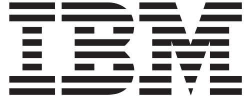
IBM WebSphere Transformation Extender



Type Designer

Version 8.1

Note

Before using this information, be sure to read the general information in "Notices" on page 155.

2006

This edition of this document applies to IBM WebSphere Transformation Extender Version 8.1; and to all subsequent releases and modifications until otherwise indicated in new editions.

To send us your comments about this document, email DTX_doc_feedback@us.ibm.com. We look forward to hearing from you.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© Copyright International Business Machines Corporation 2006. All rights reserved.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Chapter 1. Introduction to the Type Designer . . . 1
About type trees . . . 1
Data composition . . . 1
Subtypes. . . . 2
Type tree hierarchy . . . 2
Type Designer files . . . 2
Type Designer icons . . . 3

Chapter 2. Type Designer basics . . . 5
Type Designer startup . . . 5
To import a type tree . . . 5
To open an existing type tree . . . 5
To create a type tree . . . 5
To open a recently used type tree . . . 5
User interface . . . 5
Type tree window . . . 6
As work book option . . . 6
Splitting the type tree view . . . 6
Item window . . . 6
Group window . . . 7
Category window . . . 7
Properties Window . . . 7
Menu commands and tools . . . 8
File menu . . . 8
Edit menu . . . 9
Undo commands . . . 9
View menu . . . 10
Type menu . . . 10
Tree Menu . . . 11
Component menu . . . 11
Restriction menu . . . 12
Tools menu . . . 12
Window menu . . . 12
Help menu . . . 13
Status bar . . . 13
Configuring the Type Designer environment . . . 13
Options. . . . 13

Chapter 3. Working with type trees . . . 17
Creating type trees . . . 17
Creating types . . . 17
Viewing types . . . 18
Expanding a Type . . . 18
Data content . . . 18
Data objects . . . 18
Types . . . 18
Classes . . . 18
Item . . . 18
Group . . . 19
Category . . . 19
Components . . . 19
Exporting a type tree . . . 19
Exporting a type as a schema . . . 20
Type to XML conversion . . . 20
Type tree differences . . . 20

© Copyright IBM Corp. 2006

Creating a type tree exercise . . . 21
Opening an existing type tree . . . 23

Chapter 4. Type properties . . . 25
Defining type properties . . . 25
To access properties of a type . . . 25
To define the properties of a type . . . 25
Basic type properties . . . 26
Name . . . 26
Class . . . 27
Description . . . 27
Intent . . . 28
Partitioned . . . 29
Order subtypes . . . 29
Initiator . . . 29
Terminator. . . . 30
Release characters . . . 30
Empty . . . 31
National language . . . 32
Document Type . . . 32
Where used . . . 33
XML properties in the type tree. . . . 33
Support for XML constructs . . . 33
XML schema datatypes . . . 34
Identity constraints . . . 36
Namespaces . . . 36
Element type declarations . . . 37
Element and attribute wildcards . . . 37

Chapter 5. Item properties . . . 39
Item subclass . . . 39
Number item subclass properties . . . 39
Interpret as binary . . . 39
Length (bytes) . . . 40
Byte order . . . 41
Interpret as character . . . 41
Size (content) . . . 42
Separators . . . 43
Pad . . . 48
Restrictions . . . 52
National language . . . 52
None . . . 57
Places . . . 57
Text item subclass properties . . . 58
Date & time item subclass properties . . . 58
Date . . . 58
Time. . . . 59
Format . . . 59
Time zones . . . 62
Time zone format string for XML . . . 63
Optional time segments of the time format string. . . . 63
Date and time format examples. . . . 63
Special . . . 64
Syntax item subclass properties. . . . 65
Syntax objects with variable values . . . 65
Example of Variable Syntax Object as an Item Type . . . 65
Syntax objects as data . . . 66
Example of syntax objects as components of a type . . . 66
Delimiter > find . . . 66

Chapter 6. Item restrictions . . . 69
Defining item restrictions . . . 69
  Inserting new rows . . . 70
Restrictions settings . . . 70
  Value restrictions . . . 70
  Character restrictions . . . 70
  Range restrictions . . . 71
  Value not in range . . . 71
  Inserting symbols . . . 72
Ignoring restrictions . . . 72

Chapter 7. Group properties . . . 73
Group subclass . . . 73
  Properties of group subclasses . . . 73
  Choice group components . . . 73
  Unordered group components . . . 74
Sequence group formats . . . 74
Explicit format . . . 75
  Track . . . 75
  Fixed syntax . . . 75
  Explicit delimited syntax . . . 76
Implicit format . . . 76
  Floating component . . . 76
  Implicit whitespace syntax . . . 77
  Implicit delimited syntax . . . 78
  No syntax . . . 78
  Distinguishable components of an implicit group . . . 79
Specifying a delimiter . . . 79
  Literal . . . 79
  Variable . . . 80
  Location . . . 80
  Delimiter value appears as data . . . 80

Chapter 8. Components . . . 81
Components are required for group types . . . 81
  Components must be in the same type tree . . . 81
  Importance of component order . . . 81
  Component range . . . 81
Group windows . . . 82
  Nested components. . . . 82
Defining components . . . 83
  Complete type name . . . 83
  Relative type names . . . 83
  Ambiguous type names . . . 84
  Always drag components. . . . 85
  Viewing the component number . . . 85
  Specifying minimum and maximum consecutive occurrences in the component list . . . 85
  Fixed and variable ranges . . . 85
  Using the set range command . . . 85
  Viewing the range column . . . 86
  Types that can be components . . . 86
  Variable component names . . . 87
  Opening a component window . . . 87
Required and optional data . . . 87
  Significance of required data. . . . 88
Defining component rules . . . 88
  Examples of component rules . . . 89
  Component rule syntax . . . 89
  Entering object names in component rules . . . 90
  Shorthand notation . . . 90

Component rules are context-sensitive . . . 90
Special characters in component rules . . . 91
Inserting functions into component rules . . . 91
Formatting a component rule . . . 91
Comments in component rules . . . 91
Syntax errors . . . 92
Searching for components . . . 92
Finding a component by number . . . 92
Managing components . . . 92
Component attributes . . . 92
Identifier attribute . . . 92
Restart attribute . . . 93
Sized attribute . . . 93

Chapter 9. Partitioning . . . 95
Determining when to partition . . . 95
Required partitioning . . . 95
Partitioning for convenience . . . 95
Benefits of partitioning . . . 96
Partitioning types . . . 96
Partitioning items . . . 96
Partitioning an item type using initiators . . . 97
Partitioning an item type using restrictions . . . 97
Example of using restrictions . . . 97
Partitioning an item type by format . . . 98
Partitioning groups . . . 98
Partitioning a group type using initiators . . . 98
Partitioning a group type using identifiers . . . 98
Partitioning a group type using component rules . . . 99

Chapter 10. Type inheritance . . . 101
Inheritance of item properties and restrictions . . . 101
Inheritance of category properties and components . . . 101
Organizing types under a category . . . 101
Using categories for inheritance . . . 101
When not to use categories . . . 102
Propagating properties . . . 102
Properties that can be propagated . . . 102
Propagating affects types in the subtree . . . 102

Chapter 11. Managing types . . . 103
Standard windows capabilities . . . 103
Object selection . . . 103
Drag-and-drop procedures . . . 103
Moving and copying objects . . . 103
Using the move command . . . 104
Using a copy command . . . 104
Type names . . . 104
Reordering objects . . . 104
Reordering existing subtypes . . . 105
Merging types . . . 105
Before using the merge command . . . 105
Supertypes . . . 106
Existing types . . . 106
Invalid types . . . 106
Renaming types . . . 106
Using find and replace . . . 106
Using the find command . . . 106
Using the replace command . . . 107
Printing in the Type Designer . . . 107

To print the Properties window . . . 107
Print preview . . . 107
Printing type definitions. . . . 108
Printing type properties . . . 108

Chapter 12. Error detection and recovery . . . 109
Error detection . . . 109
How error detection works . . . 109
Existence indicators . . . 110
Existence versus presence of components . . . 110
Error recovery . . . 112
Restart attribute . . . 112
How the Restart Attribute works . . . 112
Mapping invalid data . . . 113

Chapter 13. Distinguishable objects . . . 115
Objects in a data stream . . . 115
Type tree analyzer and distinguishable objects . . . 115
Bound types. . . . 115
Bound components . . . 116
Component of a fixed group . . . 116
Component of an explicit delimited group. . . . 116
Component of an implicit group . . . 116
Component of a choice group . . . 117
Component of an unordered group . . . 117
Group starting set . . . 117
Group unbound set . . . 117
Unbound set of a sequence group . . . 118
Initiator-distinguishable types . . . 118
Determining if a component is initiator-distinguishable from its following set . . . 118
Determining if a partition is initiator-distinguishable from its following set. . . . 118
Determining if two types are initiator-distinguishable . . . 119
Distinguishable objects of the same component . . . 120
Content-distinguishable components . . . 121
Content-distinguishable types . . . 121
Ending-distinguishable types . . . 127
Distinguishable data objects of an implicit group . . . 129
Guidelines for defining an implicit delimited sequence . . . 129
Guidelines for defining an implicit sequence that has no delimiter . . . 129
Guidelines for defining an implicit unordered group that is delimited . . . 130
Guidelines for defining an implicit unordered group that has no delimiter . . . 130
Distinguishable data objects of an explicit group . . . 130
Guidelines for defining an explicit fixed group . . . 130
Guidelines for defining an explicit delimited group. . . . 130
Objects of a choice group . . . 131
Objects of a partitioned type . . . 131
Distinguishable syntax objects . . . 131

Chapter 14. Type Tree Analyzer. . . . 133
Internal consistency . . . 133
Mapping effects . . . 134
When to Analyze Structure or Logic. . . . 134
Logical analysis . . . 134
Structural analysis. . . . 134
Error and warning messages . . . 134

Chapter 15. Utilities for XML . . . 137
XML Type Tree/Schema Synchronization utility . . . 137
XML type tree compatibility utility . . . 138
Modifying the target type tree. . . . 139

To add a type to the target tree . . . 139
To add a component to the target tree . . . 139
Any-2-XML . . . 139
Using Any-2-XML from the Type Designer . . . 140

Chapter 16. Return codes and error messages . . . 141
Type Tree Analyzer errors and warnings . . . 141
Type tree analysis logic error messages . . . 141
Logic error and warning messages . . . 151
Type tree analysis structure error messages . . . 152
Type tree analysis structure warning messages . . . 152

Notices . . . 155
Programming interface information . . . 157
Trademarks and service marks . . . 157

Index . . . 159

Chapter 1. Introduction to the Type Designer

Use the Type Designer to define, modify, and view type trees. A type tree describes the syntax, structure, and semantics of your data. The syntax of data refers to its format including tags, delimiters, terminators, and other characters that separate or identify sections of data. The structure of data refers to its composition including repeating substructures and nested groupings. The semantics of data refer to the meaning of the data including rules for data values, relationships among parts of a large data object, and error detection and recovery.

About type trees

A type tree (.mtt) defines the entire contents of at least one input that you intend to map or one output you intend to map. A type tree is the mechanism for defining each element of your data. Similar to a data dictionary, a type tree contains a collection of type definitions.

Because data definitions are defined in a type tree, you should be familiar with the specifications that define your data before attempting to create one. A data file is a simple example. The file is made up of records and each record is made up of fields. In this case, there are three kinds of data objects: a file, a record, and a field.

In a file of records, think of the data in terms of the three data objects. For example, one type defines the entire file; another type defines the entire record contained in that file. Other types in the same type tree define the data fields of the record.

Each circular icon in the type tree identifies a datatype. The tree has a root type and other types are connected to the tree through branches of the tree. The root type is the base type from which all other types stem, representing the data objects of all types in the tree. ″ROOT″ is the default name when creating a new type tree, however, you can modify any type name.

Viewing the data in the type tree window shows the different kinds of data objects, but does not show the layout or composition of the data. For example, by looking at the tree, you cannot tell that a record consists of fields.

By default, the types are in alphabetical order (ascending) in the type tree window. You can change the order of how new types appear in the type tree window by modifying type properties of the root type.

To change the order of how types are added

1. Select the root type and click the Properties button on the toolbar.
2. Go to the **Order subtypes** property and make a selection from the drop-down list.
3. Save changes.

Data composition

The type tree window does not display data composition. The group and category windows do.

© Copyright IBM Corp. 2006

Data composition views are shown for group and category types only.

Subtypes

Subtypes are created for a number of reasons like identifying distinct data object. Another reason is that specific types of an object may have different properties such as different date formats.

Think of subtypes as different ″flavors″. Ice cream flavors can be chocolate, strawberry, or vanilla. To create a type tree to represent ice cream as data, the different flavors of ice cream could be subtypes of the type **IceCream**.

The type **IceCream** is generic and describes any kind of ice cream. The type **Chocolate**, a subtype of **IceCream**, represents a certain kind of ice cream: chocolate.

A file of purchase order data may have two different kinds of records: header and detail. The type tree representing this data might have a **Record** type with **Detail** and **Header** as subtypes of **Record**.

Type tree hierarchy

The types in a tree are arranged in a hierarchy. If a type is subordinate to another type, it is called a subtype. The type on the branch stemming above a specific type is called its supertype.

Subtypes have more specific properties. For example, two different kinds of fields, **Name** and **Date**, may be defined as subtypes of **Field**.

The item type **Field** describes any field. The item types **Date** and **Name** are more specific kinds of fields. In a classification hierarchy such as this, the deeper the subtype is in the type tree, the more specific the data characteristics.

Example scenario

Imagine that a type tree represents your house. There is a type for the entire house (**House**). A type for each room (**Bath**, **Bed**, and so on), and types for the different furnishings in these rooms (**Bed**, **Chair**, and so on). Each type represents a complete object. A house is made up of rooms. Inside these rooms are beds, chairs, couches, and so on. You know this, but you cannot see it by looking at the classification hierarchy view of the type tree. You must open one of the types in the group window to see its components.

Type Designer files

Type trees have the .mtt file name extension. If the **Backup on save** option is enabled, the backup type trees are automatically created when the type tree is saved. The backup type trees have the .bak file name extension. The backup type trees are automatically saved in the same directory as the type trees.

When a type tree is analyzed, a type tree analysis message file is automatically created in the same directory as the type tree. The type tree analysis message files have the .dbe file name extension.

The following are the Type Designer file name extensions.

Extension	File type
.mtt	Type tree file
.dbe	Type tree analysis message file
.bak	Backup type tree file

Type Designer icons

Icon	Description
	Item type
	Partitioned item type
	Sequence group type
	Choice group type
	Partitioned group
	Unordered group
	Category type
	Invalid object type

Chapter 2. Type Designer basics

Type Designer startup

Select Type Designer from the **WebSphere Transformation ExtenderDesign Studio** program menu.

When the Startup window is displayed, you can choose a startup option:

- **Import a type tree**.
- **Open an existing type tree file** and browse for the type tree file (.mtt) that you want to open.
- **Create a new type tree file**.
- **Open a recently used type tree file** by selecting one or more files from the displayed file list.

  If you do not want the Startup window to appear when you open the Type Designer, enable the **Do not show this at startup** option.

To import a type tree

1. In the Startup window, select **Import a type tree**.
2. Click **OK**.

See the Type Tree Importer documentationfor information on using the Importer Wizard.

To open an existing type tree

1. In the Startup window, select **Open an existing type tree file**.
2. Click **OK**.
3. From your file structure, select the type tree file (.mtt) that you want to open.

To create a type tree

1. In the Startup window, select **Create a new type tree file**.
2. Enter a name for the root type in the **Root type name** field.
3. Click **OK**.

To open a recently used type tree

1. Select one or more files from the list of recently used files.
2. Click **OK**.

   The **Startup** window is always accessible from the **Help** menu.

User interface

The Type Designer provides a graphical user interface (GUI) to define type trees. The name of the active type tree file is displayed in the title bar.

Type tree window

Type Trees display in the type tree window. A type tree displays data types in a hierarchy. When viewing the type tree in the type tree window you cannot see certain things, such as the layout of the data or what data objects are inside of other data objects. Also, you cannot see the kind of data that appears in each type (for example, whether it is text or numeric). ″Properties Window″ , ″Item Window″ , ″Group Window″ , and ″Category Window″ sections display details of the data.

When comparing two type trees, the results are displayed in the type tree window.

As work book option

An open type trees may be viewed in a work book.

To enable the view As Work Book option:

From the **View** menu, select **As Work Book**.

Splitting the type tree view

The type tree window displays a single type tree. However, multiple type tree windows may be open at the same time. You can split the type tree into two separate views. This may be useful for copying and moving types, or for viewing different parts of very large type trees.

Initially, the splitter is on the left border of a type tree.

To split the type tree into two views:

1. Move the mouse over the left border until you see the cursor turn into a splitter.
2. Drag the splitter until two views are displayed.
3. Expand and collapse types as desired in each side of the type tree window.

Item window

From the item window you can enter restrictions for item types. The viewable columns in the item window will differ depending on the item properties. **Value**, **Character**, and **Range** are the restriction options available.

The restrictions of an item type are the values accepted as valid data objects or rejected as invalid data objects. Defining restrictions for an item type limits the valid values of that item to a specific set. For example, the item type **Department** might have a limited set of valid values, which could be individual department abbreviations. These valid departments (for example, **Dev**, **Sal**, and so on) are restrictions for the **Department** item type that would be listed in the **Include** column. Any invalid departments would be listed in the **Exclude** column.

Insert the restrictions in the appropriate column.

The columns in the item window can be resized by dragging the splitter at the top of the grid.

Group window

The group window is used to define the components of the group type. Group types represent objects composed of other objects. A component is an object that is part of a larger object.

The components of the group are defined in the order in which they appear in the data. A group's components are the objects of that group.

For example, if **Record** consists of **Name Field**, **Address Field**, and **Phone Field**, the group window for the **Record** group type would look like this:

Note: The components in the group window are in the order in which they appear in the data, not in alphabetical order.

The group window has an individual rule bar that is used to enter and view component ranges and component rules. Component rules specify a condition that must be met for a particular component to be valid.

This rule bar can be resized to view the entire component rule. The group window view is configurable to display the range column, the component number, and whether to use ellipses for the type name display. For complete information on configuring the group window, see ″Configuring the Type Designer Environment″ .

The group window is also used to enter component attributes for additional data validation.

Drag data objects from the type tree window into the component rule column in the group window. The group window displays components in a composition view that provides visual cues for the types that are components of this group.

Category window

The category window is used to define the components of the category type. The category window, like a group window, shows components of the category type that are used for inheritance purposes, to make components available to other types.

The difference between a category window and group window is that a group window has a rule column for entering component rules. Components of a category do not have component rules.

Properties Window

The Properties window is used to define and view the properties of the currently selected type. Each type has properties that define the characteristics of that data object.

To access the Properties window:

1. Select a type.
2. From the **Type** menu, choose **Properties**.

To dock or float the Properties window:

1. Right-click in any outer gray border of the Properties window.
2. From the content menu, select **Allow Docking**.

A check mark is displayed next to **Allow Docking** in the content menu when the window can be docked.

3. After selecting **Allow Docking**, toggle between a docked window and a floating window by double-clicking the border of the **Properties** window.

Menu commands and tools

Actions are performed in the Type Designer using menu commands, tools, and shortcut keys. Not all menu commands have corresponding tools and not all tools are represented in the command menus. When working with type trees, you can activate commands in several different ways:

- Choose the command from a menu.
- Right-click in any window in the Type Designer to display the context menu.
- Click the tools on the toolbar.
- Double-click any type in the type tree window to display its window (item, group, or category).
- Press `Insert` to add types.
- Press `Insert` in the rule bar to display the Insert Function dialog box.
- Press `Delete` to delete types, rules, and restrictions.

  The Type Designer commands are available as listed above; however, accessing the command from the menu has been used throughout this document for instruction purposes. Use the access method most convenient for you.

File menu

The **File** menu provides commands that are generally available in Windows applications. To access the **File** menu using the keyboard, press `Alt F`.

Command	**Key stroke**	**Description**
New	Ctrl+N	Creates a new type tree (.mtt)
Open	Ctrl+O	Opens an existing type tree
Close	Alt+F, C	Closes the selected type tree
Save	Ctrl+S	Saves the selected type tree
Save As	Alt+F, A	Saves the selected type tree under a new name
Source Control	Alt+F, l	
Create Project	Alt+F, l+C	Creates a new project
Open Project	Alt+F, l+O	Opens a project
Get Latest Version	Alt+F, l+G	Gets the latest version of the selected file
Check Out	Alt+F, l+e	Checks out the selected file
Check In	Alt+F, l+I	Checks in the selected file
Undo Check Out	Alt+F, l+U	Undoes the check out of the selected file
Add to Source Control	Alt+F, l+A	Adds selected file to source file
Remove from Source Control	Alt+F, l+R	Removes selected file form the source control
Show Workfiles	Alt+F, l+f	Shows all work files in the project
Add Workfiles	Alt+F, l+W	Adds selected files to source control
Show History	Alt+F, l+H	Shows the history for the selected files

Command	Key stroke	Description
Show Properties	Alt+F, l+P	Shows properties for the selected file
Refresh Status	Alt+F, l+R	Refreshes the status on all the files
Type Tree Differences		Displays type tree differences
Print	Ctrl+P	Prints all or part of a type tree
Print Type Definition	Ctrl+D	Prints type definitions
Print Preview	Alt+F, V	Previews a print job
Print Setup	Alt+F, R	Specifies printer options
Recently Used File List	Alt+F, 1 or Alt+F, 2 and so on	Opens the type file selected from this recently used file list
Exit	Alt+F, X	Closes the Type Designer

Edit menu

The **Edit** menu provides editing commands that are generally available in Windows applications. It also contains the Find command that is useful when locating types.

To view the **Edit** menu by using the keyboard, press `Alt E`.

Command	Key stroke	Description
Undo	Alt+Backspace	Allows the user to undo multiple operations performed.
Cut	Ctrl+X	Cuts the selection and places a copy on the clipboard.
Copy	Ctrl+C	Copies the selection to the clipboard.
Paste	Ctrl+V	Inserts the contents from the clipboard.
Find	Ctrl+F	Displays the Find dialog box in which you can locate specified information in the active window.
Replace	Ctrl+H	Displays the Replace dialog box in which you can replace specified information in the active window.

Undo commands

The Undo command allows the user to reverse multiple operations performed. There is no limit for the number of actions that can be reversed.

The following operations can be reversed when using the Type Designer.

General	Group and Category Windows	Item Window
Delete	Add	Add
Add	Delete	Delete
Cut	Cut	Cut

General	Group and Category Windows	Item Window
Paste	Paste	Paste
Rename	Edit	Edit
Edit Moving Reorder Merging types	Move components	Move restrictions

View menu

Use the **View** menu to control the display in your Type Designer environment.

Using the keyboard, you can view the **View** menu by pressing `Alt V`.

Command	Key stroke	Description
Toolbars	Alt+V, T	Displays the Customize dialog box for toolbars
Status Bar	Alt+V, S	Shows or hides the status bar
As Work Book	Alt+V, W	Toggles the view of the open type trees as a work book

Type menu

Use the **Type** menu for choices of commands that apply to types.

Using the keyboard, you can view the **Type** menu by pressing `Alt T`.

Command	Key stroke	Description
Add	Insert	Adds a new type below the selected type
Delete	Delete	Deletes the selected type
Properties	Alt+Enter	Displays the Properties window or updates the open Properties window for the selected type
Copy	Alt+T, C	Copies the selected type
Move	Alt+T, M	Moves the selected type
Merge	Alt+T, G	Merges the selected type to another type
Reorder Subtypes	Alt+T, R	Reorders subtypes of the selected type
Open	Alt+T, O	Opens the window of the selected type (item, group, or category)
Expand All Subtypes	Alt+T, M or Alt+X	Expands all subtypes of the selected type
Select All Subtypes	Alt+T, S	Selects all subtypes of the selected type

Tree Menu

The **Tree** menu includes choices of commands that apply to type trees.

Using the keyboard, you can view the **Tree** menu by pressing `Alt R`.

Command	Key stroke	Description
Export	Alt+R, E	Exports the selected type tree.
Import	Alt+R, I	Displays the Importer Wizard dialog box, which allows you to create type trees from metadata source files.
Close	Alt+R, C	
All Windows	Alt+R, C+W	Closes all windows
All Group Windows	Alt+R, C+G	Closes all group windows
All Item Windows	Alt+R, C+I	Closes all item windows
All Category Windows	Alt+R, C+C	Closes all category windows
Analyze	Alt+R, L	
Structure Only	Alt+R, L+S	Analyzes the type tree structure only
Logic Only	Alt+R, L+L	Analyzes the type tree logic only
Logic and Structure	Alt+R, L+A	Analyzes the type tree logic and structure
View Results	Alt+R, V	Displays the analysis results for the selected type tree.

Component menu

The **Component** menu includes choices of commands that apply to components.

Using the keyboard, you can view the **Component** menu by pressing `Alt C`.

Command	Key stroke	Description
Insert Function	Alt+C, F	Displays the Insert Function dialog box in which you can insert a function into a component rule
Insert Symbols	Alt+C, Y	Displays the Symbols dialog box in which you can insert a symbol
Identifier	Alt+C, T	Toggles the identifier attribute for the selected component
Restart	Alt+C, N	Toggles the restart attribute for the selected component
Sized	Alt+C, Z	Toggles the sized attribute for the selected component
Include Self in Size	Alt+C, E	Specifies that the value of the component with the sized attribute includes the size of itself
Set Range	Alt+C, R	Displays the Set range dialog box in which you can set the range(s) for the selected component(s)
Insert	Alt+C, I	Inserts a component into the selected window

Command	Key stroke	Description
Delete	Alt+C, D	Deletes the selected component(s)
Delete All	Alt+C, A	Deletes all of the components from the selected component list
Go To	Alt+C, G or Ctrl+G	Displays the Go to dialog box in which you can specify the component number to receive current focus
Save	Alt+C, S	Saves the selected window

Restriction menu

The **Restriction** menu includes choices of commands that apply to items.

Using the keyboard, you can view the **Restriction** menu by pressing `Alt S`.

Command	Key stroke	Description
Insert Symbols	Alt+S, Y	Displays the Symbols dialog box in which you can insert a symbol
Value NOT In Range	Alt+S, V	Identifies a value as not being included in the range.
Insert	Alt+S, I	Inserts a restriction into the selected window
Delete	Alt+S, D	Deletes the selected restriction
Delete All	Alt+S, A	Deletes all restrictions in the selected window
Go To	Alt+S, G	Displays the Go To dialog box. Enter the restriction number you want to go to.
Propagate Restrictions	Alt+S, G	Propagates restrictions with an option to overwrite or append
Save	Alt+S, S	Saves the selected window

Tools menu

The **Tools** menu provides options to customize your Type Designer environment.

Using the keyboard, you can view the **Tools** menu by pressing `Alt L`.

Command	Key stroke	Description
Shortcuts	Alt+L, S	Displays the Shortcut Keys dialog box in which you can assign short cut keys or key combinations to specific Type Designer operations.
Options	Alt+L, O	Displays the Options dialog box in which you can configure your Type Designer environment.

Window menu

The **Window** menu contains commands to control open windows.

Using the keyboard, you can view the **Window** menu by pressing `Alt W`.

Command	Key stroke	Description
Close All	Alt+W, L	Closes all open windows
Cascade	Alt+W, C	Arranges all open windows so that they overlap in a descending pattern
Tile Horizontally	Alt+W, H	Arranges all open windows as horizontal, non-overlapping tiles
Tile Vertically	Alt+W, V	Arranges all open trace windows as vertical, non-overlapping tiles
Arrange Icons	Alt+W, A	Neatly arranges all minimized windows at the bottom of the main window
Recently Used Window List	Alt+W, 1 or Alt+W, 2 and so forth	Makes the selected file the active window. A check mark is displayed next to the file that is the active window

Help menu

The **Help** menu includes choices that display information about the Type Designer.

Using the keyboard, you can view the **Help** menu by pressing `Alt+H`.

Status bar

The status bar displays contextual information such as descriptive messages about a selected menu command or tool or information about the current state of an operation. The message `Ready` indicates that the Type Designer is waiting for your next action.

Configuring the Type Designer environment

The Type Designer environment can be configured to accommodate your preferences in your working environment. For example, you can:

- Specify various user interface options (font, line appearance, dialog box display, and so on)
- Select the tools to display
- Change the look of the tools on the toolbar
- Assign shortcut keys

See the Design Studio Introduction documentation for details on customizing the toolbar and creating shortcut keys.

Options

From the **Tools** menu, select **Options**. The Options dialog box is displayed and you can select choices that represent various aspects of the Type Designer environment and configure them as desired.

General options

Similar to user preferences, the General options are general customization settings for the Type Designer environment.

- **Auto-Save Tree(s)**: Specify the time interval that you want to have your type trees automatically saved.
- **Show Banner**: Show/hide the graphical banner across the top of the interface. (This option is enabled by default.)
- **Backup on save**: Enable this option to create a backup copy of the type tree with the file extension **.bak** when each type tree file is saved. (This option is enabled by default.)
- **Commit changes with**: Use this option to select a key stroke combination for committing changes made in the Type Designer. For example, if the **Enter Key** option is enabled, when you press **Enter** after typing a restriction in an item window, the text is entered and the next component cell is selected.
  - **Enter & Tab**: When enabled, the **Enter** key and **Tab** key can be used to make changes to components.
  - **Enter Key**: When enabled, the **Enter** key can be used to commit changes to components. You can create a hard return in any component rule or edit window by pressing **Ctrl Enter**.
  - **Tab Key**: When enabled, the **Tab** key can be used to commit any changes to components. You can create a hard return in any rule or edit window by pressing the **Tab** key.
- **Rule Differences - Ignore white spaces and tabs**: Use this option to specify whether to ignore white spaces and tabs in component rules. (This option is enabled by default.)
- **Default Root Name**: Sets a default type name for the root type when creating new type trees in the Type Designer. (ROOT is the default name.)

Type tree

Use the Type Tree options to define the appearance of the type tree window.

- **Font**: Use this option to change the font attributes in the type tree. Click the Font button to define the Font, Font style, Size, Effects, or Color. Arial 10-point is the default setting.
- **Lines**: You can change the appearance of the lines in the type tree window.
  - **None**: When enabled, lines do not appear in type trees.
  - **Solid**: When enabled, lines appear solid in type trees.
  - **Dotted**: When enabled, lines appear dotted in type trees.
- **Show tool tips**: When enabled, the full name of the type is displayed as a tool tip. This is extremely helpful when the window is sized too small to see the entire type name, type property, or interface label.

Group window options

The Group Window options define the appearance of the group window. There are two tabs: **General** and **Color Coding**.

