

Steen Andreasen

MORPHX IT

**An introduction to Axapta X++
and the MorphX Development Suite**

MORPHX IT

An introduction to Axapta X++ and the MorphX Development Suite

Copyright © 2006 Steen Andreassen, www.steenandreasen.com

Editor: Steen Andreassen

Layout: Steen Andreassen

Cover: Poul Cappelen and Ulla Bjulver

Photographer: Ulla Bjulver

Denmark 2006

ISBN: 87-991161-1-1

1. Edition

All rights reserved. The author has created reusable code in this publication expressly for reuse by readers. You are granted limited permission to reuse the code in this publication so long as the author is attributed in any application containing the reusable code and the code itself is never distributed, posted online, sold or commercial exploited as a stand-alone product. Aside from this specific exception concerning reusable code, no part of this publication may be used or reproduced in any manner whatsoever without the prior written permission of the copyright holder except in the case of brief quotations embodied in articles or reviews. Any other use without written consent is prohibited according to the Danish copyright law.

If you encounter any inaccuracies, please report them to the author at the following email address:

axaptabook@steenandreasen.com

Trademarks

All terms mentioned in this book that are known to be trademarks have been appropriately capitalized. steenandreasen.com cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark.

Warning and Disclaimer

You should never try out any of the examples in this book in a live environment. The information in this book is provided *as is*. The author or steenandreasen.com cannot be responsible of any loss or damages arisen from the information containing in this book.

“Thanks to my dear wife Ulla, and my son Oliver, who stood up with me and supported me while writing this book.”

S. A.

Acknowledgements

Thanks to all of you who directly or indirectly have contributed to the content of this book, providing inspiring comments and suggestions.

Special thanks to Lars Holm for his contribution to the Appendix Properties. Poul Cappelen and Ulla Bjulver www.photo-art.dk for cover design. Jens Thrane, Christian Beck, Erik Pedersen, Lars Kjærsgaard, Jim Long, Hanne Paarup, Eric Fisher www.unitederp.com, Craig Brown www.edenbrook.co.uk, Daryl Spires www.avionsystems.co.uk, who read, proofread, edited the manuscript, and most of all, encouraged me to persevere.

Acclaim for the book

"Steen Andreasen is an excellent Axapta Programmer and Technical Manager - and he is a patient teacher. In this book Steen Andreasen has worked extremely hard at taking you, the reader, for an enchanting trip into the world of Axapta development.

I would highly recommend this book as a *must have* for any developer whether experienced or novice who wants to make a career in Axapta Programming.

Thank you very much Steen Andreasen for all your efforts and generosity of offering such well-structured valuable information open to the public."

Warm regards,

Harish Mohanbabu
Microsoft Dynamics Ax - MVP
<http://www.harishm.com/>

Contents

PREFACE	15
INTRODUCTION	17
Why Is This Book Important	18
Structure of the Book	18
1 INTRO TO MORPHX	19
1.1 AOT	19
Layers	20
Properties	23
Add-ins	24
Editor	25
Debugger	27
Compiler Window	30
Import and Export	31
Compare Objects	33
Code Upgrade	34
Search	35
Infolog	36
Recycle Bin	37
User Settings	38
1.2 Project	41
Modifying a Project	41
Project Types	42
1.3 Summary	42
2 INTRO TO X++	43
2.1 Variables	43
2.2 Operators	47
Assignment operators	47
Relational operators	48
Bitwise operators	49
2.3 Control Flow Statements	50
Loops	50
Conditional Statements	52
Exceptions	55
Miscellaneous	56
2.4 Select Statements	57
2.5 Functions	64

2.6	Summary	64
3	DATA DICTIONARY	65
3.1	Tables	65
	Company	65
	Application tables.....	67
	System tables	71
	Fields	73
	Field Groups	75
	Indexes	76
	Relations.....	77
	Delete Actions.....	79
	Methods	80
3.2	Maps.....	86
3.3	Views.....	88
3.4	Extended Data Types	89
	Extended data type array	91
3.5	Base Enums	93
3.6	Feature Keys	94
3.7	Licenses Codes	94
3.8	Configuration Keys.....	95
3.9	Security Keys	96
3.10	Table Collections	97
3.11	Special Table Use	98
	Using System Classes.....	98
	External databases	100
3.12	Summary	101
4	MACROS	103
4.1	Macro commands	103
4.2	Defining constants	105
4.3	Creating macros	106
4.4	Summary	107
5	CLASSES	109

5.1	Classes Basics.....	109
	Methods	109
	Class Components	111
	Modifiers	114
	Passing Values	120
5.2	AOS	124
	Setting Tier	124
	Objects to Optimize	125
5.3	Runbase Framework	125
	Using Runbase Framework	126
	Dialog.....	130
5.4	Fundamental Classes.....	134
	ClassFactory	135
	Global	135
	Info	136
5.5	System classes	136
	Object	136
	Runtime changes.....	136
	Args.....	137
	Foundation Classes	138
	Optimized Record Operations	138
	File Handling.....	139
5.6	Special Use of Classes	139
	Using COM	140
	X++ Compiler	141
5.7	Summary	142
6	FORMS.....	143
6.1	Creating Forms	143
6.2	Form Query	146
	Joining Data Sources.....	146
	Setting Access	149
6.3	Design	151
	Creating Design	151
	Controls in Design	153
	Display and Edit Modifiers	159
6.4	Methods on a Form.....	162
	Form Methods.....	163
	Form Data Source Method	167
	Form Data Source Fields Methods.....	171
	Form Controls Methods	173
	Common Form Methods.....	173
	Overriding a Form Query.....	176
	Modifying Data Sources from X++.....	178

	Building Lookups	180
	Form Dialog	182
6.5	Special Forms	184
	Calling User Defined Method.....	184
	Overload Methods	185
	General Form Changes	188
	Colors	189
6.6	Summary	191
7	REPORTS.....	193
7.1	Report Wizard	193
7.2	Creating Reports.....	193
7.3	Report Query.....	196
7.4	Templates	199
	Report template	199
	Section template	202
7.5	Designs	202
	Creating design.....	203
	Auto design.....	205
	Generated design	209
	Controls in design	209
7.6	Methods on a Report.....	211
	Report Runbase Framework	214
	Dynamic Reports	219
	Common Report Methods	222
7.7	Special Reports.....	226
	Execute report from X++	226
	Using temporary tables.....	227
	Coloring rows.....	230
	Print using Microsoft Word	232
7.8	Summary	235
8	QUERIES.....	237
8.1	Building Queries	238
	AOT Query.....	238
	X++ Query	244
8.2	Queries in Forms and Reports	246
8.3	Summary	246
9	JOBS.....	247

9.1	Creating jobs	247
9.2	Summary	248
10	MENU ITEMS AND MENUS.....	249
10.1	Menu Items	249
10.2	Menus	250
	Locate AOT object from menu	251
10.3	Summary	252
11	RESOURCES	253
11.1	Using Resources	253
11.2	Summary	256
12	APPENDIX PROPERTIES.....	257
12.1	Data Dictionary Properties	257
	Tables, Table Maps and Table Views	257
	Table Field, Map Field	258
	View Fields	259
	Table Field Group, Map field group, View field group	260
	Table index	260
	Table Relation.....	261
	Table Relation Field.....	261
	Table DeleteAction	261
	Map Mapping	261
	Map Field Mapping	261
	Extended Data Type	262
	Base Enum	264
	Base Enum Entry.....	264
	License Codes	265
	Configuration Key, Security Key	265
12.2	Form properties	266
	Form data source.....	266
	Form Data Source Fields.....	267
	Form Design Group Controls.....	267
	Form design	271
	Type controls	273
12.3	Report Properties	288
	Report	288
	Report design	288
	Auto design.....	289
	Sections controls	290
	Section Template.....	292
	Section Group.....	292
	Type controls	293

	Field Group	301
12.4	Query properties	301
	Query	301
	Data sources	302
	Fields	302
	Sorting fields	303
	Ranges	303
12.5	Menus Properties.....	303
12.6	Menu Items Properties	304
13	APPENDIX MORPHX DEVELOPMENT TOOLS	307
13.1	Cross-reference	307
13.2	Application Objects	308
	Application objects forms.....	308
	Application management	308
	Usage data	309
	Count of application objects	309
	Locked application objects	309
	Refresh tools.....	309
	Re-index	309
13.3	System Monitoring	310
	Database tracing.....	310
	AOS tracing	310
13.4	Code Profiler	310
13.5	Application Hierarchy Tree	312
13.6	Visual MorphXplorer.....	312
13.7	Code Explorer	314
13.8	Table Definitions.....	314
13.9	Number of Records	314
13.10	Help Texts	315
13.11	Version Update	315
	Renamed application objects	315
	Create upgrade project.....	316
	Compare layers	316
13.12	Wizards	316
	Report Wizard	316
	Wizard Wizard	316
	Label File Wizard	317
	Class Wizard.....	317

COM Class Wrapper Wizard 317

13.13 Label 317

Find label 318

Label log 319

Label file wizard..... 319

Label intervals..... 319

14 APPENDIX REPORT WIZARD 321

Preface

To successfully program is primarily a matter of understanding the needs of the user and translate this need into a technically functioning solution, a system. It is furthermore essential that the programmer clearly understands how this system is adapted so adjustments are user-friendly and easy to upgrade.

This book gives an introduction to the development phase of Axapta. The book is not only an exercise in the functionality of Axapta but because it is based on my more than eight years of experience with Axapta it is as much a practical accessible book giving lots of coded examples of product development as well as developments of client solutions.

The publishing of this book has been a longstanding desire of mine. I have learned through my many years of working with Axapta that a practical and instructional programming book in this field is not available. This book, MORPHX IT, fully illustrates my professional interest in ERP systems and Axapta in particular.

My journey through the authoring process has been exiting and it has been a great inspiration for me to receive mail from hundreds of people from all over the world. People, who wrote to me with comments and suggestions after I released a chapter of the book for download. This has shown to me a strong interest and need for the book.

Hopefully the book will provide inspiration to the novices of this subject, to the participants of advanced further education, where the teaching of ERP systems is breaking through, as well as to more experienced Axapta professionals in the business environment. Many programmers have had to tediously collect the information individually, which I have published in this book, and my goal is to make the process of working in and with Axapta easier and inspire to a continued development in the profession.

Steen Andreasen

Introduction

This book is an introduction to the development environment in Axapta, also just called MorphX.

MORPHX IT is written as a practical book. By practical I mean that you should use the book while working with Axapta. This also makes the book valuable on a daily basis as the book contains a lot of examples. I have used this approach as I believe that the easiest and quickest way to learn a new development language is to start using the system right away.

You should have an Axapta application installed and have a basic knowledge of how the user interface of Axapta looks. This information can be found in the manuals in the standard package.

Focus in the book is from a developer's point of view. You will be able to use the book without having any knowledge about Axapta. However it will be easier for you to understand the contents, if you have tried using the application.

You will get most benefit from the contents by trying out the examples while reading the book. Examples used in the book are included in the zip file MORPHXIT_1ED_EXAMPLES.ZIP which came with the book. Axapta 3.0 Service Pack 4 has been used while writing the book. If running another Service Pack of Axapta 3.0 you might encounter slightly differences.

The book is intended to be read by people without any prior knowledge of programming in Axapta. You must not necessarily have a background as a programmer. The book can also be read by technical skilled persons or Axapta application consultants who want to use the development environment.

If you are a beginner I recommend you to read the chapters from the beginning of the book as more details are added throughout the chapters. There might be terms you do not understand when reading. Not all terms introduced in a chapter are explained right away. I have taken this approach on purpose to simplify the contents. Introducing all terms at once would have made the book too theoretical to be used on a daily basis. Often there will be references to other sections in the same chapter or to other chapters, where you can get more information on a certain term. As a more experienced Axapta user, you can with advantage read single chapters of the book to catch up on a specific area.

Why Is This Book Important

During my years of working with Axapta I have been aware of the lack of documentation on the Axapta development environment. See this book as the notes you wished to have when starting out programming Axapta.

MORPHX IT is the first Axapta programming book, and should be considered an alternative to attending courses to learn programming in Axapta.

With this book you will have a tool to get to know the development environment in Axapta quickly. A guide which will teach you to customize your application in such a way that your code will be easier to maintain and more user-friendly for the application users.

Structure of the Book

While writing the book I have had to choose what I found most important for an introductory book. The web framework is not part of this book. For several reasons the web framework has been left out. Focus has been to introduce what a new comer to programming in Axapta needs. Often it will be people experienced with programming Axapta who start using the web framework. Also for most customer cases you will not be using the web parts.

When explaining a topic in the book such as a tool used in the development environment not all fields, buttons or features of the tool are necessary described. Topics are often explained as you would have them told by a person sitting right next to you. This goes especially for the **Appendix MorphX Tools**.

When reading the book you will see a lot of best practice hints and recommendations. I do not distinguish between these two terms. Together these are my own set of rules when programming or managing a development project in Axapta.

The chapters in the book follow the main tree nodes in the Axapta development environment, starting with intro chapters to the development environment and the development language, followed by chapters on how to use the development environment.

In the back of the book you will find the appendix chapters which contain further information on some of the topics in the book.

1 Intro to MorphX

The development environment in Axapta is called The MorphX Development Suite or just MorphX. MorphX is an integrated development environment which consists of the Application Object Tree, the X++ programming language and several tools which provide an overview of the application.

MorphX was used to build Axapta's application modules. Assuming you have licensed Axapta's source code from Microsoft, you can edit any of the objects used in the standard package. This environment, which includes the same tools used by Microsoft to develop the software, allows you to extend the existing functionality to fit your organization's requirements. As a result, Axapta is far easier to customize than other ERP packages. It accelerates the development process as you can draw upon existing functionality, rather than having to start out creating your own code from scratch. Both Microsoft SQL Server and Oracle databases are supported, though once you have installed the database and application, you will not have to worry about the type of database. From a developer's perspective, the Axapta kernel masks all database specific issues.

Developing a modern user interface can be a time consuming process. In MorphX, the user interface, including both forms and reports, are by default almost entirely generated by the system. This means that you will not have to spend time positioning and arranging controls in designs. All this will be explained further in the chapters covering **Forms and Reports**.

1.1 AOT

The *Application Object Tree* (AOT) is the development menu in Axapta. All objects used in the application are stored in the AOT and presented to the developer in a tree organized by object type. Axapta uses a layer technology to store objects. This layers help differentiate the objects in Axapta as a typical Axapta installation is built up using not just the standard objects from Microsoft, but also objects provided by Solution Providers, partners and as those developed by customers themselves.

When expanding a node in the AOT, the sub nodes at the next level will be cached. This may take a few seconds the

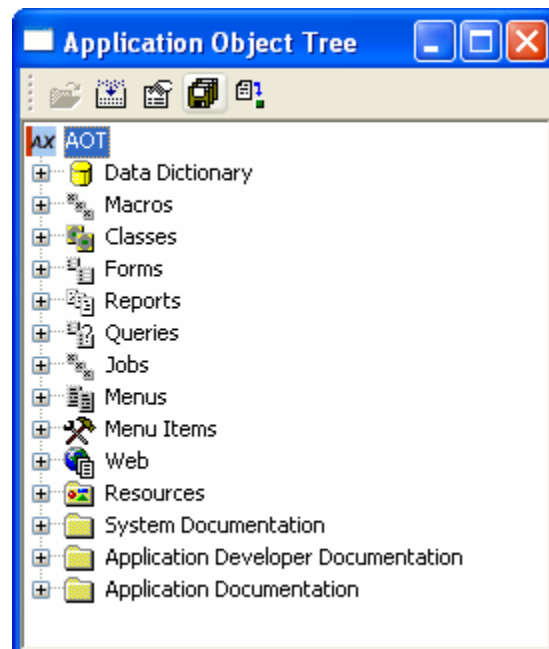




Figure 1: Application Object Tree

first time a node is expanded. You only cache the nodes used. When a cached node is later expanded you will find that it responds far more quickly.

