JHOVE2: Next-Generation Architecture for Format-Aware Characterization

JHOVE2 Architecture

Version: 2.0.0
Issued: March 24, 2011
Status: Final

JHOVE2 Project Team

California Digital Library

Stephen Abrams
Patricia Cruse
John Kunze
Isaac Rabinovitch
Marisa Strong
Perry Willett

Portico

John Meyer
Sheila Morrissey

Stanford University

Richard Anderson
Tom Cramer
Hannah Frost

Library of Congress

Martha Anderson
Justin Littman

With help from

Walter Henry
Nancy Hoebelheinrich
Keith Johnson
Evan Owens

Preface

JHOVE2 is a Java framework and application for next-generation format-aware characterization. Characterization is the process of examining a formatted digital source unit and automatically extracting or deriving representation information about that source unit that is indicative of its significant nature and useful for purposes of classification, analysis, and use. For more information, visit http://jhove2.org.

Note: JHOVE2 uses the term "source unit" to refer to the unit of characterization. A source unit can be an entire file, a subset of a file, or a set of files that collectively form a single coherent work.

Note: To avoid repetition, names of files are generally specified with POSIX file separators (**/**).

Note: Various code examples are used to illustrate significant aspects of JHOVE2. In most cases, these are edited snippets of the actual source code, with irrelevant details removed. In these examples *italics* are used to indicate generic values.

Note: In the PDF version of this document, all references to sections, figures, tables and page numbers are clickable links.

Acknowledgments

The JHOVE2 project is funded by the Library of Congress as part of its National Digital Information Infrastructure Preservation Program (NDIIPP).

Version History

Version	Date	Notes
2.0.0	March 23, 2011	First public release of document.

Contents

Introduction

JHOVE2 is a Java framework and application for next-generation format-aware characterization of digital objects. Characterization is the process of deriving *representation information* about a formatted digital object that is indicative of its significant nature and useful for purposes of classification, analysis, and use. JHOVE2 supports four specific aspects of characterization:

- *Identification*. The process of determining the *presumptive* format of a digital object on the basis of suggestive extrinsic hints and intrinsic *signatures*, both internal (for example. magic numbers) and external (for example. file extensions).
- *Validation*. The process of determining the level of *conformance* to the normative syntactic and semantic rules defined by the authoritative specification of the object's format.
- *Feature extraction*. The process of reporting the *intrinsic* properties of a digital object significant for purposes of classification, analysis, and use.
- *Assessment*. The process of determining the level of *acceptability* of a digital object for a specific purpose on the basis of locally-defined policy rules.

The object of JHOVE2 characterization, known as a *source unit*, can be a file, an arbitrary subset of a file, or an aggregation of an arbitrary number of files that collectively represent a single coherent digital object. JHOVE2 can automatically process objects that are arbitrarily nested in containers, such as file system directories or Zip files.

Additional information about JHOVE2 can be found on the public project wiki at http://jhove2.org/. JHOVE2 is made freely available under the terms of the BSD open source license.

The JHOVE2 project is a collaborative undertaking of the California Digital Library (CDL), Portico, and Stanford University, with generous funding from the Library of Congress as part of its National Digital Information Infrastructure Preservation Program (NDIIPP).

Implementation

JHOVE2 is implemented in Java. It assumes a Java 1.6 J2SE runtime environment (JRE), or development environment (JDK) if recompilation is necessary.

The JHOVE2 implementation follows a common idiom of first defining desirable function in *interfaces*, then implementing those interfaces in utility *abstract classes*, and finally extending the abstract classes in one or more *concrete classes*. For example, the methods supporting various behaviors for source units are defined in the **`Source`** interface, which is implemented by the **`AbstractSource`** class, and then extended by the **`ByteStreamSource`** class, the **`ClumpSource`** class, etc.

```
package org.jhove2.core.source;
public interface AbstractSource;
public abstract class AbstractSource
       implements Source;
public class ByteStreamSource
       extends AbstractSource;
public class ClumpSource
       extends AbstractSource;
...
```

All JHOVE2 interfaces and classes are defined in a structured package hierarchy.

Table 1: JHOVE2 package hierarchy

org.jhove2	JHOVE2 parent package
org.jhove2.annotation	JHOVE2 annotations
org.jhove2.app	JHOVE2 applications
org.jhove2.app.util	JHOVE2 utility applications
org.jhove2.app.util.documenter	Documentation utilities
org.jhove2.app.util.documenter.displayer	Displayer configuration utility
org.jhove2.app.util.traverser	Reportable traverse utility
org.jhove2.config	JHOVE2 configuration classes
org.jhove2.config.spring	Spring configuration classes
org.jhove2.core	JHOVE2 core classes
org.jhove2.core.app	Application-specific core classes
org.jhove2.core.format	Format classes
org.jhove2.core.io	Input classes
org.jhove2.core.reportable	Reportable classes
org.jhove2.core.reportable.info	Reportable reflection classes
org.jhove2.core.source	Source unit classes
org.jhove2.module	JHOVE2 modules

org.jhove2.module.aggrefy	Aggrefier module
org.jhove2.module.assess	Assessment module
org.jhove2.module.digest	Digester module
org.jhove2.module.display	Displayer module
org.jhove2.module.format	Format modules
org.jhove2.module.format.*format*…	Individual format modules
org.jhove2.module.persist	Persistence modules
org.jhove2.module.persist.berkeleybdl	BerkeleyDB persistence manager
org.jhove2.module.persist.inmemory	In-memory persistence manager
org.jhove2.util	JHOVE2 utility classes
org.jhove2.util.externalprocess	External process management

Identifiers

Various JHOVE2 entities are associated with identifiers from a variety of namespaces. Identifiers are encapsulated in instances of the I8R class.