General Tab

The following group window options are located on the **General** tab:

- **Font**: The font for text in the group window can be changed. Click the Font button to define the font attributes (font type, style, size, effects, and color).
- **Show grid lines**: When enabled, the grid lines are displayed in the group window.
- **Line color**: Choose the color of the grid lines in the group window.
- **Show range column**: When enabled, a column is displayed in the window that displays the range assigned to each component.
- **Show component number**: When enabled, a sequential number is displayed to the left of each component in the window.
- **Use ellipses**: Use this option to control the appearance of object names in the group window.
  - When this option is enabled, the abbreviated short object name displays in the component rule.
  - When this option is disabled, the full name of the type is displayed in the component rule.

Color Coding Tab

The following color-coding options are available for the group window:

- **Use color coding:** Enable option to use color coding for component rules.
- **Show errors with background**: Enable option to highlight invalid rules with a background color.
- **Color Specifications**
  - The first drop-down list contains a list of options for which you can specify a color. For example, the default setting for map names is red.
  - Use the color pallet to create custom colors.

Item and category window options

The Item Window and Category Window options define the appearance of the item and category windows. The following options are available:

- **Font**: Select the font, font style, font size, effects, and color.
- **Show grid lines**: When enabled, grid lines are displayed.
- **Line color**: Select the grid line color for the applicable window.
- **Show Component Number** (Category Window Only): When enabled, the component number displays next to a component in the Component column.

Analysis results

Use the Analysis Results option to specify the font used in the analysis results window after a type tree is analyzed.

Confirmations

Use the Confirmations options to select the actions for which you want a confirmation dialog box displayed before completing the action.

Type properties

Use the **Type Properties** option to define the appearance of the font in the Properties window.

Chapter 3. Working with type trees

Creating type trees

The following list outlines the process for creating type trees:

- Identify the data objects in your data and define each piece of data that you intend to map.
- Create types for each data object in your data.
- Define the properties of each type.
- Create component lists.
- Define component rules, if needed.
- Define item restrictions, if needed.
- Analyze and save the tree.

To create a new type tree:

Before you actually create a type tree, you should be familiar with the specifications that define your data.

From the **File** menu, choose **New**.

A new type tree appears with a root type named **Root**. The default name of each new type tree file is **TypeTree** followed by a number.

Creating types

Types are always created as a subtype of the currently selected type. A new type tree has a single root type. All types are added as subtypes of the root type and then as subtypes of other types.

By default, types at the same level of the type tree are listed in alphabetical order without regard to class or the order in which they were created. A *level* consists of all of the types that have the same supertype.

To create a type:

1. Select the type under which you want to add a type.
2. From the **Type** menu, choose **Add**.

   The default confirmation settings include adding types; therefore, you will be asked if you want to add a new type.
3. Click **Yes**.
4. Enter the name of the type and press **Enter** or click outside of the type.
5. Define the properties of the type in the Properties window. From the **Type** menu, choose **Properties**.
6. Define all type properties such as **Name**, **Class**, and **Description**.
7. After defining the properties, close the Properties window.

   The type becomes a subtype of the type currently highlighted.

© Copyright IBM Corp. 2006

Viewing types

If a type has subtypes, the subtypes beneath it can be hidden or viewed.

Expanding a Type

In the type tree window, expand a type (when there is a plus sign next to it) to view the contents.

To expand all types:

From the **Type** menu or by right-clicking on the type, select **Expand all Subtypes**. Or, with the type selected, press `Alt X`.

Data content

The entire contents of your data must be defined. Define the input data so that each data object of the source data is identified. Define the output data according to your output specifications.

How specifically you define the data is up to you. For example, if there is a large section of data that you want to process quickly, define it loosely as a chunk of text.

In the Type Designer, the data content is not defined as the source data or target data. Data input and output definitions are defined in the Map Designer. The Type Designer simply contains the definition of your data.

Data objects

A data object is a complete unit that exists in your input or is built on output. A data object may be simple (such as a date) or complex (such as a purchase order). A data object is some portion of data in a data stream that can be recognized as belonging to a specific type. When you create types, identify all the objects that make up the data: input objects and output objects.

Types

A type defines a set of data objects that have the same characteristics. For example, the type **Date** can be defined as representing data objects in the form MM-DD-YY. The type **CustomerRecord** can be defined as representing data objects, each of which consists of a **Company**, **Address**, and **Phone** data object.

Classes

A type is classified according to whether or not it consists of other objects. The color of the type icon indicates the class of the type. Each type in a type tree must be defined in one of three classes: item, group, or category.

Item

An item type represents a simple data object that does NOT consist of other objects.

Group

A group type represents a complex data object that consists of other objects.

For example, **FullName** is a group type that contains the components: **FirstName**, **MiddleInitial**, and **LastName**.

Category

A category type is used for inheritance and for organizing other types in a type tree.

For example, you might have a category named **OrderField** to organize the different kinds of order fields in your type tree.

Components

Group objects consist of other objects. An object that is part of another object is called a component. Components are added and viewed in the group window. Groups and categories may contain components.

For example, the item type **Field** is a component of the group type **Record**.

Exporting a type tree

You can export a type tree or a portion of a type tree to a document file containing a script of commands in XML format (.mts file extension). The .mts file can then be used as input to the Type Tree Maker, which creates a type tree or portion of a type tree. For more information on the Type Tree Maker documentation, see the Type Tree Maker documentation.

To export a type tree or portion of a type tree:

1. Select the type you want to export. (Select the root if you want to export the entire type tree.)
2. From the **Tree** menu, choose **Export**.

   The Save As dialog box is displayed.
3. Choose the directory in which you want the .mts file to be placed.
4. Enter the file name you want for the .mts file.
5. Click **OK**.

   For example, suppose you want to produce a document file that generates all of the **InventoryData** types in the tree Product Data.mtt. Select the type **InventoryData** and then choose **Export** from the **Tree** menu.
   a. Enter the name of the document file in the Save As dialog box.

      The default name is the name of the tree followed by an .mts extension.

The subtree of the selected type is exported. If you select a type other than the root, it is assumed that you are adding to an existing tree and the document file will have a <OPENTREE> command. If you select the root type, it is assumed that you are generating a new tree and the document file has a <NEWTREE> command.

An example of a type tree of an export script and an example map that converts an exported restriction list to a cross-reference table is included with the Design Studio examples.

Exporting a type as a schema

You can export the components of a type as an XML Schema. The resulting schema (.xsd) takes on the name of the type and is created in the type tree directory.

To export a type as a schema:

1. From the type tree window, select a type to convert.
2. Choose **Tree** → **Export As Schema**.

   When the type converts successfully, you are prompted to view the schema file. You can also view the schema (*typetreename_typename*.xsd) from the type tree directory. For example, if you convert the **Doc** group type of the ipo.mtt type tree, the name of the new schema file is ipo_Doc.xsd.

Type to XML conversion

An exported schema is designed according to XML Schema specification. Category types are not exported to a schema. The types that have subcomponents become ″complexTypes″. Item types become ″elements″.

The names of the generated schema types have the corresponding type name plus a unique identifier, which is a numerical suffix. Types that are declared more than once within a type tree, such as unordered groups, have the numerical identifier plus the ″type″ suffix. In the following schema excerpt, the **Global** group in the type tree was converted to an element with the name of **Global89**. The **AttrList** unordered group was converted to **AttrList48Type**.

```
<xs:element name="Global89">
          <xs:complexType>
            <xs:choice>
              <xs:element name="purchaseOrder86">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="AttrList48" type="AttrList48Type" />
                    <xs:element name="Pair153" type="Pair153Type" />
                    <xs:element name="Pair258" type="Pair258Type" />
                    <xs:element name="Pair361" type="Pair361Type" />
                    <xs:element name="Pair466" type="Pair466Type" />
                    <xs:element name="Pair569" type="Pair569Type" />
                    <xs:element ref="PCDATA7" />
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
<xs:complexType name="AttrList48Type">
    <xs:all>
      <xs:element name="orderDate" type="xs:dateTime" />
    </xs:all>
  </xs:complexType>
```

Type tree differences

The type tree differences feature allows you to compare two type tree files.

Two type trees are considered different when:

- Any of the types are different.

- Any of the types that exist in one type tree does not exist in the second type tree.
- When the order of the subtypes within a subtype is different.

A type is different when:
- Any of the properties are different.
- Any of the components are different.
- Any of the restrictions are different.

A restriction is different when:
- The restriction value is different.
- The description is different.

To compare type tree files:
1. From the **File** menu select **Type Tree Differences**.
   The Select First File dialog box is displayed.
2. Select the first type tree (.mtt) file and click **Open**.
   The Select Second File dialog box is displayed.
3. Select the second type tree (.mtt) file and click **Open**.
   The Type Tree Difference Analysis dialog box is displayed.

As an example, when the **Properties** tab is in red text, that indicates there is a difference in the properties of the selected type. When you scroll through the properties, you can see where the difference exists. When there are no differences in the properties, components, and restrictions of the selected type, the text in the window is black.

Creating a type tree exercise

The following sample data file represents orders from a store to an office supply company. The data is organized in rows and columns.

pencil,#2,500

pencil,#3,467

pen,blue,1000

paper,8X11,525

pen,black,1000

paper,graph,400

The specifications reveal the following information:
- Each row contains a product column (text) followed by a style column (text) and then by a quantity column (integer).
- The minimum size of each column is one byte. There is no maximum size.
- The number of rows in the order is infinite.
- A carriage return/line feed is displayed at the end of each row.

After reading the specifications, make a list of the types to be created. Your list might look something like this:

Type Name
Description

Order The type that represents the entire input or output. In this example, a type representing the entire order.

Row The type that represents the rows in the orders. The specifications show that each **Order** contains rows.

Column
The type that represents the product columns. The specifications show that each **Row** contains columns.

Column subtypes
The types that represent the different kind of columns. The specifications show three kinds of columns: **Product**, **Style**, and **Quantity**. Create these types as subtypes of the **Column** type.

Create a new type tree and start creating the types listed above.

To create the Orders type tree:

1. From the **File** menu, choose **New**.

   The new type tree is created.
2. From the **File** menu, choose **Save**.
3. For this example, save this type tree as **Orders**.

   After the type tree is created and saved, start creating the types.

To create example types:

1. Select the **Root** type.

   This is the type under which all other types are added.
2. From the **Type** menu, choose **Add**.

   A new type confirmation dialog box is displayed.
3. Click **Yes**.

   A new type is created with the default name **NewType1**.
4. Enter the name **Order** and either press **Enter** or click outside of the type.
5. Select the **Order** type to define its properties.
6. From the **Type** menu, choose **Properties**.
7. The **Order** type is a group type because it is a complex object consisting of other data objects. For the **Class** property, choose **Group**.

   The **Order** type you just created is displayed on a branch stemming down from the root.
8. Create the **Row** type. **Row** is also a group because it represents a complex data object consisting of other objects.

The **Row** group type should be on the same level as the **Order** type. Ensure the **Root** type is selected when you create the **Row** type or move the type.

1. Create the **Column** type, which is an item type.
2. As subtypes of **Column**, create the different kinds of columns: **Product**, **Quantity**, and **Style**.

Your type tree now has all of the types in your data.

Opening an existing type tree

To open an existing type tree, either double-click the type tree .mtt file in any directory listing or choose the Open command on the **File** menu.

To open an existing type tree

1. From the **File** menu, choose **Open**.
2. Browse your file structure and select the type tree you want to open.
3. Click **Open**.

Chapter 4. Type properties

The properties of a type define the characteristics of the data objects of that type.

For *item* types, properties define whether that item is text, a number, a date and time, or a syntax value. Properties include such characteristics as size, pad characters, and justification.

For *group* types, the properties are related to the format of that group. The format of a group may be explicit or implicit. In addition, type properties include syntax objects that appear at the beginning or end of the object, as well as release characters.

Defining type properties

You can individually define each type property. When you choose to view the properties of a type, the associated Properties window is displayed. This window contains the following:

- Definition of a type's **Class**, **Group** or **Item** properties
- Type **Initiator**, **Terminator**, and **Release** character
- A list of where a type is used in the definitions of other types

From the **Properties** window you can also propagate (pass along properties) to other types.

To access properties of a type

1. From the Navigator, select a type.
2. Do one of the following:
   - From the **Type** menu, choose **Properties**.
   - Right-click and select **Properties** from the context menu.
   - Press `Alt Enter`.
   - Click the Properties tool on the toolbar.

     The Properties window is displayed.

     The properties of a type are accessible from any of the methods listed above. However, the menu method is used throughout this document for instruction purposes.

Once the Properties window is open, click on any other type to view the associated properties. For example, the image below displays the properties for type **Attribute**.

To define the properties of a type

Right-click any type property in the Properties window and choose **Help** from the context menu to view a description of that property.

1. Select the type in the type tree.
2. From the **Type** menu, choose **Properties**.

   The Properties window is displayed.

© Copyright IBM Corp. 2006

3. Enter the type **Name**.
4. Select the type **Class** from the drop-down list.
5. Enter the type **Description**.

   The description of a type is recommended but not required.
6. Define other properties as needed. To define a certain property for the selected type, enter or select the property value in the **Value** column.

   You might need to expand each property to view all associated values of that property. For example, to define the type initiator as a literal value, for **Initiator**, select **Literal** from the drop-down list and then expand the **Initiator** property to define the literal initiator value.
7. After you have completely defined properties for this type, select another type to define or close the Properties window.
8. Save changes.

Basic type properties

The following type properties are common to categories, groups, and items:
- Name
- Class
- Description
- Intent
- Partitioned
- Order Subtypes
- Initiator
- Terminator
- Release Characters
- Empty
- National Language
- Document Type
- Where Used

Item-specific properties are discussed in more detail in ″Item Properties″. Group-specific properties are discussed in more detail in ″Group Properties″.

Name

The name of the type should be as descriptive as possible and reflect the information that the type represents.

A type name must conform to the following guidelines:
- A type name cannot be more than 256 characters long.
- A type name cannot be a reserved word or contain a reserved symbol. For a list of reserved words, refer to Design Studio Introduction documentation.
- A type name cannot contain only digits and/or periods.
- A type name may contain numbers and special characters.
- A type name cannot contain spaces. Use an underscore instead.
- A type name can contain the following:
  - Letters
  - Digits

- ASCII characters 128-255
- Special characters: #, ~, %, _, ? or \
- Double-byte characters (Japan edition only)

- A type name cannot have the same name as another type on the same level in the same type tree.

  A type name can be up to 256 characters in length, but use a short type name if possible. The full path name of each type is used in map rules in the Map Designer.

Class

The class of a type describes whether the type represents a simple data object (item), a complex data object (group), or whether is used to organize types (category). Define the class of the selected type by selecting the class from the drop-down list in the **Value** column.

Property
    Description

Category
    Categories organize types that have common properties. Each category has a set of item properties and a set of group properties. Subtypes of the category inherit these properties.

    A category does not define data objects in detail and it does not represent data to be used in a map. You never map a category type, so a category type does have components. A category type may be a component of a partitioned group, after a component with the identifier attribute. A category cannot be a component of a non-partitioned group.

Item
    The item type represents a simple data object that does not consist of any objects. An item does not have components.

Group
    The group type represents a complex data object that consists of components.

To change a type's class, select the type and choose the desired class for the **Class** property in the Properties window.

If you change a type to an item, all the types in its subtree become items. If you change a type to a group, all the types in its subtree become groups. This is because the type tree has a classification hierarchy and all the nested subtypes of an item must be items. Similarly, all the nested subtypes of a group must be groups.

Description

Use this property to record a brief description of the type. The description entered is for informational purposes only. It is not used for identifying data objects at runtime.

It is good practice to add a meaningful description of the type when you are defining it.

Intent

Indicates whether the type is a general type or an XML type.

The type can only be an XML type if the type tree was created using the XML DTD Importer or the XML Schema Importer.

Refer to ″XML Properties in the Type Tree″ for more information about type properties created from XML.

Validate as

The **Validate As** property contains a WebSphere Transformation Extender system generated value with an ″XML_″ prefix. These values resemble a data definition encountered in the XML Schema or DTD during the import process.

Some values are easily recognizable from the XML grammar structure, while others might be unfamiliar. For example, XML_ATTRIBUTE is easily recognizable from an XML grammar structure but XML_BODY is not. XML_BODY actually represents a global element.

The following table lists possible values for the **Validate As** property:

Value **Description**

XML_ATTRIBUTE
An XML attribute, including <anyAttribute> items.

XML_ATTRIBUTEGROUP
A global (named) attribute group <attributeGroup>. (Not applicable for DTD Importer.)

XML_ATTRIBUTELIST
Unordered groups of attributes.

XML_BODY
The global element choice when the user does not select a particular root element.

XML_COMMENT
The XML comment, which is a single item per type tree.

XML_COMPLEXTYPE
A global (named) complex type that appears in the XML Schema as the <complexType> construct. (Not applicable for DTD Importer.)

XML_CONTENT
Generic XML content types.

XML_DOCUMENT
The complete XML document.

XML_ELEMENT
An XML element, which includes <any> element items.

XML_GROUP
A global (named) group.

XML_PCDATA
Parsed character data (PCDATA) in mixed content, which is a single item per type tree.

XML_PI
The XML processing instruction, which is a single item per type tree.

XML_SIMPLETYPE
A global (named) simple type <simpleType>. (Not applicable for DTD Importer.)

XML_XMLDECL
Represents the prolog of the XML document (the group of version, encoding, and standalone attributes).

Partitioned

If the data of this type can be divided into mutually exclusive subtypes, it can be partitioned. For information about partitioning, see ″Partitioning″.

Property
Description

Yes Partitioning is enabled for this type.

No Partitioning is not enabled for this type.

Order subtypes

Choose the method in which the subtypes of this type will be added or viewed in the type tree.

Property
Description

Ascending
Add and view subtypes of this type in alphabetic or numeric order.

Descending
Add and view subtypes of this type in reverse alphabetic or numeric order.

Add First
Add subtypes of this type to the top of the type list.

Add Last
Add subtypes to the bottom of the type list.

To manually reorder the subtypes in a type tree, the **Order Subtypes** property must be either **Add First** or **Add Last**.

Initiator

An initiator is a syntax object that appears at the beginning of a data object. Defining an initiator for a type specifies that when data of that type appears, the initiator appears at the beginning of the data object. The initiator becomes part of the data type definition.

Property
Description

None There is no initiator.

Literal The initiator is literal. Expand the **Initiator** property to enter the literal initiator value and data language of the initiator value.

Variable
Allow for possible values. Expand the **Initiator** property to define the variable terminator **Default**, **Item**, and **Find** properties.

If each record begins with an asterisk `*`, define the `*` as a literal initiator of the record type.

The following data represents the classes in a college English department. An asterisk `*` is displayed at the beginning of each **ClassRecord**.

For information about other symbols used in the **Initiator Value** field, refer to the Type Tree Importer documentation.

Terminator

A terminator is a syntax object that appears at the end of a data object. The terminator becomes part of the data type definition.

Property
Description

None There are no terminators.

Literal A constant value. Expand the **Terminator** property to define the literal terminator **Value**.

Variable
Allow for possible values. Expand the **Terminator** property to define the variable terminator **Default**, **Item**, and **Find** properties.

For example, a carriage return/linefeed (`CR/LF`) at the end of a record is the record's terminator.

Generally, if the data ends with a given syntax object, you should define a literal terminator. For example, it is very common to define a `CR/LF` as a terminator of a record when you know that the record always ends with a `CR/LF`, regardless of where the record appears.

Release characters

A release character is a one-byte character in your data indicating that the character(s) following it should be interpreted as data, not as a syntax object. The release character is not treated as data, but the data that follows it is treated as actual data.

Building release characters for output data

If a release character is defined for a type, a release character is inserted for each occurrence of a syntax object in the data of any item contained in that type.

Guidelines for using release characters

Guidelines for using release characters include the following:

- Release characters apply to character data only, *not* binary data.
- Characters defined as pad characters are not released.
- The maximum size of an item does *not* include the release characters.

Release character example

The group type **Record** type has a literal delimiter of **,** and a release character of ?. The group type **Record** has three item components. Data for the record looks like the following:

Miller?, MD,Harkin Hospital,1996

The **?** releases the comma after **Miller**.

In the first field, the actual data value is **Miller, MD**. Because the comma appears as part of the data, it is necessary to have the release character **?**, which indicates that the **,** following it is data, *not* a delimiter. This data would be interpreted as the following:

Data for component #1: `Miller, MD`

Data for component #2: `Harkin Hospital`

Data for component #3: `1996`

A release character can apply to a delimiter, a terminator, or even the release character itself.

If a release character appears in the data and it is not followed by a syntax item, the release character is ignored.

Empty

The **Empty** property provides alternative type syntax for groups or items when they have no data content.

When the **Empty** property is specified for a type and there is no data content, the **Empty** syntax is displayed. For example, this can be used for XML data that contains either start and end tags or an empty tag.

You can use the **Empty** type property instead of syntax object items to potentially improve the type tree runtime processing time (during data validation).

The following options are available for the **Empty** property.

Value	Description
None	Default setting. Select **None** if you do not have an initiator, terminator, or release character.
Literal	A constant value. When **Literal** is selected, the **Value**, **Ignore Case**, **Required**, and **National language** sub-fields become available. You can use only one literal to indicate a zero-length data item.

Empty type property example

Consider an XML element called `Comment`. Presuming that this element has a simple content of an arbitrary length, it can be represented in a type tree as a text item, with the initiator value `<Comment>` and the terminator value `</Comment>`.

This item can then be used to validate the following data:

```
<Comment>Some comment...</Comment>
```

The resulting value for the item will be:

```
Some comment...
```

As another example, the following data is also successfully validated:

```
<Comment></Comment>
```

The resulting value for the item will be a zero-length string.

The problem occurs with the following data: `<Comment/>`

From the XML perspective, this data is equivalent to the `<Comment></Comment>` data shown above, as it represents the **Empty** element `Comment`. However, from the type tree perspective, this data is invalid because it does not start with `<Comment>` and does not end with `</Comment>` as required by the initiator and terminator item property values.

This is where the **Empty** property is useful. By defining the **Empty** property for the item to have a value of `<Comment/>`, it will be possible to validate the data `<Comment/>`.

Even if the data does not match the syntax described by the initiator and terminator properties, it will be validated because it will match the **Empty** property value. It will be treated as an *empty item*, that is, the resulting value for the item will be a zero-length string, similar to the example shown previously.

National language

The **National language** default value is **Western**. For initiator, terminator, and release character **Literal** values, you can optionally specify a ″Data language″.

National language options for WebSphere Transformation Extender include Japanese, French, German, Italian, Spanish (Spain), Portuguese (Brazil), Korean, Chinese (traditional and simplified).

For initiator, terminator, and release character literal values, you can optionally specify a data language.

Document Type

The **Document Type** setting is a component of Document Verification. A requirement of using Document Verification is to set the **Document Type** properties of the associated type tree object.

To use the Document Verification option for XML documents, you must do the following:

- Set the **Document Type** property to **XML**.
- Set the **Document Type Metadata** value to **Schema** or **DTD**.
- Specify the **Location** of the DTD or Schema file.

Property
Description

Default
This is the default value, which indicates a non-XML document type.

XML Indicates an XML document type. When this value is set to **XML** and the appropriate **DocumentVerification** map settings are in place, the XML document will be validated by an external program in addition to the standard data validation process.

See the Map Designer documentation for information about using the Document Verification option.

Where used

Where Used shows how and where the type is used within other types in the tree. For example, the **Where Used** property shows if the type is used as a delimiter, component, and so on.

XML properties in the type tree

When you open a type tree in the Type Designer after importing it from a DTD or XML Schema, the XML-specific properties are displayed as read-only fields.

For each type in the resulting type tree, the **Intent** property is either **General** (indicating non-XML) or **XML**. All category types created during the import process are considered non-XML properties. All group and item types created during the import process are considered XML properties.

Any types that you manually add to the XML type tree are considered non-XML and have **General** intent. In such a case where there are both XML and non-XML types present (such as EDI data), the appropriate method of validation is determined automatically.

xsi:type
The **xsi:type** attribute is not supported in WebSphere Transformation Extender. Only the XML Schema or DTD file stored as an attribute in the **Doc** group is used for validation at runtime, overriding the value specified with **xsi:noNamespaceSchemaLocation** or **xsi:schemaLocation** in the XML instance document.

See the Type Tree Maker documentation for information about creating type trees by importing XML Schemas and DTDs.

See ″Utilities for XML″ for information about tools that can assist you with upgrading your XML type trees and maps to WebSphere Transformation Extender 8.0.

Support for XML constructs

WebSphere Transformation Extender validation supports the XML constructs discussed in this section.

Character Data
During validation, character data is mapped by the XML parser to WebSphere Transformation Extender types. This includes both parsed character data (PCDATA) and unparsed character data (CDATA).

Comments and Processor Instructions
XML comments and processing instructions (PI) are mapped to floating components in type trees.

Namespaces
The XML Schema importer supports the specification of arbitrary prefixes for namespaces declared in the input grammar.

XSDL Hints
The **xsi:schemaLocation** and **xsi:noNamespaceSchemaLocation** attributes are collectively known as XML Schema Definition Language (XSDL) hints. These attributes specify the location of the XML Schema(s) that the XML parser uses for validation.

An XML Schema allows either or both of these attributes to appear within any element tag, but the XML parser only respects the location values when the XSDL hint is specified in the root element of the schema.

The **Doc** group type of type trees that are created with the XML Schema or XML DTD importer contains the location of the DTD or XML Schema that is used for XML validation. The **Intent Validate As → Location** property is a modifiable field. If you remove the file location from this field, the validation process looks to the XSDL hints specified in the XML document .

When XSDL hints are not present in either location (the **Location** field in the type tree or the XML instance document), validation fails.

Empty Elements
Only one set of initiators and terminators for the Empty element is required for validation.

Nillable Elements
A nillable element can have one of three states: absent, present with content, or present with nil content. Both the DTD and XML Schema allow the definition of optional elements in the instance document. Additionally, the XML Schema permits nillable elements where the content can be empty when it contains an **xsi:nil** attribute with a value of ″true″, despite the fact the element's content is mandatory.

The XML Schema Importer can create the content of a nillable element within a group with a range of (0:1), which makes the element content optional.

Mixed Content
When parsing elements with a mixed content model (containing both character data and child elements), the DTD and XML Schema importers generate text items within the mixed content (instead of character sequences).

Regular Expressions
An XML Schema allows the restriction of the values of simple types based on regular expressions or pattern facets. During XML validation, the parser enforces the pattern facet. The XML Schema Importer fills the appropriate type properties with pattern facets encountered in the input schema, which are viewable in the Type Designer.

XML schema datatypes

This section describes the type tree constructs that correspond to XML Schema datatypes.

The following table lists some common XML Schema datatypes with their corresponding type property values in the Type Designer.

For detailed information about XML attributes and elements as type properties in the type tree, refer to the Type Tree Importer documentation.

XML Schema Datatype	Type Property	Value
simpleType	Intent > Validate As	XML_SIMPLETYPE
complexType	Intent > Validate As	XML_COMPLEXTYPE
element	Intent > Validate As	XML_ELEMENT
attribute	Intent > Validate As	XML_ATTRIBUTE
group	Intent > Validate As	XML_GROUP

Simple types

Elements that have assigned simple types have character data content but not child elements or attributes.

Simple type elements are represented in the Type Designer type properties with the value XML_SIMPLETYPE.

Complex types

Elements that have assigned complex types can have both child elements and attributes. The order and structure of the child elements of a complex type are known as its content model. Content models are defined using a combination of model groups, element declarations or references, and wild cards. There are three kinds of model groups: **all**, **choice** and **sequence**.

Complex type elements, including the complex types using compositor elements, are represented in the Type Designer type properties with the value XML_COMPLEXTYPE.

Elements

Element declarations are either local or global. Local elements are represented in the Type Designer type properties with the value: XML_ELEMENT.

When the schema defines several global elements, the generated type tree includes a Global choice group that contains all of the global elements. A Global choice group is represented in the Type Designer type properties with the value XML_BODY.

Attributes

Attributes are another building block of XML. The import process uses attribute declarations to name attributes and associate them to particular simple types. Attribute declarations can be either local or global.

Attributes, including both local and global, are represented in the Type Designer type properties with the value XML_ATTRIBUTE.

Groups and substitution groups

One way an XML Schema allows the creation of reusable content is by using named model groups. These global groups can be referenced throughout a schema.

Substitution groups are a flexible way to designate element declarations as substitutes for other element declarations in a content model. New element declarations can easily be designated as substitutes from other schema documents or namespaces without changing the original content model.

Groups and substitution groups are represented in the Type Designer type properties with the value XML_GROUP.

Identity constraints

The XML parser in WebSphere Transformation Extender enforces declared identity constraints. The XML Schema Importer sets these properties based on the XML Schema. In general, the identity constraint definition serves in one of the roles listed below:

Identity Constraint
: **Description**

Unique
: Enforces that a value (or combination of values) is unique within a given scope. For example, all product numbers must be unique within a catalogue.

Key
: Enforces uniqueness and requires that all values be present. For example, every product must have a number and it must be unique within the catalogue.

Key References
: Enforces that a value (or combination of values) corresponds to a value represented by a key or uniqueness constraint. For example, for every product number that is displayed as an item in a purchase order there must be a corresponding product number in the product description section.

Namespaces

The XML Schema Importer supports namespace and prefix properties for all elements and attributes. The importer assigns the values of these properties based on the input grammar. For each namespace in the input grammar, the importer allows you to specify the namespace prefix to be used for output documents.

During the import process, a **Location** value is specified in the type tree without a fully qualified path. In this case, a map execution process would, by default, look for the XML Schema at this location. However, if you need to change this path, you can do so.

For example, if you refer to the proper location of the schema in the XML document, then you can manually remove the path location from the type properties. When the map execution process does not find a **Location** value specified in the type tree, the process then looks to the "XSDL Hints" specified in the XML document. When XSDL hints are not present, validation fails.

The value specified in the type tree takes precedence over the XML document.

Element type declarations

The XML DTD and XML Schema importers can generate a simple type tree when encountering character content within mixed data. During the import process, when character data (CDATA/PCDATA) is encountered, the importer incorporates it into existing text items within the type tree.

Element and attribute wildcards

DTDs and XML Schemas allow the specification of wildcard elements within a grammar. The WebSphere Transformation Extender XML Schema and DTD importers recognize element wildcards, build the appropriate types, and set the appropriate properties for those types to represent element wildcards in the type tree.

During the import process, when a wildcard element resolves to an element that is not defined in the imported grammar, it is represented by a text item.

The value of these text items is the text string from the input buffer that runs from the start of the open tag to the end of the close tag for that element.

XML Schemas allow the specification of an attribute wildcard that matches any number of undeclared attributes within the element tag. The XML Schema Importer recognizes the attribute wildcard and creates a sequence of name and value text item pairs within the attribute list of the enclosing element. The XML validation library fills this sequence with the names and values of attributes that do not match any declared attributes in the attribute list.

Chapter 5. Item properties

This chapter discusses the properties of item types. Item types define data for a selected type. Item types are divided into the following subclasses: **Number**, **Text**, **Date & Time**, and **Syntax**. Each item subclass has a specific set of properties.

