data formats produced by the sensors of a sensor network must be considered when developing software to process data originating from the sensor network. The basic approach to dealing with multiple sensor formats within a sensor network is to place the responsibility of format conversion on the data consumer, as depicted by Figure 1. This requires that any software which processes data received from the sensor network be aware of each data type produced by the sensor network. Such an approach is not very efficient, requiring all data processing software to be updated whenever a sensor with a new data format is introduced to the network. Not only does this create a

duplication of effort at both the time of implementation and the time of execution of each data consumer, it has the potential to introduce error where one consumer performs a transformation using a different algorithm than that used by another consumer, which may provide different results.

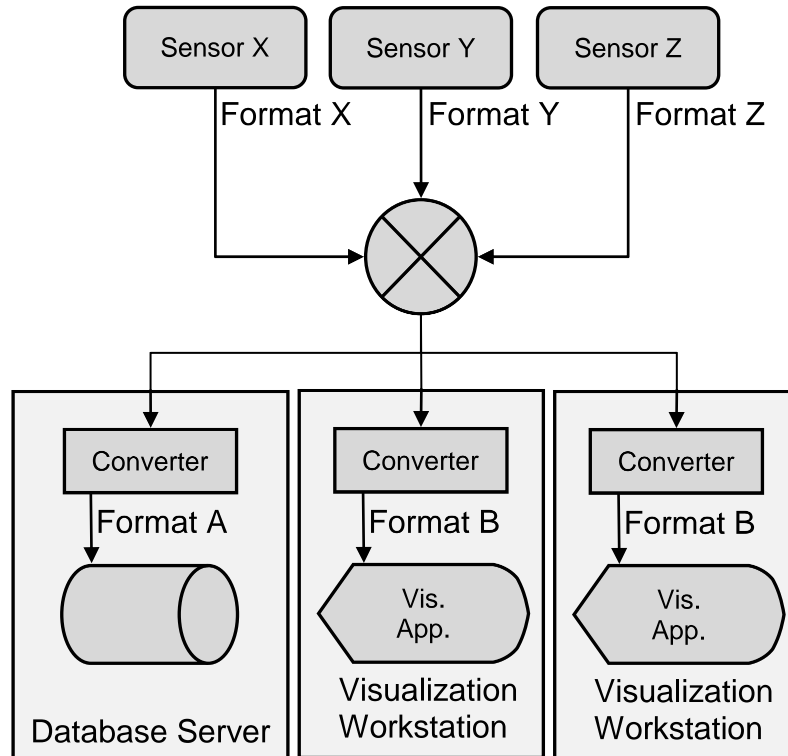


Figure 1 - Sensor Network with Multiple Data Formats

A more sophisticated approach involves the definition of a common format to represent all data produced by the sensor network. This approach, depicted by Figure 2, requires that the native data format of each sensor be transformed to the common format before transmission to the consumer. Data transformation responsibilities are now distributed such that there is a single software component for each sensor type responsible for converting the native sensor format to the common format, and a single software component responsible for converting the common format to the native format of each consumer. This reduces the amount of effort required for consumer implementation. If the consumer is specific to the sensor network, its native format may be the same as the common format, reducing the amount of processing to be performed at run time. This approach works well within the confines of the

sensor network, only requiring the creation of new data format transformation components for new sensors with new data formats.

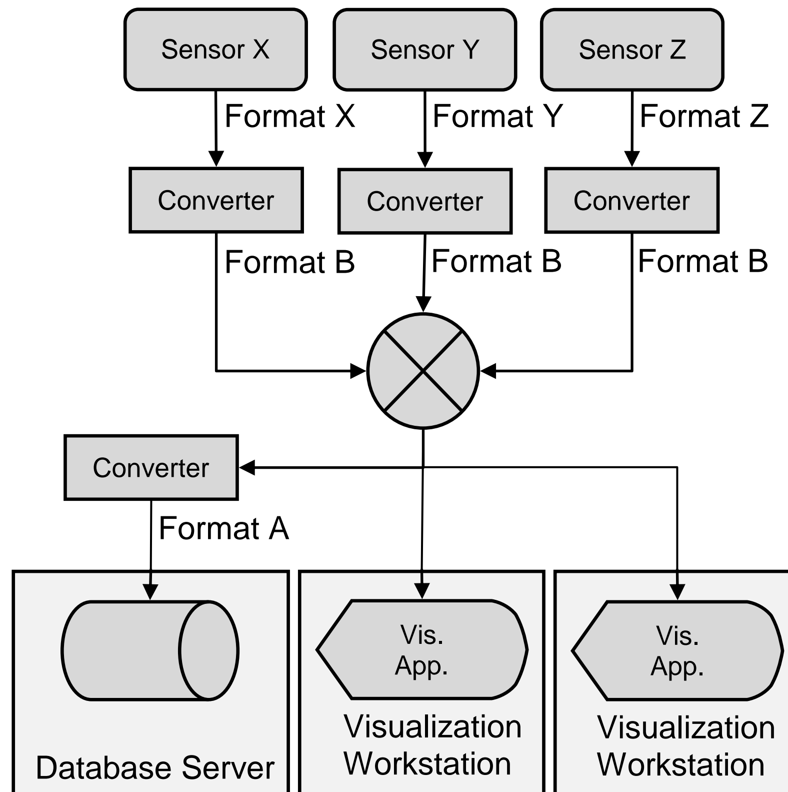


Figure 2 - Sensor Network with a Common Data Format

Although the common sensor network data format approach works well within the confines of the sensor network, problems may be encountered when attempting to share data between two different sensor networks or maintain a third party application which must integrate with multiple different sensor networks. When two independent ranges that manage sensor networks must share data as part of a joint exercise, any issues resulting from differing data formats must be addressed. Each range may have developed a format for its sensor network to meet the needs of the range mission statement and goals. Lack of alignment of range goals, and the effect of independent format development, may have led to development of formats that do not match. Each range must now either modify its data consumers to support the other range's formats, or must construct a gateway to transform the data as it moves between sensor networks.

A more difficult interoperability problem is faced by non-range centric data consumers which must interact with multiple sensor networks. Because of the cost and time required to develop full-featured data visualization and analysis applications, many ranges are adopting existing full-featured tools such as SIMDIS (Binford & Doughty, 2002). These applications are typically expected to do more than integrate with the sensor network. They must also be able to import data from files containing information such as nominal data describing an exercise to be conducted, and export data to files containing information for post-exercise analysis. Such files may be range specific, or may be specific to a range user.

Test and training ranges provide equipment, resources, and support personnel to other organizations which would like to make use of range facilities to conduct an exercise. These organizations are range users which typically perform extensive amounts of independent pre-exercise planning and post-exercise analysis, producing and consuming files containing data which may be represented with a format specific to the tools employed by the user. Data generated during the pre-exercise planning phase may be required for display during the exercise for comparison of nominal data with actual data, as depicted by the example in Figure 3. Range user specific data formats extend the interoperability problem beyond the confines of the range, generating additional data format support requirements for data processing applications.

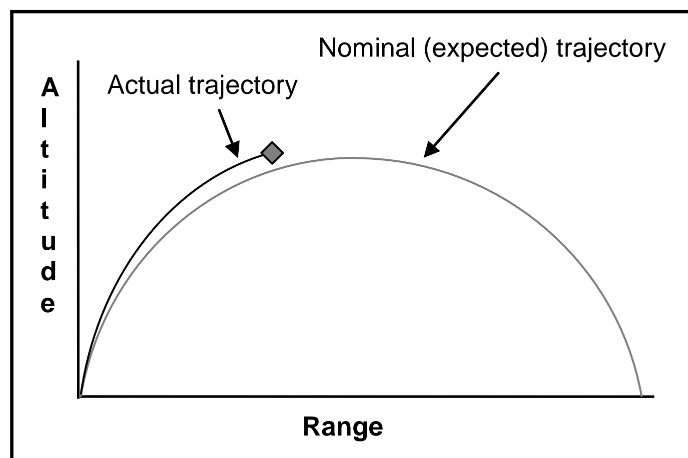


Figure 3 - Example Situational Awareness Display with Nominal and Actual Data

The Department of Defense has initiated the development of the Test and Training Enabling Architecture, or TENA, (TENA Software Development Activity, 2008) to address the issue of military facility and range interoperability. TENA provides a middleware architecture and requisite software responsible for connecting test and training range systems and applications through a common architecture promoting interoperability, reuse, and composition of range resources. While TENA provides a standard to address the application interoperability problem at the network level, the file interoperability issue remains to be addressed.

Whether it is assisting to build gateways for or integrate existing range applications with TENA, integrate applications with organizations or application domains that are not associated with TENA, or provide data file conversion support, a framework providing reusable application code and design principles common to each activity would greatly reduce the cost and effort required to add data interchange support to data processing applications.

1.3 Related Work

The framework presented by this thesis employs well-defined software engineering techniques of object-oriented framework design and reusable design patterns, and borrows linear typing concepts for packet processing from the PACLANG programming language (Ennals, 2004). Object-oriented frameworks and design patterns provide effective techniques for modular software development, composition, and reuse, while linear typing assists with the design of efficient scalable concurrent systems. The fundamental object-oriented principles of encapsulation, implementation hiding, object identity and polymorphism (Page-Jones, 1995) are key elements of the successful operation of the framework. These properties make it possible for the framework to operate on data formats and communication resources within a generic system for data format classification and transformation.

Object-oriented design promotes reuse, reliability, robustness, and extensibility (Page-Jones, 1995). Well-defined, cohesive object classes may be reused with multiple applications. Their self-contained nature makes it easy to test for correctness and handle exceptions at the object level, for increased reliability and robustness. Extensibility allows new state attributes and functionality to be added to an existing object class through class inheritance by new object classes that take on and add to the properties of the existing class. Frameworks and design patterns are natural extensions of object-oriented design, expanding beyond the object level to promote well-structured modular design at the system level.

1.3.1 Object Domains

Four major domains have been defined for organization of classes within object-oriented environments (Page-Jones, 1995). These domains are intended for classification and categorization of the generality of object classes. Each domain is less general than the next. The degree of reusability of a class may be determined by its associated domain, where generality is directly proportional to reusability. The four domains, ordered from general to specific, are:

- ***Foundation-domain:*** General purpose classes not specific to any environment.
- ***Architectural-domain:*** Classes particular to a specific architectural environment.
- ***Business-domain:*** Classes particular to a specific industry.
- ***Application-domain:*** Classes particular to a specific application.

Each domain has a set of well-defined sub-domains which help to further refine the classification process. The *foundation-domain* includes the fundamental, structural, and semantic sub-domains. Each sub-domain includes specific class types. Classes from the fundamental sub-domain include basic types such as `int` and `float`. The structural sub-domain includes classes such as data structures, or container classes. The semantic sub-domain includes general purpose types such as `time` and `date` which have

greater semantic meaning than basic types. *Foundation-classes* offer nearly universal properties which may be used within most architectural, business, and application domains.

The *architectural-domain* includes the machine-communication, database-manipulation, and human-interface sub-domains. Machine-communication classes are those which provide resources for remote communication. Database-manipulation classes are those that perform database-related transactions. Human-interface classes are those that facilitate human-computer interaction. Although classes from the *architectural-domain* are not as universally usable as those from the *foundation-domain*, they are part of a category which has potential for widespread reuse within most business and application domains.

