Defining a Flat File

This page explains the **Flat File Adapter** in Bridge context. If you were looking for the same information regarding the PAS Designer, refer to Flat File Adapter in the Designer guide.

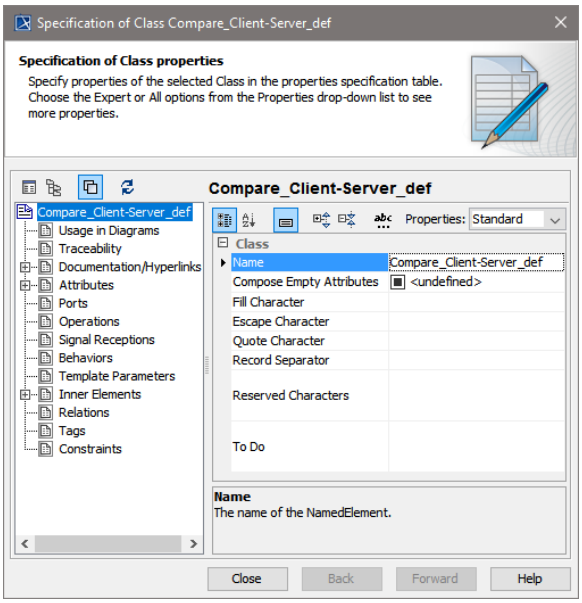
Defining the Flat File

The flat file structure is defined in a class diagram. A root class defines the file and its settings (like e.g. the record separator). This class must have stereotype <<FlatFile>>.

As this is the root of a flat file declaration no attributes are allowed. You should give this class a meaningful name, because the output object flow of the <<FlatFileAdapter>> action in the activity diagram uses this class as a type.

You can change the flat file settings on the specification dialog of the root class.

Figure: Change Flat File Settings



Value <NewLine> of tag **Record Separator** is a literal placeholder for special character \n. A complete list of all placeholders is listed below.

Placeholder	C Syntax	Character (Dec.)
<tab> **<tabulator>**	\t	9
<newline> **<unixnewline>**	\n	10
<windowsnewline>	\r\n	13, 10
<esc>	\x1B	27
<space>		32

The other optional tagged values are listed below.

On this Page:

Related Pages:

- Defining a Simple Flat File Record

Flat File Settings

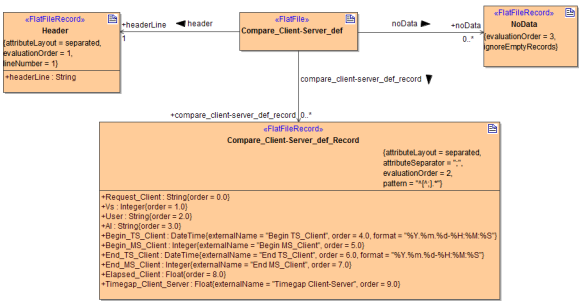
Find below a list of all tagged values corresponding to stereotype <<FlatFile>>.

Tagged Value	Description	Allowed Values		Example
escapeCharacter	Defines the character used for escaping when a reserved character is used within a field value.	any character		`/`
fillCharacter	Defines a dummy character to fill non-existent values (results in NULL). Used for fixed attribute layout only.	any character		`0`
quoteCharacter	The **quoteCharacter** will be ignored by reading field value.	any character		`/`
recordSeparator	Separator of the different records, normally line feed and carriage return. For serialized files, any other character can be defined.	any character or one of		`<newline>`
		`<esc>`		
		`<newline>`	parses correctly on Unix and Windows platforms	
		`<space>`		
		`<tab>` `<tabulator>`		
		`<unixnewline>`	composes newline	
		`<windowsnewline>`	composes newline and carriage return	
reservedCharacters	Defines a list of characters to be escaped automatically when the file is composed.	any character		`{"/", "%", "&", "(", ")"}`

Defining a Flat File Record

The record structure and the relation between records is defined by one or more child classes and the association to their parent class (e.g. the root class or a parent record class). All records have stereotype <<FlatFileRecord>>.

Figure: Flat File Record Structure



Flat File Record Groups

Additionally, (virtual) groups can be defined by stereotype <<FlatFileGroup>>. These classes have no physical representation in the flat file itself.