Item subclass

Both *category* and *item* types have the **Item Subclass** property. The subclass of an item determines the characteristics of the data.

To define the subclass of an item type:

1. Access the **Properties** for the selected category or item type.
2. For the **Item Subclass** property, select a value that defines the data for the type:

   Property
   : **Description**

   Number
   : Any number excluding active syntax objects. The interpretation can be either character or binary. See ″Number Item Subclass Properties″ .

   Text
   : Any character excluding active syntax objects. The interpretation can be either character or binary. See ″Text Item Subclass Properties″ .

   Date & Time
   : A valid date and time format based on the data. The interpretation can be either character or binary. See ″Date & Time Item Subclass Properties″ .

   Syntax
   : Syntax objects are used as separators between portions of data. A restricted set of values used to define a dynamic delimiter, initiator, terminator, or release character. The interpretation is character. A syntax object cannot be longer than 120 bytes. Syntax objects, used as syntax, cannot be mapped. See ″Syntax Item Subclass Properties″ .

Number item subclass properties

Number item subclass properties can be of **Character** or **Binary** value. When you choose **Number** as the **Item Subclass** value, you must choose an **Interpret as** value: **Binary** or **Character**.

The **Interpret as** value defines how the data should be interpreted. Items with a subclass of **Number**, **Text**, and **Date & Time** can be interpreted as either character or binary. Items with a subclass of **Syntax** can be interpreted as character only.

Interpret as binary

Binary text items have content size and pad properties. Binary data is required to be sized or of a fixed size.

Binary number items interpret the data as a binary number or as a byte stream.

© Copyright IBM Corp. 2006

Items with an item subclass of **Text** can be interpreted as character or binary.

When **Interpret as** is set to **Binary**, the following **Presentation** options are available:

- "Binary Integer Presentation"
- "Binary Float Presentation"
- "Binary Packed Presentation"
- "Binary BCD Presentation"

Binary Integer Presentation

The **Integer** value is a whole number.

Property
: **Description**

Length(bytes)
: Can have a length of **1**, **2**, or **4** bytes

Byte order
: See "Byte order" .

Sign
: See "Sign" .

Binary float presentation

The **Float** value is a number with decimals in a location as needed.

Property
: **Description**

Length(bytes)
: Can have a length of **4**, **8**, or **10** bytes

Binary packed presentation

The **Packed** value is a number that has size, decimal places, and sign properties.

The following are binary packed properties:

Property
: **Description**

Length
: Can have a length of **1** - **16** bytes

Implied places
: Can be from **0** - **31**

Sign
: Can be **Trailing-** or **Trailing+**

For **Packed** numbers, the **Implied places** cannot exceed the **Length**.

Binary BCD presentation

The **BCD** value is a binary coded decimal number that has size.

Length (bytes)

Use the **Length (bytes)** property to select the number of bytes that equals the length of the item.

The **Length(bytes)** property is an option when the **Number** item subclass property is interpreted with a **Binary** value.

Byte order

Use the **Byte order** property to define the way that bytes in the data are ordered by selecting one of the following values from the drop-down list in the **Value** column:

Value	Description
Big Endian	The most significant byte has the lowest address. (Usually systems such as IBM, HP, and Solaris.)
Little Endian	The least significant byte has the lowest address. (Usually systems such as Intel and VAX.)
Native	The order is dictated by the platform.

Byte order is an option when the **Number** item subclass is interpreted with a **Binary** value.

Interpret as character

Character number items interpret the data as symbolic data. Symbolic data has the same meaning on different computers. For example, a comma is symbolic because it has the same meaning, regardless of the computer type.

Character text items can have content size and pad properties.

When the **Interpret as** value is defined as **Character**, the **Release** property is specific to the **Character** value and is *not* available as an option for a **Binary** value. For more information, see ″Release Characters″.

When **Interpret as** is set to **Character**, the following **Presentation** options are available:

- ″Character Integer Presentation″
- ″Character Decimal Presentation″
- ″Zoned Character Presentation″

Character integer presentation

The **Integer** value is a whole number.

Further information on the **Integer** value is found in ″Integer Separators″ .

Character decimal presentation

The **Decimal** value is a number that contains decimals.

Further information on the **Decimal** value is found in ″Decimal Separators″ .

Zoned character presentation

The **Zoned** value is a number that has a size, decimal place, pad, and sign properties.

Use the ″Size (content)″ property to specify the size of the zoned number. The size of a zoned number can also be specified in terms of minimum and maximum. The minimum size must be less than or equal to the maximum size.

When the content size of a character-zoned number is used, the last digit and the sign are combined into the same digit but the content size does *not* include the initiator, terminator, release characters, or pad characters.

When the content size of a character-zoned number is important (for example, when the SIZE function is used on the character zoned number), the content size includes the sign but does not include the initiator, terminator, release characters, or pad characters.

Use the ″Sign″ property to define whether the zoned number will be signed. The default value is **No**.

The size of a zoned number is specified in terms of the minimum and maximum number of digits. If the size is specified with the **Min** and **Max** properties, the sign is *not* included.

Example

Zoned Number Not Signed
  Signed Zoned Number

1230 123{

1231 123A

(Length = 4 digits)
  (Length = 4 digits)

Places > implied

Numbers have implied decimal places. The number of decimal places cannot exceed the length specification. The list of available decimal places varies with the length selected. In the **Value** column, enter a total number of decimal places (not to exceed 31).

Size (content)

Use the Size (content) property to specify the minimum and maximum size of an item’s content.

Property
  Description

Min The minimum number of digits of the item.

  The default value is 0.

Max The maximum number of digits of the item.

The content size of a text item specifies the bytes of data, excluding any initiator, terminator, release characters, and pad characters. It is also independent of the character set.

Size is specified in bytes. Some character sets use two or more bytes per character. Digits, separators, and signs are pertinent to number items. Separators can also pertain to date items.

However, when you use the SIZE function, separators and signs are included in the count.

If there is no maximum content size, a **Max** value is not required.

Excluded from min and max size

For character number items, there are certain objects excluded from the **Min** or **Max** count. For each **Presentation** option, the following table lists what is excluded from the minimum or maximum **Size (content)** count.

Presentation:	Integer	Decimal	Zoned
	initiators	initiators	initiators
	pad characters	pad characters	pad characters
	release characters	release characters	release characters
	separators	separators	terminators
	signs	signs	
	terminators	terminators	

Size example

In the following ASCII number there are a total of 12 characters:

```
+1234567.999
```

However, because the initiator (+) and separator (.) are not counted, the number would validate for a maximum size of 10 (**Max=10**).

Separators

Integer and decimal number items can have a separator for thousands and fractional places. If you opt to have a separator, choose **Yes** and sub-options will appear that enable you to choose the format and syntax.

The **Zoned** value of the **Presentation** property does not have separators.

Integer separators

For integer numbers, the separator is the thousands separator.

Integer character numbers have separator properties of:
- Format
- 1000’s Syntax
- Value

These properties specify the placement and value of the thousands separator.

Separator > format

Use the **Separators Format** property to specify the placement and value of the thousands separator.

To define the separator format, select one of the separator formats from the drop-down list in the **Value** column.

Property
: **Description**

#[.]####
: The thousands separator is not required on input, but if present, it must be in the proper location. The separator is not built on output.

#.####
: The thousands separator is required on input in the proper location. The separator is built on output.

1000's syntax > value

Define the thousands separator as either literal or variable. Select one of the following from the drop-down list in the **Value** column:

Property
: **Description**

Literal
: The thousands syntax is a constant literal specified in the **1000's Syntax > Value** column.

Variable
: The thousands syntax allows for variable values. Use the **1000's Syntax > Default**, **Item**, and **Find** properties to define the variable fraction separator.

1000's syntax (literal) > value

For the literal thousands separator, define the literal separator by typing the literal separator character in the **Value** column or by clicking the browse button to display the Symbols dialog box from which you can insert any non-printable value.

A separator can be from one to 120 bytes in length. A separator cannot start or end with a digit and cannot be the? character. The ? character is a reserved character that can be used as a wildcard to represent any single valid character.

1000's syntax (variable) > default

Use the **1000's Syntax > Default** property to define the default variable separator value for the thousands place. The default literal separator value for thousands syntax is a comma.

Enter the default variable separator character or click the browse button to display the Symbols dialog box in which you can insert any non-printable value.

1000's syntax (variable) > item

For integer numbers with a variable thousands separator, the **Separator** property is **Yes**, and the **1000's Syntax** is **Variable**, you can specify an item type for the variable thousands separator.

You can select the desired syntax item from the drop-down list. Or, with the focus on the **Item** property (in the **Value** column), press **Alt** and drag the default separator type item from the type tree window into the **Value** column.

An item type with a class of **Syntax** must exist in the type tree to be a valid **1000's Syntax Item**.

1000's syntax (variable) > find

For integer numbers with a variable thousands separator, the **Separator** property is **Yes** and the **1000's Syntax** is **Variable** The value of the separator can be the current value or the value of the separator can be determined each time an occurrence of that type is found.

Property
Description

Yes Determine the value of the separator each time an occurrence of that type is found. After the value of that separator is found, that particular value is used until it is reset either by another Find or by the occurrence of that separator as a component.

No The system uses the value to which the separator item is currently set or, if not set, it uses the default value.

Decimal separators

For decimal numbers, the separator is the thousands separator and the fractional separator.

Decimal character numbers have separator properties of:

- Format
- 1000's Syntax
- Fraction syntax
- Value

These properties specify the placement and value of the fractional separator.

The item is assumed to have an implicit number of decimal places. To specify the number of implied decimal places, use the**Item Subclass Places** property to specify the number of implied decimal places.

Separators > format

Use the **Separators Format** property to specify the placement and value of the fraction separator.

To define the separator format, select one of the separator formats from the drop-down list in the **Value** column.

Property
Description

####[.##]
The fractional separator is present if there are fractional digits. There is no thousands separator.

####.##
The fractional separator is always present, even if there are no fractional digits. There is no thousands separator.

#[,]###[.##]
The fractional separator is present if there are fractional digits. There is an optional thousands separator.

#[,]###.##
The fractional separator is always present. There is an optional thousands separator.

#,###.##
The fractional separator is always present. There is always a thousands separator if there are more than three whole number digits.

Separators > 1000's syntax

Use the **Separators 1000's Syntax** to define the thousands syntax. Select one of the following from the drop-down list in the **Value** column:

Property
Description

Literal The thousands syntax is a constant literal specified in the **1000's Syntax > Value** column.

Variable
The thousands syntax allows for variable values. Use the **1000's Syntax > Default**, **Item**, and **Find** properties to define the variable fraction separator.

The following properties are described in the **Integer Separators** section:

- ″1000's Syntax (Literal) > Value″
- ″1000's Syntax (Variable) > Default″
- ″1000's Syntax (Variable) > Item″
- ″1000's Syntax (Variable) > Find″

Separators > fraction syntax

Use the **Fraction syntax** property to define the fraction syntax as either a constant literal value or a variable value.

You can define the **Fraction syntax** as literal or variable.

Property
Description

Literal A constant value. Use the **Fraction syntax Value** property to select a literal fraction separator from the **Symbols** dialog box.

Variable
Allow for variable fraction separator values. The following options are available to define the variable fraction separator:

- **Default** - Use this property to define the default variable separator value for fractions. The default literal separator value for fractions is a period. Enter the default variable separator character or click the browse button to display the Symbols dialog box in which you can insert any non-printable value.

- **Item** - For decimal numbers with a variable fractional separator, the **Separator** property is **Yes** and the **Fraction syntax** is **Variable**, you can specify an item type for the variable fraction separator.

  Note: An item type with a class of **Syntax** must exist in the type tree to be a valid **Fraction syntax Item**.
- **Find** - For decimal numbers with a variable fraction syntax, the **Separator** property is **Yes** and the **Fraction Syntax** is **Variable**, the value of the separator can be either the current value or the value of the separator can be determined each time an occurrence of that type is found. Select an option:

Yes - Determines the value of the separator each time an occurrence of that type is found. After the value of that separator is found, that particular value is used until it is reset by another find or by the occurrence of that separator as a component.

- **No -** The system uses either the value that the separator item is set to currently, or, if it is not set, uses the default value.

Sign

Use the **Item Subclass Sign** property to define whether the number is signed for either the **Integer** or **Decimal** value. A sign is a symbol that identifies a number as being either positive or negative. A positive sign is plus (+); a negative sign is negative (-).

Property
Description

Yes The number is signed. Expand the **Sign** property to define the sign values. If the **Sign** property is **Yes**, a minimum of at least one sign value (**If number is +**, **If number is -**, or **If number is 0**) must be specified and at least one value must be required on input.

No The number is not signed. This is the default setting.

When you select **Yes**, you can further define the sign values. The following sub-options are available:

Option Description

Leading-
The sign precedes the number when it is negative only. For input, a sign is required for negative data, but is optional for positive data.

Trailing-
The sign follows the number when it is negative only. For input, a sign is required for negative data, but is optional for positive data.

Leading+
The sign always precedes the number. A sign is required for input and output data.

Trailing+
The sign always follows the number. A sign is required for input and output data.

Custom
Defaults to **Leading-** but can be changed.

Sign values may be specified for the following:

- **If Number is +** (positive numbers)
- **If Number is -** (negative numbers)
- **If Number is 0** (value of zero)

For each value (**If number is +**, **If number is -**, or **If number is 0**), specify the following:

Leading sign

Enter the symbol to be placed before a number.

Trailing sign

Enter the symbol to be placed after a number.

Required on input

If the number has a sign, at least one value (**If number is +**, **If number is -**, or **If number is 0**) must be required on input. Select an option from the drop-down list:

Property	Description
Yes	The sign is mandatory on input.
No	The sign is optional on input.

Pad

If the data value to be mapped to the target item is smaller than the minimum length of that item, pad characters are used to pad the data to that minimum length. Input data may contain both content and pad characters. Output data is built according to the pad definitions of the types.

Property
    Description

Yes
Enables the **Pad** option. Allows the item to contain both content and pad characters. Pad properties include:
- ″Pad > Value″
- ″Pad > Padded to″
- ″Pad > Justify″
- ″Pad > Apply pad″

No
All data is assumed to be content on input; no pad characters are built on output.

Pad > value

Use the **Pad Value** property to define the one-byte pad character. The default pad value is 0.

Type the pad character symbol between angled brackets < > in the **Value** field or click the browse button to display the Symbols dialog box to insert any non-printable value.

For example, to enter a **FormFeed** value for the pad character, in the **Value** field enter <FF> or click the browse button to select the **FF** (**FormFeed**) symbol from the Symbols dialog box.

Pad > padded to

Use the **Pad Padded to** property to define whether the data item is padded to a fixed size or to the minimum content size defined for that data item.

Property
: **Description**

Fixed size
: The data item is padded to a fixed size that you specify. The item must have a value specified for the **Size Max** property. Expand this property to define a **Length** value.

Min Content
: The data is padded to the number of bytes specified as the minimum size in the **Size Min** property. This selection presents a sub-option "Padded to > CountsTowardMinContent" to count pad characters toward the length of an object.

For any item padded to a fixed size, the item must have a value specified for the **Size Max** property and the **Padded to Length** must be greater than or equal to the **Size Max** value.

Shown below are the item properties for a text item type that is padded to a length of 6. The pad character is a space. The data is padded to six bytes. The **Padded to** value is **Fixed Size** and the **Padded to Length** value is 6. For example, if there are only four bytes of data, then two bytes of pad character makes the item data size equal to six bytes.

Padded to > length

Input data may contain both content and pad characters. When an item is built for output, the item is padded to the number of bytes specified for **Length**.

The **Padded to Length** must be greater than or equal to the **Size Max** value.

Padded to > CountsTowardMinContent

When you select **Padded to** → **Min Content**, the **CountsTowardMinContent** field is displayed. Use this setting to specify if pad characters should count toward the length of an object when determining if it meets its minimum content length.

Value	Description
Yes	Pad characters are included in the count for the length of an object.
No	Pad characters are not counted toward the length of an object.

-

You can use the **Propagate** function here to propagate the present **CountsTowardMinContent** value to the subtypes of the present type, if present.

If you are using the trace option, there might be cases where in the trace file the content length appears to be different than the input length. A trace file lists the

input length (the length used to validate an object), which includes the input length of each object plus the pad characters. In the case of numbers, signs and separators are also counted in the length.

CountsTowardMinContent > AcceptAllPads

The **AcceptAllPads** property is applicable to data objects that contain only pad characters. Use this property when you need an object that contains only pad characters (no content) to either pass or fail validation depending on whether it is a mandatory or an optional type, regardless of the minimum size requirement.

Value	Description
Yes	When an object contains only pad characters, the minimum size requirement is ignored and the object passes size validation. The object then passes or fails type validation depending on whether it is a mandatory or an optional type.
No	(Default setting) Pad characters do not count toward the minimum size requirement, therefore an object that contains only pad characters and that does not meet the minimum size (content) requirement fails size validation.

Example

The results of the following examples are based on these values:

Property	Value
Size (content)	5
Pad	Yes
Pad > Padded to	Min Content
Padded to > CountsTowardMinContent	No

Default Behavior: When **AcceptAllPads** is set to **No**, the following results occur:

Input Data (X = pad character)	Valid?
ABCDE	Yes
ABC	No
ABCXX	No
XXXXX	No

When **AcceptAllPads** is set to **Yes**, the following results occur:

Input Data (X = pad character)	Valid?
ABCDE	Yes

ABC No

ABCXX
No

XXXXX
Yes (when type is optional)

No (when type is mandatory)

X Yes (when type is optional)

No (when type is mandatory)

Pad > justify

Use the **Justify** property to specify whether the data is padded to the left or right.

Property
Description

Left The data will be on the left and will be padded (if necessary) on the right.

Right The data will be on the right and will be padded (if necessary) on the left.

Use the TRIMLEFT and TRIMRIGHT functions to exclude pad characters from the justified side.

In the following example, `xxxxxxxxxx` represent 10 spaces:
For an input of `1234567891xxxxxxxxxx` with a right justified pad, the rule `=SIZE(input)` returns `20`. For the same input, the rule `=SIZE(TRIMRIGHT(input))` would return `10` (removing the padding on the right).

Pad > apply pad

Use the **Apply pad** property to specify when to apply the pad character. Choose a value from the drop-down list.

Property
Description

Fixed Group
Apply the pad characters only when the item appears in a fixed group.

Any context
Apply the pad characters when the item appears in any context.

Each item in a fixed group must be padded to a fixed size or have the same value for the minimum and maximum content size.

For example, suppose the item **Name** has a space pad character with this value:

Mary<sp><sp>

If the **Apply pad** property is **Fixed Group**, the two spaces at the end are treated as pad characters only when the item appears in a fixed group. If the **Apply pad** property is **Any context**, then the spaces are always treated as pad characters.

Each item in a fixed group must be padded to a fixed size or have the same value for the minimum and maximum content size.

Pad > fill

For padded signed numbers, this option determines placement of pad characters. Choose a value from the drop-down list.

Property
Description

After sign
Pad characters are placed after the sign.

Before sign
Pad characters are placed before the sign.

Restrictions

Restrictions of an item are the valid values of that item. When the **Interpret as** value is defined as **Character**, the **Restrictions** property is specific to the **Character** value and is not available as an option for a **Binary** value.

Depending on other **Item Subclass** settings, the **Restrictions** property can be set to **Value**, **Character**, or **Range**.

Restrictions > iIgnore case

By default, restrictions are case-sensitive. To enable or disable the **Ignore case** property of the **Restriction**, choose from one of the following options:

Property
Description

Yes Ignore case of restrictions in item type properties.

No Do not ignore case-sensitive restrictions.

For example, if **Ignore Case** = **Yes** and the restriction **Value** is `Ft`, the data values `ft`, `fT`, `Ft`, and `FT` are all valid.

Restrictions > rule

This setting indicates whether you are going to include or exclude the criteria (as ″valid″ or ″invalid″) specified for the restrictions.

Value Description

Include
Includes the restriction values as valid data.

Exclude
Excludes the restriction values as invalid data.

National language

The default **National language** value is **Western**. You can expand the **National language** property to define the **Data language**.

When the **Interpret as** value is defined as **Character**, the **National language** property is specific to the **Character** value and is *not* available as an option for a **Binary** value.

National language > data language

Use the **National language → Data language** property to define the data language or character set of this character text item.

As of version 8.1, WebSphere Transformation Extender uses International Components for Unicode (ICU) 3.2.1, supporting Unicode 4.0.

Supported code pages:

WebSphere DataStage TX version 8.1 supports the following code pages.

Arabic
BIG5
Big5-HKSCS (IBM)
Big5-HKSCS (macos)
BOCU-1
CESU-8
Cyrillic
ebcdic-803
ebcdic-ar
ebcdic-cp.roece/yu
ebcdic-cp-ar1
ebcdic-cp-ar2
ebcdic-cp-be/ch
EBCDIC-CP-DK/NO
ebcdic-cp-es
ebcdic-cp-fi/se/sv
ebcdic-cp-fr
ebcdic-cp-gb
ebcdic-cp-he
ebcdic-cp-it
ebcdic-de
ebcdic-he
ebcdic-is
EBCDIC-JP-kana
ebcdic-xml-us
EUC-CN
EUC-JP
EUC-TW
GB_2312-80
gb18030
GBK
Greek8
Hebrew
hp-roman8
HZ-GB-2312
ibm-037 (ebcdic-cp-us/ca/wt/nl)
ibm-1006
ibm-1025
ibm-1026
ibm-1047
ibm-1047-s390
ibm-1097
ibm-1098
ibm-1112

ibm-1122
ibm-1123
ibm-1124
ibm-1125
ibm-1129
ibm-1130
ibm-1131
ibm-1132
ibm-1133
ibm-1137
ibm-1140 (ebcdic-us-37+euro)
ibm-1140-s390
ibm-1141 (ebcdic-de-273+euro)
ibm-1142 (ebcdic-dk/no-277+euro)
ibm-1142-s390
ibm-1143 (ebcdic-fi/se-278+euro)
ibm-1143-s390
ibm-1144 (ebcdic-it-280+euro)
ibm-1144-s390
ibm-1145 (ebcdic-es-284+euro)
ibm-1145-s390
ibm-1146 (ebcdic-gb-285+euro)
ibm-1146-s390
ibm-1147 (ebcdic-fr-297+euro)
ibm-1147-s390
ibm-1148 (ebcdic-international+euro)
ibm-1148-s390
ibm-1149 (ebcdic-is-871+euro)
ibm-1149-s390
ibm-1153
ibm-1153-s390
ibm-1154
ibm-1155
ibm-1156
ibm-1157
ibm-1158
ibm-1160
ibm-1162
ibm-1164
ibm-1250
ibm-1251
ibm-1252
ibm-1253
ibm-1254
ibm-1255
ibm-1256
ibm-1257
ibm-1258
ibm-12712-s390
ibm-1276 (Adobe Standard Encoding)
ibm-1363 (korean)
ibm-1363_P110-1997
ibm-1364
ibm-1371
ibm-1373_P100-2002
ibm-1386-P100-2002

ibm-1388
ibm-1390
ibm-1399
ibm-16684
ibm-16804-s390
ibm-33722-P120-1999
ibm-367_P100-1995
ibm-37-s390
ibm-437
ibm-4899
ibm-4909
ibm-4971
ibm-5123
ibm-5346
ibm-5347
ibm-5348
ibm-5349
ibm-5350
ibm-5351
ibm-5352
ibm-5353
ibm-5354
ibm-737
ibm-775
ibm-8482
ibm-850
ibm-851
ibm-852
ibm-855
ibm-856
ibm-857
ibm-858
ibm-860
ibm-861
ibm-862
ibm-863
ibm-864
ibm-865
ibm-866
ibm-867
ibm-868
ibm-869
ibm-874
ibm-875
ibm-897
ibm-9005_X100-2005
ibm-901
ibm-902
ibm-921
ibm-922
ibm-930
ibm-933
ibm-935
ibm-937
ibm-939
ibm-9447

ibm-9449
ibm-949_P100-1999
ibm-949_P11A-1999
ibm-950_P110-1999
ibm-954_P101-2000
ibm-971_P100-1995
ibm-eucKR
IBM-Thai
IMAP-mailbox-name
ISO 2022, JIS, locale=ja,version=1
ISO 2022, JIS7,locale=ja,version=3
ISO 2022, JIS8,locale=ja,version=4
ISO 2022, locale=ja,version=0
ISO 2022, locale=ja,version=2
ISO 2022,locale=ko,version=0
ISO 2022,locale=ko,version=1
ISO 2022,locale=zh,version=0
ISO_2022,locale-zh,version=1
KOI8-R
KOI8-U
Latin1
Latin2
Latin3
Latin4
Latin5
Latin-9
LMBCS-1
LMBCS-11
LMBCS-16
LMBCS-17
LMBCS-18
LMBCS-19
LMBCS-2
LMBCS-3
LMBCS-4
LMBCS-5
LMBCS-6
LMBCS-8
macintosh
MS_KANJI
SCSU
Shift_JIS
shift_jis78
US_ASCII
UTF-16
UTF-16 Big Endian
UTF-16 Little Endian
UTF16 Opposite Endian
UTF16 Platform Endian
UTF-32
UTF-32 Big Endian
UTF32 Little Endian
UTF32 Oppostie Endian
UTF32 Platform Endian
UTF7
UTF-8

Windows 874
Windows 949 (korean)
windows-1256
x-iscii-be
x-iscii-de
x-iscii-gu
x-iscii-ita
x-iscii-ka
x-iscii-ma
x-iscii-or
x-iscii-pa
x-iscii-te
x-mac-ce
x-mac-cyrillic
x-mac-greek
x-mac-turkish

None

Expand the **None** or **Zero** property to specify an override data value for **None** or define the **Special value** and **Required on input** properties.

None > special value and zero > special value

Enter a special value if the item is **None** or **Zero**.

For example, if you specify * as the special value for NONE, and a text object has the value *, it will be interpreted as NONE.

If you enter a special value for **None**, enter the exact characters to be validated and built in the output data.

None > required on input and zero > required on input

Property
Description

Yes If the data object is **NONE**, use the **Special** value when building the item in the input.

No Do not use the **Special** value.

When building the item in the input, the item may be either the special value, or the default value for **NONE**. For example, consider an item containing all pad characters, and no actual data. When building that item in the input, the default value for **NONE** is used.

Places

The **Places** property allows you to specify the number of implied decimal places.

For item types defined with an **Item Subclass** of **Number**, a **Decimal** presentation, and a **Separator**, the **Places** property allows you to specify the minimum and maximum number of decimal places and whole number places. The minimum number of places must be less than or equal to the maximum number of places.

If the decimal number has no separator, use the **Places** property to specify the number of implied decimal places.

Text item subclass properties

Text item subclass properties can be interpreted with a **Character** or **Binary** value. Depending on the value, character or binary, the following are text item subclass properties:

- ″Interpret as Binary″ or ″Interpret as Character″
- ″Size (content)″
- ″Pad″
- ″Restrictions″
- ″National language″

Date & time item subclass properties

The **Date & Time** properties provide the flexibility to define multiple combinations of date-time formats.

Date & Time items can be interpreted as either binary or character. (See sections ″Interpret as Binary″ and ″Interpret as Character″).

For binary **Date & Time** items, the **Presentation** property has two different values: **Packed** and **BCD**.

Value **Description**

Packed
: The **Packed** value is a number that has size, decimal places, and sign properties.

BCD The **BCD** value is a binary coded decimal number that has size.

Other **Item Subclass** properties for **Date & Time** are displayed depending on the **Interpret as** setting (**Binary** or **Character**):

- ″Date″
- ″Time″
- ″Format″
- ″Pad″
- ″Restrictions″
- ″None″
- ″National language″

Date

For binary **Date & Time** items, use the **Date** property to define whether the date format is enabled.

Value **Description**

Yes **Date** format is enabled for this type. Expand the **Format** property to select the date format.

No **Date** is not defined for this type.

Date > format

Select the date format that the data interpretation will be based on. For binary **Date & Time** items, the **Date → Format** property has four values.

CCYYDDD and **YYDDD** Julian date formats are supported.

- CCYYMMDD
- YYMMDD
- CCYYDDD
- YYDDD

Time

For binary **Date & Time** items, use the **Time** property to define whether the time format is enabled. Select one of the following options:

Value	Description
Yes	**Time** format is enabled for this type. Expand the **Format** property to select the time format.
No	**Time** is not defined for this type.

Time > format

Select the time format that the data interpretation will be based on. For binary **Date & Time** items, the **Time → Format** property has several values. The following choices are available from the drop-down list:

- HH24MMSS
- HH24MM
- HH24:MM:SS
- HH24:MM
- Custom

Format

You can define a date-time format from within the type properties.

To define the Date & Time format:

1. Open the type properties.
2. Under **Item Subclass Format**, click the browse button.
   The Date Time dialog box is displayed.
3. Make your format selections and click **OK**.
   You can use alphabetical characters as separators. In the example, `2001-04-02T10:32:59-0500`, **T** is the separator.
   Separators are limited to 60 characters.

Supported date formats:

- CCYYMMDD
- YYMMDD
- MMDDCCYY
- MMDDYY

- CCYYDDD
- YYDDD
- DDMMCCYY
- DDMMYY

Note: The DDMMCCYY and DDMMYY formats did not exist prior to version 6.5 of the Design Studio. If you had defined either one of these formats prior to 6.5 as a custom format, it will be recognized in 6.5 as the respective new format, instead of the previously defined custom format.

Use of format elements

If you select the same format element more than once for the same item, that item may not be validated separately. If you specify **MON** twice, the day of the month is not validated separately for both months. For example, if the date format element **MON** is used twice for a single item, with the following format string:

MON D-MON D

Suppose the data is this:

Feb 28-Mar 31

28 is not validated for the month of February.

Some combinations of reserved words are invalid. A separator must follow reserved words representing a variable number of digits (**D** and **M** for date) if data follows.

For example:
- D/M/CCYY is valid.
- CCYYM/D is valid.
- DMCCYY is invalid.

See the Design Studio Introduction documentation for a list of reserve words and symbols.

Custom Date Format

After choosing **Custom** from the date format drop-down list, click browse. The Date Format dialog box is displayed.

You can only use non-alphanumeric characters (excluding the **{, }** and **[,]** non-alphanumeric characters) as separators in the custom Date Format dialog box.

The following table provides examples of date formats.

Date Format	Description	Example
CCYY	4-digit Century + Year	1999
YY	2-digit Year (00-99)	99
MM	2-digit Month (01-12)	12
M	1- or 2-digit Month (1-12)	8
MON	3-character Month (Jan to Dec)	JAN

Date Format	Description	Example
MONTH	Full name of Month	January
DDD	3-digit Day of year (001-366)	32
DD	2-digit Day of Month (01-31)	31
D	1- or 2-digit Day of Month (1-31)	7
DY	3-character Day of Week (Sun-Sat)	Fri
DAY	Full Name of Day of Week (Sunday-Saturday)	Friday
WW	1- or 2-digit Week of Year	13
Qn	Quarter of Year (Q1-Q4)	Q2
Custom	User defined custom date format	

After you define and save a custom format, the custom format string is displayed in the Properties window. For example, the custom date format **CCYYMMDD**, and the custom time format **HH24MMSS** display as:

{CCYYMMDD}{HH24MMSS}

Custom Time Format

After choosing **Custom** from the time format drop-down list, click browse. The Time Format dialog box is displayed.