The *business-domain* includes the attribute, relationship, and role domains. Attribute classes capture the specific properties of objects within a field or industry. Relationship classes define object interactions. Role classes define the types of responsibilities or activities to be fulfilled by an object. Classes from the *business-domain* are specific to an industry or organization, having potential for reuse only with applications specific to that industry or organization.

The *application-domain* includes the event-stimulus recognition and event-activity management sub-domains. Event-stimulus recognition classes monitor application state, waiting for changes to specific state attributes. Event-activity management classes respond to events by performing an application-specific action. Classes from the *application-domain* are typically too application-specific for use with other applications.

1.3.2 Frameworks

Frameworks are reusable abstract object-oriented designs (Johnson & Foote, 1988) for software components. They define collections of extensible abstract classes, and the methods through which instances of these classes collaborate (Roberts & Johnson, 1997). The intended purpose for frameworks

is the reduction of the cost and time associated with software development by promoting reuse of application code and design. A formal definition of a framework is:

> "A set of collaborating classes, arranged to be able to carry out some meaningful portion of an application. (A framework is more sophisticated than a single class, but normally less sophisticated than a subsystem.) Typically, a single class in the framework has limited reusability on its own." (Page-Jones, 1995)

Frameworks capture the general design of components common to particular types of applications. An abstract class is defined for each component, providing common component characteristics without application-specific details. Application-specific state information and functionality is added by extending abstract classes. Libraries of predefined concrete subclasses and support classes are usually included with frameworks. These classes collaborate with other framework components to achieve the goal defined for the framework. Classes may be associated with any of the four object domains, although the degree of reusability of a class is governed by its domain and may affect the overall reusability of the framework.

Two approaches to framework development have been previously identified, creating two distinct framework categories. White-box frameworks (Johnson & Foote, 1988) require the definition of new subclasses to add application-specific code. A good understanding of the design of a white-box framework is required when using the framework because the functionality added by subclasses must conform to an interface and design specified by the base classes. Black-box frameworks (Johnson & Foote, 1988) provide libraries of predefined subclasses containing application-specific code. Only the external interface to framework components must be understood when using a black-box framework, as the internal design is hidden from the framework user who may simply choose the appropriate subclasses to use with an application. Black-box frameworks are easier to understand and use but lack some of the flexibility provided by white-box frameworks.

Framework development has been shown to follow an evolutionary process (Roberts & Johnson, 1997). Frameworks typically start as white-box frameworks with small libraries of components which grow as new components are introduced through use of the framework with the design and implementation of multiple applications. As hot-spots of application-specific code are discovered, they may be localized to specific component locations and encapsulated by fine-grained concrete objects. Eventually, the library of concrete objects contained by the framework will grow large enough to form a black-box framework. The framework continues to evolve with the creation of tools for building applications through automatic composition of framework components, and reaches the final stage of evolution with the introduction of language tools for inspecting and debugging applications incorporating the framework.

1.3.3 Design Patterns

Design patterns (E. Gamma, 1995) capture common elements of object-oriented software design. They describe reusable general purpose designs for software components that may be customized to provide solutions to recurring software design problems. Object-oriented frameworks may employ design patterns to assist with the separation of application-specific code from common code that may be shared among multiple applications. Three separate categories have been defined for the classification of design patterns as creational, structural, and behavioral patterns.

Design patterns frequently used with the design of the object-oriented framework presented by this thesis include:

- **Builder:** A creational pattern to separate the construction of an object from its class, allowing multiple objects to be created through the same construction process.
- **Prototype:** A creational pattern specifying a prototypical instance of an object to be cloned to create new objects matching the prototype.

- **Adapter:** A structural pattern encapsulating an existing object to provide an interface compatible with that expected by other existing objects.
- **Decorator:** A structural pattern encapsulating an existing object to extend its capability while maintaining the same interface.
- **Façade:** A structural pattern to wrap components of a subsystem within a unified, high-level interface which simplifies subsystem interaction.
- **Command:** A behavioral pattern encapsulating an action within an object.
- **Observer:** A behavioral pattern to monitor the state of an object and notify other objects of any state changes.
- **Visitor:** A behavioral pattern to separate algorithms from class implementations through use of an external operation that is applied to elements of an object structure.

1.3.3.1 Reactor and Proactor Patterns

The Reactor (Schmidt, 1995) and Proactor (I. Pyarali, 1997) patterns are behavioral patterns responsible for demultiplexing and dispatching communication requests. Each describes a method for monitoring communication resources and dispatching communication events to designated event handlers. Event handlers are decoupled from event generators, allowing different event handlers to be dynamically associated with event generators at runtime. Both patterns provide similar functionality but have different implementations with different operational and performance characteristics.

The Reactor pattern is a behavioral pattern that can be used to synchronously monitor communication resources and dispatch communication events as they occur. Reactor includes an Initiation Dispatcher component responsible for managing Event Handlers and dispatching events. A Synchronous Event Demultiplexer, such as the UNIX `select` (Stevens, 1998) system call, is employed by the Initiation Dispatcher to monitor communication resources and detect which resources are ready to perform non-blocking operations. When the Synchronous Event Demultiplexer indicates that a resource may be safely processed without entering a blocking state, the Initiation Dispatcher invokes an

application-specific Event Handler to process the operation. The Reactor pattern offers an efficient method for demultiplexing and dispatching events from multiple resources without the need for complex synchronization techniques such as threading.

The Proactor pattern is a behavioral pattern that can be used to asynchronously monitor communication resources and dispatch communication events as they occur. A Proactive Initiator registers event Completion Handlers and Completion Dispatchers with an Asynchronous Operation Processor. Asynchronous Operations to perform operations on communication resources are invoked by the Asynchronous Operation Processor. When an Asynchronous Operation completes, the Completion Dispatcher is provided with the result of the operation which is provided as an argument to the associated Completion Handler. The Proactor pattern allows multiple operations to be processed in parallel without the need for application level threading, and operates more efficiently on high-performance systems than the Reactor pattern, which does not provide the same level of support for hardware parallelism.

1.3.4 Linear Types for Packet Processing

Linear type systems define types whose values may only have a single reference (Wadler, 1990). Values of a linear type may be neither duplicated nor discarded, allowing safe modification within concurrent environments and explicit specification of type allocation and destruction. Because pure linear type systems can be more restrictive than necessary, linear type system requirements may be relaxed such that a value may have multiple read-only references. A single reference is only required when writing. Multiple references may also be allowed if a complete copy of the value is made for each reference.

A relaxed linear typing system may help to reduce the complexity of concurrent systems by eliminating the need for explicit synchronization, fostering an environment for implicit synchronization. Implicit synchronization is an important aspect of the design of efficient scalable concurrent systems (Reinders, 2007). Eliminating the need for complex explicit synchronization and locking of concurrently accessed resources removes performance degrading delays. Concurrent systems containing tasks which

execute independently and are not required to wait for availability of shared resources are able to scale effectively within high-performance environments providing hardware parallelism.

PACLANG (Ennals, 2004) is a concurrent, linear-typed language for packet processing. References to a packet are restricted to a single thread, with multiple references to the packet allowed within the same thread. This simplifies the development of high-level packet processing applications for modern network processing systems with distributed memory architectures. PACLANG's linear type system provides a unique ownership property which eliminates the need for locking and allows a thread to safely move an object it owns to a new location. A packet value may only have a single reference wherever it is in scope. References to a packet value may be safely transferred from one thread to another without the risk of creating dangling references. These properties provide an efficient, implicitly synchronized environment for concurrent processing of packets.

1.4 Approach

To address the comprehensive data exchange problem posed for the design and development of a framework for data format classification and transformation, the presented framework defines a set of fine-grained objects that implement functionality to address each of the previously defined problem statements (Section 1.1). The framework focuses on the classification and transformation of the formatting of data exchanged through existing communication protocols and resources. It does not define new methods for communication among applications. Instead, it adapts to existing methods of communication, placing its primary focus on the processing of the data exchanged through the communication process. This allows applications to adapt to existing data formats, communication protocols, and communication resources, without requiring support for newly defined interchange format, protocol, and resource types.

An object-oriented framework design provides a foundation for constructing interoperable system components. The primary objects of the framework core are organized into three categories. Creational

objects abstract the process of application-specific format definition and creation. Structural objects encapsulate and manage the states of basic data formats and communication resources. Behavioral objects monitor and manipulate data format and communication resource states for the purpose of data communication, classification, and transformation. The framework core contains a set of abstract base classes that specify formal interfaces for structural objects, and a mix of abstract and concrete classes that specify creational and behavioral objects to create and manipulate the structural objects. Concrete object implementations providing generic data formats and common communication resource types are included with the framework. Additional objects may be defined at the application level, through extension of the core base classes, to provide support for functionality not directly incorporated by the framework.

Heavy reliance on dynamic polymorphism and design patterns allows creational and behavioral objects to process structural objects without knowing the exact details of the object implementation, making it possible for new structural objects to be added to an application at any time. Dynamic libraries may be used to load additional structural objects at run-time, incorporating new capability with an application without requiring modification to the application. The use of design patterns and scalable computational algorithms provides a flexible and extensible framework that is capable of run-time modification and able to take advantage of multi-core processor performance while hiding the details from the application developer.

1.4.1 Addressing the Problem with the Framework

The presented framework fully addresses the first six of the seven previously defined problem statements (Section 1.1). Only a partial solution to the seventh problem can be achieved with the existing framework design. Although addressing the seventh problem is a long-term goal for the project, a full solution to the problem is outside the scope of this thesis.

Problem 1 requires that the framework support multiple communication resource types. This problem is addressed with an abstract class implementing the façade design pattern. The abstract class

acts as a wrapper for existing communication resources, defining an interface that specifies the communication operations required for interaction with the framework. Concrete implementations of the class are made for each communication resource to be used with the framework. A library of concrete communication classes will be included with the framework, accumulating new classes as the need for support of new communication resource types arises and transitioning the framework toward black-box framework status.

Problem 2 requires that the framework manage multiple communication resources without impacting normal application operation. A class with a design based on either the Reactor or Proactor design pattern is employed to deal with this problem. The class is responsible for monitoring communication resources and processing communication events. Proactor may be used when the underlying system provides support for asynchronous I/O operations. Reactor is used when support for asynchronous I/O operations is unavailable. Both patterns ensure that blocking I/O operations which may suspend normal application operation are avoided.

Problem 3 requires that the framework support the definition of new data formats at the application level. An abstract format class is defined to serve as the base for definition of all data format types. Each of the format processing objects contained by the framework is designed to operate on the abstract format type. The abstract format class encapsulates a type identification field which allows other framework objects to distinguish between different format types received from an instance of a communication resource. The type identifier for a data format should be unique to a communication resource object instance, but does not have to be unique to the framework. Concrete format class implementations will be included as part of the framework's class library. A generic data format class implementing the Prototype design pattern, allowing data formats to be defined dynamically at run-time, is also included with the framework.

Problem 4 requires that the framework identify and decode unstructured data received from communication resources. This problem is addressed with a class implementing the Creator design pattern. The class is responsible for converting unstructured data received from a communication resource to a structured format defined for the framework. An abstract class implementing a creator with an interface to encode and decode structured data formats is provided with the framework. The abstract class is used by communication resource objects for the encoding and decoding of data that is received and transmitted. Concrete implementations are defined to process one or more concrete data format types and will be included as part of the framework's class library. A format classification class responsible for processing the Prototype data format, allowing format classifiers to be dynamically created at run-time, is also included with the framework. Note that objects to identify and decode unstructured data are not strictly required by the framework, as the framework could be used to process the unstructured data.

