NGDA Ingest System

Architecture and Development Guide

March 2006

1 Introduction

1.1 Intended Audience

This is a technical document, aimed at developers. In it, the details of the ingest system will be covered in great detail, including such aspects as flow-of-control and API definitions. A developer seeking to maintain, extend, or expand the functionality of the ingest system should find helpful information within this document.

It is assumed that the reader is technically proficient, with a working knowledge of Java and class diagrams. Topics such as XML and database connections are discussed, but passing knowledge of these topics should be sufficient to get some use out of this document.

1.2 Definitions, Acronyms, and Abbreviations

ADL: The Alexandria Digital Library. A system for performing spatial searches on geospatially referenced data. The Alexandria Digital Library system is one method of access for information residing in the Archive.

Archive/The Archive: An installation of the software developed on the NGDA project.

Archive Object: The logical unit of storage in the Archive. An Archive Object is described by the NGDA Data Model. As a rule, an Archive Object consists of at least one component and a manifest. Any further requirements are left purely as policy decisions. As per the NGDA Data Model, Archive Objects are implemented as simple directory structures described by a manifest. Please see the NGDA Data Model for details.

Archive Object Component: The files that compose an Archive Object are referred to as components.

Archive Object Manifest: An XML document that details the content and structure of an Archive Object. Every Archive Object must have a self-descriptive manifest file at the object's root level. Please see the NGDA Data Model for more details.

Archive Object Template: An XML document that specifies a class of Archive Object. All Archive Objects adhering to a particular template will have identically named components and have the same internal structure.

Configuration File/Configuration XML/Ingest Configuration: An XML file created by the user of the ingest system. The configuration file contains all the necessary instructions to configure and run the ingest process.

Format Registry: An archival system for storing thorough, semantic definitions for file formats. The goal of the Format Registry is to capture as much information about a file format as possible and preserve it for the future. Sufficient information should allow for the creation of file-reading software long after existing readers have faded into obsolescence.

NDIIPP: National Digital Information Infrastructure Preservation Program. A Library of Congress funded initiative aimed at developing strategies for the long-term support and preservation of digital data.

NGDA: The National Geospatial Digital Archive. A project funded by the Library of Congress through an NDIIPP grant. Sometimes used in place of "the Archive", as defined above.

NGDA Data Model: A set of rules and specifications governing the structure of Archive Objects and interactions with the Archive. The NGDA Data Model defines precisely what comprises an Archive Object, and provides specifications for the template, manifest, and ingest files. Please visit the NGDA Data Model link provided in the "References" section of this document for more details.

1.3 References

1.3.1 Informational References

NDIIP website:
http://www.digitalpreservation.gov

NGDA website:
http://www.ngda.org

1.3.2 Technical References

Ingest Configuration Language Reference
(Fix link later)

NGDA Data Model:
http://www.alexandria.ucsb.edu/~gjanee/ngda/data-model/

2 Ingest as Part of NGDA

The ingest system is but a component of the overall NGDA architecture. In order to fully understand the real purpose and role of the system, it is necessary to take a step back and view it in light of the NGDA architecture as a whole.

2.1 NGDA in Brief

The National Geospatial Digital Archive was created as a grant project under the National Digital Information Infrastructure Preservation Program. The goal of the NDIIPP was to fund a number of cooperative investigations into the problems surrounding long-term data preservation.

The NGDA system is the result of work completed under the NDIIPP grant. Like all other NDIIPP projects, the NGDA system has a particular focus on data preservation. But additional constraints make the NGDA system unique. For one, the system is focused on the unique problems surrounding the storage and searching of geospatial data, thus meeting the “geospatial” promise of the project’s name. Secondly, the NGDA system places extra emphasis on “semantic” preservation. The system’s answer to the so-called “100-year problem”—that is, how do you read a particular file one hundred years after it was produced, given the inevitable change in technology—is the creation and archiving of a Format Registry.

While the Format Registry helps the contents of the archive resist obsolescence, the archive itself must also handle this implacable force. For this reason, the archive system is designed around a series of APIs and specifications. In this way, each and every component of the system can be discarded and re-implemented as time passes. Data can be migrated from storage system to storage system. All of these changes can be handled without violating the integrity of the archive, so long as the APIs and data models are adhered to. This document discusses is the current implementation of those APIs and specifications.

The system as discussed in this document is the culmination of the first 18 months of work under the NGDA grant. The architecture shown reflects the project’s early focus on a *local* archive system. The long-term goals for the archive involve interoperability with other NDIIPP projects as well as other installations of the archive. Future improvements to the architecture should reflect this fact.