The AOT is accessed by pressing ctrl+d or clicking the icon  in the top menu. Notice that if you are running a demo installation of Axapta, you will not have access to the AOT. You must have the license codes for MorphX and X++ installed.

To create a new object, right-click a node in the AOT and choose new. A red vertical bar next to the object node indicates that the node has not been saved. The red mark is also set when a node is modified. All nodes modified or recently created can be saved by clicking the save all icon  in top of the AOT window.

Objects, fields and methods may be duplicated within the AOT. Methods and fields may also be copied. The copy and duplicate functions are accessed by right-clicking on the node and selecting the desired function. When you duplicate a node, the system will create a copy of the selected node in the AOT prefixed with *copyOf*. This is especially useful when you want to test a possible modification and would like to preserve the prior state of the object as a fallback should the change not function as desired.

The AOT also supports drag and drop. When adding controls to forms and reports, it is often much faster to drag the fields used for the controls to the form or the report directly from a data source, rather than create them from scratch. MorphX will handle creating the control with the appropriate properties.

Regardless of which node you select in the AOT, you will always have access to a context menu by right-clicking the current node. The menu is named *SysContextMenu* in the AOT and can be found under the Menu node. When browsing the menu from the AOT all menu items available in the Context menu will be listed. Depending on the selected node in the AOT you will have additional menu items available to perform context specific activities, like creating a new object, opening a new window or accessing the tools sub menu Add-Ins. Opening a new window provides a root node based on the current selected node in the AOT. You can open as many windows as needed. This feature can be used while dragging new objects to a form or a report, or simply to have cursors positioned in different places in the AOT.

Layers

The layer technology in Axapta is used to organize the objects of the standard package and customizations made. There are eight standard layers. Each of these standard layers has a corresponding patch layer for a total of 16 layers. The lower four layers are used for the standard package and are not accessible by partners or customers. Partners and customers are each allocated two layers, along with the corresponding patch layers. When you sign on, you specify the current layer in the clients Axapta Configuration. All layers except the top layer require an access code. When a customer licenses the software, they receive access codes for the top two layers and are therefore excluded

from directly modifying contents of the lower six layers. This prevents a customer from irrevocably changing the core code provided by Microsoft or their business partner. This does not prevent you from changing this code. When editing a standard object, Axapta copies part or the entire object from one of these lower layers into the current layer. Since an object may be modified at more than one layer, source of the copy is the highest level, below the current layer, in which the element you are changing exists. Your changes will be saved in the current layer. The modifications made to the top layer will always override the lower layer. If you want to start all over, just delete the object in the current layer and you will be back where you started. The current layer is shown in the status line in the bottom of the Axapta window. For an overview of the layers, see **figure 2: Layers overview**.

Layer	Access rights	Description
SYS	Read	This is the lowest layer. Here all objects created by Microsoft are stored.
SYP	Read	Patch layer for the SYS layer. Is used for Service Packs.
GLS	Read	Solution Providers are using this layer. Used for global certified modules created by sub suppliers and licensed by Microsoft, such as the CRM and HRM modules.
GLP	Read	Patch layer for the GLS layer. Is used for Service Packs.
DIS	Read	The layer is used for country localizations. Several different layers exist as the layer is grouped for countries with similar local demands. If one application is used in countries with different DIS layers, the layers must be manually merged.
DIP	Read	The patch layer for DIS. Used for Service Packs.
LOS	Read	This layer is used for local solutions. These are modules which are not certified globally as in the GLS layer. For example it is used in Denmark for a Payroll module.
LOP	Read	Patch layer for LOS. Used for Service Packs
BUS	All	The lowest layer the partners have access to. Partners can use this layer for their own modules. The layer requires a license code.
BUP	All	Patch layer for BUS. Partners can use this layer for updates.
VAR	All	Partners use this layer for their customization which is customer specific. The layer requires a license code.
VAP	All	Patch layer for the VAR layer. Can be used for updates.
CUS	All	This layer is meant to be used by customization made by the customer itself.
CUP	All	Patch layer for CUS. Can be used for updates.
USR	All	This is the highest layer. The layer is used if a user creates their own customizations like creating a report using the report wizard. Often this layer is used for test purposes.
USP	All	Patch layer for the USR layer. Meant for customer's own updates.

Figure 2: Layers overview

When logging on to an Axapta client, the current layer is set. You cannot change a layer without restarting the client. All modifications made will be saved in the current layer. Depending on the object modified, different parts of the object will be saved in the modified layer.

Modifications to objects as forms and reports, require that the complete object be created in the current layer. If a class or a table is modified, only the modified method or field will be created in the current layer.

Each of the layers is stored in physical file with the naming AX<layer>.AOD. The filename for the sys layer is AXSYS.AOD. The layers are indexed in the file AXAPD.AOI. If deleted, the index file is built automatically at startup. If a layer file is deleted or added, the index file will also be rebuilt. You might end up in a situation where you cannot locate an object in the AOT, or other strange system behavior such as the AOT crashing when a certain object is accessed. Then it might help to delete the index file and have it rebuild at startup. To rebuild the index file all users must be logged out, shut down the Application Object Server (AOS). Start a single two-tier client to rebuild the index.

Note: When modifying an object, modifications made in higher layers for the same object will automatically be updated. If a form is modified in the VAR layer, modifications made to the object in the USR layer will automatically be updated with the modifications from the VAR layer. When importing objects, the same objects will not be modified in higher layers.

If you have set the development option to show all layers at **Tools | Options**, all layers for an object will be shown in parentheses after the object name in the AOT. This gives a quick overview of which layers contain modifications for a given object. By right-clicking on a node presented in more than one layer and choosing *Layers* the node will be split up and each layer for the object will be shown indented as a separate node in the AOT. You can then browse the changes made in the single layer. However, you can still only change the current layer. Click *Layers* again to re-consolidate the layer split.

Properties

The property sheet is accessed by right-clicking an object in the AOT and choosing *Properties* or by pressing alt+enter. You may find it easier to keep the property windows open, as the property sheet is updated each time a new object is selected in the AOT. By default the property sheet is docked in the right side. If you prefer to position the property sheet yourself, you can right-click the window and choose *No Docking*. The properties can be presented in two ways. The first tab page lists all properties and the second page groups the properties. If you are running a high resolution, you can, in most cases, view all properties for an object without scrolling. To have the properties sorted alphabetically, go to **Tools | Options**. For an overview of the properties in the AOT, see the chapter **Appendix Properties**.

Note: The property sheet can be used to count numbers of object. All nodes must be cached in the AOT to be counted. Try marking all objects prefixed with Sales and press alt+enter. This will cache the selected nodes. If you now mark some of the cached nodes, the number of nodes marked will be shown in parentheses in top of the property sheet. This will only work if all objects selected are cached.

Each object properties have a set of default values. The default values are generally those which maximize MorphX's ability to automate the development process and maximize program flexibility. This means that objects like forms and reports have a default set of values for the properties making controls auto positioned and auto adjusted. When the default value of a property is changed, the property is set to bold making it easy to spot the changes made to the property sheet. If you are going to change a property on several objects, you just select all nodes to be changed in the AOT. You can multi-change any type of object, and only the properties in common for the selected objects will show up in the property sheet.

When choosing a value for a property, three different types of lookup icons are used in the property sheet. The arrow down icon is used to select between defined values like positioning or adjusting controls. The square icon is used to switch between the defined values and allow the entry of a fixed value.



Figure 3: Lookup icons, arrow down and square

The lookup icon with dots is used where a new form is opened as in entering labels or choosing a font.

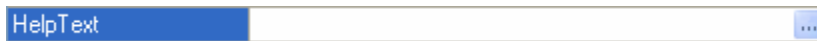


Figure 4: Lookup icon, dots

Two types of properties are colored. The AOT name for the object is colored light red, indicating the property is mandatory. Properties where a label can be set are colored yellow when no label has been entered. This does not mean that a label should be specified, as labels for forms and reports have been specified on the table or the extended data type. If a label is chosen from the label system, the yellow color is changed to white.

Add-ins

The sub menu Add-Ins in the Context menu has tools related to the current node. A standard set of tools like the Cross-reference and Check Best Practices can be called from here. Add-ins is the only menu item customizable in the Context menu. The standard items in the Add-Ins menu are created using MorphX. You can create you own items using MorphX and add them to this sub menu. Most of the Add-Ins menu items can also be called from the top menu **Tools | Development tools**.

Editor

The editor in MorphX is used for Axapta's built in language X++. To open the editor, choose a method in the AOT and double click or press enter. Select a methods node or a top node of an object like a class or a form to open a list of methods for the node.

The AOT path to the current node edited are shown in the title bar of the editor window. In the top of the editor window, icons for the common tasks like save, compile and setting breakpoints are listed. The left window shows the selected methods. To switch between the methods, click a method in the left window. A * put after the method name in the left window indicates that the method has been changed, and not yet saved. Upon closing the editor, you will be prompted for saving changes. However, if you have set the auto save option for your user in **Tools | Options**, your changes will, with intervals, be saved automatically.

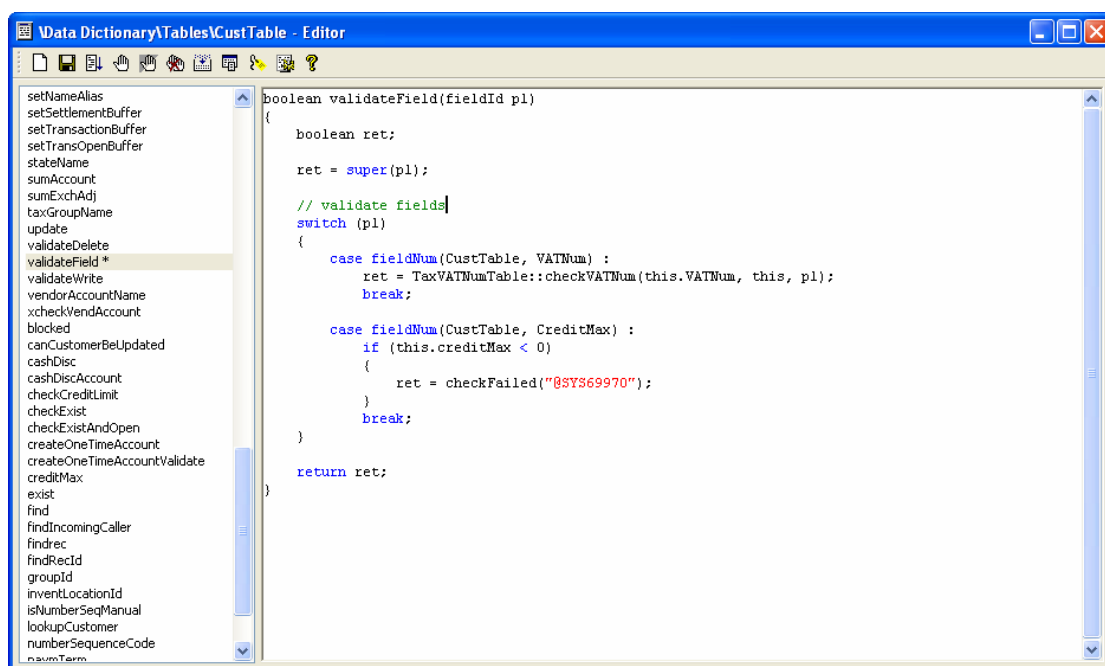


Figure 5: The editor showing the methods for the table CustTable

Code window

In the right window the X++ code for the selected method is edited. Reserved words are highlighted with blue. Comments are green and text strings are colored red. At runtime, the code entered is validated, and if an invalid sentence is entered, the error will be underlined with a red jagged line. Also, the method with the compile error will be underlined in the left window. For an overview of the most important hotkeys in the code window, see **figure 6: Common editor hotkeys**. A full overview of all hotkeys can be found in the Developers Guide located at **Help | Microsoft Axapta Developer's Guide**.

When right-clicking in the code window, you will have a menu from where you can view lists for AOT objects, look up information on your code and call editor scripts. Listing the AOT objects like tables, classes and extended data types eliminates typing as you do not have to scroll through the AOT to find the name of your objects. You just pick the object name from the list. An alternative to using the lists can be opening a second AOT window and dragging the object name from the second AOT window to the code window. You can in fact, drag any node in the AOT to the code window. The name of the node will be inserted in the code window. However some nodes will add lines of code when dragged. Try dragging a query from the AOT to the code window. You will then have all code written to run your query. The only thing needed is to declare the variables.

The lookup menu items are used to jump to the code of a method, showing the parameter profile for a method, or looking up a label in the label system. If your method has a compilation error you will not be able to use the lookup menu item beyond the error.

Editor scripts are a collection of scripts made using X++. These are used to carry out common tasks like adding code comments or formatting the code in a special sense. You can add your own scripts or modify an existing one. The class *EditorScripts* has a method for each of the existing scripts.

Function	Hotkey	Description
New	ctrl+n	Create a new method.
Save	ctrl+s	Save all methods in the left window.
Toggle breakpoint	F9	Set breakpoint.
Enable/disable breakpoint	ctrl+F9	Used to skip breakpoint without removing it.
Remove all breakpoints	ctrl+shift+F9	Remove all breakpoints set by the user.
List breakpoints	Shift+F9	List all breakpoints.
Compile	F7	Compile all methods in the left window.
Lookup Properties/Methods	ctrl+space	Show a yellow tool tip. Depending on the code clicked, the following will be shown: the base type of an extended type, parameter profile for a method or the label text for a label id.
Lookup Label/Text	ctrl+alt+space	If a label id is marked, the corresponding label id and label text is shown in the label system.
Lookup Definitions	ctrl+shift+space	Will open the selected method in a new editor window. No need to mark the method name. If the method is not overridden, no window will be opened.
Script	Alt+m	Opens the script menu.
List Tables	F2	List all tables.
List Classes	F12	List all classes.
List Types	F4	List all extended data types.
List Enums	F11	List all base enums.
List Reserved Words	shift+F2	List all reserved words.
List Built-in Functions	shift+F4	List all functions.

Figure 6: Common editor hotkeys

Debugger

When a breakpoint is set in a code line, the debugger will be loaded if the code line is entered. If debugging in a three-tier environment, you must assure that debugging is

activated on the AOS, otherwise you will only be able to debug code running at the client side. While debugging, your Axapta client will be locked. If you need access to the client while debugging, you can start another Axapta client. However, the debugger will be bound to the client from where the debugger was activated. Breakpoints are stored by user, so you will not have to worry about other users while debugging. To get a list of the current breakpoints set, you can press Shift+F9 anywhere in the AOT.

Note: If you get an error in the Infolog and want to trace the error using the debugger, you can set a breakpoint just before the error is inserted in the Infolog. Go to the class method `Info.add()` and set a breakpoint in the first code line.

The source code being debugged is shown in the top window. The code is presented as in the code editor. Breakpoint can be set both in the code editor and in the debugger. The hotkeys for setting breakpoints are listed in **figure 6: Common editor hotkeys**. Breakpoints are shown as a solid red line in the editor. Within the debugger, breakpoints are indicated by a red circle in the left margin. A yellow arrow in the left margin indicates the cursor. A toolbar for navigation is placed in the top of the debugger window. You have 4 different windows which can be enabled for monitoring purposes in the debugger. The windows can be positioned and resized as required.

Note: Do not leave your Axapta client while debugging. When debugging you will transaction lock tables in scope of the code debugged. This will not make you popular if an application user is trying to post data to the locked tables, as the application user will get a database error.