Problem 5 requires that the framework transform data as required for consumption by the application. A multi-stage pipeline is employed to perform any necessary processing of data received from communication resources. Pipeline stages are categorized as filters which remove invalid data from the pipeline, transformers which perform in-place modification of data formats, and translators which convert data formats from one format to another. Independence among data objects received from communication resources is exploited by allowing multiple objects to flow through the pipeline concurrently. A linear type system restricting data format references to a single pipeline is used by the framework. When more than one pipeline is created to process data format objects received from a communication resource, each pipeline must receive its own copy of the object.

Problem 6 requires that the framework provide a generic method for transferring data format objects to the application. An abstract data format handler is defined to serve as the terminal stage of the data format processing pipeline. The handler may either transfer ownership of the data format object to the communication resource monitor for transmission, to another pipeline for further processing, or to an application component. The communication resource monitor allows data format objects to be queued for

transmission when a non-blocking transmission operation may be safely performed. Concrete classes to transfer ownership of data format objects to the resource monitor and to other pipelines are included with the framework's class library.

Problem 7 requires that the framework provide tools to construct applications from existing framework components. This requirement is not fully addressed by the current version of the framework. The framework provides basic support for dynamic object composition to define new data format types at application run-time, but does not provide external tools to perform application construction through object composition. After the framework's class library has expanded to contain a large contingent of well-defined, fine-grained composite and concrete classes, transitioning the framework from white-box to black-box status, tools for application construction will be developed.

1.4.2 Applying the Framework to the Problem

Three approaches to addressing comprehensive data exchange among applications have been identified for use with the presented framework. The first approach adds support for format transformation to the data consumer. With this approach, each consumer must perform the processing required for data interchange. Each consumer must also be modified to support new formats as they are introduced. The second approach adds support for format transformation to the data producer. With this approach, only the producer must perform the processing required for data interchange. The third approach creates a standalone application to act as a file converter or communication gateway.

A gateway, as defined for the communication domain, is a component of a communication network that performs the necessary protocol conversions to interconnect networks that use different network protocol technologies (GSA, 1996). The gateway approach localizes data format transformation to a single connection point within the data communication system. Distribution of data may be filtered at the connection level, allowing certain types of data to be restricted to specific communication subsystems. With communication gateways and file converters, a separate application performs the

processing required for data interchange. This may be a single application that must be modified to support new formats as they are introduced, or a set of applications with one application developed for each individual format. No change is required to the applications which produce and consume data, but an extra data transformation application is required to carry out the data interchange process.

Using the framework to develop gateways for data conversion, or to perform data conversion from within the data producer, simplifies the addition of support for new data formats to the data interchange process. No modification to data consumers is required for data format interchange, and data consumers are not subject to the overhead introduced when required to perform format transformation for data format interchange.

1.5 Thesis Organization

The remainder of this thesis is organized as follows. Section 2 provides information describing the properties of communication resources, and the data that they communicate, which shape the design of the framework. Section 3 defines the type and design of framework components and Section 4 describes Phylum, an implementation of the presented framework developed with the C++ programming language (Stroustrup, 1997) and the Intel Threading Building Blocks C++ template library (Reinders, 2007) for the development of scalable parallel programs. Section 5 presents the implementation of some sample applications using the framework. The thesis concludes with Section 6, which provides a project summary and a description of the future direction for the project.

2. Important Properties of Data Formats and Sources

The purpose of the presented framework is to process data communicated through different communication resources. Communication resources provide capability to receive and transmit data, and are referred to as sources within the context of the framework. The formal definition of a source, within

the communication domain, is as the origin of a message (GSA, 1996). Sources are the point of origin to the framework for data messages, serving as access points to streams of data.

The formal definition for data is a representation of facts, concepts, and instructions suitable for communication, interpretation, or processing (GSA, 1996). Data and the data streams through which it is communicated have properties which vary by application. Variations of data format and stream properties must be carefully considered when developing a framework for data format classification and transformation. Designing the framework to support an extensive assortment of data format and stream properties ensures that the framework is as flexible and reusable as possible. A full understanding of the properties of data formats and streams also allows the data parallelism inherent to these formats and streams to be properly identified and exploited for implementation of an efficient and scalable concurrent framework.

2.1 Properties of Data Formats

Formats may be formally defined as arrangements of bits or characters within a group (GSA, 1996). The format of a data message describes the information contained by that message, assigning a structure to the data such that the message may be processed by an application or understood by a human. The design of the framework presented by this thesis depends on some important observations regarding data and its structure. The important observations regarding data and its structure are:

- **All data that may be safely processed by an application has structure.** For an application to safely process data, that data must have some structure that is recognized by the application. The generality of the structure is application dependent. The structure must only be as specific as necessary to meet the level of understanding required by an application for completion of its assigned task. Data records for an employee database application may require a very specific structure defining fields such as employee name and salary; while data processed by an

application designed to copy blocks of data from one file to another may have a vague structure where each block is treated as a simple array of bytes.

- **There exists a loose coupling between data and the specification of its structure.** An application does not need to be aware of the structure of a data segment until it must access the contents of that segment for processing. Therefore, the structure of a data segment is independent of the actual data and may be defined any-time prior to processing.
- **The structure of data may be specified at run-time.** Because the structure of a data segment does not need to be known prior to use, it does not need to be defined before the time of application execution. The structure may be statically defined in the application code at time of compilation, or may be dynamically defined at application run-time.
- **Syntax for data structure specification may be defined.** Special purpose syntax may be defined for dynamic data format specification. The syntax may be parsed by an interpreter embedded within an application for defining and registering new format types with the application.

Knowledge of these properties is important to understanding how general purpose data format classification and transformation may be achieved at application run-time. The demonstration of the separation of data from its format shows that it is possible to dynamically define and transform data formats, making it possible to specify new data format types and transformations from within an executing application.

2.2 Properties of Data Streams

Data streams may be formally defined as sequences of digitally encoded signals representing information in transmission (GSA, 1996). Some data streams may consist of aggregations of data describing multiple objects. Each object may be described by multiple messages represented with different data format types. Aggregation of data propagated through data streams may be classified as one of four distinct aggregation categories. The categories, with supporting examples, are:

- **Single object, single format.** The stream contains messages of a single data format type describing a single object. An example from this aggregation category is a single temperature sensor periodically transmitting messages of a single type, with each message containing a temperature update.
- **Single object, multiple formats.** The stream contains messages of multiple data format types describing different properties of a single object. An example from this aggregation category is a single weather station periodically transmitting multiple messages of multiple types, with each message type containing a different type of weather-related information.
- **Multiple objects, single format.** The stream contains messages of a single data format type describing multiple objects. Each object within the stream is associated with messages of a single format type describing its state. An example from this aggregation category is a network of multiple temperature sensors periodically transmitting messages of a single type, with each message containing a temperature update.
- **Multiple objects, multiple formats.** The stream contains messages of multiple data format types describing different properties of multiple objects. Each object within the stream is associated with messages of multiple format types describing different aspects of its state. An example from this aggregation category is a network of multiple weather stations periodically transmitting multiple messages of multiple types, with each message type containing a different type of weather-related information.

Different techniques for decoding and processing stream data are required for different data aggregation categories. To ensure that the data format classification and transformation framework is applicable to as many domains as possible, the framework design must carefully consider and provide support for each category. Messages within streams of multiple formats and objects will contain format type and object identifiers. Support for identifier recognition is required for design of a framework

capable of differentiating between formats and objects. The framework must also decompose aggregate data prior to processing if such an action is required by the design of the application using it.

2.3 Exploiting Data Parallelism

Decomposition of a problem into tasks capable of parallel execution may be achieved by identifying and exploiting task parallelism, data parallelism, or a combination of both (Reinders, 2007). Task parallelism occurs when a task may be decomposed into independent subtasks able to execute concurrently, and data parallelism occurs when a transformation may be simultaneously applied to each item from a data set. Large amounts of data parallelism lead to parallel applications which scale well as additional processing units become available. An abundance of potential data parallelism exists within certain types of data streams.

A great deal of data independence exists among items within streams composed of multiple objects and formats. Data describing multiple objects within a stream may be independent of other stream objects, allowing data describing each object to be processed concurrently. Formats describing different properties of an object may also be independent of each other, allowing individual elements of an object to be processed concurrently. Partial data independence may impose some restrictions requiring that some stages of the pipeline be serialized. Complete data independence allows items to flow through the pipeline with no restrictions. More independence existing among items within a data stream leads to more efficient and scalable systems for processing those items.

Data independence is not limited to streams containing multiple objects or formats. Independence may exist among items within a data stream containing data for a single object described by a single format. For example, a block compression algorithm may be concurrently applied to fixed sized blocks from a stream of audio data. Although the audio stream qualifies as a single object represented by a single block format, there is no dependence between blocks. The framework need only ensure that the original ordering of the blocks be preserved.

3. Framework Design

The framework presented by this thesis receives data from and transmits data through an abstraction of a communication resource such as a file, socket, or serial port. Data received from a resource is submitted to a data processing pipeline from which it may be retransmitted through another resource, submitted to another pipeline, or injected directly to an application. When data communication and processing is managed by the framework, the software developer is required only to create filters, transforms, or translators to be part of the processing pipeline.

The framework consists of two main modules for monitoring data sources and transforming data received and transmitted by data sources (Figure 4). The Source Monitor manages communication resources and the Pipeline Executor manages processing pipelines. Each module contains a set of components which operate on a common abstract data format type. Some components are shared between both modules and others are exclusive to a single module.

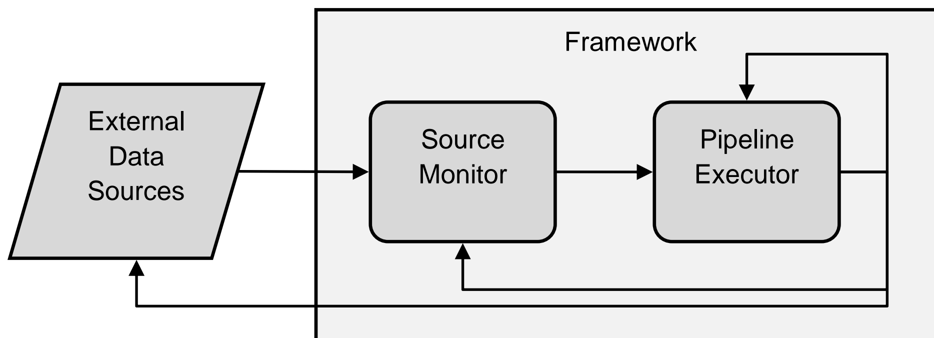


Figure 4 - Interaction of Framework Modules

A linear type system, similar to the system defined for PACLANG (Ennals, 2004), has been defined for the framework. The framework's linear type system requires that each processing pipeline have unique ownership of data items. Pipeline stages which perform write operations on data items must contain exactly one reference to a single data item. Stages which perform read-only operations on data items may contain multiple references to a single data item. Multiple operations may be performed

concurrently by read-only stages, allowing a single data item to be referenced by multiple threads. This differs from the PACLANG system which restricts references to a single thread.

3.1 Source Monitor

The Source Monitor module is responsible for detecting and executing pending communication events from communication resources. Source objects encapsulating specific communication resource types may be registered and deregistered with a Monitor object. Optional data Classifier and Cache objects may be provided to each source. Source objects are associated with Delegates responsible for dispatching data Format objects received from a Source to an associated Pipeline object or group of Pipeline objects. Figure 5 depicts the flow of data through the Source Monitor module.