2.2 NGDA System Architecture

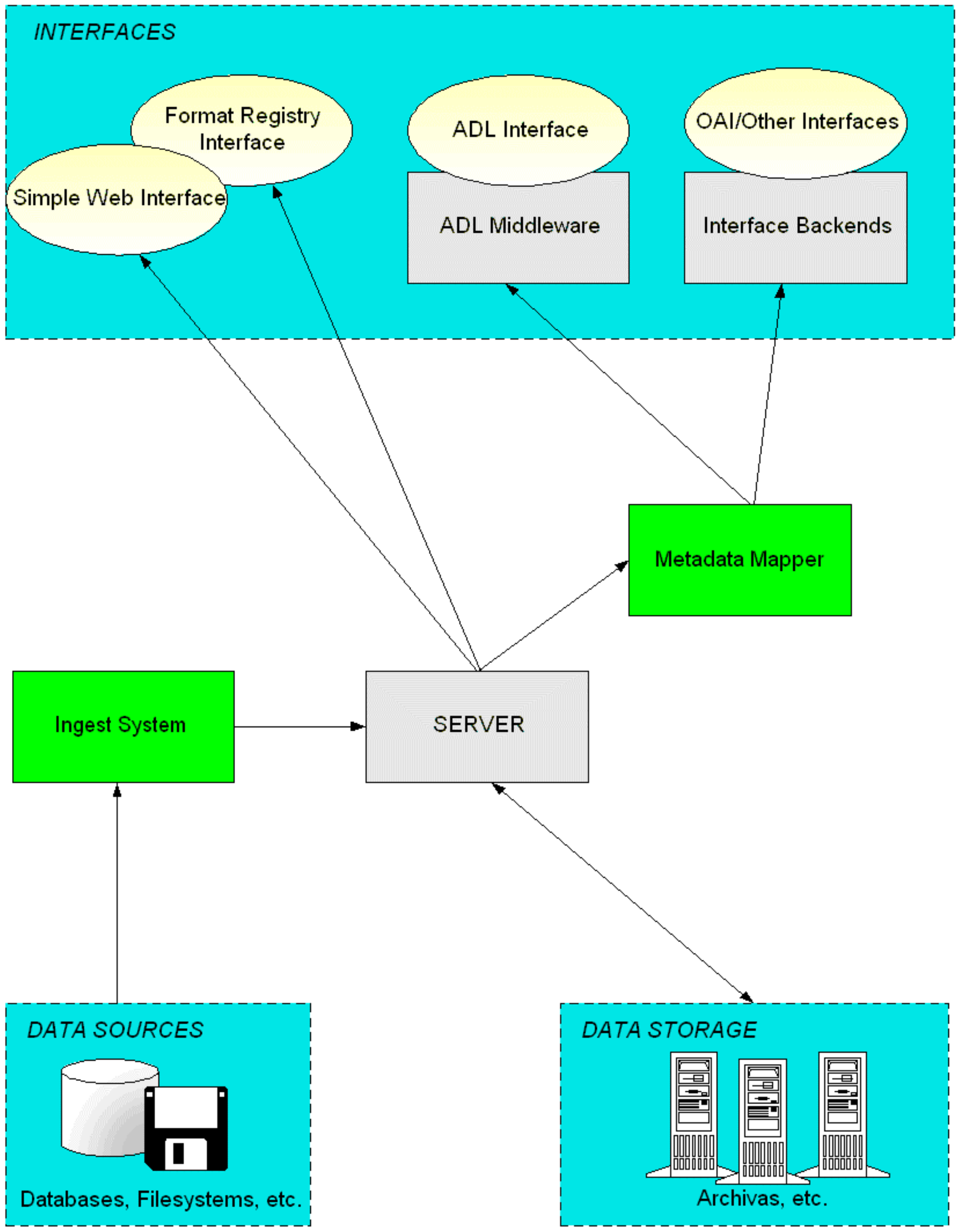
The NGDA system is a conglomerate, composed of several subsystems designed to perform specific tasks. One reason for this structure is the desire to resist obsolescence, as discussed above. Functionality is compartmentalized and interactions handled through strictly defined specifications, in order to allow components to be easily swapped out. As the structure of the system is discussed, please see the diagram (section 2.2.1) to get a visual representation of how the various components interact.

The central component of the NGDA system is the archive server. The server deals directly with the underlying storage system, allowing data to be inserted or removed as needed. The server ensures that data inserted into and removed from the archive adheres to the NGDA data model. It guarantees that Archive Objects adhere to the templates that they claim to follow. It brokers all data exchanges, and as such, is the one component that all parts of the archive must deal with.

A number of consumers derive their information from the server. Two of these, the “simple web interface” and the “Format Registry interface”, acquire information directly from the server. The simple web interface is an interface that allows a user to browse the archive as a file system. This is easily accomplished, as the NGDA Data

Model dictates that Archive Objects should be stored as simple (if strictly structured) directories. The Format Registry interface is another simple interface that allows the user to browse a specific (and important) subset of archived data, the archive's format registry.

2.2.1 Diagram: NGDA as a Whole



Another set of "more advanced" consumers exist as well. These interfaces exist to offer advanced functionality such as searching. Since both the NGDA Data Model and the archive server emphasize simplicity, neither natively support the sort of extended metadata that searching requires. This is where the metadata mapper comes into play.

The metadata mapper is a system that trolls over items stored in the archive. It uses a set of modules that allow it to recognize certain types of metadata within Archive Objects. When the mapper finds an object with a supported type of metadata, it attempts to map that metadata to a predefined set of fields required by a search interface. The exact details of how this mapping is carried out are dictated by another set of modules, which handle all interactions with the middleware behind that interface. In essence, the metadata mapper ingests data from the archive into specialized access services.

The additional access interfaces supported by the metadata mapper and illustrated on the diagram are an important part of the NGDA system. These systems perform indexing and allow for queries to be made against the holdings within the Archive. While these interfaces store their own information on the collections within the Archive, they are merely catalogues. Requests for objects must still be made to the archive server.

Ultimately, all of these interfaces are meaningless if there is no data for them to access. This is where the ingest system comes into play. The ingest system takes information from a data source, processes and packages it, and adds items into the Archive. The remainder of this document deals in detail with the specifics of the ingest system, a piece of software written for the bulk loading of data in to the Archive. It should be noted that the ingest system in this document is not intended to be the end-all be-all ingest system for the Archive—any program adhering to the proper API can ingest to the Archive, and indeed sometimes special applications warrant a more fine-tuned approach—but is simply a solution to the Archive's ingest problems.

3 Requirements Specification

3.1 General Requirements

The ingest system was designed using the following as guiding principles:

- The ingest system must be *widely applicable.*
- The ingest system must be *extensible.*
- The ingest system should be tuned toward *bulk loading*- it should load, easily, large numbers of mostly-homogenous Archive Objects.

3.2 Product Functions

The ingest system must provide the following functionality:

3.2.1 Data Gathering

The ingest system must be able to connect to and gather information from a data source. The information gathered must be in the form of discrete files, as this is the only type of data that the Archive has any conception of how to store. If ‘raw’ data sources are used, conversion routines must be specified to transform the data acquired into discrete files.

