

# Feature: Workflow Components

---

<b>Definitions .....</b>	<b>2</b>
<b>Workflows Overview.....</b>	<b>3</b>
Workflows .....	3
Components.....	3
<b>Components Overview .....</b>	<b>6</b>
Dependencies .....	6
Organization .....	7
Component Organization.....	7
Building and Running Components .....	7
Modifying and Rebuilding Components.....	9
Submitting Your Component.....	9
Automated Component Creation. ....	9
<b>The Component Schema.....</b>	<b>10</b>
Defining Inputs and Outputs.....	10
Defining Options .....	12
<b>Program Integration .....</b>	<b>17</b>
Bootstrap Program .....	17
Component Wrapper (Java).....	17
Generating Output .....	20
Error Handling.....	22
<b>Appendix A .....</b>	<b>23</b>
Existing File Types .....	23
Adding New File Types.....	24

---

## Definitions

**Workflow** – a sequence of processes chained together that can be shared or modified by the user

**Component** – a process or activity which, given a set of inputs and parameters, produces some desired output. A component is also a set of configuration files and executables which collectively make up the software process included in a workflow.

**Programs** – one or more programs to be executed by the component

**XML** – the markup language used by components to communicate with one another.

**XML Schema Definition** – the component definitions (inputs, outputs, options, and meta-data)

## Workflows Overview

### Workflows

A workflow is a component-based process model that can be used to analyze, manipulate, or visualize data. Each component acts as a standalone program with its own inputs, options, and outputs. The inputs to each component can be data or files, and the output of each component is made available after the workflow has been run. A generic workflow might consist of the following steps.

1. **import** a tab-delimited file
2. **analyze** the file
3. **visualize** the results of the analysis component

Not all components are interchangeable, as each one has a set of required inputs and options. For instance, some components will require a specific type of file, like tab-delimited or csv. Every component may have a set of inputs and user-defined options. The constraints on inputs and options are defined in each component's XML Schema Definition (XSD) which we will discuss in greater detail, later.

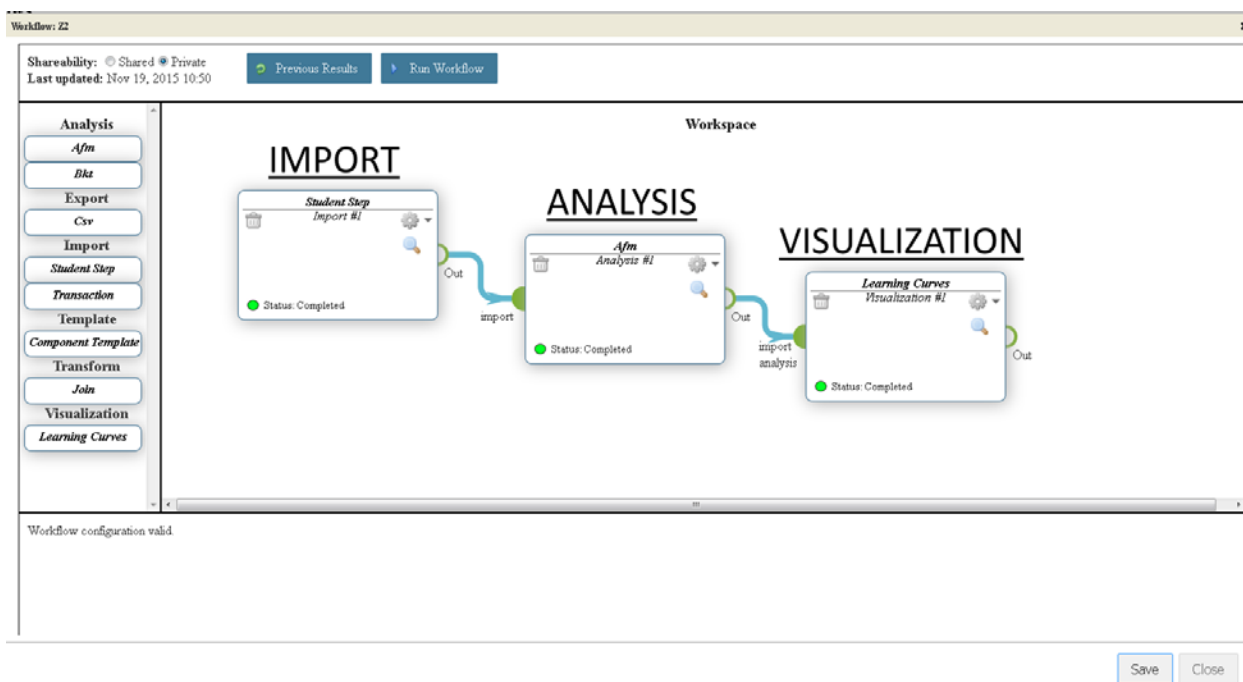


Figure 1 - Each component output in a workflow is a function of its input and user-defined options.

### Components

A component is essentially a black box which receives data from other components and carries out an arbitrary task which results in the generation of new data. The newly generated data is then passed to the component's successor. Some common component types could be *analyses*, *imports*, *transformations*, and *visualizations*. However, these types only exist for the sake of organization and to group processes with similar inputs and

outputs, but we are not limited to any particular hierarchy, as a component is merely a generic stand-in for a function. Additional component types could include *generative functions*, *sampling functions*, *transformations*, and a host of other applications. Regardless of the component type, all components match the same pattern, i.e. they all have a set of *inputs*, *options*, and *outputs*. Because of this pattern, the process for creating a component can be broken down into two parts—the **XML schema definition** and the **run-time program**.

### XML Schema Definition

The first part of a component is the XML Schema Definition (XSD) file. The XSD file defines the structure of a component.

#### Inputs

Inputs are files or data (or both) obtained from the direct predecessor of a component.

#### Options

Options are user-submitted files or data (or both) accessible from the user interface.

#### Outputs

Outputs are files or data (or both) generated by the component and passed to its direct successors.

### Basic Schema Definition

The skeleton XML schema for a workflow component describes its **inputs**, **options**, and **outputs**. These will be more thoroughly discussed in the section, The Component Schema. The sections there will describe the exact code you will need to create the desired effect. For now, simply glance over to familiarize yourself with the general concept.