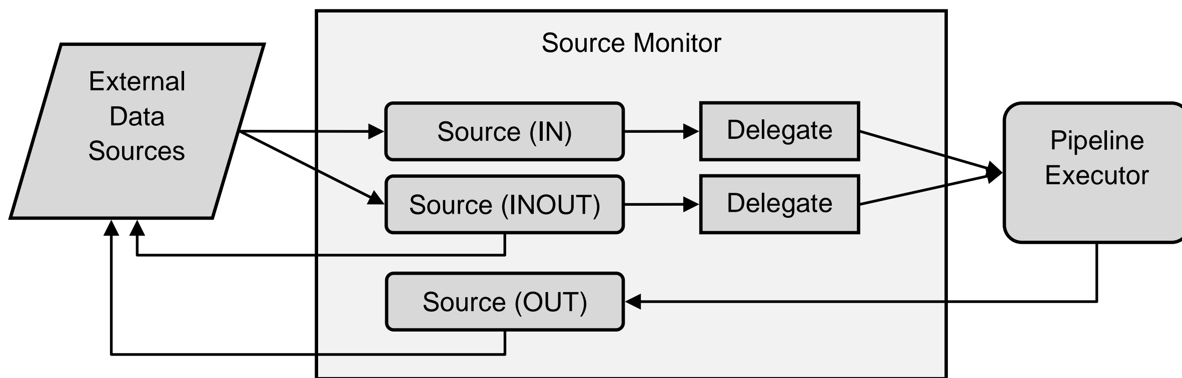


Figure 5 - Data Flow through the Source Monitor Module

The Monitor object is designed to execute within its own thread. This is done to minimize the impact of the Source Monitor on the execution of the main application thread. Within a Monitor object's thread of execution, each Source object is processed concurrently. Format objects are received from a Source, possibly transformed by a Classifier, and dispatched to a Pipeline by a Delegate from within a processing unit dedicated to that Source. Methods for registering, querying, and deregistering Source objects with a Monitor are required to be thread safe.

3.2 Pipeline Executor

The Pipeline Executor module is responsible for managing the execution of one or more data processing pipelines. Pipeline objects are added to and removed from an Executor object. Format objects are submitted by Delegate objects to Pipeline objects for processing. Pipeline objects are composed of Pipeline Function objects responsible for processing Format objects. Pipeline Functions represent stages of a Pipeline which are processed sequentially. Multiple Format objects may flow through a Pipeline concurrently. A special Terminator Function located at the end of a Pipeline object is responsible for transferring ownership of a Format object to other modules. Figure 6 depicts the flow of data through the Pipeline Executor module.

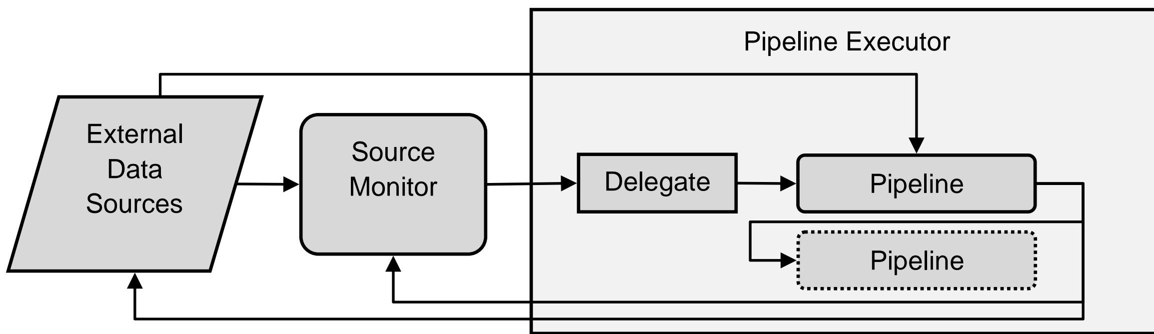


Figure 6 - Data Flow through the Pipeline Executor Module

The Executor object is designed to execute within its own thread. This is done to minimize the impact of the Pipeline Executor on the execution of the main application thread. Within an Executor object's thread of execution, each Pipeline object is processed concurrently. Methods for adding and removing Pipeline objects to and from an Executor object are required to be thread-safe.

Format objects are received from a Delegate and shifted sequentially through the stages of a Pipeline object from within a processing unit dedicated to that Pipeline. Format objects may not be shared by multiple Pipeline objects. To avoid unsafe processing conditions, each Pipeline is provided with its own copy of a Format object. Methods for submitting Format objects and adding and removing Pipeline Functions to and from a Pipeline object are required to be thread-safe.

3.3 Framework Component Specifications

Each framework component is designed to perform a specific task. The design of each component governs its behavior and interaction relative to other framework components. Details of each component design are described below.

3.3.1 Format

Format is an abstract class providing a base from which concrete data format types must be derived. Format subclasses define the structure of data received from a Source. Format objects are designed to be processed with the Visitor design pattern. Each of the framework components responsible for classifying and transforming data formats operates on objects derived from Format.

Table 1 - Format Interface

Method	Description
getType	Method providing a type identifier to uniquely identify a concrete Format object within a group of Format objects.
accept	Method to accept a visitor as part of the Visitor design pattern. The default implementation accepts all visitors. Subclasses may override the method to limit acceptance to specific visitor classes.
clone	Pure virtual method to create a complete copy of the Format object. All subclasses must implement this function.

3.3.2 Format Group

Format Group is a concrete subclass of Format. It does not define an actual data format, but defines a collection of Format objects. Format Group objects allow a group of Format objects to masquerade as a single Format object. The implementation of the Format `clone` method creates a full copy of the Format Group object and each of the Format objects which it contains.

3.3.3 Source

Source is an abstract class implementing the Façade design pattern. It acts as a wrapper for communication resources, defining a common interface for communication operations required by the framework. Source objects may contain optional Classifier and Cache objects.

Table 2 - Source Interface

Method	Description
getMode	Method reporting the I/O mode for the Source. I/O mode may be IN, OUT, or INOUT.
getStatus	Method reporting the current state of the Source. Valid states are INACTIVE, ACTIVE, DONE, and ERROR.
setClassifier	Method associating an optional Classifier object, responsible for converting between "raw" data and a structured Format class, with the Source object.
setCache	Method associating an optional Cache object, responsible for storing received Format objects, with the Source object.
getReadHandle	Method producing a system handle that may be used to determine when the Source has data available for reading.
getWriteHandle	Method producing a system handle that may be used to determine when the Source is available to receive data for writing.
read	Method receiving a Format object from a communication resource. The "raw" data received from the source will be encapsulated in a Format object containing a simple byte buffer. If a classifier has been specified, it will be used to convert the "raw" data received from the resource to a concrete Format object. A copy of the Format object will be stored in the cache, if one has been specified.
write	Method for transmitting a Format object to a communication resource. If a classifier has been specified, it will be used to convert the Format object to the "raw" Format object expected by the communication resource. Otherwise, the Format object provided to the function must be of the "raw" Format type expected by the communication resource.
enqueuForWrite	Method to push a Format object into a queue of objects to be transmitted.
writeQueued	Method to pop a single Format object from the write queue and transmit it with the `write` method. Allows multiple tasks to use a source concurrently.
readRaw	Pure virtual method to read a "raw" Format object from a communication resource.

	Used by the `read` method.
writeRaw	Pure virtual method to write a "raw" Format object to a communication resource. Used by the `write` method.
close	Pure virtual method to close a source and release its resources. Invoked at object destruction.

Subclasses of the Source class are only required to implement the `readRaw`, `writeRaw`, and `close` functions.

3.3.4 Pipeline

Pipeline is a class composed of a sequence of processing stages derived from an abstract Pipeline Function class. Format objects are submitted to a Pipeline object for processing. Stages are executed sequentially, and multiple items may be processed concurrently within a stage. Stages which should not process items concurrently may be marked as serial. Items are guaranteed to enter a serial stage with the same order that they entered the pipeline.

Methods for submitting Format objects and adding and removing Pipeline Functions must be thread-safe so that the application may manipulate the Pipeline structure from a thread different than the Pipeline object's thread of execution. Pipeline Function objects are designed to be processed with the Visitor design pattern, providing a simple and thread-safe method for sequentially accessing each Pipeline Function within a Pipeline.

Table 3 - Pipeline Interface

Method	Description
addData	Method to queue Format objects for processing.
getDataSize	Method to query the size of the Format object queue.
appendFunction	Method to append a Pipeline Function to the end of the Pipeline.
insertFunction	Method to insert a Pipeline Function at a specific position within the Pipeline.
removeFunction	Method to remove a Pipeline Function from the Pipeline.
clearFunctions	Method to remove all Pipeline Functions from the Pipeline.

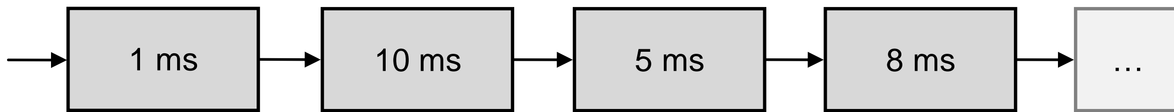
accept	Method to accept a visitor, for application to Pipeline Functions in the Pipeline, as part of the Visitor design pattern.
setTerminator	Method to append a special Pipeline termination stage to the Pipeline.
run	Method to execute the Pipeline. A processing limit may be specified to restrict the total number of items that may be processed within a single run.
suspend	Method to temporarily suspend Pipeline execution. Does not cause the `run` method to exit. No items will be submitted to the Pipeline for processing until execution is resumed. Items already "in flight" will continue to flow through the pipeline to completion.
resume	Method to resume execution of a suspended Pipeline.

A Pipeline may be composed of three distinct Pipeline Function subclasses: Filters, Transforms, and Translators. Filters remove invalid Format objects from the Pipeline, Transforms modify the contents of Format objects, and Translators convert Format objects from one type to another. Pipelines are terminated with a special purpose Pipeline Function subclass. The Terminator is responsible for transferring ownership of a Format object to another framework or application component.

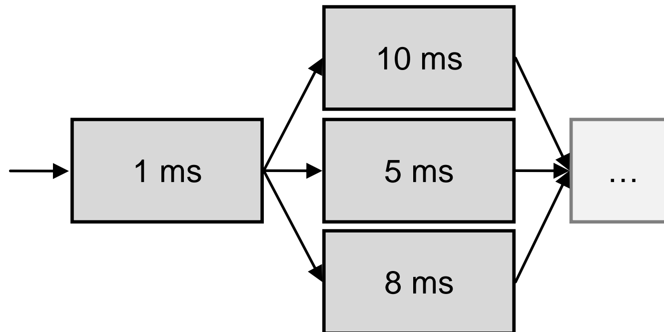
Table 4 - Pipeline Function Interfaces

Method	Description
operator(const Format*) : Boolean	Filter signature.
operator(Format*) : Boolean	Transform and Terminator signature.
operator(const Format*) : Format*	Translator signature.

The read-only nature of Filter objects makes it possible to process multiple filters simultaneously. A special purpose Pipeline Function subclass, Filter Group, has been defined to concurrently process more than one Filter. The Filter Group subclass implements the same interface as the Filter subclass. It simply contains an aggregation of Filter objects which it attempts to execute simultaneously when multiple processing units are available. Figure 7 compares a Pipeline containing sequentially arranged Filter objects with a Pipeline containing Filter objects within a Filter Group.