Flat File Record Settings

Find below a list of all tagged values corresponding to the stereotype <<FlatFileRecord>>.

Tagged Value	Description	Allowed Values		Example
attributeLayout	Defines the attribute layout (**fixed** or **separated**) of the flat file record. • **Fixed**: For attribute values that are shorter than the maximum length of the field, the field is filled with a fill character. As per default, this is blank space, but you can change the fill character in tagged value **fillCharacter** of the <<FlatFile>> class (see Tagged Values of Class <<FlatFile>>). • **Separated**: If separated is used, specify the separator using **attribute Separator** (see below).	fixed	fixed attribute layout	
		separated	separated attribute layout	
attributeSeparator	Defines the attribute/field separator.	any character	use this character as attribute separator	
		<Tab>	use tabulator as attribute separator	
attributePattern	A RegEx pattern to parse the record content into the attributes using capture groups.	a valid regular expression		`^(.?)(://)([A-Za-z0-9.])(:[0-9])(/.)$`
evaluationOrder	Defines the order in which the association of the classes starting on same parent class must be processed, see Flat File Adapter Parsing Process.	any integer		
ignoreEmptyRecords	Boolean value for ignoring empty records. If set to true, no item will be generated, if none of the defined attributes or sub records have any content. Note, that a record containing only empty **Strings** is not empty – in opposition to a record composed from `NULL`s. See **ignoreEmptyStrings** below to skip processing of records containing only empty **Strings**.	true	ignore empty records	
		false (default)	process empty records	
ignoreEmptyStrings	Boolean value for ignoring empty string attributes. If set to **true**, empty string values will be processed to `NULL`. Use this tag in combination	true	ignore empty string values	

	with **ignoreEmptyRecords** to skip processing of records containing only empty **Strings**.	**false** (default)	preserve empty string values	
lineNumber	Specifies the number of a record in the file. The first record is lineNumber=1, the second lineNumber=2, etc.	any integer		
pattern	A pattern to identify the record. The pattern is checked before the fields are separated. If no pattern is defined, all records will be parsed.	any character		
		a valid regular expression		`^Pattern.*`
suppressEscaping	Boolean value to suppress escaping. If **suppressEscaping** on a <<FlatFileRecord>> is true, <<FlatFileComplexAttribute>> that are part of this record will inherit this setting.	true	attribute values of this record will not be un-escaped (parser) or escaped (composer)	
		false	escaping /un-escaping is not suppressed	
parseMacro **composeMacro**	A macro that is executed while parsing/composing a file or complex field. This macro can contain multiple commands separated by commas or spaces. Macros on classes are executed before the processing of its attributes or associations. The ID represents a counter. The following counters are available: • eight **automatic counters** with ID `AUTO0` .. `AUTO7` • two **automatic line counters** with ID `LINE0` and `LINE1` (parsing only) • unlimited **custom counters** with ID `CUSTOM0` .. `CUSTOMx` Automatic counters are increased by 1 for each processed record. Custom counters have to be increased manually using the increase macro. All counters have the initial value of 0 when they process the first record. For more details on macro commands see Macro Expressions .	any valid macro expression (see Macro Expressions)		`GetCounter(AUTO0)`

For detailed information on associations see Flat File Group Settings or Associations Endings of <<FlatFileRecord>> and <<FlatFileGroup>> .

Flat File Group Settings

You can group multiple records in one virtual structure by using the stereotype <<FlatFileGroup>>. This virtual group does not have a representation in the flat file and therefore cannot hold any attributes. Apart from this, this element has the same behavior like a flat file record. For parsing and composing, the pattern and conditions are checked but no mapping takes place. The record details are given to the associated class where the mapping is done.

Find below a list of all tagged values corresponding to the stereotype <<FlatFileGroup>> .

Tagged Value	Description	Allowed Values	Example
pattern	A pattern to identify the record. The pattern is checked before the fields are separated. If no pattern is defined, all records will be parsed.	any character a valid regular expression	`^Pattern.*`

Associations Endings of <<FlatFileRecord>> and <<FlatFileGroup>>

All associations ending on a class with stereotype <<FlatFileRecord>> or <<FlatFileGroup>> can have additional attributes, if the association end has stereotype <<FlatFileSubRecord>> applied.