The XML schema below defines the inputs, options, and outputs for the "Analysis - AFM" component. It also includes information about the component instance in the `<component>` element which is used by the software for identification, display purposes, and for passing messages between components.

```
<xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified"
xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:include schemaLocation="../../CommonSchemas/WorkflowsCommon.xsd" />

  <xs:complexType name="InputDefinition0">
    <xs:complexContent>
      <xs:extension base="InputContainer">
        <xs:sequence>
          <xs:element type="InFileList0" name="files" />
          <xs:any minOccurs="0" processContents="skip" maxOccurs="unbounded" />
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <xs:complexType name="InputType">
    <xs:all>
      <xs:element name="input0" type="InputDefinition0" minOccurs="0" />
    </xs:all>
  </xs:complexType>

  <xs:complexType name="InFileList0">
    <xs:choice>
```

```

    <xs:element ref="student-step" minOccurs="0" />
  </xs:choice>
</xs:complexType>

<xs:complexType name="OutputDefinition0">
  <xs:complexContent>
    <xs:extension base="InputContainer">
      <xs:sequence>
        <xs:element type="OutFileList0" name="files" />
        <xs:any minOccurs="0" processContents="skip" maxOccurs="unbounded" />
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="OutputType">
  <xs:sequence>
    <xs:element name="output0" type="OutputDefinition0" />
  </xs:sequence>
</xs:complexType>

<xs:complexType name="OutFileList0">
  <xs:choice>
    <xs:element ref="student-step" />
  </xs:choice>
</xs:complexType>

<xs:complexType name="OptionsType">
  <xs:all>
    <xs:element type="FileInputHeader" name="model" id="Model" default="KC\s*(.*)\s*" />
    <xs:element type="xs:integer" name="xValidationFolds" id="Cross-validation_Folds" default="10" />
  </xs:all>
</xs:complexType>

<xs:element name="component">
  <xs:complexType>
    <xs:all>
      <xs:element type="xs:integer" name="workflow_id" />
      <xs:element type="xs:string" name="component_id" />
      <xs:element type="xs:string" name="component_id_human" />
      <xs:element type="xs:string" name="component_name" />
      <xs:element type="xs:string" name="component_type" />
      <xs:element type="xs:double" name="left" />
      <xs:element type="xs:double" name="top" />
      <xs:element name="connections" minOccurs="0" maxOccurs="1" type="ConnectionType" />

      <xs:element name="inputs" type="InputType" minOccurs="0" />
      <xs:element name="outputs" type="OutputType" minOccurs="0" />
      <xs:element name="options" type="OptionsType" minOccurs="0" />

    </xs:all>
  </xs:complexType>
</xs:element>
</xs:schema>

```

### Run-time program

The second part of the component is an run-time program. A component is essentially a standalone process. It has no knowledge of other components in the workflow chain except through its input channels. It receives changes to its options by interfacing with the user. Once its inputs and options have been processed, it will execute and send its output to any direct successors.

To make developing components easier, a Java template and a common set of Java methods handle most of the work. New components can be created with just a few lines of code and a schema definition with little overhead. Components can run programs written in any language, i.e. C/C++, Java, Matlab, Perl, Python, Ruby, R, etc.

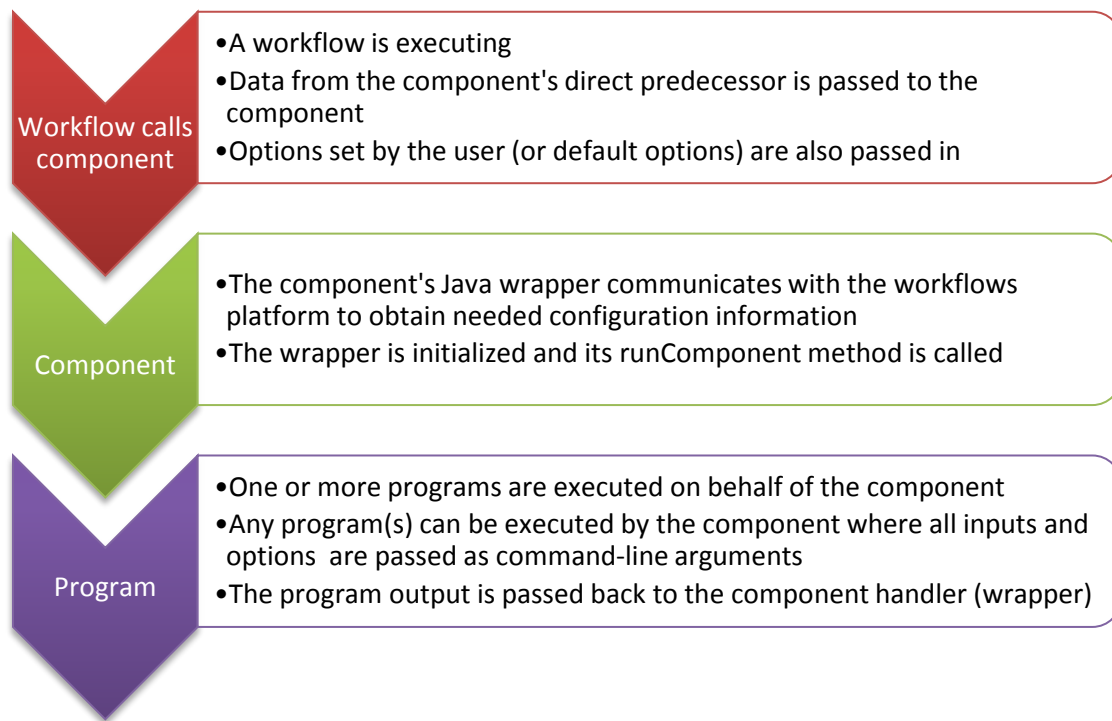


Figure 2 The component call stack.

### Language and Library Support

If you are unsure whether or not a language or library is supported or would like to request that one be added, then please inquire at [datashop-help@lists.andrew.cmu.edu](mailto:datashop-help@lists.andrew.cmu.edu).

## Components Overview

Checkout the WorkflowComponents repository with the following command.

```
git clone https://github.com/PSLCDDataShop/WorkflowComponents WorkflowComponents
```

### Dependencies

- Ant 1.9 or greater
- Java Enterprise Edition Software Development Kit (J2EE SDK)
- Eclipse, Cygwin, or a terminal / command-prompt

---

To access the components' source code, you will need to request access to the repository from [datashop-help@lists.andrew.cmu.edu](mailto:datashop-help@lists.andrew.cmu.edu).

## Organization

Programs used in components can be written in any language, but each component requires an **XML Schema Definition** (XSD) and a **Java wrapper**. The Java wrapper reduces the overhead of writing components by taking care of schema validation, pre- and post-conditions, argument passing, error handling, feedback, and program execution. It also facilitates communication between the workflows platform and components.