3.2.2 Compatibility with Archive Object templates

The ingest system must be compatible with the NGDA data model concept of object templates. The ingest system must support the assignment of data to elements defined within Archive Object templates.

3.2.3 Mapping data to Archive Object components

The ingest system must be able to map gathered files to the correct Archive Object component. The ingest system must be able to base such mappings on the identifier given to the file, the contents of the file, or both. The ingest system must solve this problem in a way that is compatible with “bulk loading”, meaning that large sets of files must be mapped as readily as small sets or single objects.

3.2.4 Identifier Mapping

The ingest system must be able to map gathered files to the correct Archive Object identifier. Since multiple Archive Object components are tied together only by the Archive Object identifier, this functionality is vital. Identifiers must be constructible from a combination of plain text and information gleaned from gathered files.

3.2.5 Output of completed data

The ingest system must output its finished data to an intermediary between itself and the Archive.

The data output to this location must include (at least) the following information:

- The location of the file to be loaded to the Archive.
- The identifier of the Archive Object that this file is a component of.
- The name of the component this file represents.

3.2.6 Use of configuration files

The ingest system must be operable through the creation of XML-based configuration files. These configuration files must allow the user to select and configure the implementations for the above-listed program functionality.

3.3 User Classes, Characteristics, and Environments

3.3.1 User Classes

There are two major potential classes of user for the ingest system: data providers, and the archive itself.

The data provider class of user is composed of any person working outside of the archive who is responsible for getting data into the archive. Content producers and accumulators (such as those operating state data clearinghouses) are the most likely candidates for this class.

The archive also makes up a class of user for this system, as circumstances may require that ingest of a collection be performed by the archive itself. It is assumed that this class will be the most common user in practice.

3.3.2 User Characteristics

In general, we can safely assume that users have few resources to devote to the act of archiving. We make this assumption because archiving as an act generates no immediate or obvious payoff, meaning that potential resources are often allocated elsewhere.

3.3.3 User Environments

It is anticipated that users will be operating the system in a wide variety of environments. Cross-platform support is therefore a derived requirement.

3.4 Assumptions and Dependencies

Assumptions:

- It is assumed that the user has a set of distinct files to load into the Archive; if instead the user has data that must be processed and separated into individual files, it is assumed that they have disk space on which to store generated files until they can be loaded to the Archive.
- It is assumed that the user has a high-speed internet connection through which their system can interact with databases.

- Adequate metadata exists for the files being loaded to the Archive. The definition of the term adequate in this context depends entirely on the policies of whatever individual archive the items are being ingested to.

Dependencies:

- The ingest system uses Java as a cross-platform solution. It is assumed that the user is operating on a platform for which Java is available.
- The ingest system will not load any data into the archive without the use of a separate data loading utility. Therefore the system is dependent upon the data loading utility.

3.5 System Use Cases

3.5.1 Primary Use Case

1. User has created and uploaded a collection template to the Archive.
2. User creates an ingest configuration file.
3. User initiates ingest system, directing it to use the configuration file produced in (2.).
4. Once ingest has completed, user runs separate programs to guarantee ingest success and load data into the Archive.

3.5.1.1 Alternate Paths

3a. If there is a problem with the user's configuration file, the user is notified with an error message. The user re-edits the configuration file and returns to step (3.).

4a. If the user is not satisfied with the results found in (4.), the user re-edits the configuration file and returns to step (3.).