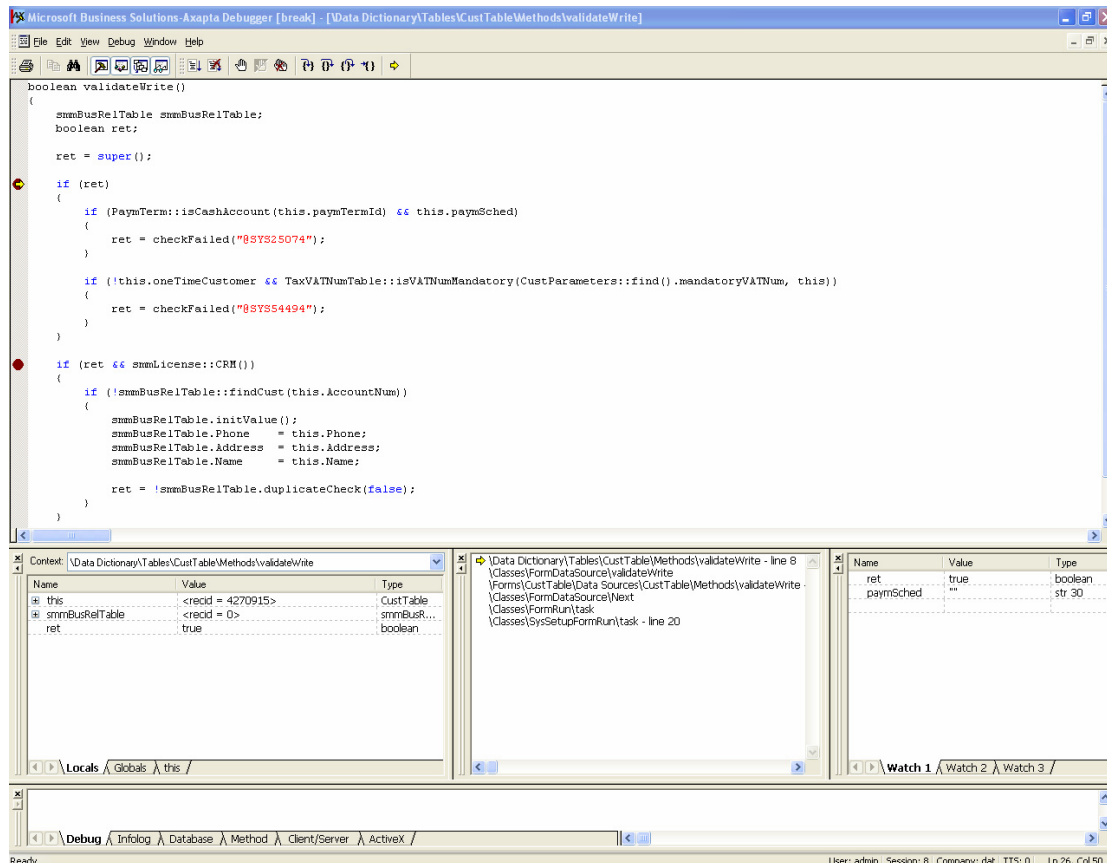


Figure 7: The debugger

Output window

The output window is by default located in the bottom of the debugger. Here you can monitor the Infolog and print commands sent from the code. Use this window if you have added lines to the Infolog or are printing comments from code for debugging purposes.

Variable window

The window is located as the left most window under the debugger window. The window lists the variables in scope of the code window. All types of variables like table fields, types and class are listed. Variables changed between two breakpoints will be highlighted. A neat feature is that you can change the value of any variable while debugging. The lookup field in the top of the variable window is used to pick a method from the call stack. When selecting a method, the variable window and the debugger window will be updated with the selected stack method.

Call Stack window

The call stack window is normally positioned in the middle below the debugger window. The window gives an overview of the methods processed. You can click any method in the call stack to jump to the method. This will update the debugger window and the variable window. Notice that you cannot jump to a non-overridden system method. A yellow arrow in the left margin indicates the current method being debugged, and if you jump to another method, a green triangle will mark the currently selected method.

Watch window

This window is the rightmost in the row under the debugger window. The watch window is used to manually pick variables which you want to trace during the debug session. The window has three similar tab pages which helps organize the information if you are tracing a lot of variables. To add a variable, mark the variable in the debugger window and drag the variable to the watch window. Use the delete key to remove a variable. As in the variable window, you can change the value of a variable added in the watch window. While a variable is in scope, the value of the variable is shown. When out of scope, an error is shown as the value for the variable. If a variable is changed between two breakpoints, the value will be highlighted. The variables set in the watch windows will be stored after completing the debug session. This speeds up debugging as you will not have to start all over if you are debugging the same code several times.

Compiler Window

When compiling X++ code, whether it is done from the code editor or from a node in the AOT, the compiler window will be triggered. The compiler window is by default docked in the bottom of the Axapta window. This takes up space as all windows are opened within the Axapta Integrated Development Environment (IDE). You can right-click the compiler window, and select *No Docking* if you prefer having the compiler window to act as a standard window. This will free up more of your working area, especially needed when viewing the property sheet.

You can either use the compiler windows or the simpler message window for the compiler result. With version 3.0, the compiler window was introduced. The message window was used in prior versions. Go to the compiler window and click on the *Setup* button and choose the *Compiler* menu item to configure the compiler. The *Output* defines whether to use the compiler window or the watch window. The watch window shows the same information as the compiler window, with a simpler interface. You can click a line in the watch window to lookup the information. If you have chosen the watch window for the output, you can re-select the compiler window in the top menu at **Tools | Options** by selecting the Compiler button. The level of compilation checks can be set in the compiler setup window. The field *Diagnostic level* defines what to include. If set to level 4, best practice checks will be included. Cross-references will be updated if marked. Be aware that updating cross-references slows down compiling. For

information on the cross-reference system see **Appendix MorphX Tools**. A log for the compilation can be made during compilation. However you might be better off just exporting the result of your compilation from the compiler window as you can import the compilation result and then use the features in the compilation window, such as lookup errors and warnings.

The compiler window consists of 4 tab pages. The first tab page shows an overview of the compilation. Depending on the settings, warnings, errors, best practice deviations and tasks will be calculated. It is recommended practice to compile the entire application before shipping modifications. During a complete compilation, you might notice a huge number of errors and warnings. You do not need to worry, as this is normal. A complete compilation consists of three loops and all objects are not recognized until the final loop.

Errors and warnings found during compilation are listed on the second tab page. The list contains errors and warnings found in methods and the property sheets. Errors are marked with a red symbol, and warnings are with a yellow symbol. The compiler window is a standard Axapta form, so the standard facilities for sorting and filtering the output are available. If double clicking an error or warning in the list, the method or property sheet contain the error or warning will be opened. You can then correct the error or warning, save the changes and close the window. As errors and warnings are corrected, they will be removed from the list.

If best practice checks have been enabled, the best practice deviations will be listed at the third tab page. Click the button *Setup* and choose *Best Practices* in the compiler window to select the best practice checks to be made. Notice that best practice checks are considered as a guideline. Use common sense while checking the outputs. Missing labels and the use of base types rather than extended types are easy to spot. With the more complex best practice checks, you should not act on the suggestions unless you are familiar with the result of your changes.

The last tab page is used to keep track of your tasks. If you put the text TODO in capitals in a method, the method will pop up in the task tab page during compilation. This is a pretty neat feature, as it helps remember places in the code which must be checked before shipping the modifications.

Note: In the Add-ins menu you will find a menu item called *Compile forward*. The menu item is available on classes, and will compile all classes inherited from the selected class. This is a quick way to compile before trying out your modifications.


Import and Export

You have two options for moving your modification from one system to another. Either copy the whole layer file, or export a selection of objects to a text file. Which option to use depends upon your case. Copying whole layer files is often used when you are updating an installation at the customer site, as this is the only option if the customer

has not licensed MorphX. To export a node, right-click the node and select *Export*. You can export object nodes such as tables, extended data types or forms, or classes, however methods cannot be exported separately. You can export several objects at once by marking the objects to be exported and right-click one of the marked objects. A dialog pops up when selecting export. Here you enter a name for the export file. The current layer is exported by default. You have the option to export another layer by selecting a layer in the dialog. This is recommended: selecting a layer, even if you are in the layer to be exported, as you will then be certain that you have the correct layer exported. Labels used in your code can be exported. Note that importing the exported labels requires knowledge about the label system, so this option is not recommended if the exported file is to be imported by a person without technical skills. However, exporting labels gives the option to have both code and labels packed in one file. This is handy as nothing is more frustrating than importing a file and realizing that the labels are missing.

Objects exported can be locked during the export. The exported objects will be marked with a keyhole icon. This is all very well, but do not expect too much of this, as locked objects can still be modified and everyone can right-click an object and lock or unlock the object. Often it is difficult to determine what has been locked and why.

All object nodes, such as tables, extended data types and classes, have a unique id in the AOT. The id is a sequence number based on the current layer. This id can be included in the exported file. The option is used later while importing to keep the id's in sync.

To import an exported file, click the import icon  in the AOT tool bar. Valid files are identified with the extension XPO. Files are imported into the current layer. To check the current layer, look at the right side of the status line in the bottom of the Axapta window. To get an overview of the imported objects, click on details in the import dialog. A tree view similar to the AOT is shown with the objects in the file. Existing objects in the AOT are marked in bold. By default, all objects in the file will be imported. You can deselect by clicking the check mark in the tree. For objects that already exist in the AOT, you can compare the object in the file with the object of same name in the AOT. This is excellent, as you can verify the objects to be updated. For a description of the compare tool, see the section **Compare objects**. When selecting import of labels, three extra tab pages will be shown in the detail window. Here you can specify the languages of the labels to be imported. The labels in the import file are listed. You will be notified if the label id is already used in the application. For each label you can choose whether to import it or create a new label id. The default label file used for creating new labels is shown in the upper right corner. The default label file is set from the labels system. For more information on the labels system, see section **Label system**.

Before choosing the option to delete table and class members during import, you must be sure that you understand this concept. If a table or class is exported, only modified objects of tables and classes are stored in the modified layer, like indexes and methods, not the whole layer of the object. When selecting to delete table and class members,

these objects of a table or a class are deleted in the AOT before importing the table or class object. This means only the methods of a class which are a part of the import file will exist after the import. This goes for the layers to which you have access. You can never delete layers like SYS or GLS.

If the imported file is exported with AOT id's, you can click the option for importing the object with id. This assures that you are keeping your applications in sync using the same id's in the AOT. You will need to do a data recovery if you are importing with id and you previously have done an import of a new table without importing the id of the table. You will be notified if changing the id of a table containing data, but you can mark the data recovery check if you are unsure.

Mark the option *Overwrite locked objects* if you are unsure if any objects imported can be locked.

Compare Objects

Comparing objects can be done either when importing a XPO file or by right-clicking on an object that exists in more than one layer and choosing *Compare* from the Add-Ins menu. When you select to compare two layers, the system will display a compare window. When clicking the lookup button for choosing the layers to compare you might notice that the same layers are listed twice, with one set marked as old layer. The old layer is from the layer files stored in the application's old folder. For information on the file structure for an Axapta installation see the manuals in the standard package. Comparing against the old layer is useful if you have done an upgrade to a new service pack of Axapta, and want to check out what modifications have been done to the old layer of an object.

The compare window is divided into two panes. In the left side you have the tree view for the object. The right side shows the comparison results for the node selected in the left pane. The comparison results have the colors red and blue. The colors are used in the result to show the differences between the two layers. Only nodes where there are differences are shown in the tree. The differences in each layer are identified by color. In the tree a blue or a red checkbox indicates that the node only exist in the layer represented by that color. A two-colored icon indicates that the node is modified in both layers. In **figure 8: Comparing two layers of an object**, two layers of the form CustTable are compared. In the compare window, lines of code which only exist in one layer are colored with the color representing that layer. A left or right arrow is displayed in the end of a colored line or a colored block of code. By clicking the arrows you can add or remove code to or from the current layer. This speeds up the process of retrofitting modifications during an update of an application, and is frequently used when you are applying a service pack update. This tool not only compares code, but property sheets as well. Like with code, changes in property values between layers may be migrated using the arrows to update the current layer. Of course there are limitations with the comparing tool: If a lot of changes have been made between the two layers,

you might be better off rewriting the object during an upgrade rather than try to resolve a confusing mix of blue and code lines.

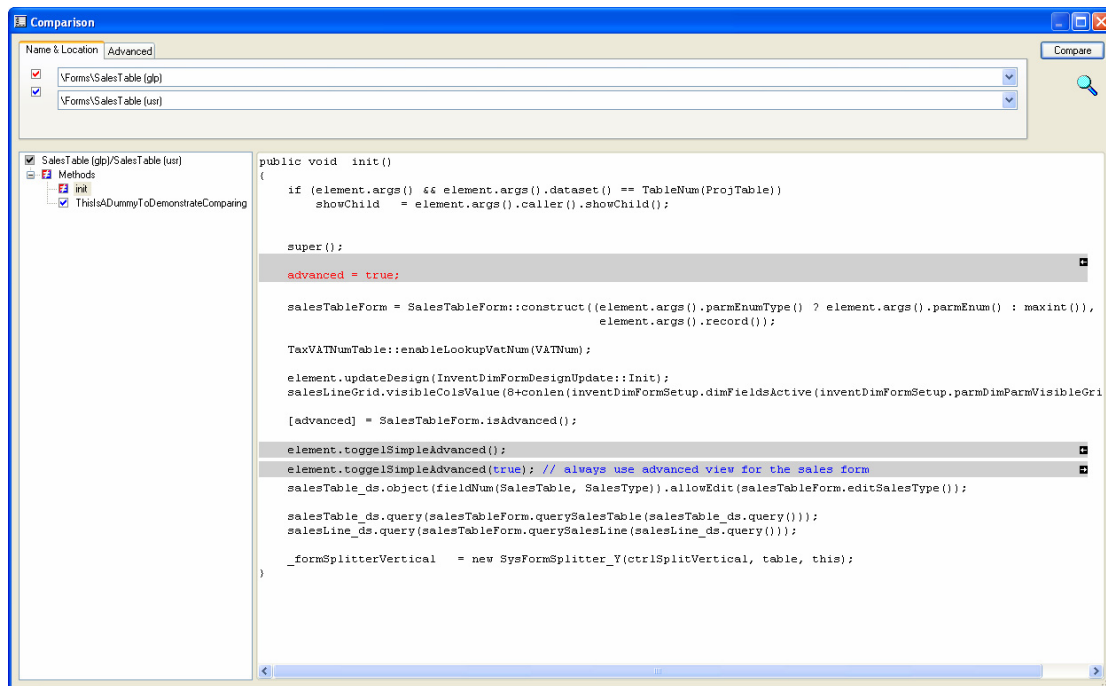


Figure 8: Comparing two layers of an object

Code Upgrade

The Code Upgrade tool is also found in the Add-ins menu. Where the Compare Object tool can be used to compare any type of object, the code upgrade tool is specialized for comparing methods. This tool is quite handy for comparing changes from an old to a new layer of a service pack or version upgrade.

Figure 9: Code upgrade shows a class that has been modified in several layers. You will see its methods listed in the left side. Methods modified in both the old layer and the new layer will be highlighted, i.e. if a method existing in the SYS layer of the old version has been modified in the new SYS layer. When you click on a method modified in more than one layer, Axapta will update the right window with a tab page representing each modified layer. The first tab page, Workspace will default to show the highest layer. From Workspace you can edit the method. It is preferable to open the code in the editor using the *Edit* button. In the sub menu *Suggestion*, the merge buttons will be active if the method is highlighted. You can use the merge buttons to merge code from all layers to the workspace. This might not complete your upgrade of a method, but it helps having all code in one place. One or two layers will be listed in the sub menu *Suggestion*. These are the layers the code upgrade tool will suggest to use as offset for the upgrade. Click the layer name to load the layer into the Workspace tab page. The *Compare* button is used to compare two layers of a method. The compared

layers are shown on a new tab page with the differences colored. For highlighted methods, the compared tab pages will automatically be created for layers with differences.