Note: The class name "I8R" is used for *identifiers*; the interface name "Identifier" is used to define the behavior of identification *modules*.

```
package org.jhove2.core;
public class I8R
       extends AbstractReportable
{
       public enum Namespace { AFNOR, AIIM, ANSI, ..., JHOVE2, ..., URI,
                                                          URL, URN, UTI, OTHER }
       public Namespace getNamespace();
       public String    getValue();
}
```

Identifiers in the JHOVE2 namespace follow the general form:

`http://jhove2.org/terms/`*`type`*`/`*`specific`*

where *type* indicates the type of entity being identified, and *specific* indentifies the specific entity. Four entity types are defined:

- **`reportable`**. An identifier for an entity encapsulating one or more reportable properties (see **Reportable Properties**). The specific portion of the identifier is based on the package-qualified name of the class embodying the reportable, with periods ("**`.`**") replaced by slashes ("**`/`**"), for example, **`http://jhove2.org/terms/reportable/org/jhove2/core/JHOVE2`**.
- **`property`**. An identifier for a reportable property (see **Reportable Properties**). The specific portion of the identifier is based on the package-qualified name of the class defining the reportable, with periods ("**`.`**") replaced by slashes ("**`/`**"), concatenated with the property name, for example, **`http://jhove2.org/terms/property/org/jhove2/core/JHOVE2/Commands`**.
- **`message`**. An identifier for a message (see **Messages**). The specific portion of the identifier is based on the package-qualified name of the class defining the message, with periods ("**`.`**") replaced by slashes ("**`/`**"), concatenated with the message name, for example,

http://jhove2.org/terms/message/org/jhove2/module/utf8/UTF8Module/ByteOrderMark.

- **format**. An identifier for a format. The specific portion of the identifier is based on the format's common name, for example, **http://jhove2.org/terms/format/utf-8**.

Reportable Properties

The characterization information that JHOVE2 provides is organized into one or more *reportable properties*. Reportable properties are themselves aggregated together into named containers known as *reportables*. A reportable is implemented by a Java class that implements the **`Reportable`** interface or extends the **`AbstractReportable`** class.

```
package org.jhove2.core.reportable;
public interface Reportable {
	public I8R    getReportableIdentifier();
	public String getReportableName();
}
public abstract class AbstractReportable
	implements Reportable;
```

The name and identifier of a reportable are determined by its defining class. The reportable name is the same as the simple class name; the identifier is a URL based on the package-qualified class name. For example, the properties of a message digest are collected into the reportable defined by the **`Digest`** class. This, the reportable name is **`Digest`** and its identifier is **`http://jhove2.org/terms/reportable/org.jhove2.core.Digest`**.

```
package org.jhove2.core;
public class Digest
	extends AbstractReportable;
```

A reportable property itself has several important sub-properties:

- *Name*. The name of the property.
- *Type*. The data type of the property.
- *Value*. The value of the property.
- *Identifier*. The unique identifier of the property,
- *Order*. The display order of the property.
- *Description*. A summary description of the property.
- *Reference*. An optional reference to a relevant specification document.

Within the reportable class a reportable property is defined by annotating the accessor method that returns the property value.

```
import org.jhove2.annotation.ReportableProperty;
protected type name;

@ReportableProperty(order=order, value="description", ref="reference")
public type getName() {
	return this.name;
}
```

The property name is derived from the annotated accessor method name by removing the leading "get". The property type is defined by the return type of the method. The property value is generally stored in an instance field. The property identifier is a URL derived from the package-qualified method name. The order, which is specified relative to all other properties in the given reportable, the description, and, optionally, the reference are defined by the annotation parameters. For example, the Digest reportable defines two reportable properties.

```
package org.jhove2.core;
public class Digest
	extends AbstractReportable
{
	protected String algorithm;
	protected String value;

	@ReportableProperty(order=1, description="Message digest algorithm.")
	public String getAlgorithm() {
		return this.algorithm;
	}

	@ReportableProperty(order=2, description="Message digest value.")
	public String getValue() {
		return this.value;
	}
}
```

The names and identifiers for these two properties are **Algorithm** and **http://jhove2.org/terms/property/org/jhove2/core/Digest/Algorithm**; and **Value** and **http://jhove2.org/terms/property/org/jhove2/core/Digest/Value**. Whenever the **Digest** reportable is displayed, the **Algorithm** property will be displayed first and the **Value** property second.

The reliance on descriptive annotations to mark reportable properties permits much of the JHOVE2 technical documentation to be generated automatically using Java reflection.

The JHOVE2 output display is structured as a hierarchical set of reportable. The result of JHOVE2 processing is the appropriate creation and population of those reportables.

Source Units

To JHOVE2, *a source unit* is any entity that can be independently characterized. A source unit might be a single file, a byte stream within a file, or a collection of files. A source unit may encapsulate subsidiary source units. In such a case the encapsulating source unit is known as a *parent*, and the encapsulated source units are known as *children*. All children of a given parent source unit are automatically characterized by JHOVE2 as they are recognized during the processing of the parent. All of the source units processed by JHOVE2 during a single invocation form a hierarchical tree. The root of this tree is either a source unit representing the single file, directory, or URL specified on the command line, or a File Set encapsulating all of the files, directories, and URLs specified on the command line.

In cases where the child source units may be arbitrary and have no obvious relationship to the parent, the parent is known as an *aggregate* source unit. Otherwise the parent is *unitary*

JHOVE2 defines two explicitly aggregate source units:

- *Directory*. A file system or container file (e.g. ZIP file) directory.
- *File set*. The set of source units (files, directories, URLs) specified on the JHOVE2 command line.

JHOVE2 also defines one explicitly unitary source unit:

- *Byte stream*. A subset of a parent source unit.

Finally, there are two JHOVE2 source units that are initially assumed to be unitary, but which may be determined to be aggregate through further processing:

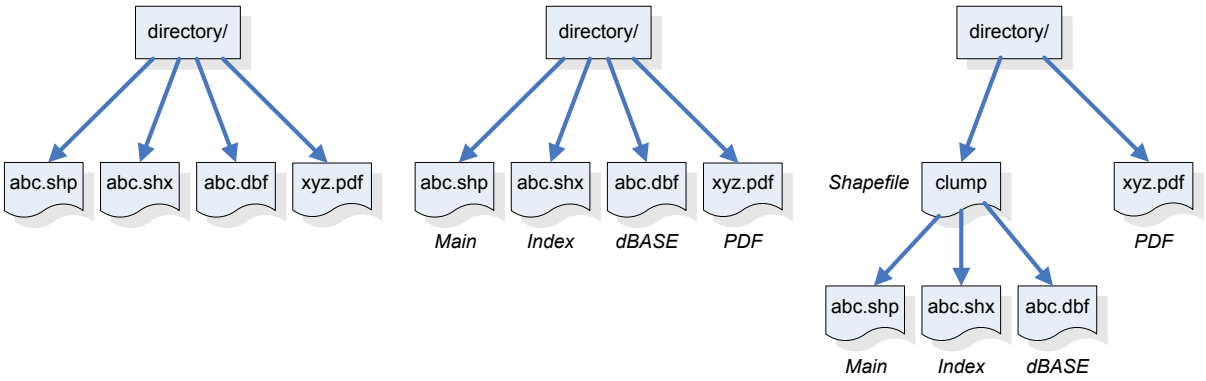
- *File*. A file system or container (e.g. ZIP file) file.
- *URL*. A URL.