4 Ingest System Architecture

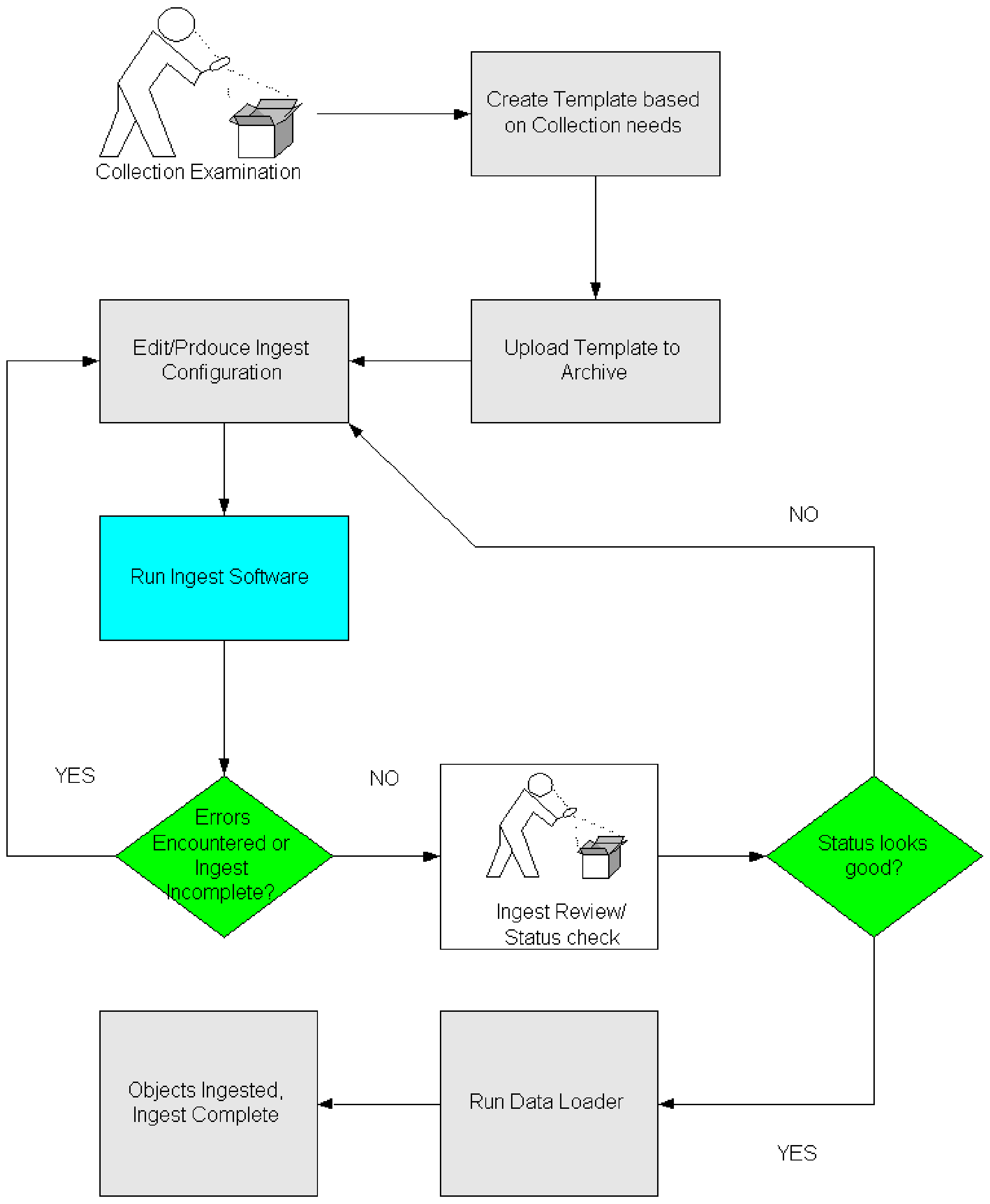
4.1 Ingest Workflow

As can be seen in the NGDA Architecture diagram (NGDA As a Whole, Section 2.2.1), the ingest system does not exist in a vacuum. It is a component of a larger, overarching system. In a similar manner, the software this document details is only part of a larger ingest workflow. To understand how the ingest system is structured and how it operates, it is important to understand the tools and events that surround its use.

The ingest process begins with data. At some point, a human being must examine an existing set of data to determine if it belongs in the Archive. During this process, the person will note details about the data, and form a preliminary idea of what should constitute an Archive Object. They may notice details about lineage and other inter-file relationships. The individual must also take stock of the existence of metadata, both at the object level and the collection level. Assuming that they are confident that they have a collection of achievable data, they move on to the next step in the process: creating a template.

Template creation amounts to the formalization of the work completed by the user thus far. The idea of the template is to create a specification that describes the items in the collection. Using the NGDA Data Model, the user specifies what components each Archive Object should possess. In collections where individual items may vary slightly, 'optional' components can be defined. For more information about templates, the NGDA Data Model should be consulted.

4.1.1 *Diagram: End to End Ingest Workflow*


Collection Examination
Create Template based on Collection needs
Upload Template to Archive
Edit/Prdouce Ingest Configuration
Run Ingest Software
YES
Errors Encountered or Ingest Incomplete?
NO
Ingest Review/ Status check
Status looks good?
NO
YES
Run Data Loader
Objects Ingested, Ingest Complete

After a template has been created, the user must upload the file to the Archive. Templates are treated in a special manner during upload, and so are ingested into the Archive in a different way than normal items. However, it should be noted that once they are placed within the Archive, templates are stored in the same manner as any other Archive Object. When a template is ingested into the Archive, it is checked against the Data Model for validity. Assuming that the template is accepted and ingests successfully into the Archive, the user can move on to creating an ingest configuration file.

In the earlier parts of the workflow, the user created a collection template in order to define the structure of the objects within the collection. In creating the ingest configuration file, the user specifies how that structure should be filled with actual data. This is a process that requires the user to define many different parameters, such as what data sources to use, how data should be mapped to components, etc. Entries in the 'components' segment of the configuration file correspond directly to entries in the collection template.

Once the user has completed the configuration file (or just partially completed it—the user needs only complete a subset of the components of the object as a whole), the user can invoke the ingest software. The software uses the configuration file as a set of instructions for determining how data-loading, component-mapping, and the writing of processed data should proceed. If any errors exist in the configuration file, an error message is printed and any processing aborts. The user must correct the issue and re-run the ingest program before proceeding. If the process runs without any errors being generated, the user can proceed. The ingest software outputs data into a database (unless directed to do otherwise), where it can be used in later steps.

The next step in the process is the status check. This allows the user to get a view of the Archive Objects that the ingest program has assembled. It can be useful to check the status of a partially-completed ingest; the user can ensure that several components are indeed mapping to the same Archive Object identifier. Users can also identify 'problem' data, such as outliers and other items that simply do not map to the correct identifier. The user can use this information to refine their mapping or to make corrections to the data itself. If the user has finished the configuration file and as such has complete Archive Objects, they can proceed to the final step in the process: data loading. If not, they can return to the configuration file armed with the knowledge that the status check has provided.

The final step in the ingest process is data loading, which is the physical process of copying data into the Archive. The data loading program takes the information generated by the ingest software and uses it to generate Ingest files, as per the NGDA data model. The Ingest files are submitted to the Archive, along with the files to be copied. (It is important to note that this process will fail if files have been moved or renamed between the operation of the ingest software and the data loading software!) The Archive reads the Ingest file as a set of instructions, and uses it to create a new Archive Object. This process repeats for all of the Archive Objects to be created. When the final object has loaded, the ingest process is complete.

4.2 Software Architecture

Now that we have described what role the software plays in the overall workflow, we will narrow our focus to the software that this document is supposed to describe.

Perhaps the easiest way to explain the architecture of the ingest software is by comparing it to a factory. The basis for this comparison comes from examining what both of these entities do. Both begin with a 'raw' material, run several processes and transformations, and output a regular, well-structured product.

If the bulk ingest software is a factory, than the IngestEngine class would be the manager’s control booth. The IngestEngine class is responsible for organizing all other components of the ingest system, and is responsible for directing flow-of-control in the program as a whole. As the configuration file is being loaded, items are added and registered to the system through the IngestEngine. The IngestEngine then oversees the rest of the process, initiating the loading of data from data sources and the processing of data in the rest of the system.

If the IngestEngine is the control booth of this factory, then the rest of the components of the system make up an assembly line. This assembly line begins with data sources. The system can take input from one or more data sources, which represent the raw materials to be used in the ingest process. These resources are marshaled into the system through a generic interface—the DataPool interface.