Each component is populated with an example in the "test" directory which allows it to be run as a standalone application, locally, if you have installed Apache Ant and the Java Development Kit. Most components will only require the Java EE SDK and a recent version of Ant. These can be acquired at <http://www.oracle.com/technetwork/java/javaee/downloads/index.html> and <http://ant.apache.org/bindownload.cgi>, respectively.

Any system with Java and Ant should be able to run the components. However, examples which use additional run-time programs, like AnalysisBkt, may require their programs be compiled on the system intended for use. Such instructions should be documented within the component, itself, in the README.md which exists in each component directory.

## Component Organization

### Directories

- "source" – contains the Java wrapper source code
- "dist" – contains the executable jar (standalone component)
- "program" – contains any run-time programs
- "schemas" – contains the XML Schema Definition (XSD) for this component
- "test" – contains example data used for running the components locally
- "unit\_test" – contains the unit test classes (to be *implemented in the next iteration*)

## Building and Running Components

Once you've obtained the "WorkflowComponents" project from SVN, you should now be able to run them as standalone applications with the test data. The preferred method for working with the projects is to create an Eclipse project. However, the simplest method is to simply use ant.

### Build and Run with Ant

1. Each component is in a separate directory and has its own **build.xml** file.
2. In your command-line prompt, change to **C:\dev\WorkflowComponents\AnalysisAfm**
3. Enter "ant -p" into the command-line. Doing so will present you with a list of available ant commands for the project.

```
c:\dev\WorkflowComponentsTrunk\AnalysisAfm> ant -p
```

```
Buildfile: c:\dev\WorkflowComponentsTrunk\AnalysisAfm\build.xml
```

---

Java-based AnalysisAfm workflow component.

**Main targets:**

clean - clean up  
compile - compile the Java wrapper  
dist - generate the jar from the Java wrapper  
javadoc - create javadoc documentation  
runComponent - run the component

Now, we can simply type "ant runComponent" in the same directory to execute the standalone component using the example data provided in the "test/" directory. The component outputs an XML entity which describes all of its output. The XML is generated automatically using built-in methods which will be covered later in this document.

```
c:\dev\WorkflowComponentsTrunk\AnalysisAfm> ant runComponent
```

**runComponent:**

```
[java] <output>
[java]   <component_id>Component1</component_id>
[java]   <component_id_human>Component #1</component_id_human>
[java]   <component_type>Analysis</component_type>
[java]   <component_name>AFM</component_name>
[java]   <elapsed_seconds>1</elapsed_seconds>
[java]   <model>Default</model>
[java]   <files>
[java]     <student-step>
[java]       <file_path>C:/dev/WorkflowComponents/AnalysisAfm/test/Component1/Step-values-with-
predictions-Component_1--081215-091902-286.txt</file_path>
[java]       <file_name>Step-values-with-predictions-Component_1--081215-091902-286.txt</file_name>
[java]       <file_type>student-step</file_type>
[java]       <label>Student Step</label>
[java]     </student-step>
[java]   </files>
[java] </output>
```

BUILD SUCCESSFUL

Total time: 1 second

The lines prefixed with [java] show the structured output of the component—most of the elements are used to describe the component except for model and the student-step file (which are specific to the AnalysisAfm component).

### ***Build and Run with Eclipse***

1. Go to File -> Import -> General -> "Existing Projects into Workspace"
2. Choose any component directory from WorkflowComponents/<AnyComponent>
3. Click 'Finish'
4. In the Ant view (Windows -> Show View -> Ant), add the desired component's build.xml to your current buildfiles, e.g. <AnyComponent>/build.xml
5. Double click the ant task "runToolTemplate". The component should produce example XML output if it is setup correctly.



- For debugging, you will want to add the jars in the directory WorkflowComponents/CommonLibraries to your build path.

## Modifying and Rebuilding Components

To rebuild a component after modifying the source, simply use the ant command "**ant dist**". Rebuilding is also done automatically when running the target: **ant runComponent**. The following sections detail the steps needed to create a new component that can be added to the workflows platform—i.e., creating the component **XML Schema Definition** (XSD) and **integrating your code**.

## Submitting Your Component

After reviewing and testing your code, it can be added to the Workflows platform and made available to all LearnSphere users. If you desire, we can also make your component available on one of our public or private GitHub repositories so that others may benefit from your work. Either send the entire component as a zipped file to us, or make it available via SVN, CVS, Git, or via the web.

## Automated Component Creation.

There are plans in place to automate the creation and submission of new components. Until then, defining the XSD and submitting your component must be done manually. Please do not hesitate to contact us for assistance. *To ask for help or to submit a new component for review, please contact [datashop-help@lists.andrew.cmu.edu](mailto:datashop-help@lists.andrew.cmu.edu).*

## The Component Schema

Each component is defined by its own XML Schema Definition (XSD) located in "<AnyComponent>/schemas/". The XSD defines the component's expected **inputs**, **outputs**, and **options**.

## Defining Inputs and Outputs

Components do not require input, but they all generate output. Both are defined similarly. **Inputs** are defined using the **InFileList** definitions. **Outputs** are defined using the **OutFileList** definitions. Generally, a component will pass its data in one or more files.

Let's look at an example, the Join component, which inputs two text files and generates a single tab-delimited file which contains the joined data files.

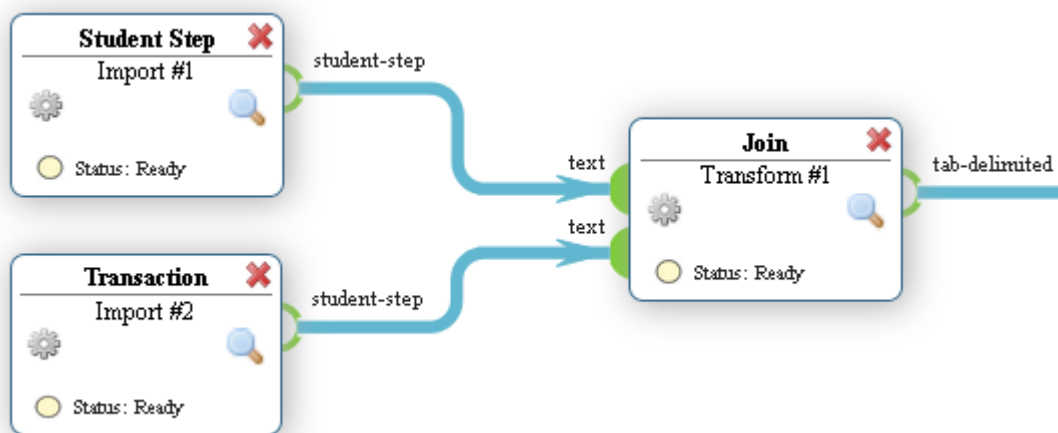


Figure 3 The Join component accepts two text inputs and outputs a tab-delimited file.