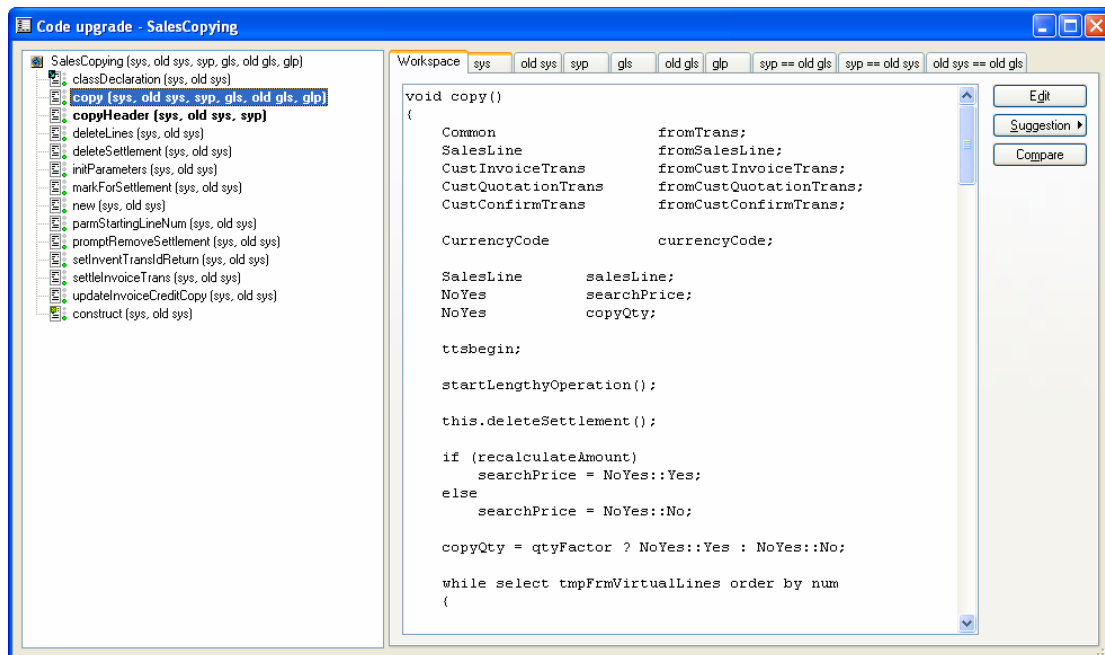





Figure 9: Code upgrade

Search

Searching through AOT objects can be done by selecting a node and pressing ctrl+f. A dialog will pop up, and by default, the search will be done on methods in the current sub tree. Methods matching the search are listed in the grid with the AOT path to the search item. By double clicking you can open the method. You can change the search to also search on properties. Change the search to select *All Nodes*, and a property tab page will be visible. The property tab page lists all properties used in the AOT sorted by name. If you want to find forms using a specific property setting, select the property and enter the property value you want to search for. For example, to search for all forms that have the property AlwaysOnTop set, mark the property and enter Yes in the range for the property. Forms where the property is set will be listed. Notice that you cannot jump to the property sheet by right-clicking. Instead, you right-click the search item and select *Properties* from the Add-ins menu.

You can search within a method by pressing ctrl+f. The search function used in methods has a find and replace function similar to that in Microsoft Word. However, if you are going to find and replace code, like changing a variable, you will be better off using the cross-reference tool as it will inform you where an object is used. For information on the cross-reference tool, see **Appendix MorphX Tools**.

Infolog

The Infolog is used to communicate with the application users. You use the Infolog to inform the user of validations, if an error occurs, or to display information to the user while processing a job. The Infolog is opened in a separate window, and opens automatically when called by a program. Information in the Infolog is removed when the Infolog window is closed, so if you need the information for later use, you can print the contents of the Infolog window by using the print menu item in the top menu. Information in the Infolog can be triggered both from code and by the kernel. Kernel information typically involves system integrity issues, such as information about mandatory database fields. From X++ you can control the information in the Infolog. There are three types of information in the Infolog. The types, which may be identified by the associated icon include: info , warning  and error . Info text is generally displayed to inform the user about actions being performed by the system. Warnings and errors typically represent issues that may require action from the user, and may indicate that a process is unable to proceed. If a help page or an action is attached to the information put in the Infolog, the icons will have a mark indicating the extra information.

```
static void Intro_Infolog(Args _args)
{
    int i;
    ;
    info("This is an info.");
    warning("This is a warning.");
    error("This is an error.");

    setprefix("prefix text");

    for (i=1; i<=3; i++)
    {
        setprefix("1. for loop");
        info("loop 1");
    }

    for (i=1; i<=3; i++)
    {
        setprefix("2. for loop");
        info("loop 2");
    }

    info("Check customer parameters.", "",
        SysInfoAction_Formrun::newFormname(formStr(CustParameters),
            identifierStr(Customer_defaultCust), ""));
    info("Check the sales form help page.", "ApplDoc://Forms/SalesTable");

    throw error("This error stop execution.", "");
}
```

This is an example on how to use the Infolog from X++. The first three lines show how to add a simple info, warning and error information. When using warning() and error()

information you will normally use the throw command as shown in the last line of the example, as the throw will stop any further action like updating a record.

If you are adding a lot of information to the Infolog, you can use the `setprefix()` function to organize the information and make the presentation more user friendly. The above example shows how you can structure a pair of nested loops so as to separately group the content in the Infolog. The prefix for the first Infolog entry represents the outer with separate indented prefixes created for each of the two loops that display info messages.

An entry in the Infolog takes three parameters. The last two parameters are options used for linking to a field in a form, or linking to a help page in the online help. The two last `info()` lines show how to use the optional parameters. The first links the Infolog entry to the customer parameter form. The system will display this form if the user double clicks on the message in the Infolog. The last `info()` entry links the Infolog entry to the sales form's help page. Notice the optional parameters are always used one at the time as you cannot link to both a form and the help system. Using links in the Infolog is a user-friendly way of communicating with the user; when an error occurs. You can provide the user with information on where exactly to go to fix the problem or provide the user with additional help on how to resolve the error. Unfortunately, changes to these links do not propagate, so if a referenced form field name or help page is changed, you will have to manually maintain any associated links.

The Infolog has a size limit of 10,000 entries. The Infolog should not be used for reporting detailed status messages representing the normal activity of batch programs. You can programmatically change the infolog's maximum size, but keep in mind that the Infolog is not meant to handle large amounts of data. Building an Infolog with several thousand entries takes time and processing resources. You are generally better off using a report.

One of the fundamental classes in Axapta is the Info class. The Info class is the handle to the Infolog. For more information on fundamental classes, see the chapter **Classes**.

Recycle Bin

The AOT Recycle bin is located in the top menu under **File | Open | AOT Recycle Bin**. The recycle bin can be used to recover objects like tables, extended data types or forms. Parts of an object, like fields of a table or methods of a form, cannot be recovered. You can only recover deleted objects for the current session, as the recycle bin is emptied when the Axapta client is closed. The deleted objects are shown in a list sorted after the objects are deleted, with the last deleted object listed as the first. If an object with the same name has been deleted twice, the objects will appear two times in the list. Notice that if a table is deleted, the recovered table will not contain any data.

User Settings

The Options form located in the top menu under **Tools | Options** is used to configure settings for your user login. These include both general settings and settings that control the development environment. To restore the default settings, use the button *Default*. For information on the *Compiler* button, see section **Compiler Window**.

General

The *General* tab page is used to define basic settings like name and password and is where you can setup single sign-ons, so your network account login can be used to automatically login to Axapta.

Status bar

The *Status bar* tab page defines the information to be shown in the status bar located at the bottom line in the Axapta window. The *Show util layer* field is especially important: When this field is checked the current layer will constantly be displayed. The *Warn company accounts change* checkbox is also important and is selected by default. When this is marked, Axapta will display a warning message whenever the current company is changed.

Fonts

You can set the default font and size to be used at the *Font* tab page. Normally the standard settings are sufficient. However you can use the font settings if you want another font size for your reports. This is a global setting. If you need a particular font within an individual form or report, you should just change that particular object.

Development

It should be no surprise that the *Development* tab page is of particular interest to developers. It is used to customize the behavior of the development environment. In the *General* field group you can select a project to be loaded at startup. This is useful when you are working on the same project over an extended period of time, as you do not have to find and open the project manually each time you start your client. For more information on projects, see the section **Projects**. The *Application object layer* field is used to set which layer of information to show in parentheses after the objects in the AOT. The *Show all layers* option will give you information on all layers an object is modified in, and can help you understand which objects are stored in each layer.

If you want to use the message window to display information regarding tracing or compiling, you might want to adjust the message limit. When the field *Development warnings* is checked the system will generate a large number of messages, many of which are of dubious value such as information on selects without an index.

The *Editor* field group has options for how to insert and add text. If *IntelliSense* is marked, methods of a current object in the editor will be looked up when a dot is typed after the object name. This is a useful feature and generally should be left on. It will save time when typing code. For example, when you type in a class name followed by a dot, the methods for the class will be looked up automatically. IntelliSense will also show yellow tooltips with information about base types and parameters when the cursor is position over variables and methods in the editor.

The *debug mode* field is normally set to its default *breakpoint*. This means if you have set a breakpoint in a method, the debugger will automatically be loaded when a breakpoint is entered.

The auto options are used to auto save modified objects. If auto save is set, objects will be saved at a defined interval. If *Auto-refresh* is marked, objects created and changed by other developers will automatically be updated. This is useful when several programmers are working on the same application and one of the programmers is creating a new table or class. The new object will then automatically be available within a defined interval. When not using auto refresh, you will have to restart your Axapta client to see the changes made by other programmers in the AOT. The garbage collection size should be set, as the number indicates the maximum number of out of scope objects that will be kept in memory. Setting this number to a larger number will tend to consume more memory but consume somewhat less CPU, as the system will run the garbage collection routine less frequently.

You can set trace options for database, methods, client/server and activeX calls. The message window is used for printing the trace options. Note, enabling method call will generate a huge number of lines in the message window, as every single method will be listed.

If you want the property sheet sorted alphabetically, you can set this field Sort alphabetically. For more information on properties, see section **Properties**.

SQL

Database tracing options are set on the *SQL* tab page. Mark the *SQL trace* field to enable tracing. You can define whether you want the tracing result to appear in the message window, the Infolog, the database or written to a file. You should only use the trace options to optimize your code. Printing the trace information to the screen using the message window or the Infolog gives you a quick way to drill down to the code. By double clicking a trace line, a form will be opened showing information about the traced line; this form allows you to directly edit the code that generated the trace line.

Confirmation

Confirmation options are important for application users. The different fields indicate the different table groups. For more information on the table groups, see chapter **Data**

Dictionary. The default settings only require the user to confirm record deletions. When testing the look and feel of an application, it is beneficial to use the same settings as the users.

Preload

This is a list of tables where all records of the table are cached. For table caching settings, see section **Data Dictionary**. Tables with few frequently referenced records are often setup to be preloaded. If you have an Axapta installation with a lot of records in one of the preloaded tables, you should consider disabling the preload of this table. The list is only a default setup and should always be verified for each Axapta installation.

Usage data


The button *Usage data* shows stored settings for objects such as forms, reports and runnable classes. When a user changes the layout of a form, changes the query of a form or enters values in a report dialog, the settings will be saved. The usage data is stored in the system table SysLastValue. Only one record exists per object per user, as only the value of the last execution is stored. If you want to see the usage data for all users, you can call the usage data form from **Tools | Development Tools | Application Objects | Usage data**.

The Usage Data form shows the content of the system table SysLastValue for the current user, divided into tab pages for each type of object. The general tab page has a button to delete all content of SysLastValue for the current user. This is quite handy if you are going to test your modifications, as you then can start out testing with the same settings as the application user will have the first time a form or report is executed. Testing modification with usage data stored for the objects is in fact a common source of errors, as an object might act differently with usage data, such as when a range is added to a query. The tab page *All usage data* lists the objects from all the usage data tab pages. Notice this tab page will also list usage data for classes.

Best Practice

This button presents a form which allows the user to set the best practice validation options. By default, all validations will be enabled but you can use the form to disable specific validation checks. Changing the best practice options can also be used if you want to run a single best practice check on all your modifications such as checking for missing labels. For more information on best practice, see the section **Compiler Window**.

1.2 Project


The AOT project window is opened by clicking the Project icon  in the top menu. Notice, that AOT Project has nothing to do with the Project application module. For differentiation, it is referred to as AOT Project.

A project is also used to group a subset of the AOT. Objects in a project act the same as they do in the AOT, meaning that you have access to the same property sheet, and identical menus when right-clicking an object. An object may exist concurrently in any number of projects; the objects will still only reside in the AOT. You should think of project objects as links to AOT object. If an object added to a project is deleted or renamed in the AOT, the link to the AOT object will still exist in the project, and the icon for the object in the project will change. Projects do not provide version control, but grouping your modifications in projects will give you a better overview of which objects have been modified for a new feature. By segregating a customized application projects you can also facilitate the application upgrade process.

Note: If you have imported the examples in this book, you will see a list of projects prefixed with MORPHXIT in the project list.

Modifying a Project

The project window shows the stored projects in a list. To open a project, double click a project node. The selected project will be opened in a new window. When adding an object to a project you can drag a node from the AOT or right-click, choose *New* and select an AOT node type. The list of available node types includes a special node type, *Group*, which is used to group objects by type as in the AOT. When you create a group node you need to specify the type of object that will be contained in that group in the group's properties. You can set the type of group like Tables of Forms. Setting the type of the group will change the icon for the group to match the icon used for the same group in the AOT. When the group type is set, only objects of the specified type can be added. You might have noticed that the group node has a property called **GroupMask**. Group masks are used for filtering using wildcards, or which object will be shown for the group. This is generally only done when initially setting up a project, as using the group mask will cause the system to replace the group's contents with those objects in the AOT that match the specified criteria.

The icon bar in top of the project window for a selected project is similar to the icon bar in the AOT. With this feature you can export the content of a project. You have an extra filter icon  not available on the AOT icon bar. The filter icon is used to build a project based on the filter options chosen. The project objects can be grouped as in the AOT by choosing AOT grouping. By clicking the *Select* button in the filter dialog you can select which AOT objects to include. The filter option will search the system table *UtilElements*, which contains information of every object in the AOT. For example, if you want to build a project with all object modified in a specific layer, this can be done by setting a range for the field *UtilElements.utilLevel*.

Project Types

There are two groups of AOT projects: private and shared. Private projects are only visible to the user who created the project. Shared projects are visible to all users. You should only use private projects for testing purposes or similar tasks. Using shared projects means that more than one developer can work on the project at the same time. To create a project, right-click the private or shared project node and choose *New*.

Beyond private and shared, there are three types of projects. The first project type just called *project* is the default project type. This is the project created if you press ctrl+n on the project node. The other two project types, Web project and Help Book are created using X++. By extending the system class *ProjectNode*, you can create your own project types. The icon in the project list indicates the type of the project. Additional options can be added to a project type by adding a menu item to the Context menu

1.3 Summary

At this point you should know how to access the development environment and how to navigate around the AOT. You should have a basic knowledge of how an Axapta application is structured using a layer technology and how these layers are used by the AOT.

The next chapter will go a step further, introducing the built in language in MorphX called X++.