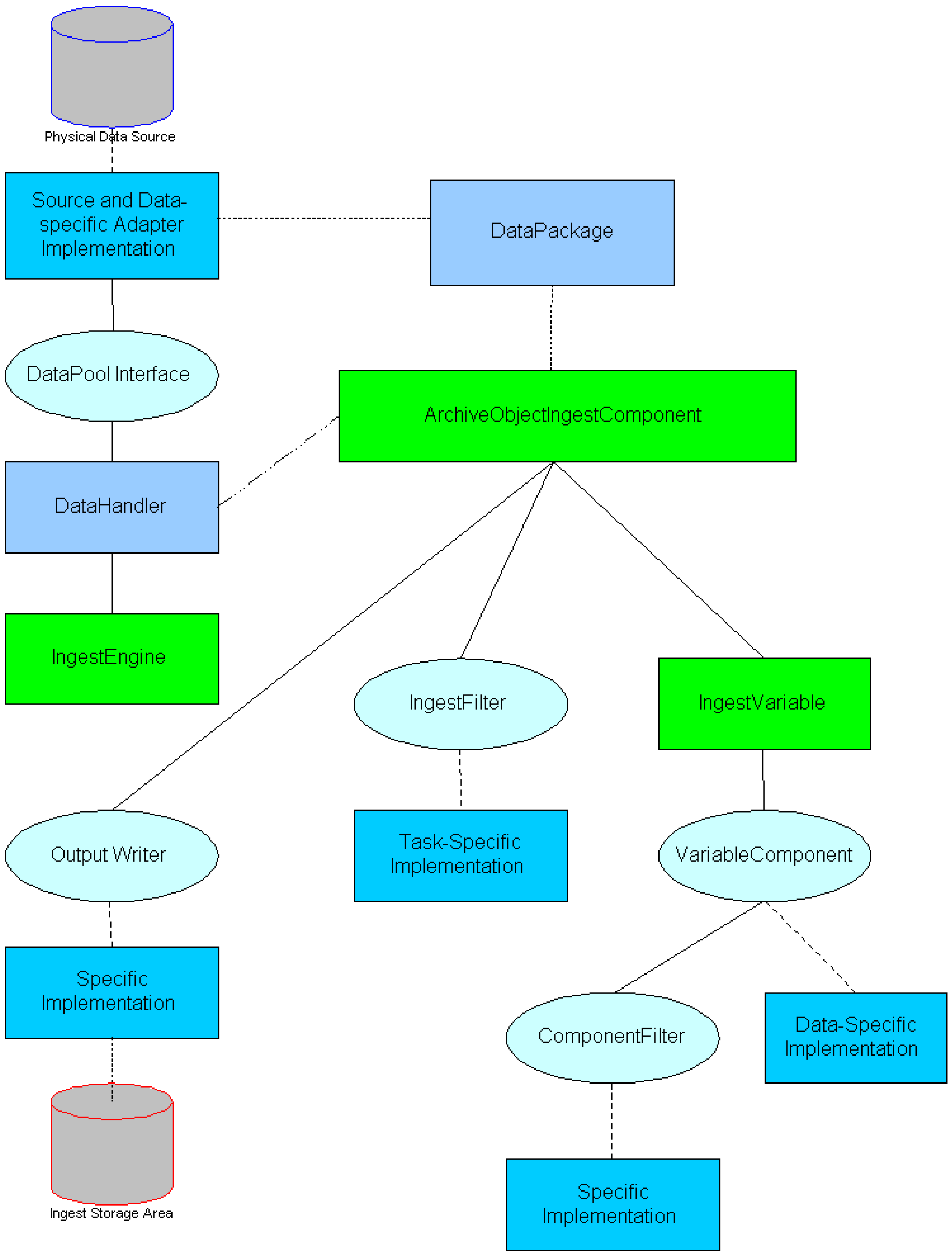
The DataPool interface allows the rest of the ingest software to deal with data sources in an abstract way. It operates by iterating through the set of items residing in the data source, allowing data items to be loaded into memory one at a time. The DataPool class is one of the modular classes within the bulk ingest software, meaning that different modules can be written to handle different data sources. To continue the factory analogy, the Data Pool is a generic box that travels through the rest of the process, allowing raw materials from any source to be handled in the same way.

Each DataPool is tied to a DataHandler. A DataHandler is a middleman between the DataPool and any interested ArchiveObjectIngestComponent. It takes the next data item from the DataPool (encapsulated by a DataPackage), and passes it to each ArchiveObjectIngestComponent that subscribes to it. The term ‘subscription’ is used to indicate that an ArchiveObjectIngestComponent might derive data from a particular DataPool. A DataHandler would be something like a programmed forklift in the bulk ingest factory, moving data from the DataPool and offering it to the various ArchiveObjectIngestComponents.

The ArchiveObjectIngestComponent represents its own mini assembly line. Each of these ‘assembly lines’ correspond to a component in a finished Archive Object. When data is passed to it from a DataHandler, it is evaluated by an IngestFilter. If the IngestFilter determines that the data being examined should be mapped to this particular component, the data is allowed to proceed down the assembly line. If not, the data is rejected, and the assembly line halts until appropriate data is found. It should be noted that the IngestFilter is another modular segment of the ingest software, so modules with varying criteria can be dropped in to determine if a particular file or piece of data should map to a component.

Once a piece of data has passed the IngestFilter, it can proceed through the rest of the processes tied to the ArchiveObjectIngestComponent. These are tied to the idea of

4.2.1 Diagram: Bulk Ingest Software Architecture



Key: Light blue ovals: interfaces/modules. Green boxes: Logic/flow of control. Grey-blue boxes: convenience abstractions.

identifier mapping—that is, tying a particular component to an Archive Object-level identifier. In the factory analogy, the next step is to process the contents of the 'data box' passed along and produce a label that dictates where it should be shipped. The accuracy of this label is vital, because if it is incorrect items will be 'shipped' incomplete or with incompatible components.