For example, when a container file, such as a ZIP file, is characterized, a File source unit is created and is initially assumed to be unitary. However, once the internal signature of the ZIP file is recognized, JHOVE2 will continue to process the source unit as an aggregate.

Aggregate source units are subject to an additional processing step by JHOVE2, known as *aggrefication* (i.e. *aggre*gate identi*fication*), to determine if the aggregate itself constitutes a coherent characterizable entity. The source unit that is created as the result of a successful aggrefication is known as a *Clump*, and is automatically inserted into the source unit tree at the appropriate location (see Figure 1).

- Clump. An aggregate source unit that can be meaningfully characterized independent of its constituent parts.
-

Figure 1: Aggrefication of a Shapefile



Note: A Shapefile is an aggregate format composed of at least three files in the same directory sharing the same root file name (in this example, "**abc**") and having the extensions "**.shp**", "**.shx**", and "**.dbf**".

Most source units are associated with a backing file, either the file that was directly supplied on the JHOVE2 command line or a temporary file explicitly created to fulfill the need for a backing file. Clump, Directory, and File Set source units do *not* have backing files.

All source units share a set of common reportable properties:

- *Children*. The set of child source units.
- *Extra properties*. Additional properties specific to a particular type of source unit. For example, the entries in a ZIP file have CRC-32 and compressed file size properties.
- *File system properties*. The file system-specific properties of the source unit or its backing file.
- *Aggregate*. Whether or not the source unit is an aggregate.
- *Messages*. The set of error, warning, and informative messages.
- *Modules*. The set of JHOVE2 modules processing the source unit.
- *Presumptive formats*. The presumptive formats for the source unit.

```
package org.jhove2.core.source;
public interface Source
```

```
        extends ReportableProperty
{
        public void                       close();
        public List<Source>               getChildSources();
        public List<Reportable>           getExtraProperties();
        public File                       getFile();
        public FileSystemProperties       getFileSystemProperties();
        public Input                      getInput(JHOVE2 jhove2);
        public InputStream                getInputStream();
        public List<Message>              getMessages();
        public List<Module>               getModules();
        public Set<FormatIdentification> getPresumptiveFormats();
}
public asbtract AbstractSource
        implements Source
        extends AbstractReportable;
```

The **`getFile()`** method returns the source unit's backing file (or **`null`** if the source unit does not have a backing file, as is the case for Clump, Directory, and File Set source units).

The **`getInputStream()`** method is provided as a convenience for examining the contents of a source unit without using the standard JHOVE2 Input abstraction (see **Inputs**). However, in order to avoid resource leaks in the Java virtual machine, it is imperative that the stream *must* be closed when it is no longer needed. It is preferable if this takes place in the same code context in which the Input was originally created.

```
Stream stream = source.getInputStream();
try {  /* Process the stream. */
       ...;
}
finally {
       stream.close();
}
```

Messages

A source unit may be associated with various messages, which fall into three classes:

- *Errors*. A message about a terminal condition generally requiring some remedial action or reaction by the user
- *Warnings*. A message about a condition that may require some further action by the user.
- *Informative*. An informative message requiring no further action

Messages are further characterized by the *context* in which the underlying condition arose.

- *Process*. A message documenting an unanticipated condition arising from the process of characterization, for example, a file not found exception.

- *Object*. A message documenting an unexpected condition in the examined source unit, for example, a validation error.

```
package org.jhove2.core;
public class Message {
       public enum Context  { PROCESS, OBJECT };
       public enum Severity { ERROR, WARNING, INFO };
       public Context  getContext();
       public String   getLocalizedMessageText();
       public Severity getSeverity();
}
```

Message texts are *localized* using the standard Java localization mechanism, based on message templates defined in the "**`config/messages/jhove2_messages.properties`**" configuration file.

Other properties particular to certain source units are defined by additional interfaces. The **`MeasurableSource`** interface is implemented by source units that are based on underlying byte streams, and thus have starting and ending offsets, relative to their parent (if one exists) and a measurable size. The **`NamedSource`** interface is implemented by source units have a meaningful name.

```
package org.jhove2.core.source;
public interface MeasurableSource
       extends Source
{
       public long getStartingOffset();
       public long getEndingOffset();
       public long getSize();
}
public interface NamedSource
       extends Source
{
       public String getSourceName()
}
public class ByteStreamSource
       implements MeasurableSource
       extends AbstractSource;
public class DirectorySource
       implements NamedSource
       extends AbstractSource;
public class FileSource
       implements MeasurableSource, NamedSource
       extends AbstractSource;
public class URLSource
```

```
        implements MeasurableSource, NamedSource
        extends AbstractSource;
```

Note that Clump and File Set source units are neither measurable nor named.

Source units are creating using a source factory retrieved from the JHOVE2 framework object (see **JHOVE2 Framework**). A source unit can be created from a file system name (file or directory), a Java File (java.io.File), a Java URL (java.net.URL), or a Java InputStream (java.io.InputStream). A byte stream source unit can be created as a subset of its parent source.

```
package org.jhove2.core.source;
public interface SourceFactory {
      public Source getSource(JHOVE2 jhove2, String name);
      public Source getSource(JHOVE2 jhove2, String name, String ...names);
      public Source getSource(JHOVE2 jhove2, List<String> names);
      public Source getSource(JHOVE2 jhove2, File file);
      public Source getSource(JHOVE2 jhove2, URL url);
      public Source getSource(JHOVE2 jhove2, InputStream stream, String name,
                              Reportable extraProperties);
      public ByteStreamSource getByteStreamSource(JHOVE2 jhove2,
                              Source parent, long offset, long size,
                              String name);
      public ClumpSource      getClumpSource(JHOVE2 jhove2);
      public DirectorySource  getDirectorySource(JHOVE2 jhove2,
                              boolean isFileSystemDirectory);
      public FileSetSource    getFileSetSource(JHOVE2 jhove2);
}
```

The **`getClumpSource()`** and **`getFileSetSource()`** methods create empty source units. The **`isFileSystemDirectory`** argument of the **`getDirectorySource()`** method *must* be set to **`true`** if the source is a file system directory and **`false`** if the directory is encapsulated inside of a container file, such as a Zip file.