You can only use non-alphanumeric characters as separators (excluding the **{, }** and **[,]** non-alphanumeric characters) in the Time Format dialog box.

The following table provides time format examples.

Time Format	Description	Example
HH24	2-digit hour in 24 hour format (00-23)	23
H24	1- or 2-digit hour in 24 hour format (0-23)	11
HH12	2-digit hour in 12 hour format (00-12)	08
H12	1- or 2-digit hour in 12 hour format (0-12)	8
MM	2-digit minute (00-59)	09
M	1- or 2-digit minute (0-59)	9
SS	2-digit second (00-59)	05
S	1- or 2-digit second (0-59)	5
AM/PM	Meridian (AM/PM)	AM
ZZZ	A time zone abbreviation. See "Time Zones" for a list of supported time zones.	EST
+/-ZZZZ	Hours and minutes before or after the Greenwich Mean Time (GMT). GMT is now referred to as Coordinated Universal Time (UTC).	+0500
+/-ZZ:ZZ	4-digit time where the format is a 2-digit hour and 2-digit minute, separated by a colon.	+05:00
+/-ZZ[:ZZ]	4-digit time where the format is a 2-digit hour and an optional 2-digit minute, separated by colon.	+05:00

Time Format	Description	Example
+/-ZZ[ZZ]	4-digit time where the format is a 2-digit hour and an optional 2-digit minute.	+0500

After you define and save a custom format, the custom format string is displayed in the Properties window. For example, the custom date format **CCYYMMDD**, and the custom time format **HH24MMSS** display as:

{CCYYMMDD}{HH24MMSS}

Time zones

The following table lists the supported time zones.

Time Zone	Abbreviation	Hours before Greenwich Mean Time	Hours ahead of Greenwich Mean Time
Greenwich Mean Time (Zulu)	GMT	0	0
West African Time	WAT	-1	+23
Azores Time	AT	-2	+22
No name; Brasilia Time	###	-3	+21
Atlantic Standard Time	AST	-4	+20
Eastern Standard Time	EST	-5	+19
Central Standard Time	CST	-6	+18
Mountain Pacific Time	MST	-7	+17
Pacific Standard Time	PST	-8	+16
Yukon Standard Time	YST	-9	+15
Hawaii Standard Time	HST	-10	+14
Nome Time	NT	-11	+13
New Zealand Time	NZT	-12	+12
No name; no location	###	-13	+11
Guam Standard Time	GST	-14	+10
Japan Standard Time	JST	-15	+9
China Coast Time	CCT	-16	+8
West Australia Time	WAT	-17	+7
Zulu + 6 (Russia zone 6)	ZP6	-18	+6
Zulu + 5 (Russia zone 5)	ZP5	-19	+5
Zulu + 4 (Russia zone 4)	ZP4	-20	+4
Baghdad Time	BT	-21	+3
Eastern European Time	EET	-22	+2
Central European Time	CET	-23	+1

Time zone format string for XML

The following time zone strings support the specification of a single character Z to indicate Coordinated Universal Time (UTC), as described by the World Wide Web Consortium (www.w3c.org) and the ISO 8601 standard:

- +/-ZZZZ
- +/-ZZ:ZZ

When the +/-ZZ:ZZ or +/-ZZZZ format string is specified, the data validation process works in the following manner:

- Valid data that corresponds to this format string will contain either a character literal Z (representing UTC) or a zone in the appropriate format: +/-ZZ:ZZ or +/-ZZZZ, as specified.
- If a character literal Z is present, it shall be interpreted as, and treated at mapping time, as UTC, which is equivalent to +00:00 or +0000.

The following time zone strings support the omission of the minute portion of the difference from UTC:

- +/-ZZ[:ZZ]
- +/-ZZ[ZZ]

Optional time segments of the time format string

This section discusses how you can specify a portion of a time-format string to be optional. For example, you can specify that either the *time zone* portion or the *hours, minutes, seconds, fractional seconds, Meridian* portion is optional.

This flexibility is also applicable to time-format strings used in functions or in type tree scripts imported by the Type Tree Maker.

Only the time portion **or** the zone portion can be optional-not both.

To specify a portion of the time-format string as optional:

1. The following procedure assumes that the **Item Subclass** property of your type tree is **Date & Time**.
2. From within the type tree properties, navigate to **Item Subclass** → **Format**.
3. Click the browse button to open the Date Time dialog.
4. From a Format field that is set to **Time**, go to the field directly below it and choose **Custom** from the drop-down menu.
5. Next to the Format field with **Custom** selected, click the browse button.

   The Time Format dialog is displayed.
6. Choose from the following tasks:
   - To specify the *hours*, *minutes*, *seconds*, *fractional seconds*, and *Meridian*-portion of the time format string as optional, enable the check box on the far left side of the dialog.
   - To specify the *time zone* portion of the time format string as optional, enable the check box.

     Only one segment of the time format string can be specified as optional.

Date and time format examples

The following list includes examples of popular standard date and time formats:

Format Description

X12 EDI
Currently supports fractional seconds. HH24MM[SS[0-2]]

HL7 The time format in HL7 is HH24MM[SS[.0-6]][+/-ZZZZ]. HL7 also has a TimeStamp format: CCYYMMDDHHMM[SS[.0-6]][+/-ZZZZ], a combination of date and time.

SAP SAP represents time as HHMMSS

ODBC

The form most commonly used in **ODBC** mapping is `HH:MM:SS[.0-9]`. The fractional part is optional and not often used.

Special

When the **Item Subclass** has a **Date & Time** value and the **Interpret as** property has a **Character** value, the **Special** property has two properties:

- If **Date & Time** is NONE
- If **Date & Time** is Zero

If Date & Time is NONE

Select one of the following from the drop-down list:

Value Description

Yes An override data value for NONE can be specified. Expand the **If Date & Time is NONE** property to define the following properties:
- Special value
- Required on input

No No special values or input requirements are defined when data is NONE.

NONE > special value

Enter a special value if the item is NONE. For example, if you specify ***** as the special value for **NONE**, and a text object has the value *****, it will be interpreted as **NONE**.

The same description applies to the ″If Date & Time is Zero″ property.

NONE > required on input

Select one of the following from the drop-down list:

Value Description

Yes If the data object is NONE, use the **Special value** when building the item in the input.

No Do not use the **Special value**. When building the item in the input, the item may be either the special value or the default value for NONE. For example, if an item contains all pad characters and no actual data, the default value for NONE is used when that item is built in the input.

The same description applies to the ″If Date & Time is Zero″ property.

If Date & Time is Zero

Select a value from the drop-down list:

Value **Description**

Yes An override data value for Zero can be specified. Expand the **If Date & Time is Zero** property to define the following properties: **Special value** and **Required on input**.

No No special values or input requirements are defines when data is Zero.

Syntax item subclass properties

Syntax objects are characters that precede, separate, or follow a particular data object. Item types with an **Item Subclass** of **Syntax** are defined and used to specify delimiters, initiators, terminators, and release characters. Syntax objects with variable values are defined as the syntax object's **Variable** property. Syntax objects appearing as actual data to be mapped as output data are defined as components of a type.

Syntax objects with variable values

Using a syntax item to specify delimiters, initiators, terminators, and release characters is required when the value of syntax objects (delimiters, initiators, terminators, and release characters) may vary.

The restrictions of syntax items define the variable literal values.

To define a variable syntax object:

1. Create an item type with an **Item Subclass** of **Syntax** to represent the syntax object.
2. Define the restrictions for the syntax item type.
3. For the item or group type with the variable syntax object, for the **Variable** delimiter, initiator, terminator, and release character, select the item from the **Item** drop-down list in the Properties window.
4. For the **Find** property, select **Yes**.

   Only item types defined with an **Item Subclass** of **Syntax** appear in the drop-down list.

Example of Variable Syntax Object as an Item Type

In this example, the group type **Header** has variable delimiter values of: **, * + ~**

1. Create an item type with the **Item Subclass** of **Syntax** and the name **Delimiter**. (The name can be any valid type name, but naming this item **Delimiter** is useful).
2. To define the item restrictions, double-click the **Delimiter** type.

   The item window is displayed.
3. Define the restrictions in the **Include** column.
4. In the Properties window, select **Syntax** for the **Item Subclass** property.
5. Open the properties for the delimited group type **Header**.
6. For the **Syntax** property, choose **Delimited**.
7. For the **Delimiter** property, choose **Variable**.

8. Expand the **Delimiter** property.
9. Define the **Delimiter Default** literal value.
10. For the variable **Delimiter Item** property, choose the item type **Delimiter** from the drop-down list.
11. Indicate that the variable delimiter should be determined for each occurrence of the object by selecting **Yes** for the **Delimiter Find** property.

Syntax objects as data

The value of a syntax object (delimiters, initiators, terminators, and release characters) may appear as actual data that can be mapped. Syntax objects that appear as actual data are defined as components of a type.

Example of syntax objects as components of a type

The following is an example of defining a syntax object as a component of a group type. The following data represents a message received from multiple departments.

Each message is made up of segments. Each department uses different segment delimiters and terminators. The message format specifies that the delimiter and terminator are the first two bytes of the message, followed by the actual data.

To define syntax objects for the delimiter and terminator:
1. Define two separate syntax objects with an **Item Subclass** of **Syntax**, one for the message delimiter and one for the message terminator. Appropriate type names might be **SegmentDelimiter** and **SegmentTerminator**.
2. Define the possible message delimiter values as restrictions of the **SegmentDelimiter** item type. Define the possible message terminator values as restrictions of the **SegmentTerminator** item type.
3. Define these item types as the first two components of the group type **Message**.
4. Expand the **Delimiter** property.
5. For the variable **Delimiter Item** property, select **SegmentDelimiter** from the drop-down list.
6. For the variable **Terminator Item**, select **SegmentTerminator** from the drop-down list.

During the data validation process, the component **SegmentDelimiter** appears in the data with a value of `*`. The group type **Segment** has a variable delimiter specified as the item type **SegmentDelimiter**, with the value `*`. Therefore, it is understood that the segment has `*` as the delimiter.

When the value of a syntax object appears as actual data, it can be mapped as data. For example, source data may use different delimiters from several different sources. Acknowledgments of this data must be sent back to the source using the delimiter used in the original data. This data can be mapped from the input to the output if these syntax objects are defined as components within the data.

Delimiter > find

When syntax objects are **Variable**, the **Find** property must be defined. The **Find** property determines whether the value of syntax object is set on each occurrence or whether the current setting (or default value) is used.

When the **Initiator**, **Terminator**, **Release**, or **Delimiter** property value is **Variable**, select one of the following from the drop-down list in the **Value** column:

Property
Description

Yes Determine the value of the syntax object each time an occurrence of that type is found. After the value of that syntax object is found, that particular value is used until it is reset by another **Find** or by the occurrence of that syntax item as a component.

INPUT: The value of the object in the input data is determined by the location in the data stream and the restrictions of the syntax item.

OUTPUT: The default value is used for building the syntax object in the output data.

No The variable syntax object is defined as the current value or, if it is not set, as the default value.

INPUT: If the syntax object is not encountered in the data stream, the value of the syntax object is the default value.

OUTPUT: If the value of the syntax item has *not* been previously set, the default value is used.

Chapter 6. Item restrictions

Item restrictions are an optional feature you can use to specify valid or invalid data objects for an item type.

Restrictions of an item type are the valid or invalid values for that item. For example, a unit of measure field in the data must be one of a set of values: `CN, BX, PK, BR`. These values should be defined as ″include″ restrictions of the item **UnitOfMeasure**.

Defining restrictions for an item restricts the valid data for that item. Partial lists of the valid values are not possible. For example, it is not possible to define the values for a certain item to be {`A`, `B`, `C`, ″`some other possible values`″}. Include restrictions are all of the valid values for an item. Exclude restrictions specify all of the invalid values.

Most items will not have restrictions. For example, you probably would not want to restrict the valid values of something like a name field. You would probably want to accept any name as valid data for that field. But you could assign restrictions to it if needed. You can define restrictions for any item.

The top section of the item window contains information about restrictions for the item type. You set the restrictions from the Properties window (for the item) in the Type Designer.

Item restrictions are grouped into three categories: **Value**, **Character**, and **Range**.

Defining item restrictions

By default, restrictions are case-sensitive. To ignore the case of restrictions, when defining the properties of the item, choose **Yes** for the **Restrictions Ignore Case** property.

In the following instructions, it is assumed that the **Enter** key is the ″commit changes″ key. (See ″Options″.)

To define restrictions of an item:

1. Double-click the item for which you want to define restrictions.

   The item window is displayed.
2. Type each restriction value in the appropriate cell and press **Enter**.

   The column headings of the restriction cells will differ based on the item properties set.
3. (Optional) In the corresponding cell, type a description of the value and press `Enter`.

   Entering a description of a value is helpful when the value of the restriction is not obvious. For example, **BR** could mean ″barrel″. A description can be especially useful when building a map with this data.

© Copyright IBM Corp. 2006

Inserting new rows

You can add new rows to the item window's columns by right-clicking or pressing the `Insert` key. The new row inserts *after* the selected row.

For example, click on the row containing ANALYZE and right-click. After choosing **Insert** from the context box, the row is inserted under ANALYZE.

Restrictions settings

In the Properties window, the **Restrictions** property has three values to choose from: **Value**, **Character**, and **Range**.

Value restrictions

When the **Item Subclass Restrictions** field is set to **Value** and **Restrictions Rule** is set to **Include**, the **Include** and **Description** columns are present in the item window.

When the **Item Subclass Restrictions** field is set to **Value** and **Restrictions Rule** is set to **Exclude**, the **Exclude** and **Description** columns are present in the item window.

- **Include** column: Specify the restrictions to use in determining if input data is valid or invalid for the item. One-to-many values are allowed. The order in which restrictions appear in the list is not relevant.

  "Include" values are considered valid. An input item is considered invalid if a character other than those specified in the **Include** column appears in the data.
- **Exclude** column: Specify the text character values to be excluded. "Exclude" values are considered invalid. This option provides a means to specify a default set of restrictions.
- **Description** column: (Optional) Enter a brief description of the value restrictions.

Character restrictions

When you are setting item type properties, you can select **Include** or **Exclude** for the **Restrictions Rule** value.

Include character restrictions

When the **Restrictions** field is set to **Character** and **Restrictions Rule** is set to **Include**, the **Include First** and **Include After** columns are present in the item window.

- **Include First** column: Defines the first character of a character list.
- **Include After** column: Defines the character list of the characters to follow the first character (defined in the **Include First** column).

  For example, the first character could be a letter and all characters that follow could be either letters or digits.

Exclude character restrictions

When the **Restrictions** field is set to **Character** and **Restrictions Rule** is set to **Exclude**, the **Exclude** and **Reference String** columns are present in the item window.

- **Exclude** column: Specifies the substrings in the input data that must be excluded and replaced with the associated reference string.
- **Reference String** column: Defines what the excluded character substrings will be replaced with. If no reference string is specified, the **Exclude** substring is not built.

  On output, a character text item is built by using the corresponding reference string when the content contains any of the **Exclude** character substrings listed.

  For example, you can define XML item content that excludes markup delimiters by using the corresponding reference strings (because XML character text items cannot contain the markup delimiters <, >, or & in the raw form).

To make it easier to specify a common range of characters, you can use reserved words and symbols. See the Design Studio Introduction documentation for a list of reserved words and symbols.

Range restrictions

Include

When the **Restrictions** value is **Range** and the **Restrictions Rule** value is **Include**, the item window displays **Include Minimum**, **Include Maximum**, and **Description** columns.

- **Include Minimum** column: (Required) Defines the minimum value of the range.
- **Include Maximum** column: (Required) Defines the maximum value of the range.
- **Description** column: (Optional) Enter a brief description of the range restrictions.

Exclude:

When the **Restrictions** value is **Range** and the **Restrictions Rule** value is **Exclude**, the item window displays **Exclude Minimum**, **Exclude Maximum**, and **Description** columns.

- **Exclude Minimum**: Defines the minimum value of the range to be excluded or considered invalid.
- **Exclude Maximum**: Defines the maximum value of the range to be excluded or considered invalid.
- **Description** column: (Optional) Enter a brief description of the range restrictions.

  Unbound restrictions are not supported.

Value not in range

The minimum value and the maximum values can be specified as not included in the range. For example, when the **Value NOT In Range** icon appears beside a number in an **Include Minimum** field, it indicates that the number is not included as the minimum value. The range will therefore extend from any value that is greater than that number to the maximum value specified. Similarly, if the **Include Maximum** value is designated as ″not in range″, the valid range would extend up to any value this is less than the maximum value.

To specify a value as ″not in range″:

1. Select the field (cell) that contains the value to be designated as ″not in range″.
2. Right click on the value and choose the **Value NOT In Range** menu item.

The **Value NOT In Range** icon appears to the left of the selected value.

Inserting symbols

Symbols are used to indicate nonpintable characters. Symbols are inserted by using the Symbols dialog box.

For example, the carriage return/line feed symbols can be inserted from the Symbols dialog box or entered in angle brackets.

Using the Symbols dialog box, the carriage return/line feed appears in literal quotation marks as "<CR><LF>".

For example, to define a **CR/LF** as an item restriction select the carriage return (**CR)** symbol and click **Insert**. Then select the line feed (**LF)** symbol and click **Insert**. Notice in the window shown below, a description of the selected symbol appears at the bottom of the window. In this case the description is **LineFeed**.

Alternatively, enter the hexadecimal values in double angle brackets. For example, enter <<00>> to indicate a hex null value.

To insert a symbol:

1. From the **Component** or **Restriction** menu, select **Insert Symbols**.

   The Symbols dialog box is displayed.
2. Select the desired symbol and click **Insert**. Or, double-click on the desired symbol

   The selected symbol appears in the **Value** field.
3. Click **OK**.

   When a symbol is selected, the description of that symbol is shown at the bottom of the dialog box.

Ignoring restrictions

There might be instances when you do not require the data for an item to match any of its restrictions. Suppose you defined restrictions for your data, but you want to run a test on some test data, and you do not want to use the restrictions for this time only. You can ignore the restrictions for a given execution of a map.

In another example, you have a list of valid part numbers that can appear in any data circulated within your company. Suppose you receive data from some other company in the same format as your internal data. The other company may be using different part numbers, so you do not want to make their data conform to the restrictions. When you map the other company's data, you could ignore the restrictions.

For instructions on ignoring restrictions during map execution, see Map Designer and Command Server documentation.

Chapter 7. Group properties

Group properties include the group's **Subclass** and **Format**, which describes how to distinguish one component of that group from another component of that group.

Group subclass

Group types have a subclass of **Sequence**, **Choice**, or **Unordered**.

Property	Description
Sequence	A partially-ordered or sequenced group of data objects. Each component of a Sequence group is validated sequentially.
Choice	Choice groups provide the ability to define a selection from a set of components like a multiple-choice question on a test. Choice groups are similar to partitioned sequence groups. A Choice group is validated as only one of its components. Validation of a Choice group is attempted in the order of the components until a single component is validated. If the Choice group has an initiator, the initiator is validated first.
Unordered	An unordered group has one or more components. Unordered groups can only have an **Implicit** format property, with the same syntax options as sequence groups: **None** or **Delimited**.

Properties of group subclasses

The distinction between Sequence and Choice group subclasses is that Choice groups have no **Partition** or **Format** properties.

Type syntax properties such as **Initiator**, **Terminator**, **Release**, and **Empty** may be applied to **Choice** groups. If a release character is defined, by default it applies to the **Terminator**.

Choice group components

The components of a choice group are similar to the partitions of a partitioned type. However, a Choice group can have both items and groups as components. A partitioned Sequence group can only have group subtypes.

Components of a Choice group must be distinguishable from each other. The components of a Choice group cannot have a component range other than (1:1). Only one component of a Choice group built in the output data.

A Choice group data type is only one of the group components. For example, the data type **Record** is a group type with a **Group Subclass** of **Choice**. The group type **Record** has three components: **Order**, **Invoice**, and **Sales**. The data validation of **Record** will be only one of the components: **Order**, **Invoice**, or **Sales**. For this reason, the components of a Choice group must have a component range of (1:1).

© Copyright IBM Corp. 2006

Unordered group components

An unordered group has one or more components that can appear in the data stream in ay order. They allow many SWIFT and FIX message types, for example, to be defined in a more natural way.

Unordered groups have no partitioned property. They have **Implicit** format properties with the same syntax options as sequence groups: **None** and **Delimited**.

When a group is defined as **Unordered**, any component can appear in the data stream. A component can be an item or a group.

Unordered group components have a range property. For example, if the unordered group, **A**, has the following component list:

```
B(1:S)
C
D(S)
```

then **A** must have one **C,** at least one **B**, and possibly some **D**s. They could appear in any order. For example, data for **A** could have the pattern: **CDDBDDD** or **BBBDDCBD**.

Component rules of an unordered group cannot reference other components of the same group. They can only reference the component to which the rule refers and the objects contained in that component.

Sequence group formats

A sequence group has either an **Explicit** or **Implicit** format.

Property
: **Description**

Explicit
: The explicit format relies on syntax to separate components. Each component can be identified by its position or by a delimiter in the data. Delimiters appear for missing components.

Implicit
: The implicit format relies on the properties of the component types. The format is not fixed. If delimiters separate components, they do not appear for missing components.

For example, if each component of a fixed group has a fixed size, the component is distinguished from the next component by its position in the data. Or, a group may have delimiters that appear for missing components. In these cases, the format is apparent; the group has an explicit format.

If a group does not have an explicit format, it has an implicit format. An implicit format relies on the properties of the component types. In this example, the components make some pattern in the data and it is possible to distinguish between them, but the format is not fixed and if delimiters separate components, they do not appear for missing components.

When deciding what format a group has, it may help to ask first whether it is clear where one component ends and another begins. Generally, a group has an explicit format if the position of each component in the data stream is always the same or if a delimiter always marks the place for each component.

Explicit format

To specify that a group has an explicit format, choose **Explicit** for the **Format** property. Select a setting for the **Track** property, and choose the group's syntax: **Fixed** or **Delimited**.

Track

The **Track** property indicates whether the system should track only the components that have content or all components, including those that do not have content. The settings are **Content** and **Places**.

For example, if a group **StudyGroup** has the component **Name(s)**, and **Track Places** is specified, any empty occurrences of **Name** are tracked.

Suppose **Name** has an `*` initiator and a space pad character. This is the data for **StudyGroup**:

*Carolyn * *Stuart *Margaret

Notice that the second **Name** is missing.

If **Track Places** is specified and you want to map the third **Name**, the empty **Name** would be counted as an occurrence, so the third **Name** would be Stuart. However, if **Track Content** is specified, the empty **Name** would not be counted as an occurrence because it does not have content and the third **Name** would be Margaret.

Fixed syntax

If a group data object is always the same size (in bytes), it has a fixed syntax. For example, a record that is always 160 bytes has a fixed syntax.

Each component of a fixed group must be fixed. If you break down a fixed group, it ultimately consists of items that are fixed. Each is padded to a fixed size or its minimum and maximum content size are equal. Do not specify the size of a fixed group. The size is automatically calculated based on the size of the group's components.

Guidelines for defining a fixed group

- Each component must be a group with a fixed syntax or a fixed item. The item is padded to a fixed size or its minimum and maximum content size are equal.
- Each component must have a specified range maximum. The maximum cannot be `s`.
- If a component range minimum is not equal to the maximum, content is not required for optional component occurrences.

  For example, if a component is the item **ShippingAddress** (0:1) and **ShippingAddress** has a minimum content size of two characters, data may either contain: 1) all pad characters, or 2) if content is in the data stream, there must be a minimum of two characters for a **ShippingAddress**. If a component is a group, no content is required for any of the items contained in an optional occurrence of that component.

Explicit delimited syntax

An explicit format group with a delimited syntax is one whose components are separated by a delimiter and the delimiter appears as a placeholder even when a component has no content. The only time a delimiter can be missing is if components following the delimiter are all optional and there is no data for these optional components.

A delimiter is a character or series of characters that separates data objects.

A delimiter cannot be longer than 500 bytes.

The delimiter of a group appears inside the group, separating its components. When a group is delimited, that indicates something about the components of the group. The delimiter inside a group is delimiting the components.

For example, the group **Employee** has an explicit delimited format because a comma delimiter appears between **Employee's** components, the items that make up the **Employee** object. In addition, the delimiter is displayed when a component is missing and there is data for components following it.

The components of **Employee** are the items **ID#**, **Name**, **Department**, **Address**, and **Age**.

Delimiter

In the **Delimiter** property, specify the value and location of the delimiter. For specific information on specifying the delimiter, see ″Specifying a Delimiter″ .

In an explicit delimited group, if the delimiter is whitespace <WSP>, the delimiter is interpreted as one byte long; each <WSP> character is interpreted as another delimiter. In the output, a <WSP> delimiter is one space.

The limit for <WSP> is 256 bytes.

Implicit format

In a group with an implicit format, the components are distinguishable not by delimiter or position, but by their pattern; something in the definition of the component types themselves. The group has no syntax property that distinguishes one component from another.

For example, the type **File** consists of **Record**(s). <CR><LF> appears at the end of each **Record**. It has been defined as the terminator of **Record**. **File**, however, has no syntax of its own. The **Record** terminator distinguishes one **Record** from another.

To specify that a group has an implicit format, choose **Implicit** for the **Format** property. Optionally, define a comment type, and choose the group's syntax: **Delimited** or **None**.

Floating component

The floating component represents an object that may appear after any component of the group.

An implicit group can have a floating component; an explicit group cannot. If the group is prefix or infix delimited, the floating component is displayed before the delimiter. If the group is postfix delimited, the floating component is displayed after the delimiter.

A floating component can be an optional component that may appear after any other component. However, it is not included in the component list because it does not appear at a specific location.

If a group has a floating component, a component must be distinguishable from a floating component. For example, components and floating component could start with different initiators.

Note: When a floating component appears in the input data, it is validated during mapping. If there are floating components in your output data, define them as actual components of the output.

A floating component can appear after the initiator (when the type has an initiator), after each component, or both. For EDI and other existing floating components, trees will be converted as "after each component".

A floating component can be specified for implicit sequence group, choice group, and unordered group definitions.

A floating component provides additional flexibility to support XML data. With the floating component property, XML element groups would have a floating component defined as an unordered group of XML comments, XML processing instructions, and/or white space.

The XML DTD Importer, for example, uses the floating component property for XML elements whose content contains other elements. For such XML element definitions, a floating component can be a choice of XML comments and/or XML processing instructions.

To define a type as a floating component:
1. In the Properties window expand the **Format** property.
2. With current focus in the **Floating Component Value** field, define the value for the floating component by pressing `Alt` and dragging the item from the tree into the **Floating Component Value** field.

Implicit whitespace syntax

Select **WhiteSpace** for the **Component Syntax** value when white spaces are not allowed it the data. When you select the **WhiteSpace** option, define the **Build As** and **Character Set** properties.

Note: Helpful for XML data.

Build as

Enter the characters that will replace white spaces.

The **Build As** property is only available when the **Component Syntax** property is defined as **WhiteSpace**.

Character set

Select one of the following from the drop-down list in the **Value** column:

Property
: **Description**

Native
: Literal values for this character text item use the machine's native character set.

ASCII
: Literal values for this character text item use ASCII (American Standard Code for Information Interchange) for the data language.

EBCDIC
: Literal values for this character text item use EBCDIC (Extended Binary-Coded Decimal Interchange Code), which is an IBM code for representing characters as numbers.

Latin1
: Literal values for this character text item use Latin1 for the data language. Latin1 uses the byte values in the 0x80-0xFF range to represent characters that are not defined in ASCII.

UNICODE Big Endian
: Literal values for this character text item use UNICODE for the data language. The rightmost bytes are most significant.

UNICODE Little Endian
: Literal values for this character text item use UNICODE for the data language. The leftmost bytes are most significant.

UTF-8
: Literal values for this character text item use UTF-8 (Universal Transformation Format 8), which is an ASCII- compatible multibyte Unicode and UCS encoding.

Implicit delimited syntax

If a delimiter separates the components of a group, but the delimiter does not appear when a component is missing that group has an implicit format, with a delimited syntax.

In certain data, the delimiter may not be a placeholder.

Delimiter

In the **Delimiter** property, specify the value and location of the delimiter. For specific information on specifying the delimiter, see ″Specifying a Delimiter″ .

In an implicit delimited group, if the delimiter is whitespace <WSP>, all contiguous <WSP> characters are treated as one delimiter. In the output, a <WSP> delimiter is built as one space.

The limit for <WSP> is 256 bytes.

No syntax

To specify a group with an implicit format that has no syntax, choose **None** for the **Component Syntax** value.

Distinguishable components of an implicit group

Each component of an implicit group needs to be recognizable. If either of two different components appears at the same place in a data stream, there must be a distinguishable difference between one component and the other.

Sometimes components of an implicit group may be distinguished because there is something in the data that distinguishes them.

After the first item **Line** of the **Form**, the next data object could be another item **Line**. Or, it could be a **Trailer Line**. When looking at a particular **Line**, there is something in the data that identifies that **Line** either as a **Header Line**, an item **Line**, or a **Trailer Line**. The type **Line** has been partitioned to distinguish between the different kinds of **Lines**.

Specifying a delimiter

For the **Delimiter** property, specify whether the delimiter is a literal or a variable value.

Literal

If the delimiter is a constant value, enter the delimiter value in the **Value** field. To enter a non-printable value, click the browse button and select a value from the Symbols dialog box.

National language

The national language is **Western**.

Data language

Choose the data language of the delimiter from the list. Select one of the following from the drop-down list in the **Value** column:

Property
: **Description**

Native
: Literal values for this character text item use the application's native data language.

ASCII
: Literal values for this character text item use ASCII for the data language.

EBCDIC
: Literal values for this character text item use EBCDIC (Extended Binary-Coded Decimal Interchange Code), an IBM code for representing characters as numbers.

Latin1
: Literal values for this character text item use Latin1 for the data language. Latin1 uses the byte values in the 0x80-0xFF range to represent characters that are not defined in ASCII.

UNICODE Big Endian
: Literal values for this character text item use UNICODE for the data language. The rightmost bytes are most significant.

UNICODE Little Endian
: Literal values for this character text item use UNICODE for the data language. The leftmost bytes are most significant.

UTF-8 Literal values for this character text item use UTF-8 (Universal Transformation Format 8), which is an ASCII- compatible multibyte Unicode and UCS encoding.