Filter sequence: minimum execution time of 1 ms and maximum execution time of 24 ms



Filter and filter group: minimum execution time of 1 ms and maximum execution time of 11 ms

Figure 7- Comparison of Filter and Filter Group Execution

3.3.5 Classifier

Classifier is an abstract class implementing the Builder design pattern. It provides functionality to convert the "raw" Format objects associated with Source objects to structured Format objects.

Table 5 - Classifier Interface

Method	Description
toRaw	Pure virtual method to convert a "raw" Format object to a structured Format object. All subclasses must implement this function.
fromRaw	Pure virtual method to convert a structured Format object to a "raw" Format object. All subclasses must implement this function.

3.3.6 Cache

Cache is an abstract class providing a base from which concrete Cache objects must be derived. It is specifically designed to store a fixed number of items received from a Source, retaining a recent history of received Format objects. Methods for adding and accessing objects must be thread-safe so that the cache may be concurrently accessed by a Source object, Pipeline Functions, and application-specific components. Cache objects are designed to be processed with the Visitor design pattern, providing a simple and thread-safe method for sequentially accessing each stored object.

Table 6 - Cache Interface

Method	Description
addData	Pure virtual method to add an item to the cache. All subclasses must implement this function.
getData	Pure virtual method to retrieve an item from the cache. All subclasses must implement this function.
accept	Method to accept a visitor, for application to items in the cache, as part of the Visitor design pattern.

3.3.7 Delegate

Delegate is an abstract class responsible for transferring Format objects received from a Source object to one or more Pipeline objects. Each Pipeline is provided with a unique copy of the Format object produced by the Format object's `clone` method. Delegate allows data to be transferred between the Source Monitor and Pipeline Executor modules without the modules having to communicate directly.

Table 7 - Delegate Interface

Method	Description
dispatch	Pure virtual method to transfer ownership of a Format object to one or more Pipeline objects. All subclasses must implement this function.

3.3.8 Decomposer

Decomposer is an abstract design concept to be implemented with the Decorator design pattern. The purpose of the Decomposer is to decompose Format Group objects for consumption by framework components which operate on individual Framework objects.

For example, a Decomposing Delegate object, derived from the Delegate class, contains a reference to an existing Delegate object and implements the `dispatch` method as a function to decompose a Format Group object. The Visitor pattern is used to apply a Visitor function to each object contained by the Format Group object. The Visitor function provides a copy of each visited Format object to the `dispatch` method of the Decomposing Delegate object's encapsulated Delegate object.

3.3.9 Monitor

Monitor is a class implementing design patterns for managing and dispatching communication events, such as the Reactor and Proactor design patterns. The actual design patterns chosen for implementation of the Monitor class will depend upon the properties of the underlying system.

Source objects are registered with a Monitor object, which will detect and process pending communication events. Data received from an input Source is transferred to one or more Pipeline objects through a Delegate object associated with the Source. Data queued for writing by an output Source is transmitted when the Source is available for writing.

Methods for registering and deregistering Sources must be thread-safe so that the application may manipulate the Monitor object from a thread different than the Monitor object's thread of execution. Finite sources which reach a completed state or sources which enter an error state are removed from processing and marked with the appropriate state indicator.

Table 8 - Monitor Interface

Method	Description
addSource	Method to register a Source object, and an associated Delegate object, with the Monitor. A specified I/O mode determines if the Source object should be monitored for reading or writing.
removeSource	Method to deregister a Source object with the Monitor.
getNumEOFStates	Method reporting the number of registered Source objects which have reached a completed state.
getNumErrorStates	Method reporting the number of registered Source objects which have entered an error state.
run	Method to start execution of the Monitor.
stop	Method to stop execution of the Monitor.

3.3.10 Executor

Executor is a class responsible for managing Pipeline execution. Pipeline objects are added to and removed from an Executor object which will execute Pipeline objects that have pending data. Methods

for adding and removing Pipeline objects must be thread-safe so that the application may manipulate the Executor object from a thread different than the Executor object's thread of execution.

Table 9 - Executor Interface

Method	Description
addPipeline	Method for adding a Pipeline object to the Executor for processing.
removePipeline	Method for removing a Pipeline object from the Executor.
run	Method to start execution of the Executor.
stop	Method to stop execution of the Executor.

3.4 Downscaling

The Source Monitor and Pipeline Executor modules have been designed to take full advantage of available hardware parallelism. Support for simple single processor systems has also been considered for each module's design. Rather than execute within a dedicated thread, each module may execute synchronously with other application modules from within a single application thread. Implementations of the executing components from each module are made to operate within this serial environment, leaving the overall framework design unchanged.

The Monitor object's `run` method is implemented such that, rather than execute continuously, it performs a single test for pending communication events. When pending events are detected, the associated communication resources are processed serially. This version of the Monitor object's `run` method is periodically executed by the main application thread. The module design remains the same, only the implementation of the method responsible for updating the module changes.

The Executor object's `run` method is implemented such that each invocation processes a fixed number of Pipeline objects that have pending data. Pipeline execution no longer runs continuously. Format objects are processed serially, restricting flow of data through the pipeline to one Format object per Pipeline execution. This version of the Monitor object's `run` method is periodically executed by the

main application thread. As with the Source Monitor module, only the implementation of the method responsible for updating the Pipeline Executor module changes.

Downscaling of the Framework is achieved by adjusting a single component from each module to operate synchronously from within a single thread. The number of tasks executed concurrently by each module is adjusted to match the number of available processing units. Updates to each module perform a limited number of operations, allowing fair sharing of processing resources among each module. These simple implementation details, transforming continuous loops to single loop iterations, allow the module to scale from support of high-performance multi-core systems to single-core embedded systems.

4. Results

The final result of the project is Phylum, an inherently scalable implementation of the framework design with support for high-performance multi-core systems. The target operating systems for the current Phylum implementation are Linux and the Microsoft Windows family of operating systems. Both core modules are implemented, along with a small component library of concrete subclasses. Included with the component library are generic Format and Classifier subclasses designed for dynamic creation of new Format and Classifier types. To reduce the impact of frequent Format object-related memory operations, techniques for scalable memory allocation and efficient memory reuse have been considered for the Phylum implementation.

Detailed descriptions for the implementations of the major Phylum components, including some unexpected implementation challenges, are provided below.

4.1 Implementation

Phylum is implemented with the C++ programming language (Stroustrup, 1997) and the Intel Threading Building Blocks library (Reinders, 2007). Threading Building Blocks (TBB) provides a library of template-based algorithms for parallel processing (Voss, Demystify Scalable Parallelism with Intel

Threading Building Block's Generic Parallel Algorithms, 2006). Concurrent containers provided by TBB simplify the task of thread-safe resource sharing (Voss, Enable Safe, Scalable Parallelism with Intel Threading Building Block's Concurrent Containers, 2006). TBB components are used extensively by Phylum's Source Monitor and Pipeline Executor implementations for parallel execution and asynchronous inter-module communication.

4.1.1 Source Monitor Implementation

The Monitor object is the chief component of the Source Monitor module. Phylum's implementation of the Monitor object includes aspects of both the Reactor and Proactor patterns. As with the Reactor pattern, system calls such as `select` (Stevens, 1998) and `WaitFormMultipleObjects` (Andrews, 1996) are used to detect pending communication events. These calls, made from within a continuous loop, do not wait indefinitely for a communication event to occur. They wait for a programmatically specified duration before returning. This allows changes to the list of monitored Source objects, made from the main application thread, to be applied to the Monitor object's run loop with minimal delay.

When communication events are detected, the `parallel_for` algorithm from the TBB library is used to concurrently process each communication operation. Data received from a Source object is transferred to one or more associated Pipeline objects through a Delegate object. Data is transferred to Pipeline objects asynchronously through the TBB `concurrent_queue` container. Concurrent processing and dispatching of communication operations is done to emulate the Proactor design pattern.

Phylum's Monitor implementation is an attempt to implement the Proactor pattern without the use of system level asynchronous I/O operations. Future work will address the implementation of a true Proactor based object that uses asynchronous I/O operations provided by the underlying operating system.

4.1.2 Pipeline Executor Implementation

The chief components of the Pipeline Executor module are the Executor object and the Pipeline object. The Executor object has a fairly simple implementation, with a continuous run loop to concurrently

execute Pipeline objects. Within the run loop, Pipeline objects are executed concurrently with the TBB `parallel_for` algorithm. The `parallel_for` construct is invoked repeatedly from within the run loop, executing any Pipeline objects which have pending data at the time of invocation. A limit is placed on the number of items that a Pipeline object may process during each execution.

Processing limits are required for Pipeline objects executing within the continuous run loop because it is possible for a Pipeline object receiving a constant stream of data to starve other Pipeline objects by preventing the `parallel_for` from completing. Placing a limit on the number of items processed per execution prevents any single Pipeline object from starving the others. Each Pipeline object yields after processing a fixed number of items to guarantee that the `parallel_for` will complete, allowing execution of Pipeline objects which have pending data items to be reinitiated. The Executor object's implementation adheres to a cooperative multitasking paradigm which requires tasks to yield to each other.

The implementation of the Pipeline object defines a class encapsulating the TBB `pipeline` algorithm. TBB's pipeline algorithm processes a sequence of TBB `filter` objects which may process items concurrently or serially. Phylum's Pipeline object implementation provides an interface for constructing and executing TBB `pipeline` objects. The Adapter design pattern is used to encapsulate the TBB `filter` class within classes specifying the Filter, Transform, Translator, and Terminator class interfaces. The Filter Group object uses the TBB `parallel_reduce` algorithm to concurrently execute multiple Filter objects, reducing the Boolean return values to a single value which is only `true` when all return values are `true`.

Methods for adding and removing Pipeline Functions must be thread-safe, as the application thread may attempt to manipulate a Pipeline object executed by a dedicated thread. Simple locking of the pipeline structure is an insufficient solution, as long runs may cause the main application thread to block for extended periods of time. For this reason, a two-phase solution to pipeline construction is

implemented with Phylum's Pipeline object. A structure representing the desired pipeline configuration is constructed by the methods for adding and removing Pipeline Functions. Creation of the actual pipeline is deferred until the Pipeline object begins execution. Changes to an executing Pipeline object are achieved by suspending and then resuming the Pipeline object.

Future work will address the implementation of the Executor object's run loop, testing the efficiency of different implementations using alternate parallel algorithms such as the TBB `parallel_while` algorithm.

4.2 Dynamic Creation of Format and Classifier Objects

Within the white-box framework model, the definition of concrete Format and Classifier subclasses is expected to be done through extension of their associated abstract base classes. Class inheritance is a static procedure which limits the ability to define new Format and Classifier types at run-time. Phylum implements generic Format and Classifier subclasses to provide support for dynamic Format and Classifier type definition. The generic classes allow new Format and Classifier type definition through the composition of fields of elements, transitioning Phylum toward a black-box framework model.

Phylum's BinaryAdapterFormat object is a concrete subclass of the abstract Format class. BinaryAdapterFormat encapsulates a collection of fields of different types. Each field contains one or more elements. Field types include single elements, fixed and variable length arrays of elements, and fixed and variable length strings. String fields are special cases of array fields. Element types correspond to basic types such as `int` and `double`.