In order to prevent resource leaks in the Java virtual machine, it is imperative that a source *must* be closed when it is no longer needed. It is preferable if this takes place in the same code context in which the source was originally created.

```
JHOVE2 jhove2;
...
SourceFactory factory = jhove2.getSourceFactory();
Source source = factory.getSource();
try {  /* Process the source unit. */
       ...
}
finally {
       source.close();
```

```
    }
```

By default, the **`close()`** method automatically deletes the source unit's temporary backing file, if one was created. The **`–T`** command line option (or its equivalent, **`--save-temp-files`**) suppresses this deletion.

Inputs

A JHOVE2 *Input* is an abstraction introduced to support uniform access to source units regardless of the underlying data structure. Inputs rely on Java's buffered I/O feature (java.nio). There are three types of I/O buffers:

- *Direct*. Direct buffers bypass much of the processing of the Java virtual machine by communicating directly with the I/O subsystem of the underlying operating system.
- *Non-direct*. Non-direct buffers use the I/O subsystem of the Java virtual machine.
- *Memory mapped*. Memory mapped buffers may use the virtual memory subsystem of the underlying computational platform.

In general, non-direct buffers require the least amount of initialization but provide the slowest subsequent performance, while memory mapped buffers require the most amount of initialization but provide the fastest subsequent performance. However, memory mapped buffers *cannot* be used for source units larger than 1.6 GB. Direct buffers provide a good balance between initialization and performance.

Inputs provide methods for reading various units of data from a source unit.

```
package org.jhove2.core.io;
public inteface Input {
       public void   close();
       public char   readChar();
       public double readDouble();
       public float  readFloat();
       public byte   readSignedByte();
       public short  readSignedShort();
       public int    readSignedInt();
       public long   readSignedLong();
       public short  readUnsignedByte();
       public int    readUnsignedShort();
       public long   readUnsignedInt();
       public long   readUnsignedLong();
       public void   setByteOrder(ByteOrder order);
       public void   setPosition(long position);
}
public abstract class AbstractInput
       implements Input;
public class DirectInput
```

```
        extends AbstractInput;
public class NonDirectInput
        extends AbstractInput;
public class MappedInput
        extends AbstractInput;
```

An Input for accessing the contents of a source unit is retrieved directly from that source unit.

```
package org.jhove2.core.source;
public interface Source {
        ...
        public Input getInput(JHOVE2 jhove2);
        public Input getInput(JHOVE2 jhove2, ByteOrder order);
        ...
}
```

In order to prevent resource leaks in the Java virtual machine, it is imperative that an Input *must* be closed when it is no longer needed. It is preferable if this takes place in the same code context in which the Input was originally created.

```
Input input = source.getInput(jhove2);
try {  /* Characterize the source. */
        jhove2.characterize(source, input);
}
finally {
        input.close();
}
```

Architecture

The overall JHOVE2 architecture is defined in terms of three conceptual layers:

- The JHOVE2 *application*, responsible for managing the command line interface presented to the JHOVE2 user.
- The JHOVE2 *framework*, which coordinates all JHOVE2 processing.
- JHOVE2 *modules*, which encapsulate specific processing behaviors.

Note: Technically, both the JHOVE2 application and framework are implemented as modules, however, due to their central, but distinct, roles in JHOVE2 processing it is useful to consider them as conceptually independent architectural levels.

JHOVE2 Application

The JHOVE2 application is responsible for managing the interface presented to the JHOVE2 user.

JHOVE2 is distributed with a single command-line application that accepts a list of *names*, which can be any combination of files, directories, and URLs, and command *options*. The command line syntax is:

```
% jhove2 [-hikT] [-b size] [-B Direct|NonDirect|Mapped] [-d JSON|Text|XML]
         [-t temp] [-o file] name ...
```

Note: Square brackets `[` and `]` enclose optional elements; a vertical bar `|` separates alternative choices, and underlining indicates default values..

Command line options can be specified in a Unix-style short form or a functionally-equivalent GNU-style long form.

Table 2: JHOVE2 command line options

`-h`	(or `--help`)	Specifies that a help message is displayed.
`-i`	(`--show-identifiers`)	Specifies that the formal identifiers for each reportable property are shown.
`-k`	(`--calc-digests`)	Specifies that message digests are calculated.
`-T`	(`--save-temp-files`)	Specifies that temporary files are not deleted.
`-b` *size*	(`--buffer-size` *size*)	Specifies the I/O buffer size (defaults to 131072 bytes).

`-B` *type* (`--buffer-type` *type*)	Specifies the buffer type: `Direct` (default), `NonDirect`, or `Mapped`.
`-d` *format* (`--display` *format*)	Specifies the display form: `JSON`, `Text` (default), or `XML`.
`-t` *temp* (`--temp` *temp*)	Specifies the temporary directory (defaults to `java.io.tempdir`).
`-o` *file* (`--output` *file*)	Specifies the name of an output file (defaults to the standard output unit).
name	One or more file or directory names or URLs to be characterized.

Every reportable property known to JHOVE2 is uniquely identified by an identifier, of the form:

`http://jhove2.org/terms/property/`*`package-qualified-class`*`/`*`property-name`*

where *package-qualified-class* is the package-qualified name of the reportable in which the property is defined, and *property-name* is the property name based on the property's accessor method name. For example,

`http://jhove2.org/terms/property/org/jhove2/core/Digest/Algorithm`

These identifiers are always displayed in JHOVE2's XML output. By default, however, they are *not* displayed in the JSON or Text output. The **`-i`** command line option (or its equivalent, **`--show-identifiers`**) specifies that the identifiers should be displayed.

A larger buffer size will tend to increase overall I/O performance; however, the specified buffer size must be consistent with the maximum heap space available to the Java virtual machine. In general, Non-direct buffers require the least initial setup overhead but provide the lowest level of subsequent performance. Alternatively, Mapped buffers required the most amount of initial overhead but provide the highest level of performance. However, Mapped buffers *cannot* be used on source units larger than 1.6 GB. The default Direct buffers provide the best balance of initial overhead and subsequent performance.