The IngestVariable is responsible for performing the identifier mapping for a piece of data. It combines a template string with information gleaned from the data at hand. In this way, a user can build up an identifier with a common prefix and combine it with a variable postfix to create a unique identifier. To return to the shipping label analogy, the IngestVariable begins with a label that says "Arizona %cityname% %Address%". The Arizona portion is constant, while the two parts enclosed in percentage marks are overwritten by information taken from the data at hand.

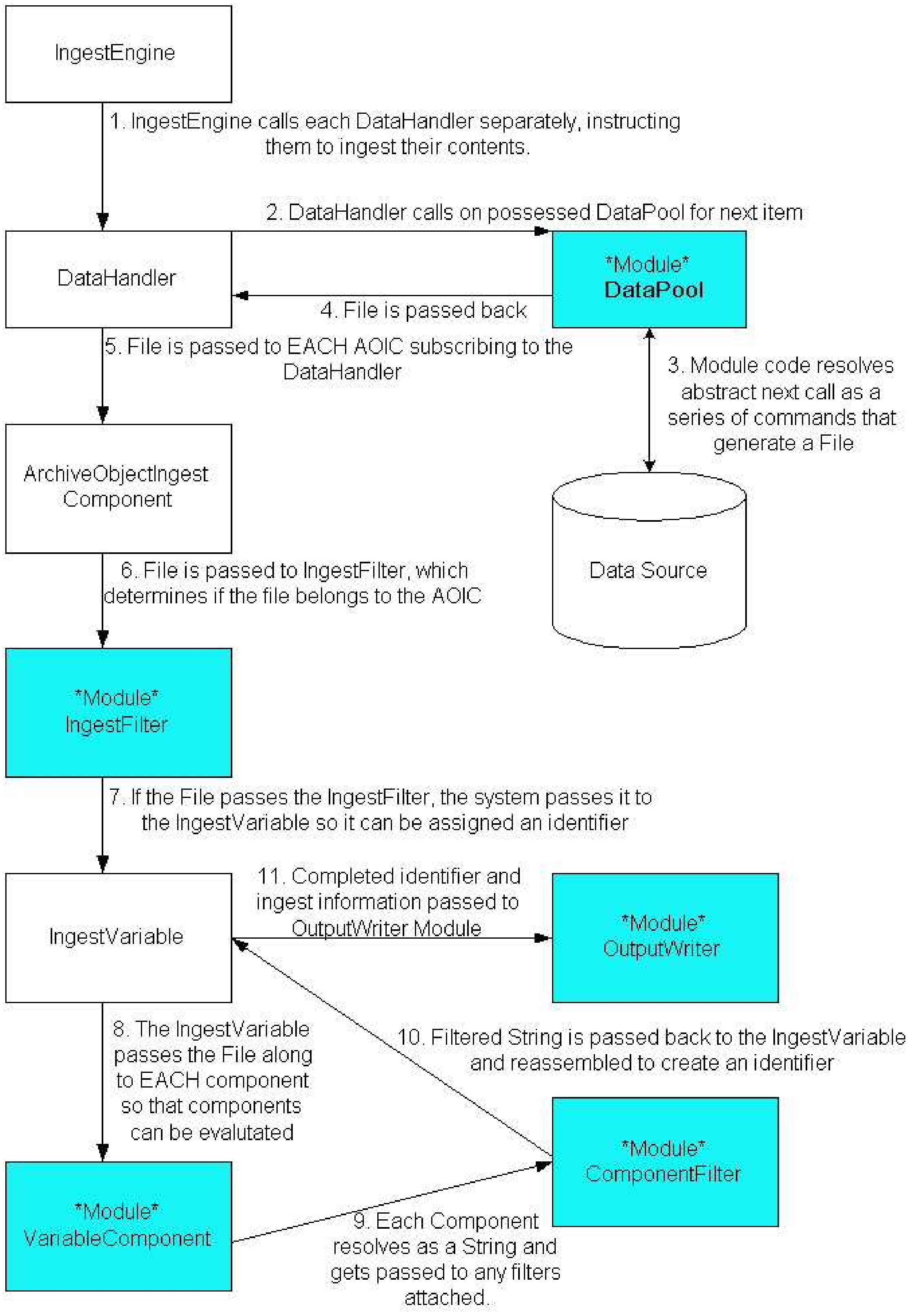
These identifier pieces are built up using VariableComponents. For each part of an identifier that needs to be substituted in, there is a VariableComponent that generates the needed data. Each VariableComponent might get its data from a different source. It could reach into the file header and pull out some piece of information. It could examine the identifier that came with the data, such as the filename. Or it could use some piece of information in the file to cross-reference it with a database. The details are unimportant, because the VariableComponent class is another modular piece of the bulk ingest software. The only requirement is that the information returned must be a string. The VariableComponent class, then, serves the purpose of a machine that generates pieces of labels for items moving through the assembly line. Before those labels can be attached, however, one more bit of processing can occur.

The ComponentFilter exists to refine the strings retrieved by the VariableComponent. This allows the VariableComponent to define a general method for creating data based off the file at hand, while specific refinements such as stripping off or reformatting data can be performed separately. In the factory, the ComponentFilter is a machine that reads and refines the label pieces generated by the VariableComponent and returns the cleaned up segments to the IngestVariable.

As the data is finally labeled, it is passed to its final stop on the assembly line: the OutputWriter. The OutputWriter is a modular class that allows the ingest process to output to any destination. To fit in to the Ingest System Workflow discussed above, however, the OuptutWriter outputs to a database. The values it writes include the identifier assigned to the data, the component that it represents, and the path to the data. The data itself remains unmoved and uncopied. The factory has merely indexed and organized it; another process must come and move it to the Archive. But nonetheless, we have reached the end of the ingest assembly line.