Variable

Sometimes you do not know what delimiter is used in the data source, especially if you receive that data from an outside source. However, you know all of the possible values that the delimiter could be. You can create an item to represent this delimiter and specify all of the possible values as restrictions of that item. The value of the delimiter in the data is found.

To specify an item as a Variable Delimiter:

1. In the Properties window for the delimited group, click in the **Delimiter Item** value field.
2. Press `Alt` and drag the item from the type tree window into the **Item** field.

Location

The **Location** property specifies the location of the delimiter with respect to the components. The options are prefix (the delimiter appears before the component), postfix (the delimiter appears after the component), and infix (the delimiter appears between components).

The following table explains each option.

Property	**Description**	**Example**
Prefix	Before each component. Before each member of a component in a series.	*a*a*b*c
Postfix	After each component. After each member of a component in a series.	a*a*b*c*
Infix	Between components. Between members of a component in a series.	a*a*b*c

Delimiter value appears as data

Delimiters do not appear as part of the actual data. For example, if the delimiter is specified as a comma, the comma in the following text item would be considered a delimiter, rather than part of the data itself:

Tom Smith, Jr.

If the data does contain the delimiter value, there are ways to format the data so that both the data and the delimiter are distinguishable. For example, text items can be enclosed in quotation marks. Or, a release character may be used. For information about release characters, see "Release Characters".

Chapter 8. Components

A component represents a data object that is part of another data object.

Components are required for group types

Categories and groups can have components. Components of group and category types display in the group and category windows. Item types do not have components.

Group types represent actual data objects, so groups must have components. The one exception is a partitioned group which is explained in ″Partitioning″.

By contrast, a category is used for organizing types and for type property inheritance reasons. Categories do not define actual data objects in detail. A category does not need components.

Each group must have at least one component, unless it is partitioned.

Components must be in the same type tree

A component must be a type in the same type tree as the type that contains the component.

You cannot define the components of a type by opening up a different type tree and dragging components from that tree. You can, however, copy types from one type tree to another.

Importance of component order

Components are listed from top to bottom in the group window in the order they appear in the data stream. The component in the first cell appears first in the data stream. The component in the second cell appears next, and so forth.

Component range

The range defines the number of consecutive occurrences of that component. A component range can be specified for any component. A component range defines the number of occurrences. The range `(s)` represents some or any number greater than one.

When a component is selected, the component name is displayed in the rule bar. Click in the rule bar after the component name to type the component range.

The range is displayed in parentheses immediately after the component name. The range is two numbers separated by a colon. The first number indicates the minimum number of consecutive occurrences of that component. The second number indicates the maximum number of consecutive occurrences of that component. The syntax of the component name and range is:

Component (MIN:MAX)

To enter a range after a component name, type in the rule bar or use the Set Range command.

© Copyright IBM Corp. 2006

The maximum component range is 2147483647.

Whenever a component has a range, each occurrence of that component may be referred to as a ″member″ of a series. The words ″occurrence″ and ″member″ are used interchangeably in the documentation.

Indefinite number

When there is no maximum number of occurrences of a component (the maximum is indefinite) use the letter **s** to stand for ″Some - you do not know how many″. Therefore, a file that contains at least one record and has no maximum number of records has this component:

Record(1:s)

If the range minimum is zero, omit the `0` and only enter `s`. If a file has a minimum of zero records and no maximum number of records, it would have this as its component:

Record(s)

Remember that when you see `(s)`, the minimum of zero is implied. Therefore,

Record(0:s) is the same as Record(s)

If you enter any number alone in the parentheses, the range changes to a minimum of 0 and a maximum of that number. For example, if you enter (5) for a range and save and close the group window, the next time you open it, you see the range displays:

(0:5)

Single occurrence

If there may be a minimum and a maximum of one consecutive occurrence of a component, its range is (1:1). This is the default. If there is no range after a component name, the range (1:1) is implied. When you drag a type to make it a component, it is automatically created with the default component range of (1:1).

Group windows

The group window displays the components of the selected group type.

The contents of the selected cell are displayed in the rule bar. To size the component rule bar, move the mouse over the bottom border of the component rule bar, and drag the bar to the desired size.

Nested components

In a group window, nested components are displayed. For example, in the group window of **CustomerList**, the component **Prolog(0:1)** may be expanded to show **Prolog**'s components. In addition, the component **XMLDesl AttList**, within **Prolog**, may be expanded to show **XMLDesl AttList**'s components.

When components are expanded, the rule cells associated with nested components are gray. Unavailable rule cells indicate that you can add component rules for components of a type, but cannot add component rules to nested components or partitions.

Defining components

Most groups need at least one component. Categories may not have components, but you can define components for a category for inheritance purposes.

To determine the components of a group, ask yourself:

Group type _____________ *consists of what?*

For example, ″The file consists of what?″ The answer might be ″records″. So, **Record(s)** is a component of that group or category type.

Suppose you have a data file containing order records for an office supply store. You need to define the components of the type **File**.

To define components:

1. In the type tree, double-click the type whose components you want to define.

   For group types, the group window is displayed. For category types, the category window is displayed.
2. Drag each component from the type tree into the group or category window.

   The type name appears in the component cell.
3. Enter component ranges where necessary.

   For example, double-click **File**.
4. Drag each component from the type tree into the group window.

   Drag **Record** from the type tree into the component cell of File window. The name **Record** appears in the component cell.
5. Enter component ranges where necessary.

   A component range indicates the number of occurrences. The number of records in the file is indefinite, so the range is (s), which represents ″some.″ When a component is highlighted, it appears in the component rule bar. Click in the component rule bar after **Record**, and type (s).
6. Press `Enter`.
7. In a group window, enter component rules, if necessary.
8. Close the window and click **Yes** when prompted to save changes.

Complete type name

The complete name of a group is displayed in the title bar of its window. A type's complete name is similar to a path name, beginning at the type and including all types in the path up to the root. Spaces appear between types in a complete type name.

You can have duplicate type names as long as they are not on the same level. Each type has a unique complete name, so there is no doubt as to which particular type is being referenced.

Relative type names

A component name is similar to a relative path name. Component names exclude types that the defined type and component type have in common in their complete names.

The relative type name excludes the names that appear in the complete names of both the type and the component type. Therefore, because **ROOT** is included in the complete type name of both **Row ROOT** and **Product Column ROOT**, it is excluded from the relative type name **Product Column**.

If more than one type in a type tree has the same relative type name as another in a group, then the full path of the type is exported instead of the relative type name. No two types that have different full type names but identical relative type names can be added to a component list.

Moving types with the same relative type name

In the following illustration, **Group Category ROOT** shows that it has one type named **Item**. There are two types with the same name **Item** in the type tree at different places. The relative type names of both types, **Item ROOT** and **Item Category ROOT** both evaluate to **Item** with respect to the **Group** component.

You are allowed to drag and drop only one **Item** into the component list. If you attempt to drag-and-drop the second, you will be blocked and the component will not be added to the list.

Ambiguous type names

A component name can refer to more than one type. If this is true, you must change type names so that the component name is no longer ambiguous. The Type Designer will not allow components with the same relative type names to be added to component lists.

If you delete a type that is used as a component of another type, and then perform an "Undo", thus adding the type back, and the component name is ambiguous; the component name remains unresolved in the component list. The "Undo" operation will not result in a complete reversal of the operation being undone.

Manual entry of types with same relative type names

Although the drag-and-drop method is the recommended method of entering component names into the component list, it is possible to type the component list manually into the text entry area of the group window. In the following example, there are two component types with the name of **Item** in the type tree.

If you type **Item** in the text entry area at the top of the group window and press **Enter**, the component name **Item** will appear in the component list, preceded by an ambiguous type name icon.

An *ambiguous type name* indicates that there is another type with the same relative type name in the type tree. There are two ways of eliminating the ambiguity:

- You can drag one of the types named **Item** from the type tree onto the type name in the component list. When you do this, the ambiguous type icon will change to a solid color to indicate that the ambiguous type name is resolved. If you attempt to drag the second type name called **Item** from the type tree into the component list you will be blocked.
- You can also type a path name in the group window as shown in the following illustration. You must type either the full path name of the type, or if you type a partial path name it must contain enough of the path name to make the type unique.

For example, if you enter **Item Category ROOT**, or **Item Category**, the type name **Item** will be unique and the ambiguity will be removed. When you enter this

unique path name, the type will be successfully entered into the component list. The ambiguity is thereby resolved, however, you will be blocked from adding the other **Item** type to the component list.

Always drag components

Because the component rule bar allows you to add a component range, the entire component name is editable. You could enter a component name by typing it. However, this is not advisable. Always drag components into component lists. This is recommended for the following reasons:

- To avoid typographical errors
- To enter type names in the exact case in which they were created. Type names are case sensitive. For example, a type named **PO_Date** is different from a type named **po_date**.
- To avoid incorrectly entering a relative type name

When you drag types into a component list, the correct relative type name is automatically entered.

Viewing the component number

You can view the number of each component in a group and category window.

To view component numbers:

1. From the **Tools** menu, choose **Options**.
2. Select **Group Window** or **Category Window**.
3. Enable the **Show component number** check box.
4. Click **OK**.

Specifying minimum and maximum consecutive occurrences in the component list

The following table contains examples of how to specify in the component list the minimum and maximum consecutive occurrences for specific data objects:

Data Object	Min	Max	How to Specify
DateField	1	5	DateField (1:5)
DetailRecord	1	100	DetailRecord (1:100)
AddressField	2	3	AddressField (2:3)

Fixed and variable ranges

Sometimes the range of a component is described as fixed or variable. A fixed range has the same minimum and maximum, (5:5) for example. A variable range has a different minimum and maximum, (1:10) or (s) for example.

Using the set range command

You can specify a component range by using the **Set** Range command.

To enter a component range using the Set Range command:

1. Select the component whose range you want to specify.
2. From the **Component** menu, choose **Set Range**.

3. In the Set Range dialog box, enter the minimum and maximum number of occurrences.
4. Click **OK**.

   For example, to enter the range (0:1) for the component **MiddleInitial Field**, select **MiddleInitial Field** and choose **Set Range** from the **Component** menu.

You can select multiple components and use the Set Range command to apply the same range to each component. For example, to assign the range (0:1) to multiple components, select them and use the Set Range command.

A range of `(1:s)` means that there will always be at least 1 occurrence, but could occur an infinite number of times. A range of `(0:s)` means that an occurrence is optional, but could occur an infinite number of times.

Viewing the range column

You can view component ranges in a separate column.

To view the component range column:
1. From the **Tools** menu, choose **Options**.
2. Click **Group Window**.
3. Enable the **Show range column** check box.
4. Click **OK**.

Types that can be components

There are certain guidelines to follow when defining types. It is not necessary to memorize these guidelines because the Type Designer assists you when you define components. The Type Designer does not allow the dragging of invalid components.

Guidelines for defining components

The guidelines below are numbered to allow references to the following type tree and to each other. The numbering does not suggest a priority or a sequence to be followed in observing the guidelines.
1. Categories may not have components.

   A category is only used for organizing your type tree and for setting common properties.
2. A partitioned group may not have components.

   A partitioned group always represents a choice among the subtypes of that group. You never map a partitioned group without its subtree, so it does not need components.

   Note: A subtree is a branch of a type tree that includes a type and all of the subtypes that stem underneath it.
3. If a group is not partitioned, it must have at least one component.

   Non-partitioned groups are sequences of data objects rather than choices. A sequence must contain at least one component.
4. A type and one of its subtypes cannot be in the same component list.
5. If a type has components, a subtype can inherit any of those components or any type in the subtree of one of those components.

6. If a type has no components, a subtype can inherit any type that could be in that type's component list.
7. A type that has an initiator and a terminator can have itself or one of its ancestors as a component.
8. A type cannot have one of its subtypes as a component.

Variable component names

A component may refer to more than one type. To refer to all possible types whose names could appear at a certain place in the component name, use the word `ANY`. The word `ANY` is like a wild card. It represents any type whose name could appear in that place.

The use of `ANY` is restricted to **Categories** and partitioned groups. For more information about using `ANY` in a partitioned group, see "Partitioning". For a list of reserved words and symbols, see the Design Studio Introduction documentation.

Opening a component window

Double-click the component whose window you want to open.

For example, if the Order window is open and it has the component **Row (s)**, when you double-click the component **Row (s)**, it opens the window of **Row**.

Required and optional data

Sometimes, a certain data object is optional; it does not have to be present in the data. For example, in purchase order data, there might be a billing address and a shipping address. If the company wants the items shipped to the billing address, the shipping address would not appear in the data. The shipping address would be optional; it might not appear in the data.

Another example is a middle name field. Some people do not have a middle name, so the middle name field might be optional.

The Type Designer needs to know what data is optional. This is evident from the component range. The range minimum tells how many occurrences of that object must be present in the data. These are the required occurrences. Optional occurrences are the ones that are not required.

For example, for the following component, the range minimum is zero. No occurrences must be present. It is optional data:

DateField (0:1)

Suppose the component looks like this:

DateField (1:5)

The range is between one and five occurrences. This means that one occurrence of **DateField** is required and the remaining four occurrences are optional.

The following table lists examples of components and explanations of their status.

Component	Status	Reason
LineItem (1:s)	1 occurrence required	Range minimum is 1
Note Field (5:5)	5 occurrences required	Range minimum is 5
RecordID	1 occurrence required	Range minimum is 1
ShipTo (0:1)	0 occurrences required (Optional)	Range minimum is 0
OrderRecord (s)	0 occurrences required (Optional)	Range minimum is 0

Significance of required data

If you define an occurrence of a component as required, you are saying that, for the data containing the component to be *valid*, this component must exist. If it does not exist, the data is *invalid*. For example, if you define **Record** as having the component **Field** (3:3), you are saying that there must be three **Fields** in the **Record**. If there are not three **Fields**, then either it is a **Record** in error or it is not a **Record**.

There are other factors, besides the existence of all required components that make data valid. The existence of required components is necessary, but not sufficient, for data to be valid.

Defining component rules

A component rule is an expression about one or more components. It indicates what must be true for that component to be valid. For given data, it evaluates to either TRUE or FALSE. A component rule is similar to a test. If the data does not pass the test, it is invalid.

Component rules are used for validating data. Some important points about component rules are:

- Only components of a group may have component rules. Components of a category cannot have component rules.
- A component rule cannot be longer than 32K.
- If a component is optional and does not appear in the data, the component rule is evaluated, after determining that the data is missing. In a component rule, you can specify relationships that depend on existence or nonexistence of data.

Sometimes components have relationships among each other. For example, an address field in purchase order data is preceded by a qualifier field which tells whether the address is a bill-to or a ship-to address. If the value of the qualifier field is `BT`, the address field following it is a bill-to address. If the value is `ST`, the following address is a ship-to address.

The qualifier and address fields are dependent on each other. They only make sense as a pair. If one of these fields is optional and is missing, the other field is not meaningful. If the qualifier is missing, you do not know whether the address is a bill-to or a ship-to. If the address is missing, the qualifier does not qualify anything!

You might want to define this kind of relationship and other relationships among data objects. To do this, use a component rule. For example, use a rule on the address field component to indicate that the qualifier must be present if the address is present.

The following instructions, assume that the **Enter** key has been assigned to the **Commit changes with** option in the **General** settings of the Options dialog.

To enter a component rule:

1. In the group window, select the rule cell to the right of the component.

   For example, to create a rule for **Quantity Column**, select the rule cell to the right of it.
2. Enter the component rule in the edit window and press **Enter**.

   For example, type the following component rule to indicate that **Quantity** must be greater than 1000:

   QuantityOrdered > 1000

Examples of component rules

A component rule can limit the acceptable values of a component as shown in the following example:

Quantity < 10000

Interest Rate > .13 & Interest Rate < .20

WHEN (PurposeCode != ″PF″, ShipValue = 200|ShipCode < 0)

The following example illustrates how a component rule can make the presence of one component mandatory, if another component is present:

WHEN (PRESENT (Address Field), PRESENT (Qualifier Field))

WHEN (PRESENT (PhoneNumber), PRESENT (AreaCode), ABSENT (AreaCode))

A component rule can compare a component to the result of an arithmetic operation as shown in the following example:

SUM (((QuantityOrdered:Item Record:Detail) = TotalQuantity:Summary Record:Detail

Account Balance = Credits - Debits

Extension = Quantity*Price

#Items Field = COUNT (Item Record IN Invoice)

Component rule syntax

A component rule is a complete expression that evaluates to either TRUE or FALSE. It may contain functions (for example, PRESENT, COUNT, and SUM). It may contain arithmetic operators (for example, - + / *).

A component rule is a statement, so it does not start with an equal sign.

For more information on the syntax of expressions, see the Functions and Expressions documentation.

Entering object names in component rules

A component rule may refer to a component within a component. The syntax for a component is the component name, followed by a colon (`:`), followed by the object of which the component is a part. Whenever you see the colon (`:`) in an expression, it can be interpreted as ″component of″ or simply ″of″.

For example, the following expression means ″The item number of the order record of the order″:

Item#:OrderRecord:Order

All of the objects that can be used in component rules are shown in the group window. You can enter an object name into a component rule by pressing `Alt` and dragging the object into the edit window. The complete object name is automatically entered in the edit window.

For example, to enter the component name for **Phone# Field** in the component rule, select the component rule. Press `Alt`, and drag **Phone# Field** into the edit window. The object name **Phone# Field:$** (where **$** represents **Customer**) is entered.

A component rule may refer to:

- The component it applies to and any nested components.
- Any component above the given component in the component list, and its nested components.

  You can enter an object name in a component rule by typing it; however, this is not advisable. It is better to press `Alt` and drag the component to the edit window.

Shorthand notation

In a component rule, the dollar sign **$** represents the component itself. When you enter an object name in a rule by pressing `Alt` and dragging the object, the dollar sign is automatically entered in the rule to represent the given component.

Component rules are context-sensitive

Component rules apply to components, not types. It applies to data in a certain context when the data is a component of a given group.

Suppose you have some order data that contains two kinds of records: a regular order record and a bulk order record. In the bulk order record, the quantity ordered must be greater than 1000. In the regular order record, the quantity can be any number. You could put a rule on the quantity ordered component of a bulk order, but not on the quantity ordered of a regular order.

The component rule in this example applies in a given context. When the **QuantityOrdered** field is displayed in the context of the **Bulk OrderRecord**, it must conform to the component rule. When it is displayed in the **Regular OrderRecord**, it does not have to conform to any component rule.

Special characters in component rules

To enter the actual value of a special character in a component rule, you must enter the Hex value for one of the characters. For example, to enter the text value <WSP>, enter the Hex value for the less than sign <<3C>>, and then the rest:

<<3C>>WSP>

See the Design Studio Introduction documentation for a list of non-printable Hex and decimal values.

Inserting functions into component rules

You can use most of the functions in a component rule. Component rules in the examples previously shown use the functions COUNT, PRESENT, SUM, and WHEN.

The function WHEN is similar to the function IF. However, WHEN evaluates to ″true″ or ″false″. IF evaluates to a data object. WHEN is most often used in a component rule.

To insert a function into a component rule:

1. Select the cell in which you want to insert the component rule.
2. Click in the edit window where you want to insert the function.
3. From the **Component** menu, choose **Insert Function**.
4. Or, click with the right mouse button and choose **Insert Functions** from the context menu.
5. Or, place your cursor in the component edit window and press **Insert** on your keyboard.
6. The Insert Function dialog box is displayed.

   Some functions are listed in more than one category.
7. Select the function you want and click **Insert**.
8. The function and a set of parentheses are entered in the component rule.
9. Click **Close**.

Formatting a component rule

To add a new line to a component rule, press **Ctrl** and **Enter** if commit changes is set to **Enter**, or press **Tab** if commit changes is set to **Tab**.

Comments in component rules

You can add comments to component rules. Comments do not affect how component rules are evaluated.

A comment begins with the characters /* and ends with the characters */. A comment may appear anywhere in a rule as long as it does not separate object names.

For example, the following component rule has a comment:

SIZE (Phone# Field:$) >=7

/* Phone numbers must include area code */

Syntax errors

If you type a component rule that is syntactically incorrect and press `Enter` , the part of the rule in error is highlighted and you cannot enter it until the error has been corrected.

Searching for components

You can search for components by using a Find operation.

Finding a component by number

Sometimes you need to search for a certain component by number. For example, you may have received an analysis error that references a particular component by number. Use the **Go** To command to locate this component.

To find a component by number:

1. Select the **Group** or Category window.
2. From the **Component** menu, choose **Go To**.
3. Enter the number of the component to which you want to go.
4. Click **Go To**.

Managing components

You can easily move components around, copy them, or delete them. For information on general drag-and-drop procedures, see "Managing Types". When you move, copy, or delete a component, if it has a range and/or a component rule, the range and rule are moved, copied, or deleted with the component.

Component attributes

Three attributes can be assigned to a component in a component list:

- Identifier
- Restart
- Sized

Component attributes are toggle commands. If an attribute is assigned to the selected component, the attribute tool and the attribute in the **Component** menu appear to be selected.

When an attribute is applied to a component, a corresponding icon is displayed to the left of the component name in the component list.

To assign an attribute to a component

1. Select the component.
2. From the **Component** menu, choose the attribute you want to assign. Or click the tool in the toolbar.

Identifier attribute

The identifier attribute can be used on a component of a group. The identifier indicates the components that can be used to identify the type to which a data

object belongs. All the components, from the first, up to and including the component with the identifier attribute, are used for type identification.

When this data is validated, it knows that, when it reaches the identifier, it has found a specific group. That group, therefore, is known to exist, even if part of the group following the identifier is missing. The Map Designer documentation discusses how data validation occurs.

The identifier attribute is also useful when identifying an object of a partitioned group. For information about using an identifier with partitioned data, see "Partitioning".

In a component list, there can be only one identifier attribute.

Restart attribute

To continue processing your input data when a data object of a component is invalid, assign the restart attribute to that component.

Do not put the restart attribute on a required component. There must be a sufficient number of valid instances to cover all the required components. If you have a required component, that is not valid, the restart attribute will not validate the data.

Sized attribute

The sized attribute is used on a component in which the value specifies the size (in bytes) of the component immediately following it. The sized attribute can be used on more than one component of a group.

For example, you may have a variable length component with a number immediately preceding it that indicates the length of the component:

The `10` indicates the size of the following component.

Some important points about using the sized attribute are:

- The component with the sized attribute must be defined as an unsigned integer.
- If a binary byte stream item does not have a fixed size, the component preceding it must specify its size and the sized attribute must be used on that component.

The size of a component is the number of bytes from the beginning of that component, up to and including the end of the component. If a component has a series range (such as [1:3]), the size includes all of the members in the series of that component. If a delimiter separates each member of that series, the delimiters must be included in the size. Also, if release characters appear in the component, they must be included in the size.

The size does *not* include delimiters that separate one component type from the next.

Include self in size

If the value of the component with the sized attribute includes the size of the component, choose **Include Self in Size** from the **Component** menu. For example, suppose the component with the sized attribute has a length of seven bytes. The

value of the component with the sized attribute would be seven. So, its data would be one byte long, to hold the character seven. If its value includes the length of itself, its value would be 7 + 1 = 8.

For example, a sized component is 7 bytes.

+ 1 byte (the length of the character 7)

If **Include Self in Size** is selected: 8 bytes

Chapter 9. Partitioning

By partitioning, you can define your data to distinguish the difference between data objects based on values in the data or differences in the syntax.

Partitioning is a method of subdividing objects into mutually exclusive subtypes.

A special icon in the type tree identifies a partitioned type. The partitioned type maintains the same class.

To partition a type:

1. Select the type you want to partition.
2. From the **Type** menu, choose **Properties**.
3. For the **Partitioned** property, choose **Yes**.
   You can only partition items and groups.

Determining when to partition

There are some cases in which you will *need* to partition your data and others in which you will *want* to partition your data. You are *required* to partition for unordered data when a data object at a certain place in the data stream can be any number of types and each type has different definitions. *Choose* to partition for convenience, either to simplify mapping rules or to put additional logic into your data's definition.

Required partitioning

Partitioning is required when components are randomly or partially ordered. The following example represents unordered data:

BGI - 13100,REM,931104,19970424...

AXR - 10930,INV,003X114,19970422...

PVY - 19496,ORD,PO-104-1499,19970425...

BGI - 13100,ORD,PO-182-2587,19970425...

AXR - 10930,INV,003X-114,19970422...

PVY - 19496,REM,931104,19970424...

The file would contain three different types of transactions: **Invoice**, **Order**, and **Remittance**. Each transaction has a different definition based on the type of information it represents.

Partitioning for convenience

You might decide to use partitioning in your type tree to build additional logic into the definition of your data. You may also use partitioning to simplify the rules needed in your map.

© Copyright IBM Corp. 2006

The following is an example of partitioning to simplify rules. The example compares the differences between rules needed *with* and *without* partitioning. In the rule without partitioning, you would specify a condition for each state abbreviation in each region. This could make your mapping rules long, difficult to read, and difficult to maintain. The mapping rule with partitioning is more concise, self-documenting, and easier to maintain.

Map rule without partitioning:

```
=IF(ShipToCode Field:.:Input="NY"|
ShipToCode Field:.:Input="NJ"|
ShipToCode Field:.:Input="PA",
F_MapEast (Record:Input), NONE)
```

Map rule with partitioning:

```
=F_MapEast (EXTRACT (Record:Input,
PARTITION (ShipToCode Field:.:Input, East)))
```

Benefits of partitioning

Using the above example, explore the benefits of partitioning.

- The rule is shorter than the rule without partitioning. The knowledge of which states belong to each region is maintained in the type tree rather than the map rule.
- It is easy to read this map rule and understand the mapping function being performed. For example, if the state belongs to the list of states in the eastern region, execute the **MapEast** functional map.
- The partitioning method is easier to maintain. If a value for **State** is added or moves from one region to another, it can be easily changed in the type tree and automatically reflected in any mapping rules that reference the partitioned object.
- Partitioning using a restriction list used with the **Ignore Case** setting eliminates the need for PROPER, LOWERCASE, or UPPERCASE functions to compare each state with a literal.

Partitioning types

When data objects of different types appear in the same place in the data, the types must be distinguishable. This means that the data needs to be distinguishable by their definitions in the type tree.

When the data object at any given point in the data may belong to any of a number of different types, there must be some way to tell the difference between them. To do this, you create a type and define a mutually exclusive subtype for each data object that may appear in the same place in the data. Once the subtypes are created, the subtypes also need to be distinguishable. Subtypes are distinguishable based on a value in the data or in the syntax of the different types.

Partitioning items

Use one of the following three methods to partition items in type trees:

- Initiators
- Restrictions
- Format

Partitioning an item type using initiators

To partition by initiator, each subtype must have an initiator and the value of the initiator must be unique for each subtype.

Partitioning by initiator is the most efficient method of partitioning.

Partitioning an item type using restrictions

If an item has restrictions, you can partition that type, create mutually exclusive subtypes, and divide the restrictions between subtypes. A restriction cannot appear in more than one subtype of that item.

Example of using restrictions

You have new data that needs to be entered into the Type Designer. Each new record contains the name of the employee and his or her department.

The **Employee List** type tree illustrates components of **Record**.

The following is the new data containing the name of the employee and the department.

Steven Barlow,Doc

Heather Proust,Qa

Mary Whiting,Doc

Genie Elks,Sup

Francine Maxwell,Dev

Mark Brown,Sup

Daryl Schwartz,Acc

Harry O'Brian,Sal

Ellen Randolph,Dev

Paula Keller,Qa

Define the values of **Department** as a character text item.

Define the example data as valid restrictions (enter in the **Include** column) of **Department**:

If the departments are located in different offices, you can divide the data into separate files; one file per office. To do this, map the data from the main office to one file, the data from the development office to one file, and the data from the support office to another file. Then create subtypes of **Department**: **MainOffice**, **DevelopmentOffice**, and **SupportOffice** and partition **Department**. When you partition **Department** and create subtypes, you are saying that a given department data object belongs to only one of the subtypes based on its value.

The subtypes of **Department** inherit the restrictions of **Department**. Now allocate restrictions among subtypes. To do this, delete the restrictions from the subtype that do not apply to that particular office. For example, the **MainOffice** item has only the departments in that office, **DevelopmentOffice** item has only the departments in that office, and the **SupportOffice** item has only the departments in that office.

Partitioning an item type by format

Subtypes that differ by their format are distinguishable from each other.

Partitioning groups

Use one of the following three methods to partition groups in type trees:

- Initiators
- Identifiers
- Component Rules

Partitioning a group type using initiators

To partition by initiator, each subtype must have an initiator and the value of the initiator must be unique for each subtype.

The method of partitioning by initiators for group types is similar to item types.

Partitioning a group type using identifiers

In a component list, only one component may have the identifier attribute.

The identifier attribute distinguishes the components that can be used to identify the type to which a data object belongs. Typically, use this technique to distinguish group partitions when components following the identifier are different for each partition, or, if you have a multilevel partitioned subtree. In the latter case, using an identifier accelerates data validation.

A partition is valid when each component up to and including the identifier is validated. If the set of components is valid, the partition exists.

If the identifier set of components is not valid, the partition is determined not to exist. Either validation occurs for the next partition at the same level (if there is one) or it is determined that the partitioned group does not exist.

If the partition exists, what occurs next depends on the position of the partition type in the subtree.

- If the partition is partitioned (that is, it has subtypes), the rest of the components are skipped and the process begins to validate subtypes until a subtype is valid, exists and is in error, or does not exist.
- If the partition has no subtypes, the remaining components are validated. If all remaining components are valid, the partition not only exists, but also is valid. If one or more components are found to be in error, the partition exists, but its type is in error. If the partition exists, but its type is in error, the error is propagated back up the partitioned subtree until the group being validated is reached. When a partition is found to exist, the system will not continue to search for partitions.

To specify a component as the identifier:

1. Select the component.
2. From the **Component** menu, choose **Identifier**.

The identifier symbol appears to the left of the component name.

Partitioning a group type using component rules

Component rules are used to partition data when a value or range of values can be used to distinguish one partition from another.

Example of using component rules

Using the Payments.mtt type tree, this example shows the use of component rules.

You can tell the difference between the types by looking at the value of the **PaymentMethod Column**. When the payment method is cash, the component rule is **$=″CASH″**. When the payment method is a credit card, the component rule is **$=″CREDCARD″**. When the payment method is a check, the component rule is **$=″CHECK″**.

Notice how the component rules are used with identifiers in this type tree to define payment types.

Chapter 10. Type inheritance

Properties, components, and restrictions of a type can be inherited by the types created as subtypes under it. Some properties of a type can also be propagated to already existing subtypes.

When you create a type, it becomes a subtype of whatever type is selected at the time. Everything that defines a type gets passed down: properties (with the exception of the **Partitioned** property), components, and restrictions. After a type is created, you can then modify any aspect of its definition.

When a group is created within a category, the item properties do not apply, so they are not inherited. When an item is created under a category, the group properties are not inherited.

Inheritance of item properties and restrictions

Properties and restrictions of items are inherited when a new item is created.