Find below a list of all tagged values corresponding to stereotype <<FlatFileSubRecord>> .

Tagged Value	Description	Allowed Values	Example
condition	A condition that must evaluate *true* if the record exists. The condition can refer to a self object which represents the current state of the parent.	any valid conditional expression	`self.UNS.exists()`
evaluation Order	Defines the order in which the associations starting on same parent class must be processed, see Flat File Adapter Parsing Process.	any integer	
offset	Define the position of this record in the flat file, starting with 0 for the first record and always relative to the parent element.	any integer	

Defining Flat File Record Attributes

All Attributes on a <<Flat File Record>> class need to have stereotype <<FlatFileAttribute>>. Depending on the layout type of the flat file (fixed or separated), you can specify different tagged values (see Attribute Settings for attributeLayout = fixed and Attribute Settings for attributeLayout = separated). Additionally, you can define record fields as to be of complex type (see Complex Flat File Record Attributes).

Attribute Settings for attributeLayout = fixed

For flat files having a fixed layout, you need to specify external length and order of the attribute, and optionally, you can specify a padding.

Tagged Value	Description	Allowed Values	Example
decimals	Replaced by **format.**		
externalLength	Number of characters of the field (only for fixed length records relevant).	any integer	
format	Pattern for formatting numeric and date & time values. For details see Number Formatting respectively Date and Time Formatting.	any valid number or dateTime pattern	`S9G999G990D00` `%Y.%m.%d-%H:%M:%`
nativeTyp deprecated	Replaced by **format**.		
order	The evaluation order of the attributes. If **offset** is not used, order reflects the field number within the record.	any integer	
offset	The character position of this field within the record.	any integer	
padding	Defines the padding rule for the field. When parsing, the characters on the left or right side are	left("<any character>")	`left("0")`

	ignored up to the first different character. When composing, the field is filled on the left or right side with the specified character.	right("<any character>")		`right(" ")`
suppressEscaping	Boolean value to suppress escaping.	true	attribute values of this attribute will not be un-escaped (parser) or escaped (composer)	
		false	escaping/un-escaping is not suppressed	
parseMacro **composeMacro**	A macro that is executed while parsing/composing a file or complex field. This macro can contain multiple commands separated by commas or spaces. Macros on classes are executed before the processing of its attributes or associations. The ID represents a counter. The following counters are available: • eight **automatic counters** with ID `AUTO0` .. `AUTO7` • two **automatic line counters** with ID `LINE0` and `LINE1` (parsing only) • unlimited **custom counters** with ID `CUSTOM0` .. `CUSTOMx` Automatic counters are increased by 1 for each processed record. Custom counters have to be increased manually using the increase macro. All counters have the initial value of 0 when they process the first record. For more details on macro commands see Macro Expressions .	any valid macro expression (see Macro Expressions)		`GetCounter(0)`

Attribute Settings for attributeLayout = separated

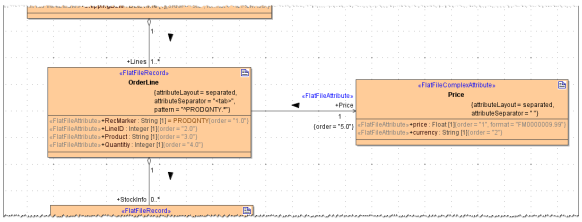
For attributes in separated flat files, you can supply an offset to specify the position of the field (attribute) in respect of the other record fields.

Tagged Value	Description	Allowed Values	Example
format	Pattern for formatting numeric and date & time values. For details see Number Formatting respectively Date and Time Formatting.		
order	The evaluation order of the attributes. If **offset** is not used, order reflects the field number within the record.	any integer	
offset	The relative position of the field in respect of the other fields in the record, e.g. field number 3 has **offset** = 2.	any integer	
suppressEscaping	Boolean value to suppress escaping.		
parseMacro **composeMacro**	A macro that is executed while parsing/composing a file or complex field. This macro can contain multiple commands separated by commas or spaces. Macros on classes are executed before the processing of its attributes or associations. The ID represents a counter. The following counters are available: • eight **automatic counters** with ID `AUTO0` .. `AUTO7` • two **automatic line counters** with ID `LINE0` and `LINE1` (parsing only) • unlimited **custom counters** with ID `CUSTOM0` .. `CUSTOMx` Automatic counters are increased by 1 for each processed record. Custom counters have to be increased manually using the increase macro. All counters have the initial value of 0 when they process the first record. For more details on macro commands see Macro Expressions .	any valid macro expression (see Macro Expressions)	`GetCounter(0)`