Let's look at how to define the inputs and outputs in its XSD. Note that the *student-step* file type is a descendant of the *text* file type. If you are uncertain which file type to use, then use the generic type *file*, as it matches any file type. See Appendix A – Existing File Types for more information on the file type hierarchy.

## Inputs

A component can have 0 or more inputs. The Join component requires two *text* inputs. We use a numeric suffix to differentiate the input nodes, e.g. InFileList0, InFileList1, etc.

The **InputDefinitionX** blocks never change. You will need one for each input, e.g. InputDefinition0, InputDefinition1. The **InFileListX** blocks define the expected file type—text in this case. The **InputType** block is used to finalize the definition and the order of the inputs (if the text1 were listed first, it would be treated as the first input node).

```

<xs:complexType name="InputDefinition0">
  <xs:complexContent>
    <xs:extension base="InputContainer">
      <xs:sequence>
        <xs:element type="InFileList0" name="files" />
        <xs:any minOccurs="0" processContents="skip" maxOccurs="unbounded" />
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
  
```

```
</xs:extension>
</xs:complexContent>
</xs:complexType>

<xs:complexType name="InputDefinition1">
  <xs:complexContent>
    <xs:extension base="InputContainer">
      <xs:sequence>
        <xs:element type="InFileList1" name="files" />
        <xs:any minOccurs="0" processContents="skip" maxOccurs="unbounded" />
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="InFileList0">
  <xs:choice>
    <xs:element ref="text" minOccurs="0" />
  </xs:choice>
</xs:complexType>

<xs:complexType name="InFileList1">
  <xs:choice>
    <xs:element ref="text" minOccurs="0" />
  </xs:choice>
</xs:complexType>

<xs:complexType name="InputType">
  <xs:all>
    <xs:element name="input0" type="InputDefinition0" minOccurs="0" />
    <xs:element name="input1" type="InputDefinition1" minOccurs="0" />
  </xs:all>
</xs:complexType>
```

## Outputs

The output definitions are similar to the input definitions. One must include **OutputDefinitionX**, **OutFileListX**, and the **OutputType**. As in the **InputType**, the **OutputType** defines the order of the output nodes. In our example, we only have one output, but we still must include a reference to it in the **OutputType**.

```
<xs:complexType name="OutputDefinition0">
  <xs:complexContent>
    <xs:extension base="InputContainer">
      <xs:sequence>
        <xs:element type="OutFileList0" name="files" />
        <xs:any minOccurs="0" processContents="skip" maxOccurs="unbounded" />
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="OutputType">
  <xs:sequence>
    <xs:element name="output0" type="OutputDefinition0" />
  </xs:sequence>
</xs:complexType>

<xs:complexType name="OutFileList0">
  <xs:choice>
    <xs:element ref="tab-delimited" />
  </xs:choice>
</xs:complexType>
```

### Components Without Inputs

Not all components have an **InputType** definition. In the example below, the Import file is uploaded via the Component Options window. We will discuss how to define a *file upload* option in the next section.



Figure 4 An Import component does not have inputs. Instead, it has a file upload option.

## Defining Options

The **OptionsType** defines a set of options which can be modified from the user-interface. The options can include *simple data types*, like strings or doubles, as well as a *file upload option*. Let's take a look at the different configurations available.

### File Upload Option

Including a *files* element in an OptionsType allows the end-user to upload a file from the component. Generally, a component will either have Inputs, or it will have a *file upload option*. Components that have both should be split into two separate components—an Import and an Analysis component. Let us look at the *file upload* definition.

Modify text in the **OptionFileList** to match your desired file type. See the section Existing File Types in Appendix A. Examples include tab-delimited, csv, image, and text, but we can invent new types and place them into a hierarchy, e.g. csv (comma-separated values) is a type of text file so any csv will suffice as a text file, but not all text files are csv files.

```
<xs:complexType name="OptionsType">
  <xs:all minOccurs="1">
    <xs:element name="files" type="OptionFileList" minOccurs="1" maxOccurs="1" />
  </xs:all>
</xs:complexType>

<xs:complexType name="OptionFileList">
  <xs:choice>
    <xs:element ref="student-step" minOccurs="1" maxOccurs="1" />
  </xs:choice>
</xs:complexType>
```

### Simple Data Types Options

Allow users to set several simple data types through your component's interface.

```
<xs:complexType name="OptionsType">
  <xs:all >
```

```
<xs:element type="xs:string" name="model" id="Model" default="Default" />
<xs:element type="xs:integer" name="xValidationFolds" id="Cross-validation_Folds" default="10" />
</xs:all>
</xs:complexType>
```

Options have several attributes which must be set. They are type, default, name, and id.

- **Type** – Can be any primitive XML data type: **xs:string**, **xs:double**, **xs:integer**, **xs:boolean**, **xs:date**, **xs:time**, ... For a full list of types, see <http://www.w3.org/TR/xmlschema-2/#built-in-primitive-datatypes>
- **Default** – Specifies the default value of the element if none is provided by the user
- **Name** – A unique name for each option. A name must start with a a-z or A-Z and be followed by any alphanumeric characters, hyphens (-), underscores (\_), or dashes (-) Additionally, if the component executes an external program which accepts command-line arguments, then the **name** attribute is used as the argument identifier, i.e. name="xValidationFolds" default="10" is converted to the command-line pair: -xValidationFolds 10
- **ID** – It follows the same naming convention as the *name* attribute. The ID is displayed in the component options pane to the user, and its underscores are replaced with spaces, e.g. *Cross-validation\_Folds* becomes "Cross-validation Folds" when displayed.

Analysis Options (Bkt)

Model: Input 0 - Column 21 - Predicted Error Rate (Default)

Structure: bySkill

Solver: GradientDescent

Conjugate Gradient Descent Type: None, Polak-Ribiere, Fletcher-Reeves, Hestenes-Stiefel

Hidden States: 2

Max Iterations: 200

Cross-validation Predict State: 1

Cross-validation Folds: 10

Tolerance: 0.01

Initial Parameters: 0.5,1.0,0.4,0.8,0.2

Lower Boundaries: 0,0,1,0,0,0,0,0,0

Upper Boundaries: 1,1,1,0,1,1,1,0.3,0.3,1

Figure 5 An Options dialog box is automatically created from the XML Schema Definition for each component.