Configuration of the JHOVE2 application is described in detail in the ***JHOVE2 User's Guide***.

The **`Application`** interface defines methods to return the invocation command line options and date/ timestamp, and the displayer module.

```
package org.jhove2.core.app;
public interface Application
       extends Module
{
```

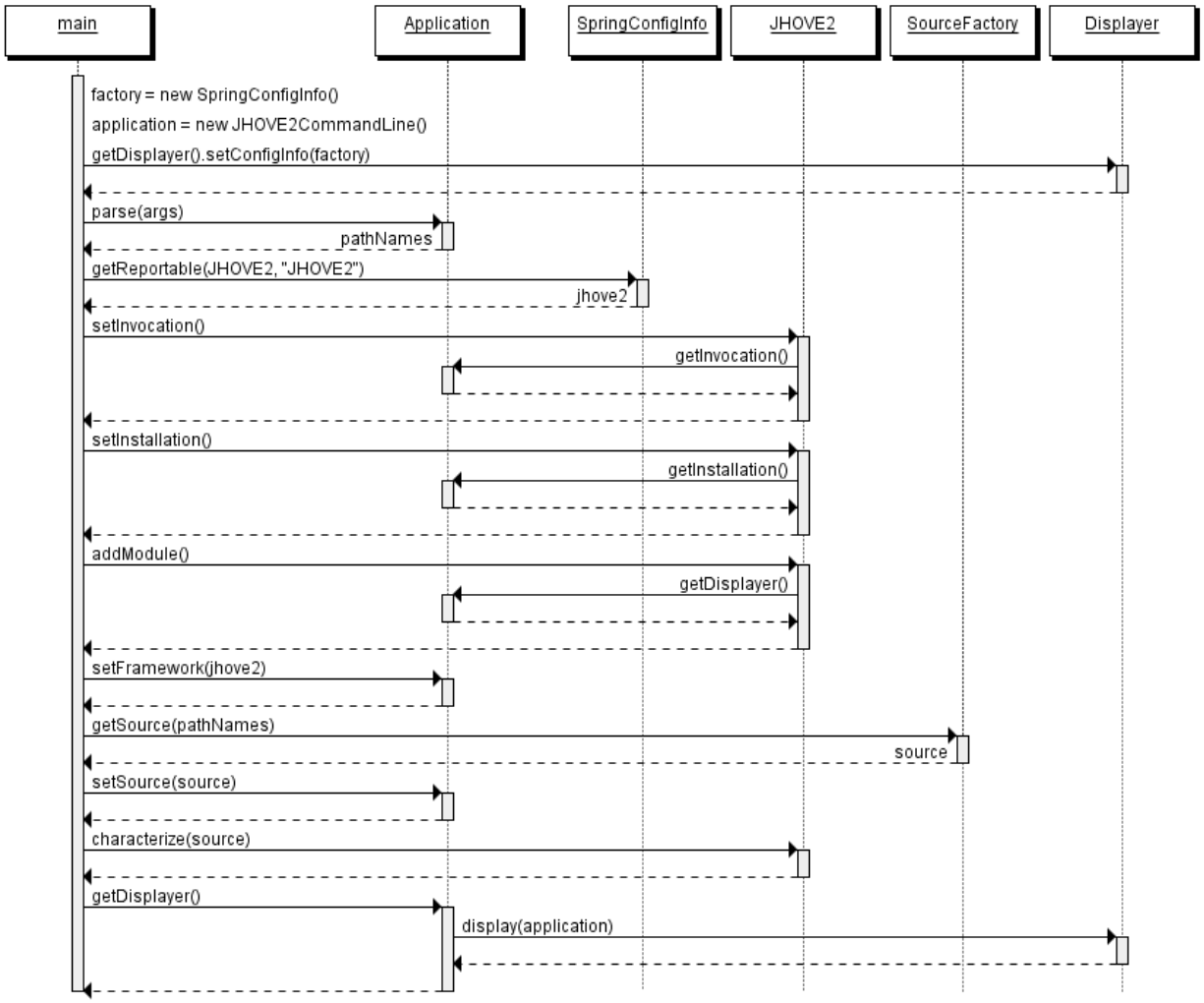
```
        public String    getCommandLine();
        public Date      getDateTime();
        public Displayer getDisplayer();
    }
    public abstract class AbstractApplication
        implements Application;
```

This interface is implemented by the JHOVE2 command line application. The essential processing of the JHOVE2 application (absent some important, though non-essential detail) is:

```
    package org.jhove2.app;
    public class JHOVE2CommandLine
        extends AbstractApplication
    {
        public static void  main (String [] args) {
            /* Instantiate the application and framework. */
            JHOVE2CommandLine app = ...;
            JHOVE2 jhove2 = ...;
            /* Parse the command line. */
            List<String> names = app.parse(args);
            /* Create the root source unit and Input. */
            Source source = factory.getSource(jhove2, names);
            Input  input  = source.getInput(source);
            try {  /* Characterize the source unit. */
                source = jhove2.characterize(source, input);
            }
            finally {
                input.close();
            }
            /* Display the source unit(s). */
            Displayer displayer = app.getDisplayer();
            displayer.display(source);
        }
        public List<String> parse(String [] args);
    }
```

The **`parse()`** method is passed the application command line arguments and sets all invocation options and returns a list of source unit names (files, directories, or URLs).