2 Intro to X++

The programming language in the MorphX Development Suite is called X++. This is an object oriented language created to write the logic for the Axapta application.

You might wonder why they created yet another programming language just for Axapta. This is in fact the key to Axapta's flexibility as X++, is optimized for the creation and modification of business objects. The language is simple, with integrated SQL syntax, so there is no need to set up and manage data source connections. Just create your select statement as you would have done using a SQL database query. X++ is also tightly integrated with the MorphX tools like the form generator and the report generator. When adding logic to your forms and reports there are several default methods where you can hook up your logic.

X++ was created with C++. The C++ kernel source code is not available. You only have a set of system objects where the parameter profiles are visible. All of the X++ source code is open source. You cannot hide your X++ code, and none of the X++ code used for the standard package is hidden. This is a great advantage, as you will quickly learn to check the standard package before creating your own modifications from scratch. Often you might find useful code to get you on right track. The language has a syntax that is similar to Java, combined with the ability to write SQL-like data manipulation statements. Though X++ is an object oriented language, you do not have the ability to inherit all type of objects like in C#. Classes in X++ can be inherited just as in any other object oriented language. Besides that, you can inherit extended data types, and a set of fundamental classes give the option to make general overloads of the user interface such as changing the behavior of all forms. For a description of the fundamental classes, see the chapter **Classes**.

If you want to try out to basics of the X++ language, you can create X++ scripts using the AOT node *Jobs*. For more information about jobs, see the chapter **Jobs**.

2.1 Variables

In X++, variables are declared in the top of the editor, before the code lines. The variables are often separated from the code lines with a line only containing a semicolon. This is due to the way the compiler interpreters the code. If you do not add the semicolon you might get an error when compiling. The semicolon is not needed in all cases, but it is now an accepted Axapta standard to use a semicolon within a method at the end of the data declaration.

For declaring variables in X++, it is best practice to use extended data types, rather than using the base types. The reason for this is that the extended data types contain information like string length and whether a string is left- or right- aligned. As variables may be used throughout an application, changing an extended data type, rather than

changing variable declarations, really speeds up development. For more information on extended data types, see chapter **Data Dictionary**.

When naming variables it is considered best practice to use variables names which make sense in the context. For a variable counting customers, name the variable `noOfCustomers` rather than just a single letter variable like `i`. If you used a reserved word for your variable, the compilation will fail. Reserved words are colored blue within the X++ editor. You can find a complete overview of reserved words in the AOT under System Documentation.

The syntax for declaring a variable is as the following:

```
CustAccount MyCustAccount;
```

Here, a variable of the extended data type `CustAccount` is declared. `CustAccount` is of the base type string. First the name of the type is entered, and second a variable name. Note that the line is completed with a semicolon. A semicolon is required at the end of each statement in X++. You can declare more than one variable of the same type in the same statement by separating the variable names with a comma. All the base types are described in **figure 10: Base types in X++**.

You can declare more than one variable at the same line like:

```
CustAccount MyCustAccount1, MyCustAccount2, MyCustAccount3;
```

This will work just fine, but you should use such declaration with caution. The compiler will recognize the type of the first variable without any problems. If one of the following variables, `MyCustAccount2` or `MyCustAccount3`, have the same name as a table or a class, the code will be compiled with errors.

Note: X++ does not have the option of declaring variables as constant. If you need to define a constant you must use macros. For information on macros, see the chapter **Macros**.

Base type	Description
Str	Contains alphanumeric characters. For string operations take a look at the functions prefixed with str.
Int	Numeric values. The functions minInt() and maxInt() can be used for returning the lowest and the highest integer values.
Real	Decimal values.
Date	Date is counted from the start of year 1901. If converted to an integer, the value will be the number of days since start 1901. The highest date is end year 2155. The Global class has several methods prefixed with date used for date operations. To calculate next and previous date for week, month, quarter and year, check the functions.
Timeofday	Count the number of seconds from midnight. This is in fact a system type, rather than a base type, but timeofday can be chosen as one of the base types for an extended data type in the AOT.
Enum	An enum represents a fixed list of values. To declare enums, one of the enums created in the AOT must be used. One of the most common enums is Boolean, which have the values true and false. Enums have a max of 255 entries.
Container	A container is like an array, except a container can hold different base types. Containers are typically used for storing the values of a query. X++ provides different functions for maintaining containers. The functions are prefixed with con.
Anytype	Anytype is often used in parameter profile, as anytype can be initialized with any of the base types. Anytype is set to one of the base types when initialized. A common example of anytype is the global method Global::queryValue().

Figure 10: Base types in X++

Variables can be initialized in the same line of the declaration. This is done by entering = <InitValue> after the variable name. Some of the base types allow automatic conversion such as setting an integer to a real value. The compiler will notify you when doing type conversions which loose information. The compiler will also catch an error if you attempt an invalid initialization, like setting an integer to a string value.

```
static void Intro_BaseTypes(Args _args)
{
    Description    myString    = "A X++ string";
    Counter        myInteger   = 100;
    Qty            myReal      = 12.25;
    TransDate      myDate      = str2Date('12-31-2005', 213);
    TimeHour24     myTime      = str2Time('14:05');
    NoYesId        myEnum      = NoYes::Yes;
    PackedQueryRun myContainer = ['12', 'test', 'tada'];
}
```

```
AnyType          myAnyType      = systemdateget();
;

print myString;
print myInteger;
print myReal;
print myDate;
print myTime;
print myEnum;
print conPeek(myContainer, 1);
print myAnyType;

pause;
}
```

This is an example of how to declare the single base types. Here, all variables are initialized at the same time, and printed to screen. If you want to check the type of a variable, this can be done by pressing ctrl+space on the variable name. The base type of the variable will be shown as a yellow tooltip.

Two different string functions are used for initializing the date and time variables. How to use functions in X++ is described later in this section. Notice the notation for initializing an enum. First the enum name is specified, followed by two colons then the entry for the enum.

The container variable is initialized with a length of 3. A container can store any of the base types, even another container. Containers must be accessed using the container functions. Here the first entry in the container is printed. In this example, the variable anytype will be of the type date, as it is initialized with a function returning the current date.

Notice that text is put in quotation marks. You can use both single and double quote marks to define a text-string in X++. It does not matter to the compiler which notation is used. Best practice says that double quotes should be used for text printed to the application users, and single quotes should be used for text only used in the code.

Base types can also be declared as arrays. To declare a variable as a dynamic array, add brackets [] after the variable name. You can specify two parameters in the brackets for the array. The first parameter is used to set the array to a fixed length, and the second parameter is used if you have a large array, and only want to hold a part of the array in memory. Note, that X++ can only handle one dimensional arrays.

```
static void Intro_Array(Args _args)
{
    CustAccount    myCustAccount[];
;

    MyCustAccount[1] = "4000";
    MyCustAccount[3] = "4001";
}
```

The example is creating an array of the extended data type CustAccount. Entry number one and three of the array are initialized. When using arrays, the entries do not have to be initialized.

You might have noticed the system class Array. If you need an array of the complex types, for example an array of objects, you will need to use the system class Array. The Array class is one of the foundations classes and is described in the chapter **Classes**.

The variable container and declaring a variable as an array are both examples of the complex data types in X++. Which can be confusing since the container variable is listed as one of the base types, and a container in MorphX is treated as a base type. Tables and classes are the other two complex data types in X++.

Note: in X++ you will not have to worry about memory allocation. When an object is no longer used, the garbage collector will automatic flush the object and free the memory. The garbage collector can be tuned for special needs for the single login from the top menu **Tools | Options**.

2.2 Operators

X++ supports both unary and binary operators. Unary operators require only one operand; binary operators require two operands. A single ternary operator exists in X++, it is a short form of an if-else statement. The ternary operator, an operator with three operands, is described in the section **Control Flow Statements**.

Operators have a predefined precedence which determines in what order the operators are executed. If an expression uses more than one operator, the predefined precedence is used to determine which order to process the operators.

```
x + y * z
```

Multiplication has a higher precedence than addition, so $y * z$ will be processed before $x + y$. You may use parentheses to overrule the precedence of the operators.

```
(x + y) * z
```

Adding the parentheses will cause $x + y$ to be processed first. Even though you are aware of the precedence of the operators, you should consider always using parentheses as it makes the code must easier to read.

Assignment operators

Assignment operators are used to change a value. This could be a variable assignment or just the returned value of a calculation.

Operator	Expression	Description
=	x = y	Set x to the value of y.
+	x + y	Increase x with the value of y.
-	x - y	Decrease x with the value of y. The operator can also be used as a unary operator as prefix for integer or real values.
*	x * y	Multiply x and y.
/	x / y	Divided x with y. If x has the value 0, the compiler will throw a division by zero error.
DIV	x DIV y	Will do an integer division of x with y. The result will round downwards.
MOD	x MOD y	Will do an integer division of x with y, and return the remaining as an integer value.
++	++x	Increase x by 1. Short for x = x + 1. This is a unary operator. The operator can both be pre-fixed and post-fixed. However the result will not differ whether pre-fixed or post-fixed notation is used.
--	--y;	Decrease x by 1. Short for x = x - 1. A unary operator. It works the same whether the operator is pre-fixed or post-fixed.
+=	x += y	Increase x with the value of y. Short for x = x + y.
-=	x -= y	Decrease x with the value of y. Short for x = x - y.

Figure 11: Assignment operators

Relational operators

A relation operator will return true or false for an expression. This is used for control flow statements as an if-else statement to determine the flow. When selecting data from a table, relation operators are used to limit the number of records fetched.

Operator	Expression	Description
>	x > y	Greater than: True if x is greater than y.
>=	x >= y	Greater than or equal: True if x is greater than or equal y.
<	x < y	Less than: True if x is less than y.
<=	x <= y	Less than or equal: True if x is less than or equal y.
==	x == y	Equal: True if x is equal y.
!=	x != y	Not Equal: True if x is not equal y.
like	x like y	Match: True if x is equal y. The operator like uses the wildcards * and ? to evaluate an expression. Often like is used in select statements where records only matching a part of a field must be fetched.
&&	x && y	Logical and: This is one of the conditional operators. Will be true if x and y both are true. Often the operator is used in combination with other relational operators, where && is used to validate two expressions.
	x y	Logical or: True if x or y are true or both x and y are true. This is also an conditional operator, often used to validate two expressions.
!	!x	Logical not: Will be true if x is false. This is the only unary relational operator.

Figure 12: Relational operators

Note: If you are using an x && y expression and x is false, then y will not be evaluated, as && only returns true if both operands are true. This is useful information, as you can optimize your code by placing slow expressions at the right side like an expression which accesses the database.

Bitwise operators

Bitwise operators are, as the name indicates, used to evaluate expression by using arithmetic and conditional operators at the bit level. These bitwise operators can only be used with integer values.

A common situation for using bitwise operators is as an alternative to having a set of variables setting true or false values if you have to control some access controls with several levels. However the use of bitwise operators will make your code more difficult to read. This might be the reason that these operators are not commonly used in the

standard application. In fact, the only place you may find yourself using the bitwise operators is when you are interfacing with external applications such as interacting directly with Windows API or need to do any low-level operations on the database.

To be able to understand the result of an expression using bitwise operators, you will have to translate the integer values to binary numbers.

```
13 & 10    // 1101 bitwise and 1010
```

The expression `13 & 10` will give the result 8. This is because 13 and 10 is compared bit by bit and only the bits set for both digits are counted. The expression is translated to binary after the comment. Only the fourth bit is equal, and thereby the result is 8.

Operator	Expresion	Description
<<	<code>x << y</code>	Shift left: The binary value of x will be shifted y positions to the left. This will increase the value of x.
>>	<code>x >> y</code>	Shift right: The binary value of y will be shifted y positions to the right. This will decrease the value of x.
&	<code>x & y</code>	Binary AND of x and y: The bits set at the same position for x and y will make the result.
	<code>x y</code>	Binary OR of x and y: Bits set for x and y will be summed.
^	<code>x ^ y</code>	Binary XOR of x and y: The bits which are not set at the same position for x and y will make the result.
~	<code>~x</code>	Binary NOT of x: All bits will be reversed.

Figure 13: Bitwise operators

2.3 Control Flow Statements

Code in X++ is executed sequentially. Often it is necessary to have a block of code executed a number of times or only execute part of the code. Control flow statements in combination with relational operators are used to set up conditions for how the code is executed.

Loops

A loop is used to repeat the same block of code. An expression can be set up for the loop in order to determine the number of times to repeat the loop.

```
static void Intro_While(Args _args)
{
    Counter    counter = 1;
    ;

    while (counter <= 10)
    {
        info(strfmt("while loop %1", counter));
        counter++;
    }
}
```

This is an example of a *while* loop. An integer variable used to control the number of iterations is set to the value 1. The while loop will be executed as long as the counter variable is less than or equal to 10. For each loop, a line is printed to the Infolog and the counter variable is increased by 1.

The function `strfmt()` is used to format any base type variables into a string. Since the `info()` method only accepts a string as its first parameter, the counter variable is formatted to a string. `Strfmt()` uses the notation `%<variable number>` to insert variables in the text.

Notice that the while loop starts and ends with brackets `{}`. The brackets indicate the beginning and end of a block of code. Only code within the brackets will be looped. It is not mandatory to set begin and end brackets, but if the brackets are not added, only the first line of code after the while will be repeated. If the bracket were skipped in this case, the loop would never end. It is considered best practice to always add begin and end brackets, even if a block only contains one line, as it makes the code easier to read. If your need add more lines later, the block of code to be looped is already defined.

In X++ while loops are often used to fetch data from the database. For examples on how to use while loops for selecting data, see the section **Select statements**.

```
static void Intro_DoWhile(Args _args)
{
    Counter    counter = 1;
    ;

    do
    {
        info(strFmt("do-while loop %1", counter));
        counter++;
    }
    while (counter <= 10);
}
```

A *do-while* loop is similar to a while loop. The main difference is that a do-while loop will always be executed at least one time, as the condition for the loop is processed after each iteration. A while loop, where the condition is evaluated at the beginning of each

iteration, may not be executed at all if the condition for the loop is not fulfilled. The example shows the same code used in the while loop. Notice that the while clause in a do-while loop must be ended with a semicolon.

```
static void Intro_For(Args _args)
{
    Container names = ["Lay", "Kai", "Zbigniew", "Rolf", "Memed"];
    Counter counter;
;

    for (counter=1; counter <= conlen(names); counter++)
    {
        info(strFmt("%1: Name: %2", counter, conpeek(names, counter)));
    }
}
```

A *for* loop functions much like a while loop but offers a more concise syntax. A for loop has three components contained within parenthesis: a *counter variable initialization clause*; a *validation expression*; and an *incrementing expression*. For each iteration of the loop, the counter variable is updated based on the incrementing expression. Braces are used to indicate the scope of the loop. The validation expression is evaluated at the start of each loop. When the validation expression evaluates to false, the loop is no longer executed and control is passed to the statement immediately following the closing brace. It is important that the code within the loop adjust the values evaluated within the validation expression in order for the loop to end. The for loop used in the example is traversing a container with five elements. The variable counter is initialized to 1 and is used to peek the first element of the container. The block is repeated as long as the counter variable is less than or equal to the length of the container.