### Options with File Upload and Simple Data Types

Allow users to set both simple data types and upload a file via the component interface.

```
<xs:complexType name="OptionsType">
  <xs:all >

    <xs:element type="xs:string" name="model" id="Model" default="Default" />

    <xs:element type="xs:integer" name="xValidationFolds" id="Cross-validation_Folds" default="10" />

    <xs:element name="files" type="OptionFileList" minOccurs="1" maxOccurs="1" />

  </xs:all>
</xs:complexType>

<xs:complexType name="OptionFileList">
  <xs:choice>
    <xs:element ref="text" minOccurs="1" maxOccurs="1" />
  </xs:choice>
</xs:complexType>
```

Any options you specify will be used to generate an **options pane** for your component. Note that "INF" is the XML symbol for infinity but is only acceptable for "xs:double" data types. If you need to allow infinity to be a valid option, the option type must be "xs:double".

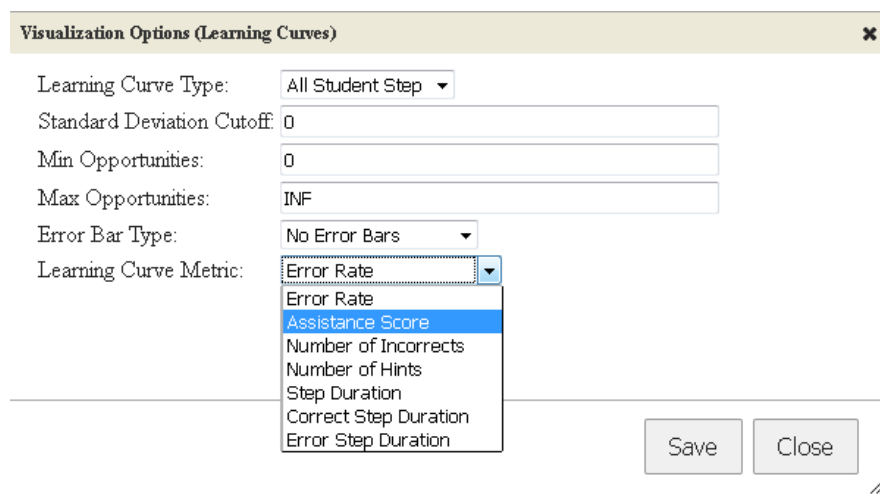


Figure 6 - An example of several different OptionType elements being displayed to the user.

### Polynomials

**Polynomials** can also be defined in the XSD. They are displayed to the user as drop-downs in the component options interface. Simply define a new enumeration in your component's XSD. For binomial values, you may use the predefined xs:boolean type. **The system requires the suffix "Type" when defining polynomials for components, e.g. errorBarType.**

```
<xs:simpleType name="errorBarType" final="restriction">
  <xs:restriction base="xs:string">
    <xs:enumeration value="No Error Bars" />
    <xs:enumeration value="Standard Deviation" />
  </xs:restriction>
</xs:simpleType>
```

```
<xs:enumeration value="Standard Error" />
</xs:restriction>
</xs:simpleType>
```

After the polynomial option is defined, add it to the **OptionsType** definition which contains all of the options. Note that the type is no longer a **simple data type**—it is the type we defined, *errorBarType*.

```
<xs:complexType name="OptionsType">
  <xs:all >

    ...

    <xs:element type="errorBarType" name="errorBar" id="Error_Bar_Type" default="No Error Bars" />

    ...
  </xs:all>
</xs:complexType>
```

### Options Based on Inputs

One useful option type is the **FileInputHeader**. If a component passes any kind of data table (csv, tab-delimited, etc.), then the next component can use the column headers from the data in its own options interface. One can filter columns by using Java regular expressions in the "default" property of the **FileInputHeader** in the **OptionsType** definition.

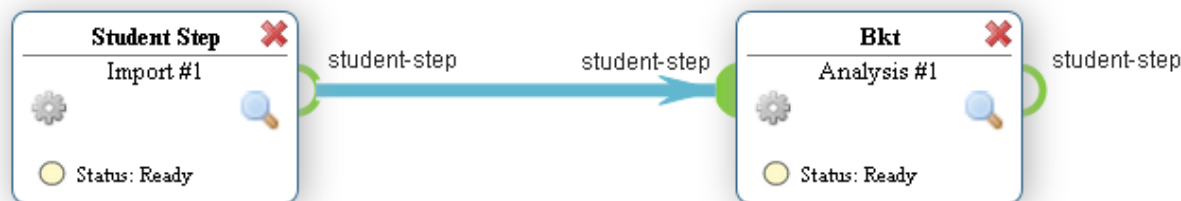


Figure 7 The column headers of the Student Step output can be utilized by the Analysis component.

For example, suppose an input file contains several columns of interest, and all of the contain the word "answer" in the column header. One can match any column header with the word "answer" in it with the Java regular expression "(?i).\*answer.\*" where (?i) declares it a case-insensitive match.

To allow the user to select from any input column, simply use ".\*" which matches any words with 0 or more characters of any kind. Regular expressions are beyond the scope of this document, but feel free to contact us if you have questions.

Match all columns which contain one or more letters (a-zA-Z)

```
<xs:complexType name="OptionsType">
  <xs:all>
    <xs:element type="FileInputHeader" name="model" id="Model" default="(?i).*answer.*" />
    <!-- Other options... -->
  </xs:all>
</xs:complexType>
```

The UI will contain an option drop-down, allowing them to select one of the columns returned by the FileInputHeader. We see that it identifies the index of the input node, the column number, and the actual column header.