Figure 1: JHOVE2 application control flow



JHOVE2 Framework

The JHOVE2 framework coordinates all JHOVE2 processing. In short, the framework is presented with a list of *names* (any combination of files, directories, and URLs) that are dispatched to a sequence of *modules* for appropriate processing and display. Each of these names is considered an independent *source unit*; they may each contain an arbitrary number of subsidiary source units, and a set of related source units may itself constitute its own, aggregate source unit.

```
package org.jhove2.core;
public class JHOVE2
       extends AbstractModule
{
       public Source characterize(JHOVE2, Source source, Input input);
       public List<Command> getCommands();
       public Installation  getInstallation();
```

```
        public Invocation    getInvocation();
        public long          getMemoryUsage();
        public SourceCounter getSourceCounter();
        public SourceFactory getSourceFactory();
    }
```

The **`characterize()`** method characterizes a source unit. The default characterization processing includes the following stages, known collectively as a characterization *strategy*, each embodied by a JHOVE2 *module*:

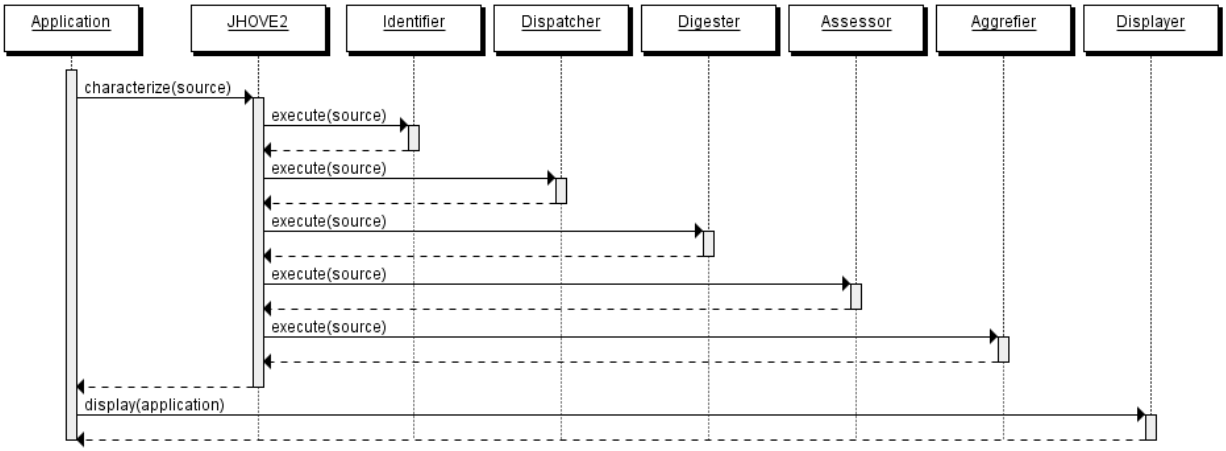
- *Identification*. Signature-based identification to determine the putative format of a source unit.
- *Dispatch*. Dispatch of a source unit, based on its putative format, to an appropriate format module for validation and feature extraction.
- *Digest*. Calculation of message digests. JHOVE2 can be configured to calculate any combination of the following digests:
    - Alder-32, CRC-32
    - MD2, MD5
    - SHA-1, SHA-256, SHA-384, SHA-512

    NOTE Digests are calculated only when the **`–k`** command line option (or its equivalent, **`--calc-digests`**) is specified. The set of digests that are calculated is specified in the **`config/spring/module/digest/jhove2-digest-config.xml`** configuration file.

- *Assessment*. Rules-based assessment of the source unit.
- *Aggrefication* (*Aggreg*ate Identi*fication*). If the source unit is an aggregate, identification of its aggregate format, separate from the formats of its children.

The framework **`characterize()`** method is automatically invoked by the JHOVE2 application to processes every source unit that is specified or detected.

Figure 2: JHOVE2 framework control flow



The **`getCommands()`** method returns the list of command modules that define a characterization strategy. The Installation reportable returned by the **`getInstallation()`** method encapsulates significant properties of the JHOVE2 application installation.

```
package org.jhove2.core;
public class Installation
       extends AbstractReportable
{
       public String getArchitecture();
       public String getClasspath();
       public String getClassVersion();
       public String getJREHome();
       public String getJREVendor();
       public String getJREVersion();
       public String getJVMName();
       public String getJVMVendor();
       public String getJVMVersion();
       public String getLibraryPath();
       public long   getMaxMemory();
       public int    getNumProcessors();
       public String getOSName();
       public String getOSVersion();
}
```

The Invocation reportable returned by the **`getInvocation()`** method encapsulates significant properties of the JHOVE2 application invocation.

```
package org.jhove2.core;
public class Invocation
       extends AbstractReportable
{
       public int     getBufferSize();
       public Type    getBufferType();
```

```
        public boolean getCalcDigests();
        public boolean getDeleteTempFilesOnClose();
        public String  getTempDirectory();
        public String  getTempPrefix();
        public String  getTempSuffix();
        public String  getUserName();
        public String  getWorkingDirectory();
}
```

The **`getMemoryUsage()`** method reports the total amount of heap memory used by the invocation, in octets. The **`SourceCounter`** reportable encapsulates the number of source units that have been processed.

```
package org.jhove2.core;
public class SourceCounter
        extends AbstractReportable
{
        public void incrementSourceCounter(Source source);
        public int  getNumByteStreamSources();
        public int  getNumClumpSources();
        public int  getNumDirectorySources();
        public int  getNumFileSetSources();
        public int  getNumFileSources();
        public int  getNumSources();
        public int  getNumURLSources();
}
```

The **`getSourceFactory()`** method returns a source factory that can be used to create source units.

JHOVE2 Modules

A JHOVE2 *module* is a reportable that encapsulates a specific processing function. Modules are characterized as being either *generic* or *specific*. A generic module's reportable properties are descriptive of the module itself, not the source unit being characterized; a specific module's reportable properties are specific to the given source being characterized.

```
package org.jhove2.module;
public interface Module {
        public enum Scope { Generic, Specific };
        public List<Agent>    getDevelopers();
        public String         getNote();
        public String         getReleaseDate();
        public String         getRightsStatement();
        public Scope          getScope();
        public TimerInfo      getTimeInfo();
        public String         getVersion();
```

```
        public WrappedProduct getWrappedProduct();
}
public abstract AbstractModule
        implements Module
        extends AbstractReportable;
```

The **`TimerInfo`** reportable encapsulates the elapsed time for the module's processing.

```
package org.jhove2.core;
public class TimerInfo
        extends AbstractReportable
{
        public Duration getElapsedTime();
}
public class Duration
        public long   getDuration();
        public String toString();
}
```

Elapsed time durations can be reported in milliseconds or as a String of the form: "*hh*:*mm*:*ss*.*msec*".

Some modules may be implemented using third-party software packages. These modules should implement the **`getWrappedProduct()`** method. The **`WrappedProduct`** reportable encapsulates properties that describe the third-party package.