Fields may be dynamically added to and removed from a BinaryAdapterFormat object. Methods for mapping fields to and from an array of bytes are provided for converting to and from the "raw" data formats used by communication resources. The BinaryAdapterClassifier contains one or more BinaryAdapterFormat objects that serve as Prototypes for conversion of "raw" communication resource data objects to structured Format objects. A Prototype copy is made for each "raw" data object

converted. Each Prototype is associated with an (offset, type, value) 3-tuple to be compared with the "raw" data object for matching raw data with the appropriate Prototype.

4.3 Memory Management

The Phylum implementation requires high frequency memory allocation and de-allocation of Format objects. A memory caching technique (Milewski, 2001) has been employed to reduce the impact of frequent memory operations. The memory cache is implemented as a template class dedicated to the recycling of memory. Each concrete Format subclass contains a static memory cache object, providing a single shared cache for each specific subclass. Overloaded `new` and `delete` operators force Format objects to perform all memory operations using the cache object. Macros to perform static memory cache, `new`, and `delete` declaration have been created to simplify memory cache integration with Format subclasses.

Phylum's memory cache uses the TBB `concurrent_queue` to store unused objects. Although this allows multiple threads to safely access the cache simultaneously, it may also limit scalability. TBB provides a scalable memory allocator based on McRT-Malloc (Hudson, 2006). McRT-Malloc assigns ownership of pre-allocated blocks of memory to individual threads, allowing lock-free memory allocation. TBB's scalable memory allocator will likely replace the current memory caching scheme for Format object allocation. Future work will be performed to evaluate the benefit derived from integration of the TBB scalable memory allocator with the Phylum's Format object memory management system.

4.4 Processing Simple Finite Data Streams

With the initial framework design, finite data streams proved to be much more difficult to manage than continuous data streams. When processing continuous data streams, programs may simply run until an external entity tells them to stop. With finite data streams, programs must detect when to stop by determining that all stream data has been processed. The framework's modular, multi-threaded design makes the process of detecting the completion of finite data stream processing rather challenging.

The finite stream processing problem stems from the fact that the Source Monitor and Pipeline Executor modules operate independently, asynchronously passing messages to each other, with no other module-to-module coordination. Consider a simple single-threaded program to compress a file. Blocks of data are read from the source file, compressed, and written to a destination file repeatedly, until all available blocks have been read from the source and written to the destination. With Phylum, the file compression procedure is very different.

To compress a file, input and output FileSource objects are created and registered with a Monitor object. The Monitor object repeatedly reads blocks of data from the source, submitting them to a Pipeline object. The Pipeline object removes items from its data queue, compresses them, and then submits them to the output source's write queue to be written to the destination by the Monitor object. These operations are all processed concurrently, encapsulated within loosely coupled framework components, making it difficult to determine that all blocks have been read from the source and written to the destination.

A seemingly anecdotal way to detect completion of finite data stream processing is to monitor the states of the FileSource and Pipleine objects. When all available blocks of data have been read from the FileSource object, it is marked as having an EOF state. Detection of processing completion is done by first waiting for the source to enter the EOF state, then waiting for the Pipeline object's data queue to become empty, and finally waiting for the destination's write queue to become empty. Although this method may appear to work, monitoring these states becomes cumbersome when the application includes multiple Pipeline and Source objects. There is also a possibility that the Pipeline object's queue and the destination's write queue both report an empty state while a data item is transitioning between modules, producing a false positive.

A better, simpler, and more elegant solution to the problem exists in the form of an EOF token. An EOF token, implemented as a special purpose Format subclass, is submitted to the Pipeline object when the input Source object enters the EOF state. The EOF token flows through the pipeline, ignored by

the processing stages, until reaching the termination stage, which may dispatch a completion notification to the application or transfer the EOF token the destination's write queue. When the Monitor object extracts the EOF token from the destination's write queue, it will dispatch a completion notification to the application. For the EOF token concept to work, the Pipeline object's termination stage must be serial to prevent the EOF token from accidentally moving past other items in the pipeline. Future work will integrate the EOF token concept with the framework.

5. Usage

The sample programs described in this section demonstrate common usage of the Phylum framework. An example is provided for each of the three approaches to applying the framework to a problem that were previously identified (Section 1.4.2). The examples introduce some of the components from the Phylum class library.

5.1 Data Converter Example

The first example describes a file conversion program responsible for converting ASCII files containing lines of comma-separated values (CSV) to files containing XML data. Comments, empty lines, and lines containing invalid data have been mixed with lines containing valid CSV data records. The `csv2xml` file conversion program must convert lines containing valid CSV data records to XML, ignoring any invalid data.

The program starts with the definition of Filter, Transform, and Translator subclasses for CSV to XML conversion. CSV to XML conversion functions operate on the TextLineFormat and BinaryAdapterFormat classes included as part of the Phylum class library. The definition of a "valid content" Filter subclass is presented here:

```
// Filter to remove lines with invalid characters
class ContentFilter : public Filter
{
protected:
```

```
  std::string validCharacters_;
public:
  // Constructor - set filter mode as non-serial
  ContentFilter(const std::string& validCharacters)
  : Filter(false), validCharacters_(validCharacters) { }

  // Filter format based on format content
  virtual bool operator()(const Format* fmt)
  {
    // Convert to TextLineFormat
    const TextLineFormat* line =
        dynamic_cast<const TextLineFormat*>(fmt);

    // Validate result of the type conversion
    if(line == NULL)
    {
      throw UnsupportedFormatException(
        "Invalid format for ContentFilter; expected TextLineFormat");
    }

    // Check for existence of invalid characters
    const std::string& s = line->getLine();
    return (s.find_first_not_of(validCharacters_) !=
        std::string::npos) ? false : true;
  }
};
```

After the implementation of the application-specific Filter, Transform, and Translator subclasses, the application must instantiate the framework components responsible for performing the file conversion. The application uses the FileSource class included with the Phylum class library for processing the file streams. It must register an input and output FileSource object with a Monitor object and construct a Pipeline to process the data. An excerpt from the initialization procedure is presented here:

```
// Create the input and output file sources
FileSource csvin(FileSource::IN, "input.csv");
FileSource xmlout1(FileSource::OUT, "output.xml",
                   FileSource::CREAT|FileSource::TRUNC);

// Create filters
// Remove all comments begining with '#'
StripCommentTransform xmlsct;

// Remove all whitespace from begining and end of line
StripWhitespaceTransform xmlswt;

// Create filter group for concurrent filter processing
FilterGroup xmlfgrp(false);
xmlfgrp.addFilter(&xmlcflt);
```

```
xmlfgrp.addFilter(&xmlcsvflt);
…
// Create pipeline
Pipeline xmlpipeline;

// Add filters to pipeline
xmlpipeline.addFunction(&xmlsct);
…
// Set the classifier for the sources
TextLinesClassifier tlsc;
csvin.setClassifier(&tlsc);

BinaryAdapterClassifier bac;
xmlout.setClassifier(&bac);

// Add the source and its delegate to the monitor
Monitor monitor;
monitor.addSourceMonitor(&csvin, Monitor::READ, &dda);
monitor.addSourceMonitor(&xmlout, Monitor::WRITE, NULL);
```

Once the framework components have been initialized, the application starts the threads responsible for Source Monitor and Pipeline Executor execution. The application will then wait for the completion of the file conversion process, after which it terminates execution. A full source code listing for the `csv2xml` conversion application can be found in Appendix A.

Many of the concrete Filter and Transform subclasses defined for this example fit within the foundation and architecture object domains. Because objects within these domains have a high degree of reusability, these subclasses are good candidates for inclusion with the Phylum class library.

5.2 Data Consumer Example

This example demonstrates the use of Phylum to add support for importing GPS data, received from a serial port, to an existing two dimensional mapping application. The mapping application plots positions specified as latitude and longitude on a map. The GPS data, formatted as NMEA (NMEA, 2002) strings, is received, filtered, and converted to an application defined Point2DFormat object by Phylum.

The implementation of this example is fairly straight forward. It requires the definition of a Point2DFormat class, two Filter subclasses to test for valid NMEA strings, and a Translator subclass to perform NMEA string to Point2DFormat conversion. Existing TextLineFormat, TextLineClassifier, and

COMSource classes included with the Phylum class library are used to process the NMEA string data. The definition of the Point2DFormat class is presented here:

```
// Point2D format
class Point2DFormat : public Format
{
public:
  double x_, y_;  // (x,y) coordinate

public:
  // Initializing constructor; type identifier is set to 0 to indicate
  // no type information is associated with the object
  Point2DFormat(double x, double y)
  : Format(0), x_(x), y_(y) { }

  // Copy constructor for use with the clone method
  Point2DFormat (const Point2DFormat & copy)
  : Format(copy), x_(copy.x_), y_(copy.y_) { }

  // Create a full copy of the object
  Point2DFormat* clone() const
  {
    return new Point2DFormat(*this);
  }

  // Declare memory caching operations
  POOL_DECLARE(Point2DFormat)
};
```

The NMEA string providing the latitude and longitude information which the program must extract has the form:

$GPGLL,4916.45,N,12311.12,W,225444,A,*31

The first element of the string indicates the NMEA message type and the last element contains a checksum. Filter subclasses to check for the appropriate NMEA message type and validate the checksum are implemented for the application. These filters are not executed concurrently with a Filter Group object, as there is no need to verify the checksum for an unsupported NMEA string type. A Translator subclass to convert the ASCII latitude and longitude values contained by the string to a Point2DFormat object is the final component that must be defined for the example.

As with the previous example, creation of the Format, Filter, and Transform subclasses is followed by the instantiation of the appropriate framework components. The application requires a Monitor object to process the input source and a Pipeline object to process the NMEA data received from the input source. The terminal stage of the pipeline is a custom Terminator subclass responsible for transferring ownership of Point2DFormat objects to the mapping application.

Once the appropriate framework components have been instantiated, the threads responsible for Source Monitor and Pipeline Executor execution are started. Because the input source is a continuous data stream, the mapping application does not have to check for source completion. The threads execute until explicitly stopped by the mapping application.

The concrete Filter subclasses defined for this example fit within the business object domain. These subclasses may be candidates for inclusion within a GPS or NMEA specific module of the Phylum class library.

5.3 Data Producer Example

The final example describes the use of the Phylum framework for streaming compressed audio data to a network. Phylum is integrated with an existing application to transform and transmit audio data that the application produces. This is a very simple example which only requires the creation of Transform subclasses to perform the audio compression. All of the other components used by the example are included with the Phylum class library.

No special Format types are required for this example. Compression is performed on blocks of bytes containing audio data. The RawFormat object included with the Phylum class library acts as a container for the blocks of bytes. RawFormat is used as the native format for the Phylum class library's UnicastSource, which is used to transmit the compressed data to the network.

Concrete Transform subclasses are implemented for the lossless transformation, quantization, and entropy coding stages of any popular audio compressing algorithm (no specific algorithm has been chosen for this example). A Pipeline object composed of the audio compression Transform objects and a TransmissionTerminator object provided by the Phylum class library is instantiated to process the uncompressed audio data. The TransmissionTerminator transmits data directly to the network, avoiding the need for a Monitor object.

Once the appropriate framework components have been instantiated, the thread responsible for Pipeline Executor execution is started. Uncompressed audio data is directly submitted to the Pipeline object by the application. The thread executes until explicitly stopped by the application.