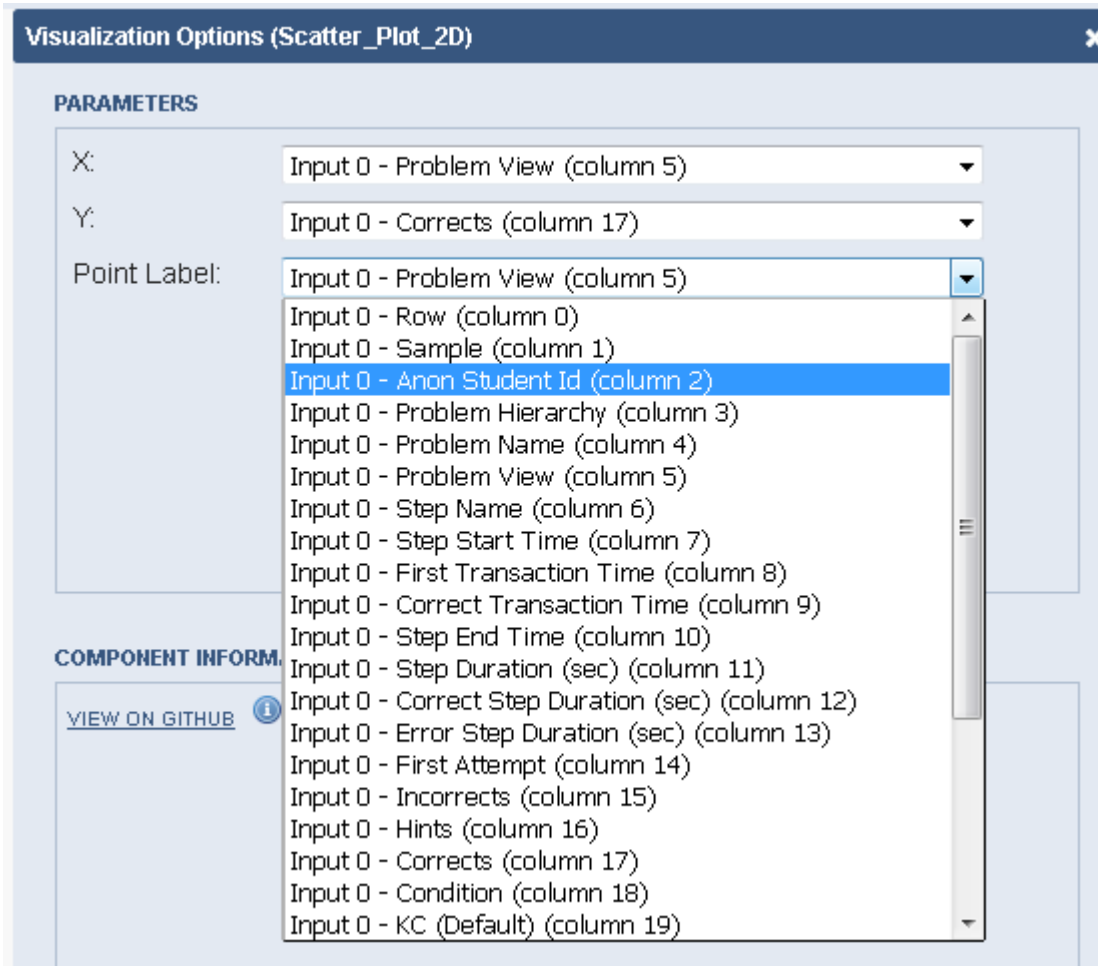


Figure 8 Matching columns using the regular expression ".\*" (any characters)



## Program Integration

A component can run any number of programs, but the bootstrap program is the one which is explicitly run by the platform. The bootstrap will then run any subsequent programs (procedure calls, file manipulation, etc.) before it returns control to the component wrapper.

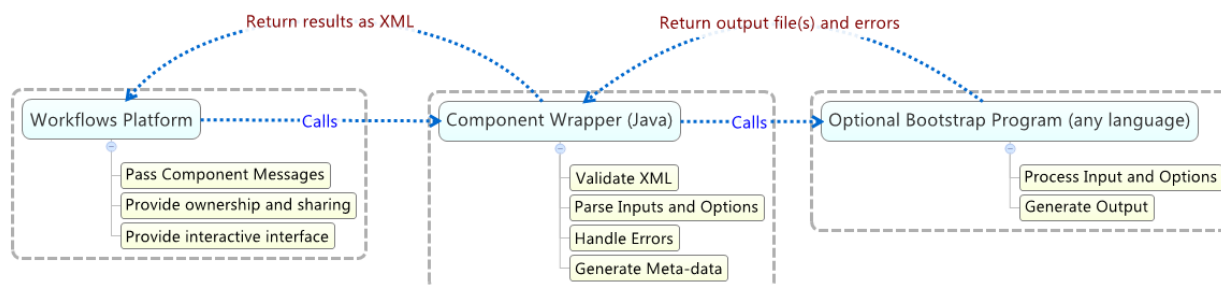


Figure 9 The Workflows call stack.

## Bootstrap Program

To ease portability, each component contains a **build.properties** file which specifies the location of the interpreter (if one is needed) and the location of the component's bootstrap program. Interpreted languages, like Python, R, and Ruby, must include the interpreter and program paths. Here is an example. RScript.exe is the interpreter that runs the R program (or script), DataShop-AFM.R.

```
component.interpreter.path=c:/R-3.2.1/bin/Rscript.exe  
component.program.path=program/DataShop-AFM.R
```

Compiled programs that do not require an interpreter only require that the program path be specified.

```
component.program.path=program/trainhmm.exe
```

Once the Bootstrap Program is loaded, it has full control and can execute additional programs before returning its output (files or stdout) to the Component Wrapper. Any messages printed to the error stream while the program is running are also passed back to the Component Wrapper.

## Component Wrapper (Java)

An essential piece to integrating new programs is the Component Wrapper. It is a standalone program written in Java which is called during workflow execution. Then, if a bootstrap program is defined, it will execute it. The Java wrapper automates schema validation, pre- and post-condition checks, argument passing, error handling, user feedback, and program execution. While the bootstrap program can be written in any language, the Component Wrapper must be in Java to utilize the existing API.

The bootstrap program will be executed by **runExternalMultipleFileOutput()**. All **files** and **simple data types** are passed to the bootstrap program via command-line arguments. A working directory will be created for each component and returned by the method. Once your bootstrap program finishes processing the data, then any files or simple data types added to its output will be passed along to the next component(s) in the workflow.

In the following example, we define a program which uses **runExternalMultipleFileOutput**. The method runs the bootstrap program and returns a single directory which contains all of the files generated by the custom code. In the example, the bootstrap program is expected to generate two files, image1.gif and text1.txt. It then adds these files to their appropriate output nodes—the file types correspond directly to the Output definitions in the component's XSD file as discussed in the section on The Component Schema.

Override the `runComponent` method in the Java wrapper to call our bootstrap program as declared in the `build.properties` file discussed earlier.

```
@Override
protected void runComponent() {
    // Run the program and add the files it generates to the component output.
    File outputDirectory = this.runExternalMultipleFileOutput();
    // Attach the output files to the component output with addOutputFile(...)
    if (outputDirectory.isDirectory() && outputDirectory.canRead()) {
        File file0 = new File(outputDirectory.getAbsolutePath() + "/text1.txt");
        File file1 = new File(outputDirectory.getAbsolutePath() + "/image1.gif");

        if (file0 != null && file0.exists() && file1 != null && file1.exists()) {

            Integer nodeIndex0 = 0;
            Integer fileIndex0 = 0;
            String label0 = "text";
            this.addOutputFile(file0, nodeIndex0, fileIndex0, label0);

            Integer nodeIndex1 = 1;
            Integer fileIndex1 = 0;
            String label1 = "image";
            this.addOutputFile(file1, nodeIndex1, fileIndex1, label1);

        } else {
            this.addErrorMessage("The expected output files could not be found.");
        }
    }

    // Send the component output back to the workflow.
    System.out.println(this.getOutput());
}
```

### Bootstrap Program Parameters

When the *component* is executed, its inputs and option values are passed to the bootstrap program as command-line arguments. The argument labels are defined in your XML Schema Definition's options and inputs. Each bootstrap program is provided a set of tagged command-line arguments for easy parsing. You can then read these parameters using your preferred programming language's standard argument handling capabilities.