Conditional Statements

Depending on the result of an expression, you often need to execute different parts of code which is the reason for having conditional statements.

```
static void Intro_IfElse(Args _args)
{
    NoYesId printToInfolog = true;
;
    if (printToInfolog)
    {
        info("Print to Infolog");
    }
    else
    {
        print "Print to window";
        pause;
    }
}
```

If statements are the most common conditional control flow statement. In the most simple form, you can use a single if condition, which consists of a relational expression contained in parentheses. If the expression in the if statement is true, the block of code in the if statement is executed. This block of code may be a single statement or a series of statements contained in braces {}. To extend an if statement an *else* clause can be added. In this case, the else condition will be executed if the expression in the if statement is false. The above example illustrates the use of an if-else statement. More complex logic can be implemented by nesting if statements. In the example below, a condition has been set for the block of code within the else clause.

```
static void Intro_IfElse(Args _args)
{
    NoYesId    printToInfolog = true;
    NoYesId    printToWindow = true;
;
    if (printToInfolog)
    {
        info("Print to Infolog");
    }
    else if (printToWindow)
    {
        print "Print to window";
        pause;
    }
}
```

In the above example, you could use two separate if statements to get the same result. However, by nesting the second if under the else, the second block will never be validated if the first if statement is true, and thus offers a slight performance advantage.

You can add as many if statements inside each other as needed. But be careful as this can make the program difficult to read and debug. Often, the need for these multiple if statements comes from the need to condition a program's execution based on the individual value contained in a variable. If the number of values is small, a normal if statement is fine, however, if the variable can take on more than two or three values, the list of if statements soon becomes unwieldy. Fortunately, X++ offers a conditional statement that is especially designed to handle this situation, the *switch* statement:

```
static void Intro_Switch(Args _args)
{
    SalesStatus    salesStatus = SalesStatus::Invoiced;
;
    switch (salesStatus)
    {
        case SalesStatus::Delivered:
            info("Salesorder is delivered");
            break;
        case SalesStatus::Invoiced:
            info("Salesorder is invoiced");
            break;
    }
}
```

```
case SalesStatus::Canceled:
    info("Salesorder is cancelled");
    break;
default:
    info("Do nothing");
}
```

If this switch was constructed using if-else statements, it would have required several levels making the code more difficult to read. In a switch statement all conditions are added at the same level making the code easy to read and easy to add additional conditions.

An expression is set on top of the switch statement which defines the variable to be evaluated. Following the switch clause are one or more case clauses. Each case clause specifies one or more values separated by commas and terminated by a colon. X++ evaluates each of the case clauses from top to bottom. If the value of the variable specified in the switch clause is equal to one of the specified values then the code following that case clause is executed. If no cases are validated true, the default case is executed. The default case is not mandatory for a switch. If you want to notify the application users if no cases are true, you should add a default. Notice a break is added to the end of each case in the switch. When X++ encounters a break clause, control is passed to the first statement following the end brace defining the scope of the switch statement. If the break command is skipped, all cases after a true case will be executed. This is in fact the most common bug when using switch statements. Running a best practice check will catch missing breaks in switch statements.

```
static void Intro_TernaryOperator(Args _args)
{
    Boolean    printCustomerName = false;
    ;

    print printCustomerName ? "Customer name" : "Customer account";
    pause;
}
```

For simple conditional expressions, there is a useful short alternative. This is a ternary operator, an operator with three operands. The first operand consists of a conditional expression, like in an if statement, followed by a question mark. The second and third operands are each statements that will be executed based on whether the first operand evaluates as true or false. If the expression is true then the first statement will be executed otherwise control is passed to the second statement. these clauses are put after the question mark and are separated with a colon. Note that the expressions defined for both the true and false conditions must resolve to the same data type.

Since the ternary operator allows a conditional statement to be written concisely on a single line, they can enhance the readability of a program. They are most useful for assignment statements when the value being assigned can only take one of two possible

values. For example, they are often used to set the visible or enabled properties for form controls.

Exceptions

Loops and conditional statements are both designed to control which block of code to be executed. Unexpected conditions like warnings and errors need to be shown to the user as they occur. Exception handling should separate information in unexpected conditions from the regular code block and takes action based on the information that triggered the error.

```
static void Intro_TryCatch(Args _args)
{
    Counter counter;

    try
    {
        while (counter < 10)
        {
            counter++;

            if (counter MOD 7 == 0)
                throw error("Counter MOD 7 is zero");

            if (counter MOD 3 == 0)
                throw warning("Counter MOD 3 is zero");
        }
    }
    catch (Exception::Error)
    {
        print ( strfmt("An error appeared at loop %1", counter));
    }
    catch (Exception::Warning)
    {
        print ( strfmt("A warning appeared at loop %1", counter));
        retry;
    }

    pause;
}
```

Try-catch statements are used for exception handling in X++. A try-catch statement consists of two or more separate blocks of code, a try block which attempts some operation, and one or more catch blocks where exceptions thrown within the try block are caught. Exceptions may either be thrown by the kernel or by using the command throw from code within the try block. The catch part takes action on exceptions. You may have multiple catch blocks, each designed to catch a particular type of exception. Action will only be taken on types of exceptions explicitly referenced in your code. The try-catch example shown will take action on the exception type's error and warning.

You can add a catch block without specifying a catch type to handle all types of exceptions not defined for any other catch blocks.

So what is the big deal about using exception handling, as you can make similar code using control flow statements? The benefit of using exception handling is that you can catch the exceptions before they are thrown to the Infolog, and take appropriate action. They are especially useful when doing database operations like updating records. When updating a record, an exception might occur due to database locks. Usually, this is a temporary condition and if the operation is retried it will succeed. Without exception handling, an unsuccessful update will be passed to the standard error handler which will display an error in the infolog and unnecessarily terminate the program. When an exception is thrown you can use the command `retry` to re-run the try block, and the user will never have noticed that an exception occurred. When coding a catch block, it is usually necessary to include logic that will detect that the condition causing the error has been unresolved, otherwise your program could wind up in an endless loop as it endlessly retries the code that generated the error.

Miscellaneous

X++ has commands which can be used to override the execution of a block of code. Often these commands are used in control flow statements, when the remaining code in that block is not to be executed.

```
static void Intro_Break(Args _args)
{
    Counter    counter;
;

    for (counter=1; counter <= 10; counter++)
    {
        print counter;

        if (counter == 5)
        {
            break;
        }
    }

    pause;
}
```

Previous examples on how to use the `break` command in switch statements to end each case has been shown. You can also use the `break` command in any block of code. If used in a control flow statement, the expression set for the loop will be interrupted. Here a `break` is set in a for loop, causing the loop to end after five runs. If you are executing a heavy block of code, like doing a search on the database, you can speed up your code adding a `break` when your needs are fulfilled, rather than waiting for the loop to complete.

Using the *return* command instead of *break* in a loop will give the same result as using *break*, however, the *break* command is generally preferred. Return is used to return a value from a method. The use of methods is described in the chapter **Classes**.

```
static void Intro_Continue(Args _args)
{
    Counter    counter;
    ;

    for (counter=1; counter <= 10; counter++)
    {
        if (counter MOD 3 == 0)
        {
            continue;
        }

        print counter;
    }

    pause;
}
```

The difference between *break* and *continue* is that *break* steps out of the loop, while *continue* will just skip the remaining code in the current iteration. You will get the same result using an *if* statement instead of using the *continue* command. It is an option, and if you are about to add a condition to a large code block, you could consider using *continue*, instead of adding an *if* statement and indenting at lots of code lines.

2.4 Select Statements

X++ uses *select* statements to fetch data from the database. Select statements can be written from code like any other statements in X++. In order to use a select statement you must first declare variables for the tables being referenced. A special form of the while statement, *while select*, can be used to create a loop that will fetch all of the records that fulfill specific criteria. The selection criteria are defined using expressions based on operators and variables. Notice that the base type *str* cannot be used in expressions, unless the length of *str* is specified. This is yet another reason to use extended data types.

```
static void Intro_Select(Args _args)
{
    CustTable    custTable;
    CustTrans    custTrans;
    TransDate    fromStartYear = systemdateget();
    ;

    while select custTable
        join custTrans
```

```
where custTrans.accountNum == custTable.accountNum
    && custTrans.transDate >= fromStartYear
{
    info(strfmt("%1, %2, %3", custTable.accountNum, custTrans.transDate, custTrans.voucher));
}
}
```

This is an example on how to use select statements in X++. The tables CustTable and CustTrans are joined, and CustTrans records with a transaction date equal to or higher than the start of year are fetched. Notice that a variable is declared which will hold the system date returned by the function `systemdateget()`. While this function could be used directly in the select statement, it would slow down the select, as the function `systemdateget()` would have to be calculated for every loop.

X++ select statements do not precisely follow the SQL standard. Also, X++ does not implement all of the standard SQL keywords. If you are familiar with writing SQL statements, you will soon adapt the way select statements are used in X++. For an overview of the keywords available in X++ sorted by keyword, see **figure 14: Select keywords overview**.

Keyword	Description
asc	<p>Set the sorting order to ascending. All selects are default fetching data ascending. Used in combination with order by or group by.</p> <p>Example</p> <pre>select custTable order by accountNum asc;</pre>
avg	<p>Aggregate keyword used to calculate the average value of a field from the fetched records. Selects using aggregate keywords using only one call to the database calculating a result based on multiple records.</p> <p>See class method <code>KMKnowledgeCollectorStatisticsExecute.runQuery()</code>.</p> <p>Example</p> <pre>select avg(amountMST) from custTrans;</pre>
count	<p>Aggregate keyword used to count the number of records fetched.</p> <p>See class method <code>KMKnowledgeCollectorStatisticsExecute.runQuery()</code>.</p> <p>Example</p> <pre>select count(recId) from custTrans;</pre>
delete_from	<p>Will delete multiple records in one call to the database. This really speeds up when deleting a lot of records, as an ordinary while select, calling delete will generate a database call for each record.</p> <p>See class method <code>InventCostCleanUp.updateDelSettlement()</code>.</p> <p>Example</p> <pre>delete_from myTable where myTable.amountMST <='1000';</pre>
desc	<p>Set the sorting order to descending. Used in combination with order by or group by. Select descending can be very slow, as indexes are always sorted ascending.</p> <p>See table method <code>CustTable.lastPayment()</code>.</p> <p>Example</p> <pre>select custTable order by name desc;</pre>
exists join	<p>Exists join is used to fetch records where at least one record in the secondary table matches the join expression. No records will be fetched from the secondary table using exists join.</p> <p>See class method <code>InventAdj_Cancel.cancelInventSettlements()</code>.</p> <p>Example</p> <pre>while select custTable exists join custTrans where custTable.accountNum == custTrans.accountNum</pre>

firstfast	<p>Instruct to select the first record faster. Can be used in situations where only one record is shown, like in a dialog.</p> <p>See class method ProjPeriodCreatePeriod.dialog().</p> <p>Example</p> <pre>select firstfast custTable order by accountNum;</pre>
firstonly	<p>Only the first record will be selected. Firstonly should always be used when not using while in selects, even if the select only will return a single record. The methods find() and exists() often created on tables uses all firstonly.</p>
forceliterals	<p>Will force the kernel to select data without using a prepared select. Used to make sure the database chooses the optimal index based on the values used in the select statement.</p> <p>To optimize joins between large tables, the kernel default fetch data using forcелiterals. The kernel uses the table property TableGroup for this. Data is fetched using forcелiterals in joins where at least two tables are not from the TableGroup Parameter or Group.</p> <p>Example</p> <pre>select forcелiterals custTrans order by voucher, transDate where custTrans.accountNum >= "4000";</pre>
forcenestedloop	<p>Used in joins to instruct the kernel to fetch a record from the primary table, before fetching any records from secondary tables. Often used in combination with forceselectorder.</p> <p>See class method ReqCalc.actionCalcDimTrans().</p> <p>Example</p> <pre>select forcenestedloop forceselectorder custTable join custTrans where custTable.accountNum == custTrans.accountNum && custTable.name >= "4000";</pre>
forceplaceholders	<p>Will force the kernel to select data using a prepared select. Used to overrule the default forcелiterals in join on large tables. The kernel default select uses forceplaceholders when not selecting from large tables.</p> <p>See class method ReqCalc.actionCalcDimTrans().</p> <p>Example</p> <pre>select forceplaceholders custTable join custTrans where custTable.accountNum == custTrans.accountNum && custTable.name >= "4000";</pre>
forceselectorder	<p>Forceselectorder will instruct the kernel to access tables in a join in the joined order. Often used in combination with forcenestedloop.</p>

forupdate	<p>If records in a select are to be updated, the keyword forupdate must be added.</p> <p>Example</p> <pre>while select forupdate custTable</pre>
from	<p>Default all fields of a table is selected. From is used to select only the fields specified. This will optimize a select, especially on tables with many fields. Use it for optimization only, as it makes the code more complex.</p> <p>Example</p> <pre>select accountNum, name from custTable;</pre>
group by	<p>Sort the fetched data group by the fields specified. Only the fields specified in the group by will be fetched.</p> <p>See class method InventStatisticsUS.calcTotals().</p> <p>Example</p> <pre>while select custTable group by custGroup;</pre>
index	<p>Set the sorting order of the fetched data. The kernel will convert the keyword index to an order by using the fields from the index. Index should only be used if the fetched data must be sorted in a specific way, as the database will choose a proper index.</p> <p>Example</p> <pre>while select custTable index accountIdx</pre>
index hint	<p>Index hint will force the database to use the specified index. The database will always try to choose the best index. Index hint should only be used if the database does not choose a proper index.</p> <p>Example</p> <pre>while select custTable index hint accountIdx</pre>
insert_recordset	<p>Used to insert multiple records in a table. Insert_recordset is useful when copying data from one table to another as it only requires one call to the database. The same number of fields must be selected from both tables, and the fields base type must match.</p> <p>See class method ReleaseUpdateDB_V25toV30.updateASPEmail().</p> <p>Example</p> <pre>insert_recordset myTable (myNum,mySum) select myNum, sum(myValue) from anotherTable group by myNum where myNum <= 100;</pre>
join	<p>Join will fetch data using an inner join. Records matching the join expression will be fetched from both tables.</p> <p>See table method SalesTable.LastConfirm().</p>

	<p>Example</p> <pre>while select custTable join custTrans where custTable.accountNum == custTrans.accountNum</pre>
maxof	<p>Aggregate keyword used to return the highest field value fetched. See class method KMKnowledgeCollectorStatisticsExecute.runQuery().</p> <p>Example</p> <pre>select maxOf(amountMST) from custTrans;</pre>
minof	<p>Aggregate keyword used to return the lowest field value fetched. See class method KMKnowledgeCollectorStatisticsExecute.runQuery().</p> <p>Example</p> <pre>select minOf(amountMST) from custTrans;</pre>
next	<p>Used to step to the next record in a select. It is preferable using a while select instead.</p> <p>Example</p> <pre>select custTable; next custTable;</pre>
nofetch	<p>Select without fetching data. Can be used if the fetched result is just passed on to another method like setting the cursor. See class method PurchCalcTax_PurchOrder.initCursor().</p> <p>Example</p> <pre>select nofetch custTable;</pre>
notexists join	<p>Opposite of exists join. Will fetch records from the primary table, where no records in the secondary table match the join expression. See class method InventConsistencyCheck_Trans.run().</p> <p>Example</p> <pre>while select custTable notexists join custTrans where custTable.accountNum == custTrans.accountNum</pre>
order by	<p>Set the sorting order of the fetched data. Used to make sure the fetched data are sorted in a exact way. If no index exists for the order by fields, this might be time-consuming. See table method CustTrans.transactionDate().</p> <p>Example</p>

	select custTrans order by accountNum, transdate desc
outer join	<p>Outer join will select records from both tables regardless if there are any records in the secondary table matching the join expression. See class method SysHelpStatistics.doTeams().</p> <p>Example while select custTable outer join custTrans</p>
reverse	<p>Fetch the data in reversed order. The keyword asc and desc are related to a single field where reverse is related to the table. See class method InventUpd_Reservation.updateReserveLess().</p> <p>Example while select reverse custTable</p>
setting	This keyword is used in combination with update_recordset.
sum	<p>Aggregate keyword used to sum values of a field fetched. See class method KMKnowledgeCollectorStatisticsExecute.runQuery().</p> <p>Example select sum(amountMST) from custTrans;</p>
update_recordset	<p>update_recordset will update multiple records in one database call. Useful to initialize fields in a fast way. The fields updated are specified after the keyword setting. See class method ProdUpdHistoricalCost.postScrap().</p> <p>Example update_recordset myTable setting field1 = myTable.field1 * 1.10;</p>

Figure 14: Select keywords overview

In previous version of Axapta, it was common practice to specify which index to use in select statements using the keywords `index` and `index hint`. Modern database optimization routines are extremely good at determining the best index to support a given query so there is no need to specify an index in select statements. Indeed, specifying the wrong index can have a significantly adverse effect on performance. You should only consider using an index if you need to ensure that the fetched data is returned in a specific order.