The next page contains a diagram detailing the process explained above. Places where modular code comes into play are clearly indicated.

4.2.2 Diagram: Bulk Ingest Software Flow-of-Control


IngestEngine
1. IngestEngine calls each DataHandler separately, instructing them to ingest their contents.
2. DataHandler calls on possessed DataPool for next item
DataHandler
Module DataPool
4. File is passed back
5. File is passed to EACH AOIC subscribing to the DataHandler
3. Module code resolves abstract next call as a series of commands that generate a File
ArchiveObjectIngest Component
Data Source
6. File is passed to IngestFilter, which determines if the file belongs to the AOIC
Module IngestFilter
7. If the File passes the IngestFilter, the system passes it to the IngestVariable so it can be assigned an identifier
11. Completed identifier and ingest information passed to OutputWriter Module
IngestVariable
Module OutputWriter
8. The IngestVariable passes the File along to EACH component so that components can be evalutated
10. Filtered String is passed back to the IngestVariable and reassembled to create an identifier
Module ComponentFilter
Module VariableComponent
9. Each Component resolves as a String and gets passed to any filters attached.

4.3 Data structures and classes

4.3.1 ArchiveObjectIngestComponent

An ArchiveObjectIngestComponent represents a component, or single file, of a completed ArchiveObject.

Each ArchiveObjectIngestComponent(AOIC) has a 'path' that dictates where in the ArchiveObject it should go. This means that it directly corresponds to an entry in a collection template.

The AOIC receives files, runs them through filters to determine if the files actually belong in the component slot that they represent. The AOIC then maps the item to the correct item identifier by processing the file's name or contents.

4.3.2 DatabaseInterface (Interface)

The DatabaseInterface is a helper interface that is not directly a part of the bulk ingest system architecture. It was created to allow a database to reside in memory and be used by multiple objects. It also has allowed generic modules to be written in other parts of the program- such as the OutputWriter derived DatabaseWriter.

Currently, MySQLDatabase is the only class implementing this interface. This may change as necessary.

4.3.2.1 API

public String getSingleValue(String tableName, String keyColumn,
String keyValue, String retrieveColumn) throws Exception;

String tableName- The name of the table to affect
String keyColumn- The name of the primary key's column.
String keyValue- The primary key for a database entry.
String retrieveColumn- The name of the column you are interested in retrieving data from.

Retrieve a single value from a database. This value corresponds to the entry in retrieveColumn in the row where the entry in keyColumn equals keyValue.

public void insert(String tableName, String[] values) throws Exception;

String tableName- The name of the table to affect.
String [] values- The values to insert into the database.

Insert a new row into the database. Implementations will probably need some initialization to configure the layout of the columns to be inserted to, as well as the name/location of the primary key.

public void update(String tableName, String keyColumn, String keyValue,
String [] values) throws Exception;

String tableName- The name of the table to affect
String keyColumn- The name of the primary key's column.
String keyValue- The primary key for a database entry.
String[] values- The values with which to update the row.

Update the entries in a row in the database. The new values to use reside in the array values. The row to update is selected by finding the entry where keyColumn equals keyValue.

public void initializeTable(String tableName) throws Exception;

String tableName- The name of the table to affect with this DatabaseInterface.

Initialize the DatabaseInterface. Create any needed connections any initialize any private variables.

4.3.3 DatabaseWriter

A helper class that implements the OutputWriter interface. The DatabaseWriter class is used to write the results of ingest into a database. The DatabaseWriter class makes use of the DatabaseInterface abstraction to relieve the user from worrying about things like database query or update syntax.

4.3.4 DataHandler

Class responsible for tying ArchiveObjectIngestComponents to the DataPools that supply them with ingest information. More specifically, this class allows multiple ArchiveObjectIngestComponents to use a single DataPool, for situations where multiple components of a single object might come out of the same filesystem.

This class is also used to specify storage options for Files produced by the retrieveContent method of DataPools. This allows for the local storage of content that requires real processing work for their production.

4.3.5 DataPackage

The DataPackage is a conveinence abstraction. Rather than pass around a full DataPool object, the lighter-weight DataPackage is used. The DataPackage carries two basic pieces of information: the local identifier for the file retrieved from the DataPool, and a link to the DataPool itself.

When an actual File needs to be generated, the DataPackage calls upon the real machinery of the fully-implemented DataPool object that spawned it. This “just-in-time” file generation is useful in situations where s file might safely be ignored based solely on its local identifier.

4.3.6 DataPool (Interface)

Generic interface for acquiring data from a data source and feeding it to the ingest process. The DataPool guarantees that the only type of item that the ingest process need to concern itself with is discrete files. Any other data must first be converted by the DataPool into a file if it is to be placed in the Archive.

4.3.6.1 API

DataPackage next();

As an iterator, retrieve the next DataPackage from this DataPool. The first call to this method should ‘set’ the iterator to the first DataPackage—no item should be skipped.

DataPackage retrieve(String identifier);

String identifier- The local identifier of the DataPackage to be retrieved.

Retrieve a DataPackage by its local identifier. In some cases, this method may need to be stubbed out (if, for example, data is being retrieved by a stream.) But at the moment I can’t guarantee that such a strategy wouldn’t break the rest of the functionality. Will need to check that later.

File retrieveContent(String identifier);

String identifier- The local identifier of the DataPackage/file of interest.

Retrieve the File indicated by identifier.

4.3.6.2 Development Notes

I should really look into forcing this interface to implement/derive from the Iterator class, since some of the limits/functionality are linked...

- next() MUST return null ONLY when the DataPool has iterated through all items that it stores.
- The first call to next() MUST not skip any items in the implementing DataPool. In other words, the first call to next() should point the iterator at the first item to be returned.
- retrieve() MUST NOT disrupt the iterator model used by next().
- When generating DataPackages for a content consumer, it is suggested that the 'asFileId' or 'asFile' parameter used in creating the DataPackages be a unique identifier for the file. While this isn't vital- the DataHandler will detect and take simple corrective measures for duplicates- it will make the ingest operations faster and make the locally stored files neater.