### Example Parameters

```
/path/to/MyComponent/program/analysis.exe

-programDir /path/to/MyComponent

-workingDir /path/to/MyComponent_WorkingDirectory

-xValidationFolds 100

-someOtherOption value2
```

---

`-file0 input.txt`

### Example Parameter Summary

**programDir** – the base directory of the component, i.e. the directory containing all of the component code

**workingDir** – the working directory for each component in each workflow, e.g.  
/data/workflows/1/ImportComponent3/ -- the working directory is the CWD used to run the external program

**xValidationFolds** – an example option

**someOtherOption** – an example option

**file0** – the relative path to the input file on Node 0

**file1** – the relative path to the input file on Node 1

...

The **simple data types** are passed as strings while **files** are passed as absolute file paths.

Note: Components are executed on Linux so programs should expect absolute file paths in accordance with this fact. For more information on the Linux file system and what to expect, see <https://help.ubuntu.com/community/LinuxFilesystemTreeOverview>. Almost all programs can simply ignore this requirement unless they utilize the programDir or workingDir arguments in some platform-specific way, an unlikely event.

### Simple Input Parameters

For simple data types, the **name** attribute prefixes the argument as a tag, e.g. -xValidationFolds.

```
<xs:element type="xs:integer" name="xValidationFolds" id="Cross-validation_Folds" default="10" />
```

The XSD element is converted to the command-line argument:

```
-xvalidationfolds 10
```

### File Input Parameters

- Components may have 0 or more input nodes but only 1 file per input node
- Files passed to the bootstrap program are tagged, e.g. -file0 someFile.txt -file1 anotherFile.txt
- -file0 is the file passed to the first input node, -file1 is the second, etc.

```
<xs:complexType name="InputType">  
  <xs:all>  
    <xs:element name="input0" type="InputDefinition0" minOccurs="0" />  
    <xs:element name="input1" type="InputDefinition1" minOccurs="0" />  
  </xs:all>  
</xs:complexType>
```

## Parameter Order

Parameters will always follow this order

- [1] `-programDir /path/to/WorkflowComponents/GeneratePfaFeatures/`
- [2] `-workingDir /path/to/workflows/16/Transform-1-x432172/output/`
- [3] data type options (alphabetical order), e.g. `-model Default -nonce 123 -xValidationFolds 4`
- [4] input files (ordered by node index), e.g. `-file0 somefile.txt -file1 someOtherFile.txt`

## Generating Output

A component outputs an XML entity which contains all the data needed by the workflows platform and other components. The XML is automatically generated once the component's **program** has completed. There are several options for generating component output. The most direct way is using the output stream. The **output stream** of your program is written to a file which can be added to the component output, if desired.

### From Output Stream

The output stream of a program is written to a file that can be attached to the component output. Each language has a standard set of calls for writing to the output stream. In Java, this is done via the `System.out.println` statement. In C or C++, this is done via `printf` or `cout` statements.

In the following example, we see how to interact with the component output. The bootstrap program specified in the `build.properties` file is executed, and a file is automatically created from the program's standard output stream. Then, the file is attached to the component's output (as a file path), and then the component output is sent to the next component.

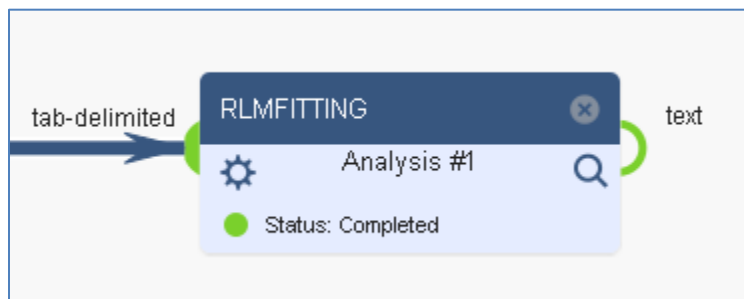


Figure 10 Component with a single file output.

```
@Override
protected void runComponent() {
    // Run the program and return its stdout to a file.
    File output = this.runExternal();

    Integer nodeIndex = 0;
    Integer fileIndex = 0;
    String fileLabel = "text";

    this.addOutputFile(output, nodeIndex, fileIndex, fileLabel);
}
```

```
// Send the component output back to the workflow.  
System.out.println(this.getOutput());  
}
```

### *File Manipulation*

The alternative way of generating file output is to simply write files from your external program to the "output" subdirectory of the working directory. Then, you can add the file(s) to the component output in the Java wrapper with a set of commands similar to those when using the standard output stream.

```
...  
File outputDirectory = this.runExternalMultipleFileOutput();  
File outputFile = outputDirectory + "/MyOutput.txt";  
this.addOutputFile(outputFile, nodeIndex, fileIndex, fileLabel);
```

The above lines will run the external file and the file "MyOutput.txt" to the component output.

Any number of files can be added to the output so long as they are defined in the XML Schema Definition.

### *Summary of Methods*

#### **public void** runExternal()

This method executes the component's program, passing all inputs and options to it via the command-line. The standard output stream of the program is automatically written to the file returned by this method.

#### **public String** runExternalMultipleFileOutput()

This method executes the component's bootstrap program declared in the build.properties file. When executed, all inputs and options are passed via the command-line to the bootstrap program. The program will then be expected to create one or more output files which are added to the component output.

#### **public void** addOutputFile(File file, Integer nodeIndex, Integer fileIndex, String label)

This method attaches an arbitrary file to the component's output. It requires

- file – the output file
- nodeIndex – the output node index (0 being the first output node)
- fileIndex - more than one file may be attached to an output node but usually the relationship is 1 file to 1 output node
- label – the file label; the most generic label is **file**, while more specific types include **student-step**, **transaction**, **zip**, or **image**. See *File Types* in **Appendix A**.

#### **getOutput()**

This method produces the program output and component meta-data required by the system to process the workflow. It should be written to the standard output stream at the end of the runComponent() method which is required by all components except Import components.