The keywords *forceliterals*, *fornestedloops*, *forceplaceholders*, *forceselectorder* can be used to optimize query performance. These keywords change the default behavior of how data is retrieved from the database also called the execution plan. Be careful how

you use these. You might get better performance on your test data, but realize worse performance on live data. The impacts of these keywords are dependent on the composition of the data. Don't use them unless you know what you are doing.

When using select statements to retrieve records to be modified such as deleting and updating records the *Transaction Tracking System* referred to as *TTS* must be used. The TTS allows you to define logical transaction boundaries. A logical transaction may involve the update of records in several different tables. Often the integrity of the database is dependent on the relationship between the updated records. The TTS ensures that should a single update in the transaction fail, the other transactions in the group will be rolled back to their pre-transaction state in order to preserve the integrity and consistency of the database. The system will not allow you to update existing records without using the TTS, and will generate an error. For more information on TTS, see the chapter **Data Dictionary**.

2.5 Functions

X++ has a set of system functions that you can draw upon to perform various tasks. These functions are written in C++ and are part of the Axapta kernel. Several of the functions can be used for type conversion like converting an integer to a string. There are also a collection of useful functions that perform date and string operations. An overview of the system functions can be found under the *System Documentation* node in the AOT.

When using functions in X++, you just have to type the function name and enter the parameters in parentheses. Advance X++ programmers may recognize that functions are addressed in the same way as Global class methods. From an X++ standpoint, they look the same. One way to identify the origin is by right-clicking to lookup the function or method, as you cannot lookup a function.

2.6 Summary

The programming language X++ used by MorphX has been introduced in this chapter. You should by now have acquired knowledge on how the syntax of X++ is. How to declare variables, using flow statements and how data are selected from the database. Learning the X++ language is the first step, when getting to know how to making your own modifications for Axapta. The next chapters will show how to use the individual nodes in the AOT, and how to add logic using X++ to the AOT objects.

3 Data Dictionary

The data model for the Axapta application is defined in the *Data Dictionary*. When modifying an Axapta installation this is the area in which you would commence designing your new data model. The data in an Axapta installation is stored in a relational database. In short, this means that data is separated in tables to prevent redundant data occurring and relations are defined between the tables making it possible to gain access to the related data. Basic knowledge about creating a database relationship model and how to normalize data will be beneficial when designing your data dictionary modifications for Axapta. Explanation of relational databases, their development and subsequent support is not detailed within this book. There are many books available in this subject as well as vast amount of knowledge now available on the internet.

An Axapta installation uses either a Microsoft SQL Server or an Oracle database for storing data. Tables, views, fields and indexes are synchronized to the database when created, changed or deleted from Axapta. This definition of the data, is the only information about the data dictionary stored in the database, the actual data is physically stored within the Microsoft SQL Server or Oracle database. Information regarding the actual table relations and delete restrictions may be found in MorphX. Visual MorphXplorer can be used to build the actual entity relationship diagram (ERD).

3.1 Tables

Two main categories of tables exist:

- *Application Tables*. These are used to define and build the application modules. On the initial definition of your Axapta installation, you will determine which specific application tables you will be using. This will be defined based on the specific configuration keys which are enabled and will be synchronized to the database software at start-up.
- *System Tables*. These are tables that contain information specific to the operation of the Axapta infrastructure and are created in the kernel of the application. Systems tables are automatically synchronized to the database operating system.

If discovering synchronization problems, the SQL tool located in the main menu under **Administration | Periodic | SQL Administration** will be useful. Running a Check/Synchronize will fix the most common synchronization errors.

Company

Axapta has the option of using the same application for information from different companies. Even though all of the data is kept on the same *physical* database, Axapta

will determine which data is to be used for each company defined – even though the application will appear to be exactly the same in its look, feel and functionality.

This is useful for a number of areas:

- You may have test data that you wish to use for training new staff on your current Axapta system. They can sign in to Axapta under the ‘test’ company and you are safe in the knowledge that none of your critical ‘live’ data will be damaged.
- You may wish to have a ‘planning’ or ‘budget’ company where you can prepare future business models.
- You may have more than one physical company that handles completely different business, but uses the same systems functionality. You can have separate information to support each company: For example, distinct General Ledger definitions, Customer and Creditor specifications, stock and inventory management rules and even employee structure. All of this information will be stored under a separate company id.

Axapta manages this by using the system field *dataAreaId* to define which company the data belongs to. When adding a record, *dataAreaId* is automatically initialized with the current company id. Whenever a user requests to view data from forms and reports or when the application fetches data using a select statement, Axapta will always return data from the current company, as a filter for *dataAreaId* is automatically added by the kernel.

```
static void DataDic_ChangeCompany(Args _args)
{
    DataArea  dataArea;
    CustTable custTable;
;

    while select dataArea
    {
        if (!dataArea.isVirtual)
        {
            info(strfmt("Company: %1", dataArea.id));

            changeCompany(dataArea.id)
            {
                custTable = null;


                while select custTable
                {
                    info(strfmt("%1, %2", custTable.dataAreaId, custTable.accountNum));
                }
            }
        }
    }
}
```

The above example illustrates what to do if you need to fetch data from more than one company and print customer accounts printed from all non-virtual companies. The `changeCompany()` is used to switch to the selected company. The company is changed within the `changeCompany()` code block and the starting company is automatically set afterwards. Notice that `CustTable` is set equal null. The table variable must be reset otherwise data will be fetched from the default company. `ChangeCompany()` should be used with precaution as you might end up modifying data in the wrong company. Virtual companies are described in the section **Table Collections**.

Application tables

When a new field or a new form is required in Axapta, the data being entered must be stored in the database in an appropriate table. New fields can either be added to an existing table, or to a new table. If you choose to add new fields to an existing table, the new fields will be added to the current layer even though the modified table belongs to another layer. This goes for all objects on a table, except delete actions. When upgrading an application to a new release, this is really an advantage as you will only have to manually merge nodes modified in more than one layer. The Tables node is located under *Data Dictionary* in the AOT.

Table Browser

The table browser is accessed by right-clicking a table and selecting *Table Browser* from the Add-Ins menu. The table browser is created using a standard form which is called `SysTableBrowser`. You can call the table browser from any application table or system table. You may select the Table Browser from any data source. By default, all fields with the property **Visible** set to “true” are shown. To reveal the common fields of a table, switch to show Auto-report fields. You can filter the records shown in the table browser by writing a select statement, but it is far easier to filter using the query icon  from the top menu.

To get an overview of data in a table, the table browser is quite handy as you will not have to know from where in the menu the table is accessed. For testing purposes the table browser can be used to create data or alter existing data. However, it is extremely important you only use this method for testing. Data changed used the table browser *will* be validated upon the business logic defined in the table scripts, but the form which has been created within Axapta must be used to key in live data for the table as this form may contain additional logic. You may damage your data using the table browser in a live application and therefore this is not an application user tool.

Creating tables

It is important that you initially consider how your new tables are going to be used. When the new tables are installed in a live application, and data has been entered to the

tables it will be more complex to redesign your tables and indexes as you also will have to convert existing data.

Performance is also an important issue. If you are creating a new transaction table, you must assure that you have a well defined index for fetching data, both for simple selects and when using the new table in joins to other tables. Where additional fields are needed on an existing table, you might have chosen to create a new table with the required fields. This may be a better solution; just keep in mind that each time you need to access the new fields you will have to join the new table with the existing table. This can be frustrating, and slower, when fetching the data from additional tables. As long you have considered your design this might be the right solution. The point is, when creating tables you must have the overview of how the tables are going to be used before shipping your modifications.

Example 1: Creating a table**Objects used from MORPHXIT DataDictionary project**

- Table, MyTable

In this first example, a table to store customer information will be created. Initially a few fields is added to the table. Throughout this chapter more features will be added.

1. Create a new table by going to the *Tables* node in the AOT, right-click and choose *New Table*. A new table called "Table1" will be created. Open the property sheet and rename the table to "MyTable".
 2. Expand the new table node so the nodes at the next level are visible. Open another instance of the AOT and go to *Data Dictionary/Extended Data Types*. Locate the extended data type AccountNum and drag the extended data type AccountNum to the *Fields* node of MyTable.
 3. Locate the extended data types CustName, CustGroupId and CurrencyCode and then drag them all to the *Fields* node of MyTable.
 4. Save the table by pressing ctrl+s at the MyTable node. The table is now synchronized to the database and you have created your first table!
-

When the extended data types were dragged to the *Fields* node of the table, a new field was created of the same name as the extended data type. This is the easiest way of creating new fields. You can multi-mark the extended data types and drag them at one time. You can then define the individual properties for the new fields.

By using the table browser, you can now add records to MyTable. When creating a new record notice that the fields' custGroupId and currencyCode both have a lookup button and only values from the related lookup table can be chosen. The reason for this is that the extended data types for two of the fields you created have a relation to the tables

CustGroup and Currency. Without any coding you have already created a new table with relations!

The table property **TableGroup** is used to group tables. This is a property which is often forgotten when creating a new table. This property is used to specify the content of a table. For example, whether the table is a main table or a transaction table; such as the customer and customer transaction tables respectively. Browsing the existing tables will give you an idea of how to set the table group property. The default value of the property TableGroup is "Miscellaneous". As data confirmations and data exports can be filtered based on the table group property it must be set. If not defined correctly for tables containing huge amount of data, such as transaction tables, joined select statements will not perform optimally as the kernel uses the table group property to determine whether to select using forceLiterals. For more information on selects, see chapter **Intro to X++**.

There are still some more properties which require setting, especially properties related to security and performance. The common table properties will be explained in this chapter.

For a complete overview of the properties for the data dictionary, please see the **Appendix Properties**.

Table variables

Tables are declared in X++ like any other variable. To initiate a table variable with a value from the database select statements are used. You can also set a table variable equal to another as long as both tables variables are of the same type.

```
static void DataDic_OneCursor(Args _args)
{
    CustTable custTable, newCustTable;
;

    select firstly custTable;

    newCustTable = custTable;
}
```

In this example with the CustTable, only one cursor position will exist if one table variable is set to equal another. If one of the table variables is used to select another record, both table variables will point to the new record.

```
static void DataDic_TwoCursors(Args _args)
{
    CustTable custTable, newCustTable;
;

    select firstly custTable;

    newCustTable.data(custTable);
}
```

```
}
```

To have two separate cursors the method data() must be used. Now the table variable custTable and newCustTable will each have a cursor.

Temporary tables

A temporary table contains no data and is not synchronized to the database. A temporary table may be used as any normal table in joins and selects. The table property **Temporary** determines whether a table is temporary. Any application table can be set to temporary by setting the table property Temporary to "Yes". Caution: Setting a table containing data to temporary will cause existing data to be deleted! To easily locate temporary tables in the AOT, all temporary tables are prefixed with Tmp*. A common reason for using temporary tables is for the sorting of data. You might have to present data using a specific sort in a form or a report which cannot be accomplished using a select statement or a query. Instead you can create a temporary table with the fields needed and the sorting to be used. Data must be fetched from the tables and inserted in the temporary table according to the sort requirement. As long the temporary table is in scope, the temporary table will contain data. The content of a temporary table is stored in the file system as a temporary file.

```
static void DataDic_TmpTable(Args _args)
{
    CustTable          custTable;
    CustTrans          custTrans;
    TmpCustLedger      tmpCustLedger;
;

    while select custTable
    {
        tmpCustLedger.accountNum = custTable.accountNum;
        tmpCustLedger.name       = custTable.name;
        tmpCustLedger.insert();
    }

    while select tmpCustLedger
    {
        info(strFmt("%1, %2", tmpCustLedger.accountNum, tmpCustLedger.name));
    }
}
```

In this simplified example, the customer account numbers and names for all customers are inserted in the temporary table TmpCustLedger. The content of TmpCustLedger is looped and printed to Infolog. Data inserted will usually be filtered and fetched from one or several tables.

Using temporary tables will generate an overhead as initially, the data must be fetched and inserted in a temporary table. Next, the temporary table must be looped. If you have a form with a complex query which makes the form act slow while navigating,

using a temporary table might give a better solution. It will initially take a longer time to load the form, but the application will perform much faster once the data is loaded and thereby provide a more user-friendly system.

For more information on using temporary tables in forms, see chapter **Forms**.

```
static void DataDic_SetTmp(Args _args)
{
    CustGroup custGroup;
    ;

    custGroup.setTmp();

    custGroup.custGroup = "10";
    custGroup.name      = "Test customer 10";
    custGroup.insert();

    while select custGroup
    {
        info(strFmt("%1, %2", custGroup.custGroup, custGroup.name));
    }
}
```

An instance of a table can be set to temporary from X++. Data will not be deleted from the database, only the table variable will be empty. In the example above, the table CustGroup has been set to temporary. A single record is inserted in the temporary instance of CustGroup. When looping the temporary table, only the single record which has been inserted is printed. You should be careful if using existing tables as temporary tables. If used in a live system and someone uncomments the code line setting the table to temporary, the result will be fatal. You can find examples of this in the standard package. However, the extra work of creating a new temporary table instead might be paid off in the long run.

If you have temporary tables with a lot of records you should consider building your temporary table at the server side. The first record inserted in a temporary table will determine from where the temporary table will run. It can be an advantage creating a class for maintaining your temporary table, as an entire class can be set to run on the server.

For more information on defining where to run your code, see chapter **Classes**.

System tables

Unlike application tables, system tables are non-changeable. You cannot modify the data model of a system table. System tables are tables used by the kernel to handle tasks such as keeping the database in sync with the application, handling licensing and user information. The system tables can be found in the AOT under the node *System Documentation/Tables*.

You can use system tables as any application table in your modifications. If using system tables, keep in mind that the data stored in a system table can be vital for the application. The system tables prefixed with sql*, stores information to keep the database in sync with the application. Changing data in these tables may cause the application to be unstable.

When updating system information such as users or license keys, the system tables are being updated by the application objects. However, kernel tasks maintain most of the system tables. There is one particular system table to take note of. The system table SysLastValue stores usage data. The table is frequently used in the application as it stores the last user settings for an object. You might not see SysLastValue declared from code, as the table is wrapped in the runbase framework or used by the class xSysLastValue.

For more information on usage data, see the chapter **Intro to MorphX**.

Common Table

The system table Common is the base for all tables. The table contains no data, and the table can be compared with a temporary application table. Common is used like the base type anytype. You can set any table equal to the table Common. This is useful if you need to transfer a table as a parameter and not knowing the exact table before execution time.

```
static void DataDic_Common(Args _args)
{
    Common      common;
    CustTable    custTable;
;

    common = custTable;

    if (common.tableId == tablenum(custTable))
    {
        while select common
        {
            info(common.(fieldnum(custTable, name)));
        }
    }
}
```

The example above is initializing Common with CustTable. A check is made to assure that Common is now equal to CustTable. All records from CustTable are looped, and the customer name is printed in the InfoLog. Notice, the system function fieldnum() is used to get the field name printed. If you do not know the fieldname at runtime, you can enter the field id in parentheses instead.