For example, the item **Department** is defined as a character item with an **Item Subclass** of **Text**. It has a content size minimum of 2 and maximum of 3.

When the type **MainOffice** is created as a subtype of **Department**, it inherits the properties and restrictions of **Department**. You can then delete the restrictions that are not in the **MainOffice**.

Inheritance of category properties and components

Categories can be used for organizing types and for inheritance reasons. Generally, you would use categories when you want to put items, groups, and possibly other categories under it as subtypes.

Organizing types under a category

If you have two tables defined in one type tree, you might divide the types of the two tables into different categories. Under one category would be the types of one table and under another category, would be the types of the other table.

A benefit when you use a category is that you can have any class of subtype: other categories, groups, and items. For example, in the **Categories** type tree, notice how **ClassInfo** and **LabInfo** contain items and groups.

Using categories for inheritance

The properties of a category include group and item properties. Assign properties to a category that you want types beneath the category to inherit.

Any group created under a category inherits the property of the category. Any item created under a category inherits the category's item properties. A category created under a category inherits both the group and item properties. Each type created under a category inherits the other properties of the category, such as initiator and terminator.

© Copyright IBM Corp. 2006

If most groups in your **LabInfo** data are infix delimited with ~ and most items in your **LabInfo** data are unsigned integers, you would have defined these as the properties of the category **LabInfo**. When you created the types under **LabInfo**, the properties would be inherited.

You can change the properties of the root type which is a category. Categories can have components, so you can also use them for inheriting components.

When not to use categories

Suppose you create a type and plan on creating subtypes under it. If you are only going to create items under that type, or only groups under that type, you probably do not want to make that type a category. Suppose you have a type named **Field**, and under it, you are only going to create items. If you make **Field** a category type, each time you create a subtype beneath it, you must define the **Class** as **Item** in the **Properties** window. If you make **Field** an item type, the **Class** of **Item** is automatically selected.

Propagating properties

Creation time is not the only time properties can be passed from a type to its subtypes. You can pass certain properties after types have been created. This is called propagation. Propagation passes the settings of a particular property from a given type to all types in its subtree.

To propagate a property:

1. In the Properties window for the type, right-click the property you want to propagate.
2. Click **Propagate**.

Properties that can be propagated

You can select any property and propagate it to types in the subtree of the selected type. When a property does not apply to a type, it is not propagated. For example, if the selected type is an item with **Number Character Decimal** properties, and you propagate the **Separator** property, it affects only types in the subtree that also have **Number Character Decimal** properties. If the selected type is a fixed group and you propagate the **Include trailing white space** property, only fixed groups in the subtree are affected.

Propagating affects types in the subtree

When you propagate a property of a certain type, the property gets passed down to the types in its subtree. For example, if you propagate the terminator of **Report**, the terminator is assigned to all types in the subtree of **Report**.

Chapter 11. Managing types

Standard windows capabilities

The Type Designer supports standard Windows mouse and keyboard capabilities.

In this section, the word "object" refers either to a graphical object (such as a type) or text (such as text in a component rule).

Object selection

The Type Designer supports standard Windows selection capability.

- To select multiple, contiguous objects, select the first object, press `Shift` and select the last object.
- To select multiple, noncontiguous objects, press `Ctrl` while selecting each object.

Drag-and-drop procedures

The following is a summary of drag-and-drop procedures:

- To move an object, drag it to the destination.
- To copy an object, press `Ctrl` and drag it to the destination.
- To merge a type, press `Shift` and drag the type to the destination tree.
- To add a component name to a component rule, press `Alt` and drag the component to the component rule bar.
- To specify a type as a comment type or a variable delimiter, initiator, terminator, or release character in the Properties window, press `Alt` and drag the type from the tree to the Properties window.

Moving and copying objects

You can move and copy the following objects within the Type Designer by using the drag-and-drop method:

- Types (within the same tree)

  When you drag a type from one tree to another, it is copied to the other tree because it is copied from one type tree file to another.
- Components (from one window to another)
- Restrictions (from one window to another)

  Duplicate restrictions are not permitted.

When you move or copy a type using the drag-and-drop method, the entire subtree of the type is also moved or copied. The properties, components, component rules, and restrictions are also copied.

If a type cannot be a subtype of the type you are dragging it to, you will not be able to drag it. For example, you cannot move an item under a group.

Note: You cannot place a type under another type if it cannot be a valid subtype.

© Copyright IBM Corp. 2006

Using the move command

In addition to drag-and-drop, use the Move command to move a type.

To move a type using the Move command:
1. Select the type you want to move.
2. From the **Type** menu, choose **Move**.
3. Select the type under which you want to move the given type.
4. Click **Move**.
5. Click **Close**.

Using a copy command

In addition to using the drag-and-drop method, use the Copy command to copy a type.

To copy a type using the Copy command:
1. Select the type you want to copy.
2. From the **Type** menu, choose **Copy**.
3. Select the type under which you want to copy the given type.
4. To change the name of the copied type, rename the type in the **As** field.
5. To copy the subtree of the given type, click **Copy sub-tree**.
6. Click **Copy**.
7. Click **Close**.

Type names

When you move or copy a type, if that type is referenced by another type (for example, if it is used as a component), the reference is updated to reflect the new relative type name. In addition, if a type you move or copy references other types, these references are automatically updated. Referenced type names include component names, syntax item names, and comment type names.

When you move a syntax item that is referenced as a number separator, initiator, terminator, delimiter, or release character, its referenced name changes.

When you move a type that references other types, the references are automatically updated.

Reordering objects

You can reorder the following objects within the Type Designer by using the drag-and-drop method:
- Subtypes - to reorder subtypes, press `Ctrl + Shift` while dragging a subtype to the desired location.

  Subtypes may be reordered only when the type whose subtypes you are reordering has the **Add First** or **Add Last** setting for the **Order subtypes** property.
- Components - to reorder components, drag the component(s) to the desired location in the component list.
- Restrictions - to reorder restrictions, drag the restriction(s) to the desired location in the restriction list.

Reordering existing subtypes

If the **Order Subtypes** option is set to **Add First** or **Add Last**, you can arrange the subtypes in any order.

To reorder subtypes:

1. Select the type whose subtypes you want to rearrange.
2. Open the Properties window to confirm that the **Order subtypes** property is **Add First** or **Add Last**.
3. From the **Type** menu, choose **Reorder Subtypes**.

   The Reorder Subtypes dialog box is displayed.
4. To move a subtype, drag it to a position.
5. Click **Close**.

For example, to arrange the subtypes of **Field** in the order that they appear in the data, select **Field** and choose **Reorder Subtypes** from the **Type** menu.

Merging types

You can merge types from one tree to another. Merging a type copies that type and all types referenced by that type to another tree. The referenced types include any components, comment types, and any syntax items used as a number separator, initiator, terminator, release character, or delimiter. You can merge the type itself or the type with its subtree.

Before using the merge command

Before using the Merge command, perform the following:

- Analyze the type tree you are merging from.
- Analyze the type tree you are merging to.
- Ensure that the root of the type tree you are merging to has the same root name as the tree you are merging from.

To merge a type from one tree to another:

1. Select the type you want to merge into another tree.
2. From the **Type** menu, choose **Merge**.
3. Enable the **Merge subtree** check box to merge the entire subtree.
4. Open the destination type tree if it is not already open.
5. Select the destination tree.
6. Click **Merge**.
7. Click **Close**.

For example, to merge **Detail Record** from **Sales Tree** to **New Tree**:

- Select **Detail Record** in **Sales Tree**
- Choose **Merge** from the **Type** menu
- Click the **Merge sub-tree** option to merge all the types in **Detail Record's** subtree
- Select **New** Tree

Detail Record and the types referenced by **Detail Record** are automatically copied. In addition, the types in **Detail Record's** subtree and their referenced types are copied.

Supertypes

When you merge a type from one tree to another, the supertypes of that type are created. In the example above, notice that **Detail**'s supertype, **Record**, is included in the copied types.

Existing types

The Merge command copies a type only if it does not already exist in the destination tree. If the type already exists in the destination tree, it is not copied.

Invalid types

The Merge command will only merge a type if it can exist in the new tree. For example, if you try to merge an item into a tree where it would exist as a group, the type is not merged.

Types referred to by ANY in a component are not copied in a merge.

Renaming types

One way to rename a type is to press `Ctrl Alt` and click on the type. Then, type the new name and press `Enter`.

Another way to rename a type is to use the Properties window.

To rename a type:

1. Select the type.
2. From the **Type** menu, choose **Properties**.
3. Enter the new name in the **Name** field.
4. Close the Properties window.

   When you rename a type, it may move to a different position on the same level in the tree if types on that level are in alphabetic order.

Using find and replace

The Type Designer supports standard search facilities. You can find and replace the following objects:

- Types
- Components
- Restrictions

Using the find command

Use the Find command to find types, components, or restrictions.

To use the Find command:

1. Select the window in which you want to search. For example, the type tree window or the group window).

2. From the **Edit** menu, choose **Find**.
3. In the **Find what** field, enter the text you want to find.
4. Optionally, choose **Match whole word only** or **Match case**.
5. Select the direction in which to search: **Up** or **Down**.
6. Click **Find Next**.
7. Continue to click **Find Next** for as long as you want to search, then click **Cancel**.

   If a match is detected, the match is selected. Otherwise, a message asks if you want to continue searching.

Using the replace command

The Replace command replaces text. For example, you can rename types by using the Replace command.

To replace text:

1. Select the window in which you want to search. For example, the type tree window or the group window.
2. From the **Edit** menu, choose **Replace**.
3. In the **Find what** field, enter the text you want to find.
4. In the **Replace with** field, enter the text with which you want to replace it.
5. Optionally, choose **Match whole word** only or **Match case**.
6. Click **Find Next** and then **Replace** if you want to replace that occurrence. Continue until you are finished replacing. Or, click **Replace All**, to replace every occurrence.

Printing in the Type Designer

You can print the contents of any window and type definition in the Type Designer.

To print the Type Tree window and type windows:

1. Select the window to print.
2. From the **File** menu, choose **Print**.
3. Click **OK**.

To print the Properties window

1. Select the window to print.
2. From the **File** menu, choose **Print Type Definition**.

   You can also right-click and select **Print Properties** from the context menu.

Print preview

You can perform a print preview for a type tree.

To preview a print job:

1. Select the type tree to be printed.
2. From the **File** menu, choose **Print Preview**.
3. The Print Preview window is displayed.

Printing type definitions

To print the definition of a type and the types in its subtree, choose **Print Type Definition**.

To print type definitions:

1. Select the type whose definition you want to print.
2. From the **File** menu, choose **Print Type Definition**.

   The Print Type Definition dialog box is displayed.
3. Disable any check box next to the definition you do not want to print.
4. Disable the **Include Sub-tree** check box if you do not want to print the definitions of types in the subtree.
5. Click **OK**.

   The Print dialog box is displayed.
6. Click **OK**.

Printing type properties

You can print a type's properties from the Properties window.

To print type properties:

1. Click the type you wish to print.
2. Right click and select **Print Properties**.

   The Print dialog box is displayed.
3. Click **OK**.

Chapter 12. Error detection and recovery

Error detection

During map execution, the input data is compared to the data definition in the type tree. If the data does not match the definition, it is invalid or ″in error″.

To validate a data object as belonging to a certain type, the data must be matched to its type definition. For data to be valid, the following must be true:

- The data must have the properties defined in the Properties window.
- If the type is an item that has restrictions, the data object must match one of the restrictions.
- If the type is a group, the components of the data object must match those defined in the group window, and each component rule must evaluate to TRUE at map execution time.

Ultimately, all data objects, no matter how complex, consist of items because items represent the smallest unit of data. When all of the items that collectively comprise a component are found, the component has been ″found.″

When executing a map, invalid data is recorded in the trace file. You can decide what you want the system to do when errors are encountered: you can map the errors to an output and also have the system ignore invalid data. For methods on ignoring invalid data, see ″Restart Attribute″ .

How error detection works

When data is validated, the system may encounter data that does not match its type definition. A combination of how that type is defined and what data shows up determines what happens next. As data is validated, the system tracks valid data objects, data objects that exist but are in error, and data that cannot be recognized as belonging to any type.

If a data object is valid, the information is recorded, as needed, and validation continues.

If the system is looking for a data object of a particular type, and something in the data does not exactly match the type definition, the system bases the determination on whether to continue validation on whether that data object is known to exist.

If a data object contains enough information to determine its existence, errors associated with that data object are recorded and validation continues.

If a data object exists but contains errors, it is marked as an error. If a restart point is specified for the component of the type in error, these errors are ignored and validation continues.

If a data object does not contain enough information to determine its existence, the following occurs:

- If no validation recovery mechanism is specified, validation is stopped because it will get lost if it continues. Validation recovery mechanisms are discussed in the section ″Error Recovery″ .

© Copyright IBM Corp. 2006

- If recovery mechanisms are specified, the system returns to the nearest restart point and resets what it is looking for. It proceeds by examining the data byte-by-byte until it either recognizes something or reaches the end of the data stream. All rejected bytes (from the first byte not associated with a type to the last unrecognizable byte) are collectively marked as an unidentified foreign object (UFO). A UFO is data in error with no valid data contained within it.

In summary, the data object of an input card could be valid, yet contain errors. If the data object of any input card is invalid, output data is not built. If all input card data objects are valid, the input is mapped to the output based on the map rules. Operators, most functions, and map references operate on valid data. To map invalid data, use the REJECT function.

Existence indicators

When the source or destination of a data object exists, the system knows that the entire data object of that source or destination exists. For example, if a source specified to contain a transaction is a file, and that file exists, the transaction exists. On the other hand, if you get a `Source not available` message, the data object of that source does not exist because the source itself does not exist.

When the data object of a source or destination exists, specific information about errors will appear in the data. For example, you might have any of the following:

- The data object is valid because it conforms to its type definition.
- The type of the data object exists, but has no content.
- The data object is in error because it does not conform to its type definition.
- The data object is valid but contains errors. You indicated that you wanted to ignore certain errors.

In addition to these conditions, the system can also tell if there is any unknown data remaining after the card object has been recognized. This can occur if there really is ″junk″ at the end. It may also be that enough data was identified to determine the status of the entire object and the data at the end could not, for some reason, be determined to have anything to do with the data object.

When a data object exists, the status of that object is determined by validating its type properties. If an entire data object is an item, it is easy to determine what the status is: whether it is valid or whether it does not match its item format, whether its value is not one of the specified restrictions, or whether it is missing an initiator or terminator. If the source or destination exists and the entire data object is an item, there is nothing else it could possibly be.

If the type of the entire data object is a group, the status of each component must be determined. The system uses existence indicators to determine whether a group component exists. If a component exists, the system can also tell you its status. If a component is required in another component that exists, the system will notify you if that required component is missing.

Existence versus presence of components

A component exists if the data object in which it is contained in exists and if the component takes up space (if there is at least one byte in the data stream representing that component).

For example:

- If a delimiter appears as a placeholder for that component.

- If the type of that component has an initiator that distinguishes that component from any other component that may appear at that position in a data stream, the component exists if the initiator is there.
- If the type of that component is a group with an identifier and all components up to and including that identifier have been found to be valid.
- If the type of the component is an item with a restriction list and one of its restrictions appear in the data.
- If the type of the component is an item with a **Padded To** length specified and the item contains at least the number of bytes in the **Pad To** length.
- If the type of an optional component is valid, but the component rule evaluates to FALSE, the component only exists if there is a syntactic placeholder.

Data might exist, but might not be present. The presence of data is determined by whether the data has content. The following definitions are used in other parts of the documentation:

Function
    Description

EXISTS
    Something in the data represents that data object.

CONTENT
    The data contains at least one byte other than a syntactical placeholder, such as a pad character or delimiter.

PRESENT
    The data object exists and has content.

ABSENT
    The data object does not exist or it exists and has no content.

Required components must exist. They do not necessarily have to be present. For example, a required item with no minimum content size can exist, be absent, and still be valid. However, if some minimum content size is specified, a required component is not valid if there is no minimum content.

The existence of optional components depends on the context in which they are used. For example:

- In a delimited group, an optional component must exist if data follows.
- In a fixed group, an optional component must exist if trailing white space is required. If no trailing white space is required, an optional component must exist if data follows.
- In an implicit group, an optional component is not required.

Unlike a required component, when an optional component exists, it might not have content - even when the minimum content specification is greater than zero bytes. The following table explains the conditions that must be met for required and optional components.

	EXISTS	CONTENT	PRESENT	ABSENT
Required Component	Must exist in any context	Must conform to content specification	Must be present if minimum content specification is greater than zero bytes.	May be absent only if minimum content specification is zero.

	EXISTS	CONTENT	PRESENT	ABSENT
Optional Component	Depends on context	Need not have content	Not necessary	May be absent

Error recovery

You might want to map your data even when invalid data is encountered. For example, you have a file of records and if some of the records are invalid, that is fine. You do not want to stop; you want to go on and process all the valid records.

Restart attribute

The restart attribute provides instructions for handling errors encountered in a data stream.

The restart attribute may be assigned to a component. It applies when data is validated - input data is validated as the first step in the mapping process. When output data is audited by using Audit Settings in the Map Designer, output data is also validated.

If you are mapping data of a component with a restart attribute only the valid occurrences of that component are mapped. For example, suppose your input is a file of records, and the **Record** component of **File** has the restart attribute. Suppose some of the records are invalid. When you map the **Record** objects, only the valid records are mapped. To map the invalid records (for example, to an error file) use the REJECT function.

For information on the REJECT function, see the Functions and Expressions documentation.

Typically, you would assign the restart attribute to a component that has a series range. For example, (1:20) or (s) and the objects in the series are independent of one another. For example, if each record in your file is independent from the others, it makes sense to ignore an error if it encounters an invalid one. A bad record does not make the next record meaningless. But suppose the records are related. Maybe they are records related to one purchase order; if any record is bad, you want to stop after validation because it is important to have all records. Losing any records would make the overall data incomplete.

To assign the Restart attribute to a component:

1. Select the component.
2. From the **Component** menu, choose **Restart**.

How the Restart Attribute works

The restart attribute has the following properties:

- During validation, the restart attribute is used to tell the system where to start over when an unidentified foreign object (UFO) is encountered in the data. All unrecognized data is considered to be an error of the type with the restart assigned.
- The restart attribute is used during validation to mark both UFOs and existing data objects in error that are ignored when mapping input to output.
- The restart attribute is used to identify valid data objects that contain objects in error.

If an invalid data object is a component that does not have the restart attribute, that component is marked in error. If components, from the beginning of the data are marked in error because none of them has a restart, the system stops after validation. It does not map the input data to the output.

Do not put the restart attribute on a required component. There must be a sufficient number of valid instances to cover all the required components. If you have a required component that is not valid, the restart attribute will not validate the data.

Mapping invalid data

You can map the invalid or rejected data which you can use to determine what is wrong with the data. To do this, you use the restart attribute, in conjunction with the REJECT, CONTAINSERRORS, and ISERROR functions in map rules. For information on using these functions, see the Functions and Expressions documentation.

Chapter 13. Distinguishable objects

Differences can be identified for objects in a data stream. This information is helpful when a type tree analysis produced an error message concerning objects that are not distinguishable.

Objects in a data stream

A data stream is a byte-by-byte flow of data. Objects in a data stream include both data objects and syntax objects. Syntax objects indicate where a data object begins or ends. They include separators, initiators, terminators, delimiters, release characters, and pad characters. Sometimes, data values are used to identify where another data object begins or ends.

It might be necessary to distinguish between data objects in a component series, data objects of different types, or even syntax objects.

Type tree analyzer and distinguishable objects

The type tree analyzer indicates whether your data definitions are sufficient to distinguish the objects in your data stream. The following discussion explains how you can define your data so that the objects that need to be distinguishable are distinguishable. Chances are, if you analyzed your tree and received a message that involves distinguishable objects, you need to define the types differently or more specifically.

Bound types

A type is bound if its definition makes it clear where an instance of that type ends. If a type is bound, different objects of that type can be distinguished in a data stream. A bound type is easier to distinguish between an object of that type and an object of another type. The following tables describe how types may be bound.

An object of this type:
Is bound if any of the following is true:

Item
It is padded to a fixed size or its minimum and maximum content size are equal.

It has a terminator.

It has an include restriction list.

Partitioned item
Each non-partitioned item in the subtree is bound.

Sequence Group
It has an explicit fixed format.

It has a terminator.

Its last component is bound.

It is postfix delimited and its last component has a fixed range. For example, **Comment Field (3:3)** has a fixed range of **(3:3)**.

Partitioned Group
Each non-partitioned group in its subtree is bound.

Choice Group
It has a terminator.

The type of each selection component is bound.

Unordered Group
It has a terminator.

Bound components

A component is bound if it is possible to tell if a data object belongs to that component without comparing it to a component that follows it.

Component of a fixed group

Condition
Example

A range maximum is specified (it is not ″s″), and its type is either a fixed group or an item whose length is fixed.
The component **InventorySection (0:3)** is bound because it is assumed there will be spaces in the data stream for 3 **InventorySections**.

Component of an explicit delimited group

Condition
Example

A range maximum is specified (it is not ″s″) if a component of the same group follows.
In the component list, **Security Sequence (0:10) Trailer** is bound in an explicit delimited group. It is assumed there will be a delimiter for all 10 **Security Sequences** in the data stream because **Trailer** is required.

The component is the last one and the explicit group has a terminator.
In the component list, **Security Sequence (s)** is bound in an explicit delimited group with a terminator. The system assumes the terminator will appear to indicate that there are no more **Security Sequences**.

Component of an implicit group

Condition
Example

Its range is fixed and its type is bound.
The component **Inventory Record (3:3)** is bound, if the type **Inventory Record** is bound according to any of the conditions listed under ″Bound Types″ .

It has a component rule that binds it.
The component **PO Record (s)** is bound, if it has the following component rule which binds it:

PO# Field:PO Record = PO# Field:PO Record[LAST]

The type tree analysis checks only that the component has a rule that refers to the component itself. The analysis does not check that the rule binds the component. You must ensure that the rule is one that binds the component.

It is sized (using the Sized attribute) by the component that precedes it.
The component **Name Field (0:2)** is bound, if the previous component in the group has the Sized attribute.

Component of a choice group

Condition
Example

A component is bound if the type of a component is bound.
The component **Customer Record** is bound if the type **Customer Record** is bound.

Component of an unordered group

Condition
Example

A component is bound if its range is fixed and its type is bound.
The component **Address Field** is bound if its range is fixed, for example **(3:3)**, and the type **Address Field** is bound.

Group starting set

Data objects in a data stream need to be distinguishable when they could belong to two different groups. Specifically, the difference between the data that can come first in one group and the data that can come first in the other group. All of the possible types of data that may appear first in a group are referred to as the group's starting set.

The following describes the starting set of a group based on its group format:

Group Format	Starting Component Set
Explicit • Delimited • Fixed	Includes the type of its first component.
Implicit • Sequence • Unordered	• Includes the type of all components up to and including the first component that has a minimum range of at least one. • Includes the type of each component.
Choice	Includes the type of each component.

Group unbound set

Based on the nature of a bound group, the end of a bound group is determined without analyzing the data that follows it. However, if a group is not bound, the data that belongs to that group must be distinguishable from data that belongs to another type that might follow it in the data stream.

The unbound set of a choice or unordered group consists of the type of each component. The unbound set of a partitioned group consists of the unbound set of each unbound partition.

Unbound set of a sequence group

The unbound set does not include a type that could come last, if that type is a required occurrence. For example, if the type **Comment Record** could appear last and the component is specified as **Comment Record (2:2)**, it is not in the unbound set, because the two occurrences of **Comment Record** are required. However, if **Comment Record** could appear last and the component is specified as **Comment Record (1:2)**, then **Comment Record** is in the unbound set, since its second occurrence is optional.

To determine the unbound set of a group, start with the last component of the group and proceed backward up the component list. If a component is unbound or has a minimum range of zero, continue up the list. Stop at the first component that is bound or that is unbound but has a minimum range greater than zero. Essentially, a group's unbound set is everything that is not clearly defined at the end of the group.

Initiator-distinguishable types

Initiator-distinguishable types are used during validation to determine existence of a type object based on the existence of its initiator.

Determining if a component is initiator-distinguishable from its following set

A component is initiator-distinguishable from its following set if:

- The component is not a member of the identifier set and
- The type of the component has an initiator and
- The following set is empty or
- The type of the component is initiator-distinguishable from each type in its following set.

Type trees are analyzed to determine if components are initiator-distinguishable. Each implicit sequence group, choice group, and unordered group is analyzed to determine if their components are initiator-distinguishable. A component is marked as initiator-distinguishable when that component is initiator-distinguishable from its following set. The basis for this determination is found in ″Determining If Two Types are Initiator-Distinguishable″ .

Determining if a partition is initiator-distinguishable from its following set

In a partitioned type, a partition is initiator-distinguishable from its following set if:

- The type of a partition has an initiator and
- The following set is empty or
- The type of the partition is initiator-distinguishable from each partition in its following set and the following set of a partition is the type of each partition that may follow.

Type trees are analyzed to determine if partitions are initiator-distinguishable. Each partitioned type is analyzed to determine if its partitions are initiator-distinguishable.

Determining if two types are initiator-distinguishable

The following table lists ways two types may be initiator-distinguishable. This is helpful if data validation errors indicate a type does not exist.

Type1	Type2	How to define them as initiator-distinguishable
item	item	If Type1 and Type2 have an initiator, and the initiators are different.
item	sequence group	Either: • Type1 and Type2 both have an initiator and the initiators are different. or • Type1 has an initiator, Type2 does not, Type2 has no delimiter, and Type1 is initiator distinguishable from type of each component in the starting component set of Type2.
item	choice group or unordered group	Either: • The Type1 and Type2 both have an initiator and the initiators are different. or • Type1 has an initiator, Type2 does not, Type2 has no delimiter, and Type1 is initiator distinguishable from type of each component of Type2.
item	partitioned item or partitioned group	Type1 has an initiator and is initiator distinguishable from each partition of Type2.
partitioned item	item or sequence group	Type1 has an initiator and each partition is initiator distinguishable from Type2.
partitioned item	partitioned item or partitioned group	Type1 has an initiator and each partition is initiator distinguishable from each partition of Type2.
sequence group	item	Type1 has no identifier, Type1 and Type2 both have initiators, and the initiators are different.
sequence group	sequence group	Type1 has no identifier and • Type1 and Type2 both have initiators and the initiators are different or • Type1 has an initiator, Type2 does not have an initiator, Type2 has no delimiter, and Type1 is initiator distinguishable from the starting component set of Type2.
sequence group	choice group or unordered group	Type1 has no identifier and • Type1 and Type2 both have initiators and the initiators are different. or • Type1 has an initiator, Type2 does not have an initiator, Type2 has no delimiter, and Type1 is initiator distinguishable from the type of each component of the Type2.

Type1	Type2	How to define them as initiator-distinguishable
sequence group	partitioned item	Type1 has no identifier and Type1 must be initiator distinguishable from each partition of Type2.
sequence group	partitioned group	Type1 has no identifier and the Type1 must be initiator distinguishable from each partition of Type2.
partitioned group	partitioned item	Type1 has no identifier and each partition Type1 must be initiator distinguishable from Type2.
partitioned group	sequence group	Type1 has no identifier and each partition of the Type1 must be initiator distinguishable from Type2.
partitioned group	partitioned item	Type1 has no identifier and each partition of Type1 must be initiator distinguishable from each partition of Type2.
partitioned group	partitioned group	Type1 has no identifier and each partition of Type1 must be initiator distinguishable from each partition of Type2.
choice group or unordered group	item	Type1 and Type2 both have an initiator and the initiators are different.
choice group or unordered group	partitioned item	Type1 is initiator-distinguishable from each partition of Type2.
choice group or unordered group	choice group or unordered group	Either: • Type1 and Type2 both have an initiator and the initiators are different. or • Type1 has an initiator and Type2 does not, Type2 has no delimiter, and Type1 is initiator-distinguishable from the type of each component of Type2.
choice group or unordered group	sequence group	Either: • Type1 and Type2 both have an initiator and they are different. or • Type1 has an initiator and Type2 has no initiator and no delimiter, and Type1 is initiator-distinguishable from the type of each component in the starting component set of Type2.
choice group or unordered group	partitioned group	Type1 is initiator distinguishable from each partition of the Type2.

Distinguishable objects of the same component

When a component has a series range, for example, (1:10) or (s), one occurrence of that component must be distinguishable from the next occurrence of that same component.

Different data objects of a component are distinguishable if any of the following is true:

- The component type is bound.

- The component type is a non-partitioned group with an unbound set that is content distinguishable from the component type as a whole.
- The component type is a partitioned group and the unbound set of each non-partitioned group in its subtree is content-distinguishable from the component type as a whole.

Content-distinguishable components

A component is distinguishable from its following set when the component is:

- Initiator-distinguishable from its following set.

or

- Content- distinguishable from its following set.

A component is content-distinguishable from its following set when:

- The following set is empty.

or

- The type of the component is content-distinguishable from each component in its following set.

Content-distinguishable types

The following table lists the ways in which two types may be content-distinguishable. The table is helpful particularly if you analyzed your tree and received an error indicating that two types are not distinguishable. Look up the combination of types in the first two columns and read the list of ways to define them that would make them distinguishable.

For example, if you get an error indicating that **X** is not distinguishable from **Y**, where **X** is an item, and **Y** is a partitioned group, you would find the row in the table where **Type1** is an item and **Type2** is a partitioned group.

Remember that the order in which you compare two types matters. When you ask the question ″Is type A content-distinguishable from type B?″, this is not the same question as ″Is type B content-distinguishable from type A?″

If you are trying to determine whether two types are distinguishable and you follow the guidelines in the following tables, you may encounter a situation in which you are comparing a type to itself. A type is never distinguishable from itself.

Another thing to keep in mind is that is that the context in which a type is used matters. When you ask the question, ″Is type A, as a component of type C, content-distinguishable from type B?″ this is not the same question as, ″Is type A, as a component of type D, content-distinguishable from type B?″

The following table shows how to define types as content-distinguishable.

Type 1	Type 2	How to define them as content-distinguishable
item	item	The first component or partition is either: • An item and marked as initiator distinguishable and Both items are different partitions of the same partitioned subtree, or The second type has an initiator and those initiators are mutually exclusive, or Both types have the same initiator and the value of the first item is distinguishable from the value of the second item, or The second type has no initiator and the initiator of the first type is distinguishable from the value of the second type. • An item and not marked as initiator distinguishable and Both items are different partitions of the same partitioned subtree, or The first item has an initiator, second has an initiator and initiator value of first is distinguishable from initiator value of the second. Both types have initiators and they are the same, or both types have no initiator and the value of the first item is distinguishable from the value of the second item. The first item has an initiator, second does not, and value of initiator distinguishable from the value of second item. The first item has no initiator, second does, and the first item's value is distinguishable from the value of the second item's initiator.
item	sequence group, choice group, or unordered group	The first component or partition is either: • An item marked as initiator distinguishable and The second type has an initiator and those initiators are mutually exclusive, or The second type has no initiator and the item is initiator-distinguishable from each type in the starting component set of the group. • An item not marked as initiator distinguishable and The item and group both have an initiator and the initiator of the item is distinguishable from the initiator of the group, or The item has no initiator, the group has an initiator, and the item's value is distinguishable from the value of group's initiator, or Both types have initiators and they are the same or both types have no initiator, and the item's value is distinguishable from the type of each component in the starting component set of the group, or The item has an initiator, the group has no initiator, and the item is content-distinguishable from the type of each component in the starting component set of the group, or The item has no initiator, the group has an initiator, and the item's value is distinguishable from each type in the starting component set of the group.