The standard JHOVE2 distribution includes a number of modules that fall into the following categories:

- *JHOVE2 command line application* (see **JHOVE2 Application**).
- *JHOVE2 framework* (see **JHOVE2 Framework**).
- *Commands*.
- *Strategy modules*.
- *Format modules*.
- *Format profiles*.
- *Displayers*.

The JHOVE2 application and framework, and all commands and displayers are *generic modules*; all format modules and profiles are *specific* modules.

Command Modules

A JHOVE2 *command* module is a special module whose purpose is to execute another module.

```
package org.jhove2.module;
public interface Command
       extends Module
{
       public void execute(JHOVE2 jhove2, Source source, Input input);
}
public abstract class AbstractCommand
       implements Command
       extends AbstractModule;
```

All of the processing stages in the JHOVE2 characterization strategy invoked by the JHOVE2 framework – by default, *identification*/*dispatch*/*digest*/*assessment*/*aggrefication* – are represented by command modules.

```
public org.jhove2.module.identify;
public class IdentifierCommand
       extends AbstractCommand;

package org.jhove2.module.format;
public class DispatcherCommand
       extends AbstractCommand;

package org.jhove2.module.digest;
public class DigesterCommand
       extends AbstractCommand;

package org.jhove2.module.assess;
public class AssessorCommand
       extends AbstractCommand;

package org.jhove2.module.aggrefy;
public class AggrefierCommand
       extends AbstractCommand;
```

The **`DispatcherCommand`** is responsible for invoking the appropriate format command, and thus indirectly, the format module, based on the putative format identification associated with a source unit by the Identifier module.

Strategy Modules

A JHOVE *strategy* module is one that implements a specific stage of the characterization strategy that is invoked by the JHOVE2 framework for every source unit, by default, *identification*/*digest*/*assessment*/*aggrefication*. Note that there is no dispatch module, only a dispatch command.

Identifier Module

```
package org.jhove2.module.identify;
public interface Identifier
       extends Module
{
       public Set<FormatIdentification> identify(JHOVE2 jhove2, Source source,
                                                 Input input);
}
public class IdentifierModule
       implements Identifier
       extends AbstractModule;
```

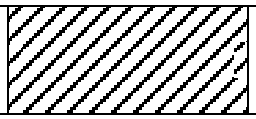
The **`FormatIdentification`** reportable returned by the Identifier module encapsulates a set of putative formats, and the level of confidence associate with each putative identification.

```
package org.jhove2.module.identify;
public class FormatIdentification
       extends AbstractReportable
{
       public enum Confidence { Negative, Tentative, Heuristic,
                                PositiveGeneric, PositiveSpecific,
                                Validated };
       public Confidence    getConfidence();
       public I8R           getIdentificationProduct();
       public I8R           getJHOVE2Identifier();
       public List<Message> getMessages();
       public I8R           getNativeIdentifier();
}
```

The supported confidence levels are the union of those defined by the DROID format identification tool (http://droid.sourceforge.net/) and the DSpace format identification framework (http://wiki.dspace.org/index.php/BitstreamFormat_Renovation).

Table 3: Format identification confidence levels

JHOVE2	*DROID*	*DSpace*	*Confidence level*
Negative	Negative	Unidentified	The source unit does match any format signature.
Tentative	Tentative	Circumstantial	The source unit matches an external, but not an internal, signature.
Heuristic		Heuristic	The source unit coarsely matches known characteristics of a generic format.
PositiveGeneric	Positive (generic)	Positive-generic	The source unit matches a generic format signature.
PositiveSpecific	Positive (specific)	Positive-specific	The source unit matches a specific format signature.

Validated		Validated	The source is valid with respect to all normative requirements of the reported format.

The **`getNativeIdentifier()`** method returns the format identifier native to the identification product (by default, DROID); the **`getJHOVE2Identifier()`** method returns the format identifier in the JHOVE2 namespace.

Digester Module

```
package org.jhove2.module.digester;
public interface Digester
       extends Module
{
       public void digest(JHOVE2 jhove2, Source source, Input input);
}
public class DigesterModule
       implements Digester
       extends AbstractModule;
```

Assessment Module

```
package org.jhove2.module.assess;
public interface Assessor
       extends Module
{
       public void assess(JHOVE2 jhove2, Source source, Input input);
}
public class AssessmentModule
       implements Assessor
       extends AbstractModule;
```

Aggrefier Module

```
package org.jhove2.module.aggrefy;
public interface Aggrefier
       extends Module
{
       public Set<ClumpSource> identify(JHOVE2 jhove2, Source source,
                                        Input input);
}
public class AggrefierModule
       implements Aggrefier
       extends AbstractModule;
```

Format Modules

A JHOVE2 *format* module is a special module that can parse and extract features from formatted source units. Optionally, a format module can *validate* the source units, that is, determined the level of their conformance to the normative syntactic and semantic rules established for the format. Parsing and validation behavior are defined by specific interfaces.

```
package org.jhove2.module.format;
public interface Parser {
	public long parse(JHOVE2 jhove2, Source source, Input input);
}
public interface Validator {
	public enum Coverage { Inclusive, Selective };
	public enum Validity { True, False, Undetermined };
	public Validity validate(JHOVE2 jhove2, Source source, Input input);
	public Coverage getCoverage();
	public Validity isValid();
}
```

The parse() method returns the number of octets consumed during the parse.

Validity is reported relative to a particular *coverage*.

- *Inclusive*. The module verifies *all* normative syntactic and semantic rules established for the format.
- *Selective*. The module verifies *some* of the format rules.

and in terms of three values:

- *True*. The source unit conforms to all format rules examined by the module.
- *False*. The source unit does not conform to at least one format rule examined by module.
- *Undetermined*. The conformance of the source unit cannot be determined.

The standard distribution of JHOVE2 includes modules for the following formats:

- *Directory.*
- *File set.*
- *ICC color profile.*
- *SGML text.*
- *Shapefile.*

- *TIFF image.*
- *UTF-8 text.*
- *WAVE audio.*
- *XML text*.
- *Zip container*.

Configuration

JHOVE2 offers extensive opportunities for local configuration. Configuration information is defined in several forms:

- Spring XML configuration files.
- Localized messages in Java properties files.
- JHOVE2 Java properties files.
- DROID XML configuration files.

Note: All configuration files *must* be found on the Java classpath at runtime.