Complex Flat File Record Attributes

Flat file record attributes can by of complex type. The complex type must have stereotype <<FlatFileComplexAttribute>> applied, then.

Figure: Flat File Complex Attribute



You can use this class to divide a record field into sub-fields. You can think of this like a <<FlatFileRecord>> placed within a single field. Most of the tagged values that are valid for a flat file record are valid for a complex attribute, too. Though, the scope of these settings is not the record, but the field.

Tagged Value	Description	Allowed Values		Example
attributeLayout	Defines the attribute layout (**fixed** or **separated**) of the complex attribute. • **Fixed**: For attribute values that are shorter than the maximum length of the field, the field is filled with a fill character. As per default, this is blank space, but you can change the the fill character in tagged value **fillCharacter** of the <<FlatFile>> class (see Tagged Values of Class <<FlatFile>>). • **Separated**: If separated is used, specify the separator using **attributeSeparator** (see below).	fixed	fixed attribute layout	
		separated	separated attribute layout	
attributeSeparator	Defines the attribute/field separator.	any char	use this character as attribute separator	
		<Tab>	use tabulator as attribute separator	
attributePattern	A RegEx pattern to parse the field content into a complex structure using capture groups.	a valid regular expression	`^(.?)(://)([A-Za-z0-9.])(:[0-9])(/.)$`	
suppressEscaping	Boolean value to suppress escaping.	true	attribute values of this attribute will not be un-escaped (parser) or escaped (composer)	
		false	escaping/un-escaping is not suppressed	
parseMacro	A macro that is executed while parsing/composing a file or complex field.	any valid macro expression (see Macro Expressions)		`GetCounter(0)`

compose Macro	This macro can contain multiple commands separated by commas or spaces. Macros on classes are executed before the processing of its attributes or associations. The ID represents a counter. The following counters are available: • eight **automatic counters** with ID `AUTO0` .. `AUTO7` • two **automatic line counters** with ID `LINE0` and `LINE1` (parsing only) • unlimited **custom counters** with ID `CUSTOM0` .. `CUSTOMx` Automatic counters are increased by 1 for each processed record. Custom counters have to be increased manually using the increase macro. All counters have the initial value of 0 when they process the first record. For more details on macro commands see Macro Expressions .		

Macro Expressions

Available macros are counters:

- eight **automatic counters** with ID `AUTO0` .. `AUTO7` (for parsing and composing)
- two **automatic line counters** with ID `LINE0` and `LINE1` (for parsing only)
- unlimited **custom counters** with ID `CUSTOM0` .. `CUSTOMx` (for parsing and composing).

Automatic counters automatically increase with each processed record, whereas custom counters have to be increased manually using the **IncreaseCounter**() macro. When parsing/composing the first record, all counters have the initial value of 0.

Macro	Available on	Available for	Description	Example
ResetCounter(`ID`[, *Value*])	Classes Attributes	`AUTO` `CUSTOM`	Reset the counter `ID` to 0 or a given *Value*.	`ResetCounter (AUTO0, 1)`
IncreaseCounter(`ID`)	Classes Attributes	`AUTO` `CUSTOM`	Increases the counter `ID` by 1.	`IncreaseCounter (CUSTOM2)`
GetCounter(`ID`)	Attributes	`AUTO` `LINE` `CUSTOM`	Read the value of a counter and store it in the current attribute.	`GetCounter(LINE0)`
VerifyCounter(`ID`)	Attributes	`AUTO` `LINE` `CUSTOM`	Compare the value of a counter with the current attribute. This macro will throw an exception if the values are not equal.	`VerifyCounter (CUSTOM0)`

For **GetCounter()** and **VerifyCounter()** only attributes of type **Integer** are supported.