### *Accessing Options from within the Component Wrapper (Java)*

You may want to access component options or inputs prior to the actual execution of the bootstrap program. You can access them via built-in method calls in Java. Otherwise, you may access these same options via the command-line arguments passed to the bootstrap program.

### Example XSD Option Definition

```
<xs:complexType name="OptionsType">
  <xs:all >
    <xs:element type="xs:string" name="model" id="Model" default="Default" />
    <xs:element type="xs:integer" name="xValidationFolds" id="Cross-validation_Folds" default="10" />
    <xs:element type="xs:double" name="weight" id="Weight" default="1.0" />
    <xs:element type="xs:boolean" name="useSampling" id="Use_Sampling" default="false" />
  </xs:all>
</xs:complexType>
```

### Example Option Accessors

```
String model = getOptionAsString( "model" );
Integer xValidationFolds = getOptionAsInteger( "xValidationFolds" );
Double weight = getOptionAsDouble( "weight" );
Boolean useSampling = getOptionAsBoolean( "useSampling" );
```

### Accessing Inputs from within the Java wrapper

```
protected File getAttachment(int nodeIndex, int fileIndex)
```

nodeIndex - the input node index (0 is the first)

fileIndex - the file index of the file associated with the given input node

### Example XSD InputData Definition

```
xs:complexType name="InputData">
  <xs:complexContent>
    <xs:extension base="InputContainer">
      <xs:sequence>
        <xs:element type="xs:string" name="model" />
        <xs:element type="xs:integer" name="xValidationCrossFolds" />
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

### Example Input Accessors

```
File inputFile0 = getAttachment(nodeIndex, fileIndex);
File inputFile1 = getAttachment(1);
String model = getInputAsString(fileIndex, "model");
```

## Error Handling

### Error Stream

When a component executes a program, it reads from the *stderr* stream. If any error messages exist, they are passed pack through the wrapper and presented to the user on the workflows message pane. Most languages have their own functions for writing to the error stream—e.g., Java's *System.err.println* or the C++ object *std::cerr*.

### Pre- and Post-condition Checks

The XSD is the primary driver for pre- and post-condition checks. If defined correctly, you can always expect a component's inputs and options to meet the criteria defined in the component schema. If any expected inputs are options are absent, or if one of the values does not match their designated type, then an error will be

returned to the user and the component will exit before executing its program. For example, if "abc" is entered into an xs:integer type option, the user will receive the following feedback.

```
Previous run results - Analysis #1
cvc-datatype-valid.1.2.1: 'abc' is not a valid value for 'integer'.
```

In addition to type checks, **files** attached to a component (input or options) are automatically tested to ensure they exist and can be read. If not, the component will not run its program and will return an error to the user.

### User-defined Tests

More advanced tests can be implemented by the user. User-defined tests can test input and options values. If any of the tests fail, the component will return a specific error before running the program. This can be useful when the XSD restrictions are not enough. To create the user-defined test method in the Java wrapper, simply create and override the component's **test** method.

```
/**
 * The test() method is used to test the known inputs prior to running.
 * @return true if passing, false otherwise
 */
@Override
protected Boolean test() {
    Boolean passing = true;

    // Constrain the xValidationFolds options to be between 2 and 100.
    Integer xValidationFolds = this.getOptionAsInteger("xValidationFolds");

    if (xValidationFolds < 2 || xValidationFolds > 100) {

        // Add the error to the user interface
        this.addErrorMessage("Cross-validation folds " + xValidationFolds
            + " must be between 2 and 100, inclusive.");
        passing = false;
    }

    for (String err : errorMessages) {
        logger.error(err);
    }

    return passing;
}
```

## Appendix A

### Existing File Types

From TableTypes.xsd (as of 1/18)

```
<xs:element name="file" type="FileContainer" />
<xs:element name="text" substitutionGroup="file" />
<xs:element name="tab-delimited" substitutionGroup="text" />
<xs:element name="analysis-summary" substitutionGroup="text" />
<xs:element name="user-sess-map" substitutionGroup="tab-delimited" />
```

```
<xs:element name="user-map" substitutionGroup="tab-delimited" />
<xs:element name="resource-use" substitutionGroup="tab-delimited" />
<xs:element name="outcome" substitutionGroup="tab-delimited" />
<xs:element name="resource-use-to-outcome" substitutionGroup="tab-delimited" />
<xs:element name="student-step" substitutionGroup="tab-delimited" />
<xs:element name="transaction" substitutionGroup="tab-delimited" />
<xs:element name="student-problem" substitutionGroup="tab-delimited" />
<xs:element name="csv" substitutionGroup="text" />
<xs:element name="xml" substitutionGroup="text" />
<xs:element name="html" substitutionGroup="text" />
<xs:element name="ctat-Log" substitutionGroup="xml" />
<xs:element name="sql" substitutionGroup="text" />
<xs:element name="discoursedb" substitutionGroup="text" />

<xs:element name="image" substitutionGroup="file" />
<xs:element name="png" substitutionGroup="image" />
<xs:element name="jpg" substitutionGroup="image" />
<xs:element name="gif" substitutionGroup="image" />
<xs:element name="psd" substitutionGroup="image" />
<xs:element name="bmp" substitutionGroup="image" />

<xs:element name="compressed" substitutionGroup="file" />
<xs:element name="zip" substitutionGroup="compressed" />
<xs:element name="bz2" substitutionGroup="compressed" />
<xs:element name="gz" substitutionGroup="compressed" />
<xs:element name="_7z" substitutionGroup="compressed" />
```

## Adding New File Types

File types are defined in "CommonSchemas/TableTypes.xsd" in the WorkflowComponents package. Any elements defined in a FileList must exist in the TableTypes.xsd file. Currently, adding a new type is straightforward. Simply add it to one parent that best describes it, or create a new category entirely. As an example, let's add a new "tab-delimited" file type called "blackboard-quiz-scores". All we need to do is set the name and substitutionGroup.

```
...

<xs:element name="file" type="FileContainer" />

<!-- All of the following are types of "file" elements.
      If the subheading you want doesn't exist, either create a new one with a unique name
      or use a more generic type, like "text" or "image". -->
<xs:element name="text" substitutionGroup="file" />
  <xs:element name="tab-delimited" substitutionGroup="text" />
    <xs:element name="resource-use" substitutionGroup="tab-delimited" />
    <xs:element name="student-step" substitutionGroup="tab-delimited" />
    <xs:element name="blackboard-quiz-scores" substitutionGroup="tab-delimited" />
  </xs:element>
</xs:element>

...
```