As with the previous example, the concrete Transform subclasses defined for this example fit within the business object domain. These subclasses may be candidates for inclusion within a compression module of the Phylum class library.

6. Conclusion and Future Direction

A flexible framework for data format classification and transformation, to be used with the development of interoperable applications, has been presented. The framework addresses the design of efficient and scalable data processing systems. A modular framework design and reusable component library simplify the integration of new data format types with existing applications. The communication resource management and concurrent data processing aspects of interoperable application design are abstracted from the application developer, reducing design complexity. Application code duplication is reduced through the direct use of existing framework components, or the creation of new application-specific components through the composition and extension of existing framework components.

Framework components have been designed to exploit data parallelism, providing a high-performance solution for processing high-rate data streams. Real-time management and manipulation of

live data streams is performed by custom data processing pipelines composed of operations for data filtering, transformation, and translation. Support for dynamic data stream-to-pipeline association and dynamic pipeline composition provides a high degree of support for the adjustment of applications at run-time. Although the framework was initially intended to address a specific problem domain, its general design allows it to be applied to many other areas, such as distributed data processing and compression of live data streams.

The long-term goal for the framework is the production of a domain specific language that supports run-time definition, creation, and manipulation of framework objects from within a language interpreter. This goal will be realized through an evolutionary framework development process starting with the creation of a white-box framework. As the framework's library of fine-grained concrete classes grows, it will transition to black-box framework status. Once the black-box framework has been realized, tools for composing applications from framework components may be created. Finally, the implementation of an interpreter to control these tools will produce a domain-specific language for data format classification and transformation.

This thesis has addressed the first stage of project evolution, the design and implementation of a white-box framework with a small library of concrete classes representing the initial stage of a black-box framework. Future work will focus on the further development of the black-box framework, the creation of framework tools for constructing applications, and the design of a domain-specific language syntax and interpreter for dynamic application creation and modification. Ultimately, a domain-specific language operating within a secure, distributed environment consisting of interconnected language interpreters which may be manipulated remotely, for the purpose of controlling a network of dynamically adjustable network gateways, will be generated.

7. Acknowledgements

I owe a great debt of gratitude to the faculty and staffs at both The George Washington University Department of Computer Science and the Naval Research Laboratory, whose assistance and support have made this work possible. I would particularly like to thank the following:

- From GW, I would like to thank Professors Rahul Simha and Bhagi Narahari for providing guidance and advice regarding this work, and for being patient and accommodating as project scope and direction fluctuated.
- From NRL, I would like to thank John Binford and William Doughty for allowing my continued participation in the Edison Program and providing the opportunity to complete this project; Ned Brekelbaum for the end-token suggestion; and the co-workers whom I subjected to scattered descriptions of the project while attempting to improve my own understanding.

I would also like to thank Michael Shintani, John Shishido, Steve Pringle, and the other members of the PMRF Data Systems branch, past and present, responsible for creating the PMRF instrumentation network and providing a model for sensor data collection network design.

Finally, I would like to thank anyone with whom I have discussed this work and who has reviewed this work, improving the overall quality of the project.

Bibliography

Andrews, M. (1996). *C++ Windows NT Programming, 2nd ed.* New York: M&T Books.

Binford, J. Q., & Doughty, W. A. (2002). Advanced Visualization for Test and Evaluation and Training Ranges. *NRL Review* , 143-145.

Dana, P. H. (1999, December 15). *Coordinate Systems Overview.* Retrieved from The Geographer's Craft Project, Department of Geography, The University of Colorado at Boulder: http://www.colorado.edu/geography/gcraft/notes/coordsys/coordsys.html

Dougherty, D., & Robbins, A. (1997). *sed & awk, 2nd ed.* Sebastopol, CA: O'Reilly & Associates, Inc.

DVD Forum. (2006). *HD DVD - Video Product Requirement & Guideline Version 1.02.* Tokyo: DVD Format/Logo Licensing Corporation.

E. Gamma, R. H. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software.* Reading, MA: Addison-Wesley.

Ennals, R. S. (2004). *Linear types for packet processing (extended version).* Cambridge: Tech. Rep. UCAM-CL-TR-578, University of Cambridge Computer Laboratory.

GSA. (1996). *Federal Standard 1037C.* General Services Administration.

Hall, M. L. (1993). A Standard Data Format for instrument data interchange - HP's SDF standard for analyzers - Technical. *Hewlett-Packard Journal* .

Hudson, R. L. (2006). McRT-Malloc - A Scalable Transactional Memory Allocator. *Proceedings of the 2006 International Symposium on Memory Management* (pp. 74-83). New York: ACM Press.

I. Pyarali, T. H. (1997). Proactor: An Object Behavioral Pattern for Demultiplexing and Dispatching Handles for Asynchronous Events. *4th Annual Pattern Languages of Programming Conference.*

Johnson, R. E., & Foote, B. (1988). Designing Reusable Classes. *Journal of Object-Oriented Programming* , 22-35.

Milewski, B. (2001). *C++ In Action: Industrial Strength Programming Techniques.* Reading, MA: Addison Wesley.

Morrison, J. (1985). *"EA IFF 85" Standard for Interchange Format Files.* Electronic Arts.

NIST. (1996). *FIPS PUB 161-2: Electronic Data Interchange (EDI).* National Institutes of Standards and Technology.

NMEA. (2002, January). *Publications and Standards from the National Marine Electronics Association (NMEA) / NMEA 0183*. Retrieved from The National Marine Electronics Association Website: http://www.nmea.org/pub/0183/index.html

OMG. (2008). *Common Object Request Broker Architecture (CORBA) Specification, Version 3.1.* Object Management Group.

Page-Jones, M. (1995). *What Every Programmer Should Know About Object-Oriented Design.* New York, NY: Dorset House Publishing.

Reinders, J. (2007). *Intel Threading Building Blocks.* Sebastopol, CA: O'Reilly Media, Inc.

Roberts, D., & Johnson, R. (1997). Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks. In *Pattern Languages of Program Design 3.* Addison Wesley.

Sayood, K. (2006). *Introduction to Data Compression.* San Francisco, CA: Morgan Kaufmann Publishers.

Schmidt, D. C. (1995). Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching. In e. J. O. Coplien and D. C. Schmidt, *Pattern Languages of Program Design* (pp. 529–545). Reading, MA: Addison-Wesley.

Stevens, W. R. (1998). *UNIX Network Programming, 2nd ed.* Upper Saddle River, NJ: Prentice Hall PTR.

Stroustrup, B. (1997). *The C++ Programming Language, 3rd ed.* Reading, MA: Addison-Wesley.

TENA Software Development Activity. (2008). *TENA: Introduction*. Retrieved from TENA SDA Website: http://tena-sda.org/

U.S. Naval Research Laboratory. (2008). *SIMDIS web site*. Retrieved from https://simdis.nrl.navy.mil

Voss, M. (2006, October 26). *Demystify Scalable Parallelism with Intel Threading Building Block's Generic Parallel Algorithms.* Retrieved from DevX: http://www.devx.com/cplus/Article/32935

Voss, M. (2006, December 11). *Enable Safe, Scalable Parallelism with Intel Threading Building Block's Concurrent Containers.* Retrieved from DevX: http://www.devx.com/cplus/Article/33334

Wadler, P. (1990). Linear types can change the world! In M. Broy, & C. Jones, *IFIP TC 2 Working Conference on Programming Concepts and Methods* (pp. 347-359). North Holland.