Spring Configuration

Spring configuration files are found in the "**`config/spring`**" directory, whose subdirectory structure follows the hierarchical arrangement of the package structure.

```
config/
       spring/
              module/
                     aggrefy/
                              jhove2-aggrefy-config.xml
                     digest/  ...
                     display/ ...
                     format/
                              bytestream/
                                        jhove2-bytestream-config.xml
                              directory/ ...
                              fileset/   ...
                              icc/       ...
                              riff/      ...
                              sgml/      ...
                              shapefile/ ...
                              tiff/      ...
                              utf8/      ...
                              wave/      ...
                              xml/       ...
                              zip/       ...
                     identify/...
```

```
persist/
        jhove2-persist-config.xml
jhove2-config.xml
```

More information about the Spring framework is available at www.springsource.org.

Messages

All JHOVE2 messages can be internationalized to locales using standard Java Locale capabilities. Messages for all JHOVE2 components in the standard distribution package are defined in the "**`messages/jhove2_messages.properties`**" file. Messages for newly-developed components not yet in the standard distribution can be added in files of the form:

```
component_messages.properties
```

For example,

```
warcfile_messages.properties
```

Any such file must be placed on the Java classpath. All files on the classpath conforming to this naming convention will be detected automatically by the Spring configuration.

It is recommended that the key used to map from a message code to a message value follows the "dotted" Java path naming convention. For example:

```
org.jhove2.module.format.icc.ICCTagTable.MissingRequiredTag=Missing required
                                                              tag\: {0}
```

Note: To conform to the semantics of Java message files, all colons ("`:`") in the message text *must* be escaped with a backslash ("`\`").

More information about Java message internationalization and locales is available at http://java.sun.com/docs/books/tutorial/i18n/index.html.

JHOVE2 Properties

JHOVE2 uses Java properties files define specific attributes meaningful to various components. The three main types of properties are:

- Displayer properties.
- Units of measure properties.
- Unicode properties.

Displayer Properties

The display of individual reportable properties can be controlled by directives in displayer properties files, which are defined on a per-reportable basis. These files conform to the naming convention "*reportable*`_displayer.properties`". They are arranged in a subdirectory hierarchy that matches the JHOVE2 package structure, rooted at the "**`config/properties/module/display/displayer`**" directory.

The display directives take the form:

```
identifier directive
```

where *identifier* is the unique identifier for a reportable, collection, or property; and *directive* is a directive that is evaluated with respect to the property value to determine whether or not the property is displayed:

- Never (never display).
- IfFalse (display if *value* = false).
- IfTrue (display if *value* = true).
- IfZero (display if *value* = 0).
- IfNonZero (display if *value* ≠ 0).
- IfNegative (display if *value* < 0).
- IfPositive (display if *value* > 0).
- IfNonNegative (display if *value* ≥ 0).
- IfNonPositive (display if *value* ≥ 0).
- Always (always display).

For example:

```
http\://jhove2.org/terms/property/org/jhove2/core/source/Source/ChildSources
                                                                    Always
http\://jhove2.org/terms/property/org/jhove2/core/source/Source/NumChildSources
                                                                  IfNonZero
```

Note: To conform to the semantics of Java message files, all colons ("`:`") in the message text *must* be escaped with a backslash ("`\`").

The default display directive is "Always"; that is, all properties will be displayed if a more restrictive directive is not explicitly provided.

Displayer properties files for newly-created JHOVE2 modules should conform to the displayer properties file naming convention described above, and should be placed on the classpath. Developers creating new modules can use the **`org.jhove2.app.util.DisplayerPropertyFileGenerator`** utility to generate displayer properties files for any new reportable classes they create.

Units of Measure Properties

Units of measure can be associated with individual reportable properties in units of measure property files, which are defined on a per-reportable basis. These property files conform to the naming convention "*reportable*`_units.properties`". They files are arranged in a subdirectory hierarchy that matches the JHOVE2 package structure, rooted at the "**`config/properties/module/display/units`**" directory.

The unit of measure definition takes the form:

```
identifier unit
```

For example:

```
http\://jhove2.org/terms/property/org/jhove2/module/format/utf8/UTF8Character/Size byte
```

Note: To conform to the semantics of Java message files, all colons ("`:`") in the message text *must* be escaped with a backslash ("\").

Note: Setting a unit of measure for a property does *not* change the underlying property's value, it merely adds a descriptive label to the output display.

Unit of measure properties files for newly-created JHOVE2 modules should conform to the unit of measure properties file naming convention described above, and should be placed on the Java classpath. Developers creating new modules can use the **`org.jhove2.app.util.UnitsPropertyFileGenerator`** utility to generate displayer properties files for any new reportable classes they create.

Unicode properties

The Unicode properties define the C0 and C1 control characters and the code blocks.

```
resources/
          properties/
                    unicode/
                              c0control.properties
                              c1control.properties
                              codeblock.properties
```

The C0 and C1 control character sets are complete so these files should never need to be modified. The format of these files is "*control value*", where *control* is the two or three character control character mnemonic and *value* is the two digit hexadecimal code point.

```
NUL 00
SOH 01
...
DEL 7F

XXX 80
XXY 81
...
APC 9F
```

The code block definitions can be amended as additional code blocks are defined in the Unicode standard. The "**codeblock.properties**" code block file uses the same format as the "**Blocks.txt**" file in the Unicode database (UCD).

```
0000..007F;     Basic Latin
0080..00FF;     Latin-1 Supplement
...
100000..10FFFF; Supplementary Private Use Area-B
```

More information about the UCD is available at http://www.unicode.org/ucd.

DROID Configuration

The existence of the DROID configuration file, "**config/droid/DROID_config.xml**", is required by the DROID software, but most of the properties defined in the file are not used directly. There should be no need to modify the file supplied in the standard distribution package.

The DROID signature file, "**config/droid/DROID_Signature_V20.xml**" contains a list of format signatures from the PRONOM registry. JHOVE2 does *not* support the dynamic update of signature data from PRONOM. It is the responsibility of the JHOVE2 installer to ensure that an appropriate up-to-date copy of the signature file is available.

The absolute path names of the DROID configuration and signature files *must* be properly defined in the Spring Identifier module configuration file, "**config/spring/module/identify/jhove2-identify-config.xml**".

More information about DROID and PRONOM is available at http://sourceforge.net/projects/droid