4.3.7 FilePool

An implementation of the DataPool class that creates DataPackages from files in a file system. The local identifier assigned to each file is the full path of the file in the file system. Uses a stack-based recursion model for navigating the file system. The FilePool will recursively navigate subdirectories and produce any files present at or below the root directory provided at instantiation.

4.3.8 IngestEngine

The IngestEngine is the central class that ties together and drives the ingest process.

A number of DataHandlers are registered to the IngestEngine. Then a number of ArchiveObjectIngestComponents- one for each component that will eventually appear in the Archive Object- are registered to those DataHandlers through the IngestEngine.

Finally, the IngestEngine runs each DataHandler in series. These DataHandlers call upon the DataPool objects they encapsulate, iterating through the DataPackages that those DataPools produce. Each new DataPackage is dispatched to any ArchiveObjectIngestComponent registered to the DataHandler.

The files are then examined by each individual AOIC, and ingested or rejected saccordingly.

4.3.9 IngestFilter (Interface)

The IngestFilter is an interface, not a class. The role of the IngestFilter is to determine whether or not a file being processed by the ingest system should be mapped to a particular component. The interface is open-ended, allowing any sort of constraints upon the contents of a DataPackage to be implemented. The IngestFilter returns true if a file maps to the filter's associated component.

4.3.9.1 API

```
public boolean approve(DataPackage data);
```

DataPackage data- The data package to be tested by this IngestFilter.

This method should examine the DataPackage and determine if it indeed maps to the component associated with this filter. Any implementation should return true when data does correspond to the associated component, and false otherwise.

4.3.10 IngestVariable

This class is meant to allow the construction of values during the ingest process from the DataPackage being examined.

This is used to create the identifiers that map all components to a single Archive Object. It can also be used to gather information from inside files during ingest, or from the context surrounding those files, so that potentially important information can be preserved.

When creating Archive Object identifiers, it is important to remember that different components depend on this mapping to form a complete object. If components of what *should* be a single object map to even slightly different identifiers, the system will recognize them as different objects entirely. Without a correct mapping, the components effectively form separate objects.

4.3.11 MySQLDatabase

The MySQLDatabase class is an implementation of the DatabaseInterface interface. It allows database operations to be performed on MySQL-compliant databases. On instantiation, it creates and holds a connection with a running database.

Through the use of the DatabaseWriter, (itself an implementation of the OutputWriter interface) data can be written to the database represented by the MySQLDatabase class. A DataPool could be implemented to read from it as well.

4.3.12 OutputWriter (Interface)

The OutputWriter is an interface, not a class. The OutputWriter exists to allow the system to output to different locations, be it databases or flat files. The current workflow for the ingest system as a whole depends on data being written to a database, but other workflows could be devised. It is for this reason that this interface exists.

4.3.12.1 API

The following methods must be implemented by any class adhering to the OuputWriter interface:

```
public boolean write(String key, String value, String varName);
```

String key- The Archive Object identifier assigned to an item.
String value- The location of the file to be ingested to the Archive.
String varName- The full path of the component this item belongs to.

The write method takes the data accumulated during the ingest process and writes it somewhere for later use in the ingest process.

```
public boolean write(String key, String value);
```

Technically, this method is deprecated.

4.3.13 PathComponent

A VariableComponent used for capturing information from the path of a file. Can be used in any scenario where the DataPackage uses a path as an identifier. The PathComponent class implements the VariableComponent interface.

4.3.14 RegularExpressionFilter

The RegularExpressionFilter is an IngestFilter that tests the local identifier of a DataPackage against a regular expression to determine whether the file it represents should be mapped to a particular component. The user of the class may choose between two behaviors of the filter: approving the DataPackage when the local identifier matches the regular expression, or approving it when it does *not* match the regular expression.

4.3.15 RemoveLeadingTextFilter

The RemoveLeadingTextFilter class implements the VariableComponentFilter interface. It allows the user to strip off a prefix by selecting a marker. Either single characters or sub-strings can be used as markers. Anything before the marker will be discarded, while the rest of the string will be returned unchanged.

4.3.16 RemoveTrailingTextFilter

The RemoveTrailingTextFilter class implements the VariableComponentFilter interface. It allows the user to strip off an unwanted postfix by selecting a marker. Either single characters or sub-strings can be used as markers. Anything after the marker will be discarded, while the rest of the string will be returned unchanged.

4.3.17 VariableComponent

The VariableComponent class is stubbed class meant to be extended and fully implemented. A VariableComponent should exist for every variable part of an identifier as constructed by an IngestVariable. The VariableComponent represents a procedural method for generating portions of an identifier.

4.3.17.1 API

The following methods must be implemented by any class extending the VariableComponent class:

```
public String evaluate(DataPackage data)
```

DataPackage data- The DataPackage you are attempting to generate an identifier for.

In classes that exend this one, this method should run some operation on data, and return a String for substitution into an IngestVariable. Since the only argument to this method is the DataPackage, other methods should handle set-up and configuration of the component.

4.3.18 VariableComponentFilter (Interface)

The VariableComponentFilter is an interface, not a class. VariableComponentFilters allow a user to refine the Strings returned by a VariableComponent. This additional filtering is necessary when the results returned by a generally-implemented VariableComponent need additional processing.

4.3.18.1 API

```
String filter(String input);
```

String input- The string returned by a VariableComponent during identifier creation.

Process the String input. Return an altered version as a String. The specifics of the alteration are of course dependent on the implementing class.

4.3.19 XMLLoader

The XMLLoader loads ingest configuration files and creates the data structures necessary to run the ingest process. It effectively translates the ingest configuration language into instructions on how to run the bulk ingest software.

The XMLLoader is special because it allows the user to specify the use of different code modules.

4.4 UML Diagrams

Please see the following pages for a UML diagram of the architecture as discussed above.