Appendix A

Appendix A contains the full source code listing for the example `csv2xml` file conversion application (Section 5.1). The full version of the program contains an additional pipeline, not previously described, to illustrate the association of a data source with multiple processing pipelines. Lines of comma-separated values (CSV) received from an ASCII file are transferred to two separate pipelines. The first pipeline performs the previously described CSV to XML conversion. The second pipeline simply strips comments and invalid lines from the original file. Both pipelines produce two separate output files, to demonstrate the ease with which data redirection to multiple consumers may be done.

```
#ifdef WIN32
#include <windows.h>
#define usleep(x) Sleep(x/1000)
#define TRETURN DWORD WINAPI
#else
#include <pthread.h>
#include <unistd.h>
#define TRETURN void*
#endif

#include <iostream>
#include <tbb/tick_count.h>
#include <tbb/task_scheduler_init.h>
#include "Exceptions.h"
#include "ForwardTerminator.h"
#include "PipelineDispatcher.h"
#include "Pipeline.h"
#include "Monitor.h"
#include "FileSource.h"
#include "TextLinesClassifier.h"
#include "TextLinesDecomposer.h"
#include "TextLineFormat.h"
#include "TextLineClassifier.h"
#include "DecomposingDelegateAdapter.h"
#include "BinaryAdapterFormat.h"
#include "BinaryAdapterClassifier.h"

using namespace std;
using namespace phylum;

// Monitor's thread execution function
TRETURN run_monitor(void* monitor)
{
  tbb::task_scheduler_init tsi;
  try
  {
    static_cast<Monitor*>(monitor)->run();
```

```
  }
  catch(Exception e)
  {
    std::cout << e.getMessage() << std::endl;
    exit(-1);
  }
  return NULL;
}

// Executor's thread execution function
TRETURN run_executor(void* executor)
{
  tbb::task_scheduler_init tsi;
  try
  {
    static_cast<PipelineExecutor*>(executor)->run();
  }
  catch(Exception e)
  {
    std::cout << e.getMessage() << std::endl;
    exit(-1);
  }
  return NULL;
}

// Strip comment from a line
class StripCommentTransform : public Transform
{
public:
  // Constructor - set transform mode as non-serial
  StripCommentTransform() : Transform(false) { }

  // Filter based on format content
  virtual bool operator()(Format* fmt)
  {
    TextLineFormat* line = dynamic_cast<TextLineFormat*>(fmt);
    if(line == NULL)
    {
      throw UnsupportedFormatException("Invalid format for
StripCommentTransform; expected TextLineFormat");
    }

    std::string s = line->getLine();
    std::string::size_type pos = s.find_first_of('#');
    if(pos != std::string::npos)
    {
      line->setLine(s.substr(0, pos));
    }

    return true;
  }
};

// Transform to remove whitespace from begining and end
// Empty strings are filtered
class StripWhitespaceTransform : public Transform
{
```

```
public:
  // Constructor - set transform mode as non-serial
  StripWhitespaceTransform() : Transform(false) { }

  // Filter based on format content
  virtual bool operator()(Format* fmt)
  {
    TextLineFormat* line = dynamic_cast<TextLineFormat*>(fmt);
    if(line == NULL)
    {
      throw UnsupportedFormatException("Invalid format for
StripWhitespaceTransform; expected TextLineFormat");
    }

    std::string s = line->getLine();
    std::string::size_type first = s.find_first_not_of(" \t\n\r");

    if(first == std::string::npos)
    {
      return false;
    }

    line->setLine(s.substr(first, s.find_last_not_of(" \t\n\r")-first+1));

    return true;
  }
};

// Filter to remove lines with invalid characters
class ContentFilter : public Filter
{
protected:
  std::string validCharacters_;
public:
  // Constructor - set filter mode as non-serial
  ContentFilter(const std::string& validCharacters)
  : Filter(false), validCharacters_(validCharacters) { }

  // Filter based on format contents
  virtual bool operator()(const Format* fmt)
  {
    const TextLineFormat* line = dynamic_cast<const TextLineFormat*>(fmt);
    if(line == NULL)
    {
      throw UnsupportedFormatException("Invalid format for ContentFilter;
expected TextLineFormat");
    }

    const std::string& s = line->getLine();
    return (s.find_first_not_of(validCharacters_) !=
            std::string::npos) ? false : true;
  }
};

// Filter to check for the correct number of values per line
class CSVFilter : public Filter
{
```

```
protected:
  unsigned int num_;
public:
  // Constructor - set filter mode as non-serial
  CSVFilter(unsigned int num) : Filter(false), num_(num) { }

  // Filter based on format contents
  virtual bool operator()(const Format* fmt)
  {
    const TextLineFormat* line = dynamic_cast<const TextLineFormat*>(fmt);
    if(line == NULL)
    {
      throw UnsupportedFormatException("Invalid format for CSVFilter;
expected TextLineFormat");
    }

    const std::string& s = line->getLine();
    unsigned int count = 0;
    for(std::string::const_iterator iter = s.begin();
        iter != s.end();
        ++iter)
    {
      if(*iter == ',')
      {
        if(++count > num_)
        {
          break;
        }
      }
    }

    return (num_ == count) ? true : false;
  }
};

// Transform to add a line feed to the end of a line
class AppendLineFeedTransform : public Transform
{
public:
  // Constructor - set transform mode as non-serial
  AppendLineFeedTransform() : Transform(false) { }

  // Filter based on format contents
  virtual bool operator()(Format* fmt)
  {
    TextLineFormat* line = dynamic_cast<TextLineFormat*>(fmt);
    if(line == NULL)
    {
      throw UnsupportedFormatException("Invalid format for
AppendLineFeedTransform; expected TextLineFormat");
    }

    std::string s = line->getLine();
    s.push_back('\n');
    line->setLine(s);

    return true;
```

```
  }
};

// String tokenizer from
http://www.oopweb.com/CPP/Documents/CPPHOWTO/Volume/C++Programming-HOWTO-
7.html
void tokenize(const std::string& str,
              std::vector<std::string>& tokens,
              const std::string& delimiters = " ")
{
  // Skip delimiters at beginning.
  std::string::size_type lastPos = str.find_first_not_of(delimiters, 0);
  // Find first "non-delimiter".
  std::string::size_type pos = str.find_first_of(delimiters, lastPos);

  while (std::string::npos != pos || std::string::npos != lastPos)
  {
    // Found a token, add it to the vector.
    tokens.push_back(str.substr(lastPos, pos - lastPos));
    // Skip delimiters.  Note the "not_of"
    lastPos = str.find_first_not_of(delimiters, pos);
    // Find next "non-delimiter"
    pos = str.find_first_of(delimiters, lastPos);
  }
}

// Converts the CSV lines to XML data
// Uses BinaryAdapterFormat to store the XML data,
// simulating creation of binary data
class CSV2XMLTranslator : public Translator
{
protected:
  BinaryAdapterFormat template_;
public:
  CSV2XMLTranslator() : Translator(false), template_(0, false)
  {
    template_.addTermStringField("time", '\n');
    template_.addTermStringField("x", '\n');
    template_.addTermStringField("y", '\n');
    template_.addTermStringField("z", '\n');
    template_.addTermStringField("vx", '\n');
    template_.addTermStringField("vy", '\n');
    template_.addTermStringField("vz", '\n');
  }

  // Filter based on format contents
  virtual Format* operator()(const Format* fmt)
  {
    const TextLineFormat* line = dynamic_cast<const TextLineFormat*>(fmt);
    if(line == NULL)
    {
      throw UnsupportedFormatException("Invalid format for CSV2XMLTranslator;
expected TextLineFormat");
    }

    // Separate the CSV line to a vectr of string values
    std::vector<std::string> tokens;
```

```
    std::string s;
    tokenize(line->getLine(), tokens, ",");

    BinaryAdapterFormat* adapter = template_.clone();

    // Mix XML tags with data
    s = std::string("<Data>\n\t<Time>") + tokens[0] + std::string("</Time>");
    adapter->setFieldValue("time", s);

    s = std::string("\t<PositionX>") + tokens[1] +
        std::string("</PositionX>");
    adapter->setFieldValue("x", s);

    s = std::string("\t<PositionY>") + tokens[2] +
        std::string("</PositionY>");
    adapter->setFieldValue("y", s);

    s = std::string("\t<PositionZ>") + tokens[3] +
        std::string("</PositionZ>");
    adapter->setFieldValue("z", s);

    s = std::string("\t<VelocityX>") + tokens[4] +
        std::string("</VelocityX>");
    adapter->setFieldValue("vx", s);

    s = std::string("\t<VelocityY>") + tokens[5] +
        std::string("</VelocityY>");
    adapter->setFieldValue("vy", s);

    s = std::string("\t<VelocityZ>") + tokens[6] +
        std::string("</VelocityZ>\n</Data>");
    adapter->setFieldValue("vz", s);

    return adapter;
  }
};

int main()
{
  tbb::task_scheduler_init tsi;

  tbb::tick_count start = tbb::tick_count::now();

  try
  {
    // Create input and output sources
    FileSource csvin(FileSource::IN, "input.csv");

    FileSource lineout1(FileSource::OUT, "lineoutput1.csv",
                        FileSource::CREAT|FileSource::TRUNC);
    FileSource lineout2(FileSource::OUT, "lineoutput2.csv",
                        FileSource::CREAT|FileSource::TRUNC);

    FileSource xmlout1(FileSource::OUT, "xmloutput1.xml",
                       FileSource::CREAT|FileSource::TRUNC);
    FileSource xmlout2(FileSource::OUT, "xmloutput2.xml",
                       FileSource::CREAT|FileSource::TRUNC);
```

```
// Create filters

// Remove all comments begining with '#'
StripCommentTransform linesct, xmlsct;

// Remove all whitespace from begining and end of line;
// lines only containing white space are eliminated from pipeline
StripWhitespaceTransform lineswt, xmlswt;

// Reject lines containing invalid values
ContentFilter linecflt("0123456789,.+-e"), xmlcflt("0123456789,.+-e");

// Reject lines without exactly 7 columns (6 commas)
CSVFilter linecsvflt(6), xmlcsvflt(6);

// Create filter group for concurrent filter processing
FilterGroup linefgrp(false), xmlfgrp(false);
linefgrp.addFilter(&linecflt);
linefgrp.addFilter(&linecsvflt);
xmlfgrp.addFilter(&xmlcflt);
xmlfgrp.addFilter(&xmlcsvflt);

// Append '\n' to the line before writing to a file
AppendLineFeedTransform alft;

// Translate CSV to XML
CSV2XMLTranslator xmlt;

// Write the reults to one or more files
ForwardTerminator lineterm(true);
ForwardTerminator xmlterm(true);

// Create pipelines
Pipeline linepipeline;
Pipeline xmlpipeline;

// Add filters to text-line based pipeline and the xml pipeline
linepipeline.addFunction(&linesct);
linepipeline.addFunction(&lineswt);
linepipeline.addFunction(&linefgrp);
linepipeline.addFunction(&alft);

xmlpipeline.addFunction(&xmlsct);
xmlpipeline.addFunction(&xmlswt);
xmlpipeline.addFunction(&xmlfgrp);
xmlpipeline.addFunction(&xmlt);

// Add pipeline terminator to forward to the output sources
lineterm.addSource(&lineout1);
lineterm.addSource(&lineout2);
linepipeline.setTerminator(&lineterm);

xmlterm.addSource(&xmlout1);
xmlterm.addSource(&xmlout2);
xmlpipeline.setTerminator(&xmlterm);
```

```
    // Add pipeline to delegate
    Pipelines pipelines;
    pipelines.push_back(&linepipeline);
    pipelines.push_back(&xmlpipeline);
    PipelinesDelegate delegate(&pipelines);

    // Create a decomposing delegate decorator to decompose groups of lines
    // prior to submission to the pipeline for processing
    TextLinesDecomposer tld;
    DecomposingDelegateAdapter dda(&delegate, &tld);

    // Add the pipeline delegate to the executor
    PipelineExecutor executor;
    dispatch.addSourceDelegate(&dda);

    // Set the classifiers for the sources
    TextLinesClassifier tlsc;
    csvin.setClassifier(&tlsc);

    TextLineClassifier tlc;
    lineout1.setClassifier(&tlc);
    lineout2.setClassifier(&tlc);

    BinaryAdapterClassifier bac;
    xmlout1.setClassifier(&bac);
    xmlout2.setClassifier(&bac);

    // Add the source and its delegate to the monitor
    Monitor monitor;
    monitor.addSourceMonitor(&csvin, Monitor::READ, &dda);
    monitor.addSourceMonitor(&lineout1, Monitor::WRITE, NULL);
    monitor.addSourceMonitor(&lineout2, Monitor::WRITE, NULL);
    monitor.addSourceMonitor(&xmlout1, Monitor::WRITE, NULL);
    monitor.addSourceMonitor(&xmlout2, Monitor::WRITE, NULL);

    // Start monitor and dispatcher threads
#ifdef WIN32
    HANDLE t1, t2;
    t1 = CreateThread(NULL, 0, run_monitor, (void*)&monitor, 0, NULL);
    t2 = CreateThread(NULL, 0, run_dispatcher, (void*)&dispatch, 0, NULL);
#else
    pthread_t t1, t2;
    pthread_create(&t1, NULL, run_monitor, (void*)&monitor);
    pthread_create(&t2, NULL, run_dispatcher, (void*)&dispatch);
#endif

    // Wait for eof
    while(csvin.getState() != Source::DONE)
    {
      usleep(100);
    }

    // Wait for pipeline data queue to clear
    while(linepipeline.getDataSize() > 0 || xmlpipeline.getDataSize() > 0)
    {
      usleep(100);
    }
```

```
    // Wait for write queues to clear
    while(lineout1.getWriteQueueSize() > 0 ||
          lineout2.getWriteQueueSize() > 0 ||
          xmlout1.getWriteQueueSize() > 0 ||
          xmlout2.getWriteQueueSize() > 0)
    {
      usleep(100);
    }

    monitor.stop();
    dispatch.stop();

#ifdef WIN32
    WaitForSingleObject(t1, INFINITE);
    WaitForSingleObject(t2, INFINITE);
#else
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
#endif
  }
  catch(Exception e)
  {
    std::cout << e.getMessage() << std::endl;
  }

  std::cout << "File processed in "
            << (tbb::tick_count::now() - start).seconds()
            << " seconds"
            << std::endl;

  return 0;
}
```

The content of the input file to be processed should be similar to this sample data set:

```
# Sample CSV file containing position and velocity data sorted by time
# There are seven values per row which represent a single time entry
# TIME, X, Y, Z, VX, VY, VZ

0,15,15,15,0,0,0  # 0
NaN,NaN,NaN,NaN,NaN,NaN,NaN   # Invalid data
NaN,NaN,NaN,NaN,NaN,NaN,NaN   # Invalid data
NaN,NaN,NaN,NaN,NaN,NaN,NaN   # Invalid data
0.1,15,15,15,0,0,0      # 0.1
0.2,15,15,15,0,0,0      # 0.2
0.3,15,15,15,0,0,0      # 0.3
0.4,15,15,15,0,0,0      # 0.4
0.5,15,15,15,0,0,0      # 0.5
0.6,15,15,15,0,0,0      # 0.6
0.7,15,15,15,0,0,0      # 0.7
0.8,15,15,15,0,0,0      # 0.8
0.9,15,15,15,0,0,0      # 0.9
```