Type 1	Type 2	How to define them as content-distinguishable
item	partitioned item or partitioned group	The first component or partition is content-distinguishable from each partition of the second type.
partitioned item	item	Each partition of the first type is content-distinguishable from the second type.
partitioned item	sequence group	Each partition of the first type is content-distinguishable from the second type.
partitioned item	choice group	Each partition of the first type is content-distinguishable from second type.
partitioned item	unordered group	Each partition of the first type is content-distinguishable from second type.
partitioned item	item	Each partition of the first type is content-distinguishable from second type.
partitioned item	partitioned group	Each partition of the first type is content-distinguishable from second type.
group	item	Each partition of the partitioned group is content-distinguishable from the item.
partitioned group	partitioned item	Each partition of the partitioned group is content-distinguishable from each partition of the item.
partitioned group	sequence group	Each partition of the partitioned group is content-distinguishable from the second group.
partitioned group	partitioned group	Each partition of the first partitioned group is content-distinguishable from each partition of the second partitioned group.
partitioned group	choice group	Each partition of the partitioned group is content-distinguishable from the choice group.
partitioned group	unordered group	Each partition of the partitioned group is content-distinguishable from the unordered group.
sequence group	item	The first component is either: • A group marked as initiator distinguishable and The item has an initiator and the initiator value of the group is distinguishable from the initiator value of the item, or The second type has no initiator and the initiator value of the first type is distinguishable from the value of the second type. • A group not marked as initiator distinguishable and The group has an initiator, the item has an initiator and the initiator value of the group is distinguishable from initiator value of the item, or The group has an initiator, the item does not, and value of the initiator is distinguishable from value of the item, or Both types have initiators and they are the same, or both types have no initiator and each type in the starting component set of the group is content-distinguishable from the value of the second item.

Type 1	Type 2	How to define them as content-distinguishable
sequence group	sequence group	The first component is either: • A group marked as initiator distinguishable, and Both groups are different partitions of the same partitioned subtree, or The second type has an initiator and those initiators are mutually exclusive, or The second type has no initiator and the first group is initiator-distinguishable from each type in the starting component set of the second group. • A group not marked as initiator distinguishable, and Both groups are different partitions of the same partitioned subtree, or Both groups have an initiator and the initiator value of the first group is distinguishable from the initiator value of the second group, or The first group has no initiator, the second group has an initiator and the type of each component in the starting component set of the first group is content-distinguishable from the second group, or The first group has an initiator, the second group has no initiator and the first group is content-distinguishable from the type of each component in the starting component set of the group. Both types have initiators and they are the same or both types have no initiator and The first group is content-distinguishable from the starting component set of the second group, or The second group is content-distinguishable from the starting component set of the first group, or The starting component set of the first group is content-distinguishable from the starting component set of the second group.

Type 1	Type 2	How to define them as content-distinguishable
sequence group	choice group or group	The first component is either: • A group marked as initiator distinguishable and Both groups are different partitions of the same partitioned subtree, or The second type has an initiator and those initiators are mutually exclusive, or The second group has no initiator and the first group is initiator-distinguishable from the type of each component of the second group. • A group not marked as initiator distinguishable and Both groups are different partitions of the same partitioned subtree, or Both groups have an initiator and the initiator of the first group is distinguishable from the initiator of the second group, or The first group has an initiator, the second group has no initiator and the first group is content-distinguishable from the type of each component of the second group, or The first group has no initiator the second group has an initiator, and each component in the starting component set of the first group is distinguishable from the second group, or Both types have initiators, either the initiators are the same or both types have no initiator, and each component in the starting component set of the first group is content-distinguishable from the type of each component of the second group.
sequence group	partitioned item or partitioned group	The first component is either: • A group marked as initiator-distinguishable and the group is initiator-distinguishable from each partition of the second type, or • A group not marked as initiator-distinguishable and the group is content-distinguishable from each partition of the second type.

Type 1	Type 2	How to define them as content-distinguishable
choice group or unordered group	item	The first component is either: • A group marked as initiator distinguishable and The item has an initiator and the initiator value of the group is distinguishable from the initiator value of the item, or The second type has no initiator and the initiator value of the first type is distinguishable from the value of the second type. • A group not marked as initiator distinguishable and Both types have an initiator and the initiator value of the group is distinguishable from initiator value of the item, or The group has an initiator, the item does not, and the value of the initiator is distinguishable from value of the item, or The group has no initiator, the item has an initiator and the starting component set of the group is content-distinguishable from the item, or Both types have initiators and they are the same, both types have no initiator and the type of each component of the group is distinguishable from the value of the second item.
choice group or group	sequence group	The first component is either: • A group marked as initiator distinguishable and Both groups are different partitions of the same partitioned subtree, or The second type has an initiator and those initiators are mutually exclusive, or The second type has no initiator and the first group is initiator-distinguishable from each type in the starting set of the second group. • A group not marked as initiator distinguishable, and Both groups are different partitions of the same partitioned subtree, or Both groups have an initiator and the initiator value of the first group is distinguishable from the initiator value of the second group, or The first group has an initiator, the second group has no initiator and the first group is content-distinguishable from the type of each component in the starting component set of the second group. The first group has no initiator, the second group has an initiator and the type of each component of the first group is content-distinguishable from the second group, or. Both types have initiators and they are the same or both types have no initiator, and the type of each component of the first group is content-distinguishable from the type of each component in the starting component set of the second group.

Type 1	Type 2	How to define them as content-distinguishable
choice group or group	choice group or group	The first component is either: • A group marked as initiator distinguishable and Both groups are different partitions of the same partitioned subtree, or The second type has an initiator and those initiators are mutually exclusive, or The second group has no initiator and the first group is initiator-distinguishable from the type of each component of the second group. • A group not marked as initiator distinguishable and Both groups are different partitions of the same partitioned subtree, or Both groups have an initiator and the initiator of the first group is distinguishable from the initiator of the second group, or The first group has an initiator, the second group has no initiator and the first group is content-distinguishable from the type of each component of the second group, or The first group has no initiator the second group has an initiator, and each component of the first group is distinguishable from the second group, or Both types have initiators and the initiators are the same or both types have no initiator, and the type of each component of the first group is content-distinguishable from the type of each component of the second group.
choice group or group	partitioned item or partitioned group	The first component is either: • A group marked as initiator-distinguishable and the group is initiator-distinguishable from each partition of the second type, or • A group not marked as initiator-distinguishable and the group is content-distinguishable from each partition of the second type.

Ending-distinguishable types

Ending-distinguishability is used to determine if the end of a data object is distinguishable from the start of any other data object that could be next in the data.

The following table describes how two types may be ending-distinguishable. This is helpful if you are validating data and you receive a message that says a type exists, but it belongs to the wrong component.

Type1	Type2	How to define them as ending-distinguishable
item	item or sequence group or choice group or unordered group	Type1 is bound or the value of Type1 is content-distinguishable from Type2.
item	partitioned item or partitioned group	Type1 is bound or the value of Type1 is content-distinguishable from each partition of Type2.

Type1	Type2	How to define them as ending-distinguishable
sequence group	item or sequence group or choice group or unordered group	Either: • Type1 is bound. or For each component in the unbound set of Type1 • If the component has a fixed range it must be ending-distinguishable from Type2. • If the component range is variable, it must be content-distinguishable from Type2 and ending-distinguishable from Type2.
sequence group	partitioned item or partitioned group	Either: • Type1 is bound. or For each component in the unbound set of Type1 • If the component has a fixed range it must be ending-distinguishable from each partition of Type2. • If the component range is variable, it must be content-distinguishable from each partition of Type2 and ending-distinguishable from each partition of Type2.
choice group or unordered group	item or sequence group or choice group or unordered group	Either: • Type1 is bound. or • For each component of Type1, The type of that component is bound, or The type of that component is unbound, and that type must be ending-distinguishable from Type2.
choice group or unordered group	partitioned item or partitioned group	Either: • Type1 is bound. or • For each component of Type1, The type of that component is bound, or The type of that component is unbound, and that type must be ending-distinguishable from each partition of Type2.

Type1	Type2	How to define them as ending-distinguishable
partitioned item or partitioned group	item, sequence group, choice group, or unordered group	For each partition of Type1, either: • The partition is a partitioned type and ending-distinguishable from the second type, or • The partition of the first type is bound, or • The partition of the first type is unbound and ending-distinguishable from the second type.
partitioned item or partitioned group	partitioned item or partitioned group	For each partition of the first type, either: • The partition is a partitioned type and ending-distinguishable from the second type, or • The partition of the first type is bound, or • The partition of the first type is unbound and ending-distinguishable from the second type.

Distinguishable data objects of an implicit group

When a data object belongs to an implicit sequence, determining the component a data object belongs to depends on the format of the sequence, the type definition of a component, and the component properties.

Guidelines for defining an implicit delimited sequence

- If the type of a component has a terminator, that terminator must be different from the delimiter of the explicit group. If the delimiter and terminator are both defined as literal values, a type analysis confirms both have different values.
- The type of a component cannot be a binary item whose length is not fixed or sized.
- If the type of a component or a contained component is not bound, the type of the component cannot have a delimiter that is the same as the delimiter of the implicit sequence.
- If the range of a component is not bound, the type of that component must be content-distinguishable from the type of each component in the following set of that component.
- If the type of a component has both an initiator and terminator, the nested delimiters do not have restrictions. If this is not the case, all contained delimiters must be different.

Guidelines for defining an implicit sequence that has no delimiter

- The type of a component cannot be a binary item whose length is not fixed or sized.
- If the range of a component is bound, but the type of the component is unbound, the type must be ending-distinguishable from each type in the component’s following set.
- If the range of a component is not bound, all the following rules apply:
  - The type of the component must be content-distinguishable from each type in the component’s following set.

- If the maximum range for that component is greater than one, and the type is unbound, the type must be ending-distinguishable from itself.
- If the type of the component is unbound, the type must be end-distinguishable from each type in the component's following set.

Guidelines for defining an implicit unordered group that is delimited

- The type of a component cannot be a binary item whose length is not fixed.
- The type of a component must be content-distinguishable from the type of each other component.
- If the type of a component has both an initiator and terminator, the nested delimiters do not have restrictions. If this is not the case, all contained delimiters must be different.

Guidelines for defining an implicit unordered group that has no delimiter

- The type of a component cannot be a binary item whose length is not fixed.
- The type of a component must be content-distinguishable from the type of each other component.
- If the type of a component is unbound, the type must be ending-distinguishable from the type of each other component.
- If the maximum range is greater than one and the type is unbound, the type must be ending-distinguishable from itself.

Distinguishable data objects of an explicit group

When a data object belongs to an unordered group, distinguishing one data object from another depends on the format of the unordered group and the type definition of each component.

Guidelines for defining an explicit fixed group

- In a fixed group, each component must be either an item of fixed size, an item padded to a fixed length, or an explicit fixed group.
- The range maximum for each component must have a specific value; it cannot be **s**. A fixed amount of space is assumed in the data stream for each series member of a component in a fixed group.

Guidelines for defining an explicit delimited group

A component of an explicit delimited group can be a group or item, with the following restrictions:

- If the type of a component has a terminator, that terminator must be different from the delimiter of the explicit group. If the delimiter and terminator are both defined as a literal, analysis confirms if both have different values.
- If a component of an explicit delimited group is a binary item whose length is not fixed, that component must be sized by the component that precedes it.
- If the type of a component, or a contained component, does not have a terminator and the type of the component has a delimiter that is the same as the delimiter of the explicit group, the type of the component *must* be an explicit sequence. In this case, the delimiter appears as a placeholder for every data object if data of the same delimiter follows it.

- If an explicit group has a delimiter, the range maximum of any component other than the last one must have a specific value; it cannot be **s**. A delimiter is assumed in the data stream for each series member of a component of an explicit group with a delimiter.

Objects of a choice group

When a data object belongs to a choice group, which component the data object belongs to is based on the order the components appear in the type definition.

In a choice group, if a component is unbound it must be content-distinguishable from the type of each component that follows it in the component list.

Objects of a partitioned type

When data objects of different types appear in the same place in the data, the types must be distinguishable. The data must be distinguishable in the type definition. When a partition is part of a partitioned object, the process of cycling through the partitioned subtree begins to make the determination of which partition the data object belongs to.

In a partitioned type, each partition must be content-distinguishable from the type of each partition that follows it.

Distinguishable syntax objects

The system must be able to distinguish between different syntax objects contained within a group. To ensure that the syntax objects are distinguishable, use the following guidelines:

- If a component is unbound, make sure its delimiter and any delimiter contained in it is distinguishable from the delimiter of the group. If you get an analysis error, look for missing placeholders or components with a range maximum of **s**.

  In a component or a contained component with the same delimiter as the type delimiter, the system assumes that the delimiter appears as a placeholder for every data object if data with the same delimiter follows it.
- If the delimited group has a terminator, make sure that the delimiter and any contained delimiter is distinguishable from the terminator.

  The type tree analysis verifies if the above requirement is met in a non-partitioned group.

Chapter 14. Type Tree Analyzer

Use the type tree analyzer to analyze your type definitions.

The type tree analyzer analyzes type definitions and ensures internal consistency. For example, if you defined a group as fixed, but accidentally defined one of its items with no **Padded To** length, errors occur during analysis.

The analyzer checks your data definitions for logical consistency. It does not compare your definitions to your actual data. The resulting analyzer messages indicate whether your type tree definitions are acceptable; not whether they match your data.

To analyze a type tree:
1. Select the type tree.
2. From the **Tree** menu, choose **Analyze**.
3. Choose **Structure Only**, **Logic Only**, or **Logic and Structure**.
4. Click **Results** to view the results.

Error scenario

After analysis, the Analysis Results dialog displays error L201.

The error concerns the type **File**. **File**'s first component, **Record(s)** is a series. The way the data is currently defined does not allow one **Record** to be distinguishable from the next **Record** in the series.

There are a number of reasons why the records might not be distinguishable.

A good way to figure out how to resolve an error is to look at the data. When you look at the data, it is clear where a record ends. Ask the question: *How can I tell the difference between one record and the next?* Each record ends with a carriage return/linefeed.

In this case, you forgot to define CR/LF as the terminator of **Record**.

To resolve the problem, open the Properties window for **Record** and define the terminator for **Record**. After you define the terminator for **Record**, analyze the tree again and receive no errors.

Internal consistency

The analyzer checks the logic of your data definitions. For example, suppose you defined a group **PO** as fixed and consists of **Line(s)**. For the **PO** to be fixed, it cannot have an indefinite number of **Lines**. An analysis error would occur, indicating that you defined **PO** as fixed but it has a variable number of components.

© Copyright IBM Corp. 2006

Mapping effects

The analyzer helps you define your data by locating objects in your input data and creating the objects in your output data. The analysis indicates whether there is something in your definition that may prevent a correct mapping your data.

In the preceding example using orders.mtt, if the orders data was your input and you did not terminate each record with a CR/LF, the Map Designer would not know when a record ends. It would not be able to find each record in the input.

If the orders data was the output and you did not define the terminator, CR/LF would not be placed at the end of each record and the records would be received one after the other, wrapped, with no line breaks.

If you do not analyze a tree before you map the data or you analyze a tree and do not resolve the errors, you will be warned that you may receive unpredictable results when you map.

When to Analyze Structure or Logic

You can choose to analyze the structure or the logic of your type definitions, or both.

Logical analysis

Logical analysis addresses the integrity of the relationships that you define. Logical analysis detects, for example, undefined components, components that are not distinguishable from one another, item restrictions that do not match the properties of that item, and circular type definitions. The analyzer also checks delimiter relationships to each other and to components, undefined inherited relationships, and logic errors contained in component rules.

Structural analysis

Typically, you should not encounter structural analysis errors. Structural analysis addresses the integrity of the underlying database. Structural analysis may be able to detect and possibly correct defects caused by system environment failures.

Error and warning messages

Analysis results may contain error and warning messages.

Warnings are relatively insignificant. They indicate an inconsistency that occurred when you changed something in the tree that was resolved. For example, if you change a group to an item, the analyzer removes the components of that type because items do not have components. To remind you of this change, the analyzer issues a warning.

Errors are important. An error is a problem in your type definitions that you should correct. An error may result in unpredictable results in your mapping.

Occasionally there are errors that prevent the analysis of the rest of your definitions (for example, when a component type cannot be found in the tree). `Analysis halted before completion` is displayed. This error might be due to one of the following:

- Undefined COMPONENTs found: ending analysis.

- Circular reference found in COMPONENT list: ending analysis.

When this occurs, click **Results**, correct the errors, and analyze the tree again.

Chapter 15. Utilities for XML

There are XML utilities provided with the Type Designer.

- Use the XML Type Tree/Schema Synchronization Utility to synchronize a type tree with an XML schema or DTD.
- Use the XML Type Tree Compatibility Utility to add non-XML types back into a target type tree. The XML Type Tree Compatibility Utility is also available from the command line (dsxmlconv).
- Use Any-2-XML to create a map that can transform any input data into XML output. Any-2-XML is also available from the command line (dtxany2xml)

There is also a Map Migration Utility (dsmapconv) that updates map source files and the associated XML type trees to the current XML format that is available from the command line.

See the Utility Commands documentation for information about all command line utilities.

XML Type Tree/Schema Synchronization utility

Use this utility to synchronize a type tree with an XML schema or DTD.

You can only use this utility to update 8.0 type trees that have been generated from a DTD or schema using the XML DTD or XML Schema importers.

To synchronize a type tree with a an XML DTD or schema:

1. In the Type Designer, select **Tree** → **Synchronize**.

   The XML Type Tree/Schema Synchronization Utility appears.
2. From the drop-down list, select the importer that was originally used to create the type tree: XML, which is the default setting, or DTD.
3. Click the browse button and choose a type tree.
4. In the next field, click the browse button and select the modified DTD or schema file that you want to synchronize the type tree to.

   The **Create backup file** option is enabled by default and the location for the backup file is automatically displayed (based on the type tree you selected). You can change the location and filename extension if desired.
5. If applicable, select the national language of the original type tree.
6. Click **Next** to begin synchronization.

   Possible results include:
   - If no modifications have been made to the original type tree created from the DTD or schema, the synchronization process completes successfully.
   - If modifications have been made to the original type tree created from the DTD or schema, they are displayed in the left pane of the utility in red text. At this point you can incorporate any differences into the new (target) type tree.

   For example, you can take an attribute that was added to the original type tree and add it to the target type tree (see ″Modifying the Target Type Tree″).
7. Make changes as needed and click **Next**.

© Copyright IBM Corp. 2006

When you see that the type tree was synchronized successfully, click **Finish** to close the utility.

XML type tree compatibility utility

If you have type trees generated with the XML Schema or DTD importers of WebSphere Transformation Extender 6.7 or 7.5 or older, and you are now using version 8.0, you must update your type trees in order to use the current XML validation. To perform the task of updating your type trees, you can use the XML Type Tree Compatibility Utility. This utility is also available from the command line (`dsxmlconv`).

When using the XML Type Tree Compatibility Utility, there are two important requirements. One is that you must specify the correct version of the WebSphere Transformation Extender XML Schema or DTD importer that was used to create the original type tree. The other is that you must have the original DTD or Schema.

To make a type tree compatible with WebSphere Transformation Extender 8.0 XML validation:

Note: The original DTD/schema file is required when using the XML Type Tree Compatibility Utility.

1. In the Type Designer, select **Tree** → **Convert**.

   The XML Type Tree Compatibility Utility appears.
2. From the drop-down list, select the importer that was originally used to create the type tree: **XML Schema**, which is the default setting, or **DTD**.
3. Click the browse button and choose a type tree.
4. From the drop-down list, choose the version of the WebSphere Transformation Extender importer used to create the type tree. Choices include: **6.7 (268)**, **6.7.1 (306)**, **6.7.2 (342)**, and **7.5**.

   Note: It is essential that you specify the correct version of the WebSphere Transformation Extender importer used to create the original type tree. If you use an incorrect version number, the conversion process could complete successfully, however, the type tree would be invalid.
5. In the **Select the XML grammar location** field, accept the default value for the original XML DTD or Schema used to create the type tree if it is accurate. Otherwise, enter the correct location of the XML grammar.
6. Enable or disable the **Create backup file** option. By default, the option is enabled and the location for the backup file is automatically displayed in the subsequent field (based on the type tree you selected). You can change the location and filename extension if needed.
7. If applicable, select the national language of the original type tree.
8. Click **Next** to begin the update process.

   Possible results include the following:
   - If no modifications have been made to the original type tree created from the DTD or schema, the process completes successfully.
   - If modifications have been made to the original type tree, they are displayed in the right pane of the utility in red text. At this point you can incorporate any differences into the new (target) type tree (see ″Modifying the Target Type Tree″).

For example, the non-XML types that were added to the original type tree can be added to the new (target) type tree at this point.

9. When you see that the type tree was synchronized successfully, click **Finish** to close the utility.

Modifying the target type tree

When the XML utility encounters any non-XML types that have been added to the type tree after it was first created from a schema or DTD, the unrecognized types are displayed in red text.

The instructions below describe how to add the types back into your target tree, however, you have the ability from within this utility to make any necessary modifications to the target type tree in the same manner as you would from the type tree window (using add, delete, copy, and paste operations).

The following procedures can be used for both XML utilities: XML Type Tree Compatibility Utility and XML Type Tree/Schema Synchronization Utility.

To add a type to the target tree

1. In the left pane of the XML utility, right-click on the type and select **Edit** → **Copy**.
2. In the right pane (which displays the target type tree), right-click on the corresponding (parent) type and select **Edit** → **Paste**.

   The type appears in the target tree.
3. After making all changes, click **Next** to continue.

To add a component to the target tree

1. In the right pane (which displays the target type tree), select the type to add the component to.
2. At the bottom of the pane, select the **Components** tab.
3. Press **Alt** and then select-and-drag the type to the **Component** column of the **Components** tab.
4. After adding all components, click **Next** to continue.

Any-2-XML

The Any-2-XML utility is an extension of the type tree Export as Schema functionality that you can use to create a map that can transform any input data into XML output, based on a type tree structure that you select.

How it works

Choose a type within a type tree that models the XML output you want to produce. The utility converts the type into an XML Schema from which it creates a new XML type tree. A new map source file is created containing the new map. The input card of the new map references the original type tree and input and the output card references the new XML type tree and the new output file name. The result of running the new map is XML output.

Using Any-2-XML from the Type Designer

1. Using the Type Designer (version 8.1 or later), open a type tree that models the XML output structure you are seeking.
2. In the type tree window, select a type that represents the portion of the input file that you want to export. (This must be a group or item.)
3. Select **Tree** → **Export as Schema**. The Export as Schema window opens.
4. Enable the **Create a map that transforms the input data described by the type tree into XML output** check box.
5. In the **Input file name** field, click the navigation button and select the input file that corresponds to the type tree.
6. In the **Output file name** field, enter a filename and path for the XML output file that will be generated as a result of the executing the new map.
7. In the **Map file name** field, enter a name for the new map and map source file.
8. Select **OK** to start the utility. The following files are created:
   - XML Schema (*orig_type_tree _name_plus_exported_type_name*.xsd)
   - Type tree (*orig_type_tree _name_plus_exported_type_name*.mtt)
   - Map source file (.mms)

   A log file (*map_name*.log) is created only when an error occurs.
9. Using the Map Designer, open the new map.
10. Build and run the map. An XML output file is created.

Chapter 16. Return codes and error messages

Type Tree Analyzer errors and warnings

The Type Tree Analyzer checks the logic and internal consistency of your data definitions. Type tree analyzer error and warnings messages are issued on the type of analysis that is performed: logic and structural.

- Logical analysis addresses the integrity of the relationships that you define in the type tree.
- Structural analysis addresses the integrity of the underlying database.

Warnings indicate a successful analysis and are relatively insignificant. Warning messages provide information about inconsistencies that occurred and were automatically resolved when the type tree was changed.

Errors are important. Error messages provide information about errors in your type definitions that you should correct. An error may result in unpredictable results in your mapping.

Words in italics represent information that varies, indicating information specific to the type for which the message is generated.

Type tree analysis logic error messages

The following table lists the logic error messages that result from a logical analysis of a type tree:

Return Code
Message

L100 COMPONENT neither inherited nor local: \`*type name*' of TYPE: \`*type name*'

Hint: Look at the super-type's component list. The component is a valid type, but the supertype has a component list that restricts you from using this type as a component. You may have added subtype components before adding supertype components. Either remove all supertype components or add the components in error to the component list of the supertype.

L101 This GROUP must have at least one component - TYPE: \`*type name*'

Hint: If you want to map this group, add components. If you do not want to map it, make it a category.

L102 Circular reference found in COMPONENT # (\`*type name*') - TYPE: \`*type name*'

Hint: Look at the type of the component in error. It is probably missing an initiator or terminator.

L103 Circular reference found in Floating Component type - TYPE: \`*type name*'

Hint: Look at the floating component type in error. It is probably missing an initiator or terminator.

L104 DELIMITER for TYPE - \`*type name*' must have a value

© Copyright IBM Corp. 2006

Hint: All delimited groups need a delimiter. Edit delimited group properties to insert the missing delimiter.

L105 DELIMITER type neither inherited nor local - TYPE: `*type name'*

Hint: The delimiter name has been entered incorrectly. It should be the name of a local type, or the name of an inherited delimiter, or the name of a type in the sub-tree of the inherited delimiter.

L106 Default DELIMITER not specified - TYPE: `*type name'*

Hint: This Type was specified with a FIND option for its delimiter. Please add a default value to define what to use for building outputs.

L107 Default DELIMITER not in restriction list - TYPE: `*type name'*

Hint: This delimiter was specified as a syntax item. Add the default value to the restriction list for that syntax item.

L108 DELIMITER type is not a SYNTAX ITEM - TYPE: `*type name'*

Hint: Delimiters specified as an item must be specified to be interpreted as SYNTAX to set the value of the delimiter if it appears as a component in a data stream.

L109 DELIMITER type has no restriction list - TYPE: `*type name'*

Hint: All syntax items need a restriction list.

L110 RELEASE CHARACTER neither inherited nor local - TYPE: `*type name'*

Hint: The release character name has been entered incorrectly. It should be either the name of a local type, the name of an inherited release character, or the name of a type in the sub-tree of the inherited release character.

L111 Default RELEASE CHARACTER not specified - TYPE: `*type name'*

Hint: This Type was specified with a syntax item for its release character. Please add a default value to define a value for the release character that has not been encountered in the data.

L112 Default RELEASE CHARACTER not in restriction list - TYPE: `*type name'*

Hint: This Type was specified with a syntax item for its release character. Please add the default value to the restriction list of that syntax item.

L113 RELEASE CHARACTER type is not a SYNTAX ITEM - TYPE: `*type name'*

Hint: Release characters specified as an item must be specified to be interpreted as SYNTAX to set the value of the release character if it appears as a component in a data stream.

L114 RELEASE CHARACTER type has no restriction list - TYPE: `*type name'*

Hint: All syntax items need a restriction list.

L115 Floating Component TYPE neither inherited nor local - TYPE: `*type name'*

Hint: The floating component name has been entered incorrectly. It should be either the name of a local type, the name of an inherited floating component, or the name of a type in the sub-tree of the inherited floating component.

L116 INITIATOR type neither inherited nor local - TYPE: `type name'

Hint: The initiator name has been entered incorrectly. It should be either the name of a local type, the name of an inherited initiator, or the name of a type in the sub-tree of the inherited initiator.

L117 Default INITIATOR not specified - TYPE: `*type name'*

Hint: This Type was specified with a syntax item for its initiator. Add a default value to define a value for that initiator has not been encountered in the data.

L118 Default INITIATOR not in restriction list - TYPE: `*type name'*

Hint: This Type was specified with a syntax item for its initiator. Add the default value to the restriction list of that syntax item.

L119 INITIATOR type is not a SYNTAX ITEM - TYPE: `*type name'*

Hint: Initiators specified as an item must be specified to be interpreted as SYNTAX to set the value of the initiator if it appears as a component in a data stream.

L120 INITIATOR type has no restriction list - TYPE: `*type name'*

Hint: All syntax items need a restriction list.

L121 TERMINATOR type neither inherited nor local - TYPE: `*type name'*

Hint: The terminator name has been entered incorrectly. It should be either the name of a local type, the name of an inherited terminator, or the name of a type in the sub-tree of the inherited terminator.

L122 Default TERMINATOR not specified - TYPE: `*type name'*

Hint: This Type was specified with a syntax item for its terminator. Add a default value to define a value for that terminator has not been encountered in the data.

L123 Default TERMINATOR not in restriction list - TYPE: `*type name'*

Hint: This Type was specified with a syntax item for its terminator. Please add the default value to the restriction list of that syntax item.

L124 TERMINATOR type is not a SYNTAX ITEM - TYPE: `*type name'*

Hint: Terminators specified as an item must be specified to be interpreted as SYNTAX to set the value of the terminator if it appears as a component in a data stream.

L125 TERMINATOR type has no restriction list - TYPE: `*type name'*

Hint: All syntax items need a restriction list.

L126 COMPONENT range minimum (#) greater than range maximum (#) - COMPONENT `*type name'* - TYPE: `*type name'*

Hint: The minimum range must be less than or equal to the maximum range.

L127 COMPONENT range minimum (#) less than inherited range minimum(#) - COMPONENT `*type name'* - TYPE: `*type name'*

Hint: The component in error has been inherited. Look at the range of the component with the same name in the super-type's component list.

L128 COMPONENT range maximum (#) greater than inherited range maximum(#) - COMPONENT `*type name'* - TYPE: `*type name'*

Hint: The component in error has been inherited. Look at the range of the component with the same name in the super-type's component list.

L129 COMPONENT RULE references a COMPONENT later in the component list - `*type name'* - TYPE: `*type name'*

Hint: Move the component rule to the component later in the list.

L130 COMPONENT RULE references undefined type - COMPONENT # of TYPE: `*type name*'

Hint: Verify the spelling of the data object name. The rule should reference a data object name of the component or a data object name of a component earlier in the component list.

L131 Reserved for future use.

L132 Invalid partitioning: TYPE has no SUBTYPES - TYPE: `*type name*'

Hint: Remove the partitioned option from the class window or add sub-types to the Type in error.

L133 Type of COMPONENT exists, but its relative name is not valid: `*type name*' in TYPE: `*type name*'

Hint: To get the correct relative name, drag the type you want to use as a component and drop it in the component list of the Type. (Remember to delete the invalid component!)

L134 Reference to `ANY' not allowed: COMPONENT number # of TYPE: `*type name*'

Hint: In this case, the Type in error is a group and it is not the root of a partitioned tree. ANY cannot be used if that component needs to be validated. So, if that group is partitioned, you cannot use ANY for a component up to and including the identifier (if there is one). If that group is not partitioned, you cannot use ANY at all.

L135 COMPONENT number # cannot reference a CATEGORY in TYPE: `*type name*' (because group is not partitioned)

Hint: In this case, the Type in error is a group and it's not the root of a partitioned tree. A category cannot be used if the component must be validated. So, if that group is partitioned, you cannot use a category for a component up to and including the identifier (if there is one). If that group is not partitioned, you cannot use a category as a component at all.

L136 COMPONENT `*type name*' occurs more than once in list - TYPE: `*type name*'

Hint: Each component in the same component list must have a unique type name. Try to make sub-types of the type name in error and replace each non-unique component with one of the new sub-types.

L137 COMPONENT `*type name*' and its super-type cannot be in same COMPONENT LIST (in TYPE: `*type name*')

Hint: Try making another sub-type of the super-type and replace the super-type reference with the new sub-type.

L138 COMPONENT `*type name*' is same type as delimiter - TYPE: `*type name*'

Hint: A component and a delimiter cannot have the same name. You may need to add sub-types to the type name used in error to resolve this one.

L139 COMPONENT `*type name*' is sub-type of delimiter - TYPE: `*type name*'

Hint: This occurs when a syntax item is used to specify a delimiter. You can add another sub-type to the syntax item and replace the delimiter name with the new sub-type name.

L140 COMPONENT `*type name*' is super-type of delimiter - TYPE: `*type name*'

Hint: This occurs when a syntax item is used to specify a delimiter. You can add another sub-type to the syntax item and replace the component name with the new sub-type name.

L141 COMPONENT `*type name'* is same type as initiator - TYPE: `*type name'*

Hint: A component and an initiator cannot have the same name. You can add sub-types to the type name used in error and replace both the component name and the initiator name.

L142 COMPONENT `*type name'* is sub-type of initiator - TYPE: `*type name'*

Hint: This occurs when a syntax item is used to specify an initiator. You can add another sub-type to the syntax item and replace the initiator name with the new sub-type name.

L143 COMPONENT `*type name'* is super-type of initiator - TYPE: `*type name'*

Hint: This occurs when a syntax item is used to specify an initiator. You can add another sub-type to the syntax item and replace the component name with the new sub-type name.

L144 COMPONENT `*type name'* is same type as terminator - TYPE: `*type name'*

Hint: A component and a terminator cannot have the same name. Try adding sub-types to the type name used in error and replace both the component name and terminator name with one of the new sub-types.

L145 COMPONENT `*type name'* is sub-type of terminator - TYPE: `*type name'*

Hint: This occurs when a syntax item is used to specify a terminator. You can add another sub-type to the syntax item and replace the terminator name with the new sub-type name.

L146 COMPONENT `*type name'* is super-type of terminator - TYPE: `*type name'*

Hint: This occurs when a syntax item is used to specify a terminator. You can add another sub-type to the syntax item and replace the component name with the new sub-type name.

L147 COMPONENT `*type name'* is same type as Floating Component - TYPE: `*type name'*

Hint: Make both the floating component name and the component name sub-types of the floating component.

L148 COMPONENT `*type name'* is sub-type of Floating Component - TYPE: `*type name'*

Hint: Make both the floating component name and the component name sub-types of the floating component.

L149 COMPONENT `*type name'* is super-type of Floating Component - TYPE: `*type name'*

Hint: Make both the floating component name and the component name sub-types of the floating component.

L150 COMPONENT `*type name'* is same type as release character - TYPE: `*type name'*

Hint: This occurs when a syntax item is used to specify a release character. You can add sub-types to the syntax item and replace both the component name and the release character name with the new sub-type names.

L151 COMPONENT `*type name'* is sub-type of release character - TYPE: `*type name'*

Hint: This occurs when a syntax item is used to specify a release character. You can add another sub-type to the syntax item and replace the release character name with the new sub-type name.

L152 COMPONENT `*type name'* is super-type of release character - TYPE: `*type name'*

Hint: This occurs when a syntax item is used to specify a release character. You can add sub-types to the syntax item and replace the component name with the new sub-type name.

L153 DELIMITER `*type name'* is same type as initiator - TYPE: `*type name'*

Hint: A delimiter and an initiator cannot have the same name. You may need to add sub-types to the type name used in error and replace both the delimiter and initiator names to refer to the new sub-types.

L154 DELIMITER `*type name'* is sub-type of initiator - TYPE: `*type name'*

Hint: This occurs when a syntax item is used to specify both an initiator and a delimiter. You can add another sub-type to the syntax item and replace the initiator name with the new sub-type name.

L155 DELIMITER `*type name'* is super-type of initiator - TYPE: `*type name'*

Hint: This occurs when a syntax item is used to specify both an initiator and a delimiter. You can add another sub-type to the syntax item and replace the delimiter name with the new sub-type name.

L156 DELIMITER `*type name'* is same type as terminator - TYPE: `*type name'*

Hint: A delimiter and a terminator cannot have the same name. You may need to add sub-types to the type name used in error and replace both the delimiter and terminator names to refer to the new sub-types.

L157 DELIMITER `*type name'* is sub-type of terminator - TYPE: `*type name'*

Hint: This occurs when a syntax item is used to specify both a delimiter and a terminator. You can add another sub-type to the syntax item and replace the terminator name with the new sub-type name.

L158 DELIMITER `*type name'* is super-type of terminator - TYPE: `*type name'*

Hint: This occurs when a syntax item is used to specify both a delimiter and a terminator. You can add another sub-type to the syntax item and replace the delimiter name with the new sub-type name.

L159 DELIMITER `*type name'* is same type as release character - TYPE: `*type name'*

Hint: A delimiter and a release character cannot have the same name. You may need to add sub-types to the type name used in error and replace both the delimiter and release character names to refer to the new sub-types.

L160 DELIMITER `*type name'* is sub-type of release character - TYPE: `*type name'*

Hint: This occurs when a syntax item is used to specify both a delimiter and a release character. You can add another sub-type to the syntax item and replace the release character name with the new sub-type name.

L161 DELIMITER `*type name'* is super-type of release character - TYPE: `*type name'*

Hint: This occurs when a syntax item is used to specify both a delimiter and a release character. You can add another sub-type to the syntax item and replace the delimiter name with the new sub-type name.

L162 DELIMITER `*type name'* is same type as Floating Component - TYPE: `*type name'*

Hint: A delimiter and a floating component cannot have the same name. Try adding sub-types to the type name used in error and replace both the delimiter and floating component names to refer to the new sub-types.

L163 DELIMITER `*type name'* is sub-type of Floating Component - TYPE: `*type name'*

Hint: This occurs when a syntax item is used to specify both a delimiter and a floating component. You can add another sub-type to the syntax item and replace the floating component name with the new sub-type name.

L164 DELIMITER `*type name'* is super-type of Floating Component - TYPE: `*type name'*

Hint: This occurs when a syntax item is used to specify both a delimiter and a floating component. You can add another sub-type to the syntax item and replace the delimiter name with the new sub-type name.

L165 INITIATOR `*type name'* is same type as terminator - TYPE: `*type name'*

Hint: An initiator and a terminator cannot have the same name. You may need to add sub-types to the type name used in error and replace both the initiator and terminator names to refer to the new sub-types.

L166 INITIATOR `*type name'* is sub-type of terminator - TYPE: `*type name'*

Hint: This occurs when a syntax item is used to specify both an initiator and a terminator. You can add another sub-type to the syntax item and replace the terminator name with the new sub-type name.

L167 INITIATOR `*type name'* is super-type of terminator - TYPE: `*type name'*

Hint: This occurs when a syntax item is used to specify both an initiator and a terminator. You can add another sub-type to the syntax item and replace the initiator name with the new sub-type name.

L168 INITIATOR `*type name'* is same type as release character - TYPE: `*type name'*

Hint: This occurs when a syntax item is used to specify both an initiator and a release character. You can add sub-types to the syntax item and replace both the initiator name and the release character name with a new sub-type name.

L169 INITIATOR `*type name'* is sub-type of release character - TYPE: `*type name'*

Hint: This occurs when a syntax item is used to specify both an initiator and a release character. You can add another sub-type to the syntax item and replace the release character name with the new sub-type name.

L170 INITIATOR `*type name'* is super-type of release character - TYPE: `*type name'*

Hint: This occurs when a syntax item is used to specify both an initiator and a release character. You can add another sub-type to the syntax item and replace the initiator name with the new sub-type name.

L171 INITIATOR `*type name'* is same type as Floating Component - TYPE: `*type name'*

Hint: An initiator and a floating component cannot have the same name. Try adding sub-types to the type name used in error and replace both the initiator and floating component names with the new sub-types.

L172 INITIATOR `*type name'* is sub-type of Floating Component - TYPE: `*type name'*

Hint: This occurs when a syntax item is used to specify both an initiator and a floating component. You can add another sub-type to the syntax item and replace the floating component name with the new sub-type name.

L173 INITIATOR `*type name'* is super-type of Floating Component - TYPE: `*type name'*

Hint: This occurs when a syntax item is used to specify both an initiator and a floating component. You can add another sub-type to the syntax item and replace the initiator name with the new sub-type name.

L174 TERMINATOR `*type name'* is same type as release character - TYPE: `*type name'*

Hint: A terminator and a release character cannot have the same name. You may need to add sub-types to the type name used in error and replace both the terminator and release character names with the new sub-types.

L175 TERMINATOR `*type name'* is sub-type of release character - TYPE: `*type name'*

Hint: This occurs when a syntax item is used to specify both a terminator and a release character. You can add another sub-type to the syntax item and replace the release character name with the new sub-type name.

L176 TERMINATOR `*type name'* is super-type of release character - TYPE: `*type name'*

Hint: This occurs when a syntax item is used to specify both a terminator and a release character. You can add another sub-type to the syntax item and replace the terminator name with the new sub-type name.

L177 TERMINATOR `*type name'* is same type as Floating Component - TYPE: `*type name'*

Hint: A terminator and a floating component cannot have the same name. You may need to add sub-types to the type name used in error and replace both the terminator and floating component names with the new sub-types.

L178 TERMINATOR `*type name'* is sub-type of Floating Component - TYPE: `*type name'*

Hint: This occurs when a syntax item is used to specify both a terminator and a floating component. You can add another sub-type to the syntax item and replace the floating component name with the new sub-type name.

L179 TERMINATOR `*type name'* is super-type of Floating Component - TYPE: `*type name'*

Hint: This occurs when a syntax item is used to specify both a terminator and a floating component. You can add another sub-type to the syntax item and replace the terminator name with the new sub-type name.

L180 RELEASE CHARACTER `*type name'* is same type as Floating Component - TYPE: `*type name'*

Hint: A release character and a floating component cannot have the same name. You may need to add sub-types to the type name used in error and replace both the release character and floating component names to refer to the new sub-types.

L181 RELEASE CHARACTER `*type name'* is sub-type of Floating Component - TYPE: `*type name'*

Hint: This occurs when a syntax item is used to specify both a release character and a floating component. You can add another sub-type to the syntax item and replace the floating component name with the new sub-type name.

L182 RELEASE CHARACTER `*type name'* is super-type of Floating Component - TYPE: `*type name'*

Hint: This occurs when a syntax item is used to specify both a release character and a floating component. You can add another sub-type to the syntax item and replace the release character name with the new sub-type name.

L183 COMPONENT NAME ambiguous: `*type name'* in TYPE: `*type name'*

Hint: This type has a component whose relative name can be associated with more than one type in the type tree. Rename the conflicting types.

L184 RESTRICTION longer than max TYPE size - RESTRICTION # of TYPE: `*type name'*

Hint: The Type in error is an item. Either change the maximum size of the item or remove the restriction.

L185 RESTRICTION used in an earlier partition - RESTRICTION # of TYPE: `*type name'*

Hint: Item Partitions must have mutually exclusive restrictions. Remove the restriction from one of the partition restriction lists.

L186 Type of COMPONENT does not exist - `*type name'* in TYPE: `*type name'*

Hint: You probably entered an incorrect type name. Try the drag and drop approach to get the correct one.

L187 TYPE must be partitioned (since in a partitioned tree and has sub-types) - TYPE: `*type name'*

Hint: All types in a partitioned sub-type must have mutually exclusive data objects. Set the partitioned property for the type in error.

L188 Reserved for future use.

L189 TYPE is FIXED, but COMPONENT # is not fixed - TYPE: `*type name'*

Hint: If the component is not intended to be fixed in size, change the group format for the Type to implicit. If the group format is intended to be fixed, check the component: if that component is an item, make sure it has a Padded To length; if that component is a group, change its type to be of fixed syntax.

L190 BINARY text ITEM used as COMPONENT neither FIXED nor SIZED - COMPONENT # of TYPE: *`type name'*

Hint: The size of a binary text item must either have a Padded To length or it must be sized by the previous component.

L191 COMPONENT with SIZED attribute is not an UNSIGNED INTEGER ITEM TYPE - COMPONENT # of TYPE: *`type name'*

Hint: A component used to size the component that follows it must be defined as an unsigned integer item type.

L192 The last COMPONENT in the COMPONENT LIST may not have a SIZED attribute: TYPE: *`type name'*

Hint: Specify a component to follow the one with the sized attribute.

L193 Range of COMPONENT # must have a maximum value to indicate how many placeholders are needed for its series in TYPE: `type name'.

Hint: Change the range maximum to a specific value (not ″s″) if you may re-define the data this way.

L194 Cannot distinguish delimiter from terminator in TYPE: `type name'.

Hint: Make the range of the last component in the type fixed or make the delimiter of the type different from its terminator.

L195 Cannot distinguish delimiter contained in COMPONENT # from terminator of TYPE: `type name'.

Hint: Make that component bound or make that contained delimiter different from the type terminator.

L196 Cannot distinguish delimiter of COMPONENT # from delimiter of TYPE: `type name'.

Hint: Make that component bound or make that component's delimiter different from the type delimiter.

L197 Cannot distinguish delimiter of COMPONENT # from delimiter of TYPE: `type name'.

Hint: Make that component bound, or make that component's delimiter different from the type delimiter, or specify a range maximum that has a specific value (not ″s″) for the last component of COMPONENT #.

L198 Cannot distinguish delimiter contained in COMPONENT # from delimiter of TYPE: `type name'.

Hint: Make that contained component bound or make that contained component's delimiter different from the type delimiter.

L200 Cannot distinguish delimiter contained in COMPONENT # from delimiter of TYPE: `type name'.

Hint: Either make that contained component bound, make that contained component's delimiter different from the type delimiter, or specify a range maximum that has a specific value (not ″s″) for the last component of the contained component.

Logic error and warning messages

The tables in this section list the logic warning messages that result from a logic analysis of a type tree.

The following table lists the warnings than can result when a map is compiled.

Warnings should be resolved because they may produce unpredictable results at mapping time.

Return Code
Message

L199
COMPONENT # is not distinguishable from COMPONENT # that may follow in TYPE: `type name'.

Hint: Make the first COMPONENT bound, or look at the tables in "Distinguishable objects" to see how you can define the two component types as distinguishable.

L201
Different data objects of COMPONENT # are not distinguishable in TYPE: `type name'.

Hint: See "Distinguishable objects" for more information about distinguishable objects.

L202
RESTRICTION list deleted: TYPE is not an ITEM - TYPE: *`type name'*

Hint: Type class was changed from an item to a group or category, so the restriction list was deleted. If this was not your intent, change it back to the way it was.

L203
COMPONENT list deleted: TYPE is an ITEM - TYPE: *`type name'*

Hint: Type class was changed from a group to a item or category, so the program deleted its component list. If this was not your intent, change it back to the way it was.

L204
DELIMITER deleted: TYPE is not a DELIMITED GROUP - TYPE: *`type name'*

Hint: Group format was changed from delimited to something else, so the program deleted its delimiter. If this was not your intent, change it back to the way it was.

L205
COMPONENT RULE deleted: TYPE is a CATEGORY - TYPE: *`type name'* (warning)

Hint: Type class was changed from a group to a category, so its component rule was deleted. If this was not your intent, change it back to the way it was.

L206
DELIMITER cannot be found (because first component is not required) - TYPE: *`type name'* (warning)

Hint: If the delimiter is missing, a previously set initiator value or the default value is used.

L251
COMPONENT NAME could apply to more than one type:'type name' in TYPE: `type name' (warning).

Type tree analysis structure error messages

The following table lists the structure error messages that result from a structural analysis of a type tree:

Return Code
Message

S100 Invalid TYPE Name: SubTYPE # of TYPE: `*type name'*

S101 Invalid TYPE chain: SubTYPE # of TYPE: `*type name'*

S118 Invalid TYPE NAME WhereUsed chain - TYPE NAME: `*type name'* (error).

S133 Referenced COMPONENT not `InUse' - COMPONENT # of TYPE: `*type name'* (error).

S134 COMPONENT previously referenced - COMPONENT # (COMP #) of TYPE: `*type name'* (error).

S149 Bad Parent COMPONENT Index - COMPONENT `*type name'* - TYPE: `*type name'* (error)

Type tree analysis structure warning messages

The following table lists the structure warning messages that result from a structural analysis of a type tree:

Return Code
Message

S102 Unused DELIMITER deleted: `*type name'* (at index #)

S103 Invalid DELIMITER pointer deleted - TYPE: `*type name'*

S104 Invalid default DELIMITER pointer deleted - TYPE: `*type name'*

S105 Invalid RELEASE Char pointer deleted - TYPE: `*type name'*

S106 Invalid default RELEASE Char pointer deleted - TYPE: `*type name'*

S107 Invalid INITIATOR pointer deleted - TYPE: `*type name'*

S108 Invalid default INITIATOR pointer deleted - TYPE: `*type name'*

S109 Invalid TERMINATOR pointer deleted - TYPE: `*type name'*

S110 Invalid default TERMINATOR pointer deleted - TYPE: `*type name'*

S111 Resetting DELIMITER Use Count (was # now #) - DELIMITER: `*type name'*

S112 Unused DESCRIPTION deleted: `*type name'* (at index #)

S113 Invalid DESCRIPTION pointer deleted - TYPE: `*type name'*

S114 Resetting DESCRIPTION Use Count (was # now #) - DESCRIPTION: `*type name'*

S115 Invalid Floating Component TYPE pointer deleted - TYPE: `*type name'*

S116 Invalid TYPE UsedInComp chain repaired - TYPE: `*type name'*

S117 Unused TYPE NAME deleted - TYPE NAME: `*type name'* (at index #)

S119 Resetting TYPE NAME use count (was # now #) - TYPE NAME: `*type name'*

S120 Repaired empty TYPE NAME WhereUsed chain - TYPE NAME: `*type name'*

S121 Unused RESTRICTION NAME deleted: `*type name'* (at index #)

S122 Invalid RESTRICTION NAME deleted no DESCRIPTION was available - TYPE: `*type name'* .

S123 Invalid RESTRICTION NAME deleted DESCRIPTION was `*type name'* - TYPE: `*type name'*

S124 Resetting RESTRICTION NAME Use Count (was # now #) - RESTRICTIONS: `*type name'*

S125 Unused RESTRICTION DESCRIPTION deleted: `*type name'* (at index #)

S126 Invalid RESTRICTION DESCRIPTION deleted - TYPE: `*type name'*

S127 Resetting RESTRICTION DESCRIPTION Use Count (was # now #) - RESTRICTIONS: `*type name'*

S128 Unused RULE deleted: `*type name'* (at index #)

S129 Invalid RULE pointer deleted - COMPONENT # of TYPE: `*type name'*

S130 Resetting RULE Use Count (was # now #) - RULE: `*type name'*

S131 Invalid COMPONENT TYPE Description pointer - COMPONENT #

S132 COMPONENT marked `InUse' found in Free Chain- COMPONENT #

S135 COMPONENT in Free Chain referenced by a TYPE - COMPONENT #

S136 COMPONENT recovered and added to Free Chain - COMPONENT #

S137 TYPE in Free Chain referenced by another TYPE - TYPE #

S138 TYPE recovered and added to Free Chain - TYPE X'%04X'

S139 TYPE marked `InUse' but not referenced - TYPE #

S140 Referenced TYPE not marked `InUse' - TYPE #

S141 TYPE Free Chain not in order: sorting

S142 COMPONENT Free Chain not in order: sorting

S143 Overlap found in LIST Free Chain

S144 Free Chain extends into unallocated region

S145 Overlap found in COMPONENT LIST SPACE: list cleared COMPONENTS will be deleted

S146 Invalid COMPONENT LIST pointer: all COMPONENTS DELETED - TYPE: `*type name'*

S147 Resetting COUNT in COMPONENT LIST: some COMPONENTS may be lost

S148 RULE truncated (due to internal error): `*type name'* (at index #)

S150 CATEGORY `*type name'* was missing GROUP and/or ITEM attributes

S151 GROUP `*type name'* was missing GROUP attributes

S152 ITEM `*type name'* was missing ITEM attributes

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

© Copyright IBM Corp. 2006

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
577 Airport Blvd., Suite 800
Burlingame, CA 94010
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Programming interface information

Programming interface information, if provided, is intended to help you create application software using this program.

General-use programming interfaces allow you to write application software that obtain the services of this program's tools.

However, this information may also contain diagnosis, modification, and tuning information. Diagnosis, modification and tuning information is provided to help you debug your application software.

Warning: Do not use this diagnosis, modification, and tuning information as a programming interface because it is subject to change.

Trademarks and service marks

The following terms are trademarks or registered trademarks of International Business Machines Corporation in the United States or other countries, or both:

AIX
AIX 5L
AS/400
Ascential
Ascential DataStage
Ascential Enterprise Integration Suite
Ascential QualityStage
Ascential RTI
Ascential Software
Ascential
CICS
DataStage
DB2
DB2 Universal Database
developerWorks
Footprint
Hiperspace
IBM
the IBM logo
ibm.com
IMS
Informix
Lotus
Lotus Notes
MQSeries
MVS
OS/390
OS/400
Passport Advantage
Redbooks
RISC System/6000
Roma
S/390
System z
Trading Partner
Tivoli

WebSphere
z/Architecture
z/OS
zSeries

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Intel, Intel logo, Intel Inside, Intel Inside logo, Intel Centrino, Intel Centrino logo, Celeron, Intel Xeon, Intel SpeedStep, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

This product includes software developed by the Eclipse Project (http://www.eclipse.org/).

IBM WebSphere Transformation Extender, Version 8.1

Index

Special characters

.bak file name extension 3
.dbe file name extension 3
.mtt file name extension 3

A

analysis
  logic 134
  structural 134
analyzer
  type trees 115, 133
ANY
  reserved word 87
Any-2-XML utility 139, 140
arrange icons
  window 13
As Work Book option 6, 10
attribute
  restart 112
attributes
  identifier 92
  restart 92, 93
  sized 92, 93

B

big endian 41
binary values 40
  BCD 40
  float 40
  integer 40
  packed 40
bound types 115
byte order property
  big endian 41
  big native 41
  little endian 41

C

cascade
  window 13
category
  definition of 19
  inheritance 101
  organizing 101
  when not to use 102
  window 7
CDATA 37
character values 40
  decimal 41
  integer 41
  zoned 42
character zoned numbers
  places 42
  sign 42
  size (content) 42
choice group
  objects 131
class
  definition of 27
commands
  copy 104
  find 9, 106
  menu 8
  merge 105
  move 104
  replace 107
  undo 9
comments
  component rules 91
common item properties
  national language 52
  none 57
  pad 48
  special value 57
components 7, 81
  attributes
    identifier 92
    restart 92
    sized 92
  defining 86
  definition of 19, 83
  distinguishable 120
    implicit group 79
  guidelines 86
  inheritance 101
  manipulating 92
  name 83
  number 85
  optional 87
  order of 81
  range 81, 82
    fixed 85
    variable 85
    viewing 86
  required 87
  restart attribute 112
  rules 88, 89, 90
    comments 91
    examples 89
    formatting 91
    inserting functions 91
    object names 90
    shorthand notation 90
    syntax 89, 92
  searching for 92
  variable 87
configuring 13
CONTAINSERRORS function 113
content distinguishable
  components 121
copy
  command 104
  edit 9
cut
  edit 9

© Copyright IBM Corp. 2006

D

data
- composition 1
- defining 18
- object
  - definition of 18
- objects
  - subtypes 2

date & time subclass properties 58
- custom date format 60
- custom time format 61
- date 58
- examples 63
- format 59
- presentation 58
- special 64
- time 59

date format
- custom 60

date property 58

defining
- type properties 25

delimited syntax
- explicit group 76
- implicit group 78

delimiter 80
- definition of 76
- literal 79
- location of 80
- variable 80

description
- types 27

distinguishable data
- explicit group 130
- implicit group 129

distinguishable objects 115
- components 121
- implicit group 79
- of same component 120
- types 121

document type 32

document verification 32

drag and drop
- types 83

E

edit menu 9

element type declarations 37

Empty type property 31

errors
- detection 109
- messages 134
- recovery 112
- syntax 92

examples
- using AcceptAllPads type property 50
- using Empty type property 31

expanding
- types 18

explicit format 75

explicit group
- distinguishable data 130

export
- type trees 19

extensions See file name extensions 2

F

file name extensions 2, 3

find
- command 106
- edit 9

fixed syntax 75

floating component 76

flow chart
- properties 39

format
- date & time 59
  - custom 60, 61
- group
  - explicit 75, 76
  - implicit 76, 77, 78

G

group
- component 7
  - name 83
- components 81, 83
- definition of 19
- formats
  - explicit 75, 76
  - implicit 76, 77, 78
- window 7, 82

group subclass 73
- choice 73
- sequence 73
- unordered 73

H

Help
- menu 13

hex values 72, 91

hierarchy
- of a type tree 2

I

icons 3

identifier attribute 92, 99

implicit group
- distinguishable data 129

Importer Wizard 5

inheritance
- category 101
- components 101
- item properties 101
- restrictions 101

initiator 29

inserting functions 91

interpret as 39

ISERROR function 113

item
- definition of 18
- restrictions 69, 70

item properties 39
- date & time subclass properties 58
- flow chart 39
- interpret as 39
- item subclass 39
- number item subclass 39

item properties *(continued)*
    syntax objects 65
    text item subclass properties 58
item subclass 39

L

language options 32
length property
    bytes 40
literal
    delimiter value 79
little endian 41
location property 80
logic errors
    type tree analysis 141
logic warnings
    type tree analysis 151
logical analysis 134

M

menu
    commands 8
    edit 9
    help 13
    tools 8, 12
    tree 11
    view 10
    window 12
merge command 105
messages
    errors 134
    warnings 134
move command 104

N

name
    relative 83
    type 83
namespaces 36
national language 32, 52
native 41
no syntax group 78
non-printable values 72
number item subclass property 39
    byte order 41
    length (bytes) 40
    places 57
    separators 43
    sign 47

O

objects
    copying 103
    distinguishable 115, 120
    moving 103
    names
        component rules 90
    reordering 104
    selection 103
optional component 87
options dialog box 12
options dialog box *(continued)*
    analysis results 15
    confirmations 15
    general 13
    group window 14
    item window 15
    type properties 15
    type tree 14

P

pad characters 49
pad property 48
partitioning 95
    benefits of 96
    convenience 95
    groups 98
        component rules 99
    items 96, 97, 98
    required 95
    types 29, 96, 131
paste
    edit 9
PCDATA 37
places property 57
presentation property 58
printing
    type definitions 108
    type properties 108
propagate
    types 25
properties
    flow chart 39
    inheritance of 101
    national language 32
    where used 33
    window 7
        dock 7
        float 7

R

range
    default 82
    fixed 85
    variable 85
    viewing 86
recently used window list
    window 13
REJECT function 112
relative type name 83
release characters
    definition of 30
    guidelines 30
rename types 106
replace
    command 107
    edit 9
required component 87
reserved words and symbols
    ANY 87
restart attribute 93, 112
restrictions
    defining 69
    inheritance of 101
    settings 70

rules
- component 88, 89
  - examples 89
  - syntax 89

S

separators 43
- decimal 45
- integer 43

set range command 85
shortcuts
- tools 12

shorthand notation
- component rules 90

sign property 42, 47
size
- command
  - include self 93
- digits 42

sized attribute 93
special property 64
startup window 5
status bar
- view 10

structural analysis 134
structure errors
- type tree analysis 152

structure warnings
- type tree analysis 152

subtree 27, 86, 98, 102, 104, 105
subtypes 2
- order 29
- reordering 105

supertype
- definition of 2

symbols
- dialog box 72
- inserting 72

syntax
- errors 92
- initiator 29
- none 78
- objects 65
  - data 66
  - variable values 65
- of a component rule 89
- release character 30
- terminator 30

T

terminator 30
text item subclass properties 58
- interpret as 40

time format
- custom 61

time property 59
time zones 62
toolbars
- view 10

tools menu 12
trace option 49
track property 75
Type Designer
- configuring 13

type tree analysis
- logic errors 141
- logic warnings 151
- structure errors 152
- structure warnings 152
- Type Tree Analyzer 141

type trees
- analyzer 115, 133
- As Work Book option 6
- creating 17
- differences 20
- exporting 19
- file name extension 3
- hierarchy 2
- icons 3
- importing 5
- opening 23
- views 6
- window 6

types
- bound 115
- category 19
- class 18, 27
- copying 103
- creating 17
- definition of 18
- description 27
- distinguishable 121
- drag and drop 83
- expanding 18
- groups 19, 81
  - components 19, 83
- icons 3
- inheritance 101
- item 18
- moving 103
- name 26, 83
- organizing 101
- partitioned 29
- propagate 25
- properties
  - defining 25
  - window 7
- relative name 83
- renaming 106
- reordering 104
- subtypes 2
- supertypes 2, 17

U

undo command 9

V

Value Not In Range
- range restriction 71

variable component name 87
variable delimiter 80
view menu 10

W

warning messages 134
where used 33

whitespace syntax
  build as 77
  character set 78
wildcards
  element and attribute 37
window
  category 7
  close all 13
  group 7, 82
  menu 12
  properties 7
  type tree 6
  view
    cascade 13
    tile horizontally 13
    tile vertically 13
wizard
  importing 5
work book option 6

X

XML 139, 140
  and exporting a type tree 19
  data and floating components 77
  excluding delimiters 71
  identity constraints 36
  properties in the type tree 33
  supported constructs 33
  time zone formats 63
  Type Tree Compatibility Utility 138
  Type Tree/Schema Synchronization Utility 137
  using Empty property 31
  wildcards 37
XML DTD Importer 37
XML Schema
  datatypes and type tree constructs 34
XML Schema Importer 36, 37
XML type property See document type 32

IBM®

Printed in USA