Util Tables

The system tables prefixed with Util* are in fact not real tables. You will not find any of these tables in the database. They are stored in the layer files, but appear in the AOT as system tables. The Util* tables contain information on any object in the AOT, even in the old layers if you have upgraded your application, and contain a huge amount of data. As such, you should not use these tables in complex select statements, as even joining two Util* tables will take a very long time. Also, when using the Util* tables remember to use an efficient index. You can check the available indexes by using the Application Hierarchy Tree.

The Util* tables are used for a lot of the tools in MorphX. If you are going to develop your own tools for MorphX you will find these tables useful. For more information on tools using the Util* tables, see **Appendix MorphX Development Tools**.

Fields

Both extended data types and base enums can be dragged to create a new field. If neither of these fulfills your needs, you should start creating new ones before creating your fields. If selecting and exiting extended data types or base enums, you should check whether a configuration key is defined in the property sheet, as your fields will not be viewable for the application users if the related configuration key is disabled. All fields should have an extended data type or a base type set in the properties. You can of course add fields manually to a table, but by making it a habit of dragging the types to be used, you will not forget to fill out the properties for extended data types or base enums. The benefit of using extended data types or base enums is that if changing property values such as field length or alignment, this will affect all fields using this extended data type or base enum. When an application user right-clicks a key field such as the customer account, and chooses *Rename* from the Record info form, the extended data type for customer account is used to locate all places in the application where that extended data type is used. This is a very powerful feature of Axapta. There are few applications in the market-place that provide application users with such flexible functionality.

Best practice states that all fields should start with lower case letters. As all extended data types and base enums begin with an upper case letter, you would have to manually change the fields when dragging to create. This is a time consuming rule which does not make much sense in the real world. You will not gain much when looking at the code in the editor as some fields are created using lower case and some are created with higher case.

When looking at the fields in the AOT, you might have noticed the icons in front of the fieldnames. The icon informs you about the base type of a field. Extended data types and base enums also used the icons. This is a very good aid when picking a field or a type when using the X++ editor.

Common Field Properties

In the previous MyTable example a table was created and fields were added. Some of the properties also needed to be set. The field AccountNum is defined as the key field for the MyTable. Key fields must always have the property **Mandatory** set to “Yes” as this will prevent a record being stored with a blank value in the key field. From the table browser or from a form you will easily spot mandatory fields as they are underlined with a red color. Normal practice in Axapta is that key fields must only be filled when inserting a new record. The rename option mentioned earlier must be used to change the value of a key field. The property **AllowEdit** is set to “No” and the property **AllowEditOnCreate** is defaulted to ‘Yes’ making it possible to enter a value when inserting a record.

The field labels that application users see come from the extended data type or the base enum. Unless you want to override the label, you should not specify any label in the property sheet for a field. If you often override the label of an extended data type, you should consider creating a new extended data type.

If you have fields in a table which have no use to the end-user, such as a container field described below, you should set the property value **Visible** to “No”. This will make the field invisible for the application users. Note that the field will not show up in the table browser.

Container Fields

Fields of the base type container are often use for storing binary files like bitmaps. You should be cautious of adding such fields to an existing table as this type of field is often large in comparison to other fields. The table CompanyInfo is an example of this; if adding a large bitmap for the company logo it may affect the performance of the application. The table CompanyInfo is widely used and as the normal practice is to select all fields from this table, the bitmap will always be loaded when fetching a CompanyInfo record. In this case you should either consider using small bitmaps or create a related table for your bitmap. The table CompanyImage is a generic table whose purpose is to store bitmaps. An alternative is to add your binary files to the Resources node in the AOT. For more information on resources, see chapter **Resources**.

System Fields

All tables have a set of system fields which are automatically added when creating a new table. The system fields are not listed in the AOT. You can get the list of system fields by checking the fields for the system table Common. The fields prefixed with modified* and created* contain no value unless you set the properties with the corresponding name at the table property sheet. By default those values are not logged as it will affect performance slightly.

RecId is the system field which is used as a unique id for a record. This is an integer counter shared by all tables. The system table SystemSequences keeps track of the counter. Each time a record is inserted in a table the counter is increased by one and the field recId is initialized with that value. The use of recId is one of the main reasons for using COM, in the event of inserting records to the Axapta database from an external

system. If these records are inserted directly to the database, the true status of recId value will not be realized.

For an explanation of the field dataAreaId, see the start of section **Tables**.

Field Groups

Fields visible for the application users must be part of at least one field group. You can use fields in MorphX which are not part of a field group. However, only by using field groups will you obtain the full benefit of MorphX. Field groups are widely used when creating forms and reports. Instead of adding fields one by one to a design, you should add *field groups*. If a field group is later changed, e.g, when adding a new field to a group, then the new field will automatically be available everywhere that specific field group is used.

The application users can view all fields from a table by right-clicking a form and selecting *Show all fields* from the Record info form. The fields are grouped by field groups. Fields which are not part of a field group are all shown in a default group. If all new fields are added to logically named field groups, the application user will get an more logical and easier overview of fields in that table.

Example 2: Adding field groups


Objects used from MORPHXIT DataDictionary project

- Table, MyTable

In the previous example the table MyTable was created and fields were added. Now field groups will be added to the previously created MyTable.

1. Locate the field group node *AutoReport*. Mark all 3 fields and drag the fields to the AutoReport node.
 2. Go to the field group node *AutoLookup*. Drag the fields accountNum and custName to the AutoLookup node.
 3. Create a new field group called “Identification” by right-clicking the *Field Groups* node and choose *New Group*. The name of the field group is set by using the property sheet. Add the field accountNum to the field group.
 4. Create a new field group called “Overview” and add all 4 fields to the field group.
 5. Finally, create a new field group called “Details” and add the fields custName, custGroupId and currencyCode.
 6. Save the table.
-

You might wonder why so many field groups are needed for a few fields: AutoReport and AutoLookup are reserved field groups. The other 3 field groups are used for grouping the fields on forms and reports.

When selecting the print icon  in a form, the fields of the AutoReport group are printed. If no fields are added, the report will be empty. This feature is referred to as auto reports and is used by the application user to print simple reports directly from forms. The common fields should be added to the AutoReport field group.

AutoLookup can be used to determine which fields to be shown when pressing the lookup button. If no fields are added to the AutoLookup, the table properties fields TitleField1 and TitleField2 will be used together with the first index field from each index. Form lookups are further explained in the chapter **Forms**.

It is recommended to always create the two field groups Identification and Overview. The field group Identification should contain all of the key fields for a table. If the table has a unique index, all fields from the unique index should also be added. Consider the field group Overview as a summarization of the table. This field group is used for the tab page called Overview which most forms have. All the common fields and at least all mandatory fields should be added to this group.

Aside from the Overview field group, all fields should be part of at least one other field group for making a logical grouping of all fields. The field groups Identification and Details are the logical grouping of the fields in MyTable. Before using the field groups on a form or a report, labels must be added. A label can be added to a field group using the property sheet for a field group.

Not only fields can be added to a field group. Display and edit methods can also be added. However the solution is not bulletproof as methods added to a field group is identified by a consecutively method number. If a method is added or deleted from the table, the methods will be re-numbered and you will have to check that the correct display and edit methods are attached to the field groups. How to use methods is described in section **Methods**.

Indexes

Any table must have at least one index. If a table has a unique index you will be able to fetch an exact record. This is however not always possible and not always the best solution. Transaction tables will often have several records with the same values making it impossible creating a unique index. As transaction tables are frequently addressed and contain a lot of records, having a unique index will impact performance. The point is the better index you have, the better your application will perform. Main tables and group tables like the customer table and customer group table should have unique indexes, whereas tables with a lot of records like transaction tables should not have a unique index. Be aware that creating many indexes on tables will impact performance, so create your indexes with care. Each time data is changed in a table, the database will update the indexes for that table.

Example 3: Adding indexesObjects used from MORPHXIT_DataDictionary project

- Table, MyTable

The next step is to add indexes to MyTable. In this example two indexes will be created.

1. Locate the node *Indexes* in MyTable. Right-click and choose *New Index* to create an index. Use the property sheet to name the index “AccountIdx”. Drag the field `accountNum` to the index.
 2. Go to the property sheet for AccountIdx and set the property **AllowDuplicates** to “No”.
 3. Create a new index named GroupIdx. Add the fields `custGroupId` and `accountNum` in this order.
 4. Save the table.
-

A unique index has been added to MyTable by setting the property AllowDuplicates to No. This will prevent entering two records with the same values for the fields which are part of the unique index. If a table contains data and you subsequently change an index of that table to unique, you will force an error if duplicate records exist for the index you are trying to set unique. You might not even have any data in the current company, but *none* of the companies must have duplicate records for the new index. It is possible to create more than one unique index for a table but this is not particularly effective database design. In additions, having several unique indexes will make the use of the table unnecessary complex.

Tables with a unique index should have the properties PrimaryIndex and ClusterIndex set. Only unique indexes can be selected as a primary index. The primary index is used as the default sorting when data is fetch from the table. Caching of the table will also use the primary index. A unique index is specified as a ClusterIndex for better performance. Clustered indexes contain both index and data, whereas a normal index contains only a sorted list of the index fields. When a record is fetch in a normal index the database must first lookup the index and subsequently find the record.

Relations

If you click the Transactions button in the customers form you will open the customer transaction form. Try selecting another customer in the customer from. Notice that the customer transaction form will automatically be refreshed with the customer transactions for the selected customer. This is one of the reasons for using relations. Relations will connect your tables and let MorphX know how the data model looks like. Using relations will also save you from writing many lines of code. The example shown below with customer transactions is made without writing any lines of code.

Example 4: Adding relationsObjects used from MORPHXIT DataDictionary project

- Table, CustTable

A relation to MyTable will be created in CustTable making it possible to relate a customer account to an account in MyTable.

1. Find the table CustTable and go to the *Fields* node. Add a new field using the extended data type AccountNum. Rename the field to “altCustAccountNum”.
 2. Go to the field group Delivery in CustTable and add the field altCustAccountNum.
 3. Locate the node Relations in CustTable. Right-click the node *Relations* and choose *New Relation*. Rename the new relation to “MyTable” using the property sheet.
 4. Right-click the new relation MyTable and choose Normal. Go to the property sheet and select altCustAccountNum as **Field** and an accountNum as **RelatedField**.
 5. Save the table.
-

A new field was created in CustTable and related to the table MyTable. The new field altCustAccountNum is part of the field group Delivery. Try opening the form CustTable and go to the tab page Setup. In the field group Delivery you will find the new field with the label Account number. Notice that the field has a lookup button. As a relation to MyTable was made in CustTable, a lookup button will automatically be added. You can now pick an account number from MyTable using the lookup button. If an account number which is not created in MyTable is entered in the new field in CustTable, an error will occur. A relation will, by default, secure that only values from the related field are valid. If needed, you can disable relation validation in the property sheet.

The relation described in CustTable was created as a normal relation using a field as the relation. Any number of fields can be added to define the relation. As MyTable only has one field in the primary index, only one field is needed. All of the fields part of a primary index must be added as normal relation fields. Besides a normal relation, two other relation types exist, field fixed and related field fixed. These are used to narrow the choices of a relation and are often used for filtering data depending on the value of an enum, or even defining which table to relate. Take for example the table LedgerJournalTrans which contains manually entered journal lines. Depending on the account type, when the user selects the Lookup button on the account field, a different table will be selected such as ledger, customer or vendor tables. If you check the relations LedgerTable, CustTable and VendTable in the table LedgerJournalTrans, you will notice the relations are created using two normal relation fields and one field fixed relation for the account type.

This example was using different tables. You could also use the fixed field relation and the related field relation for filtering data on the same table, as more than one relation can be created for the same table. Remember the relation fixed field relation and related field relation will be added to your expression.

The use of lookups is further explained in the chapter **Forms**.

An Extended data type can also have a relation to a table. However, if a table has relations on the same field, the table relation will overrule an extended data type relation. Having relations in two places might seem confusing when getting to know MorphX. Using the Visual MorphXplorer will be a help, as relations from both tables and extended data types will be shown in the diagrams.

Delete Actions

To assure data consistency delete actions are used. If you have created a table to enter information about customers. This table is related to the customer table. If you decide to delete a customer from the customer table, the data in your table will still remain, but you will no longer require the information. This will cause in-consistence in your data both now and in the future: For example, if your newly created table was for customer transactions and a new customer was created with the same id, the data in the related table would be visible again, but for another customer.

To prevent this situation, delete actions will either delete data in related tables if a customer is deleted, or prevent a customer from being deleted if related tables contain data for the customer. Typically, transactions for a customer will prevent the customer to be deleted, whereas information only relevant for the customer like personal data will be deleted when the customer is deleted.

Example 5: Adding delete actions

Objects used from MORPHXIT_DataDictionary project

- Table, CustTable

In this example delete actions will be added to both CustTable and MyTable to ensure data consistency.

1. Go to the node *DeleteActions* in MyTable. Right-click and choose *New DeleteAction*. Use the property sheet for the new delete action and set to the CustTable. The Delete action mode should also be set to Restricted.
 2. Now locate CustTable and add a new delete action to MyTable. Open the property sheet for the new delete action and choose MyTable. Set the delete action mode to Cascade.
 3. Save the table.
-

Open the form CustTable, create a new customer and select an alternative account number from MyTable for the new customer. Once you have completed this, from the table browser, try deleting the account number from MyTable that is related to the new customer, and you will get an error. The delete action set to “restricted” in MyTable will prevent deleting a record that is used in related tables. If you try deleting the new customer and then check MyTable in the table browser, you will notice that the account number related to the deleted customer is also deleted. Setting a delete action to cascade will delete related records.

Delete actions use relations to figure out whether to delete or prevent deleting related data. If no relation has been specified for a table used in a delete action, the delete action will have no affect.

```
static void DataDic_DeleteActions(Args _args)
{
    MyTable myTable;
;

    ttsbegin;
    select forupdate myTable
        where myTable.accountNum == "10";

    if (myTable.validateDelete())
        myTable.delete();
    ttscommit;
}
```

When deleting records using X++ some rules must be followed to have the delete actions validated. In this example a record is fetched from MyTable. Before the record is deleted a validation is made. If the record was just deleted without calling validateDelete(), the delete action would not be validated, which could cause inconsistent data.

A delete action can be set to a third mode called “Cascade + Restricted”. This delete action mode will act as restricted if used from the table browser or from a form. Deleting a record using X++, this mode will perform a cascade delete from the related table without calling validateDelete(). Reviewing the code example above, this would have been the same as deleting a record in CustTable from the customer form and subsequently causing related MyTable record to be deleted. Confused? The delete action mode Cascade + Restricted is hardly ever used, and there might be a good reason not doing so.

Methods

Methods are used for adding X++ code to your application. The code in methods is also referred to as business logic. When records are changed, inserted or deleted from a table different default methods are executed. You can add you own methods and have them executed from one of the default methods. When changing a default method you are overriding a method. To override a method go to the *Methods* node of a table, right-

click and choose *Override Method*. A list of the default methods will be shown. If you select the method `validateField()`, you will have the following code shown in the editor:

```
public boolean validateField(fieldId _fieldIdToCheck)
{
    boolean ret;

    ret = super(_fieldIdToCheck);

    return ret;
}
```

Note the `super()` call. This executes the code from the corresponding method in the parent class. Actually, when overriding a default table method you inherit a method from a system class. When overriding one of the default table methods you will be inheriting a method from the system class `XRecord`.

Table methods and classes are preferred for adding logic to the application. Having no business logic in the user interface will make it easier to reuse and to upgrade the code. You cannot avoid having code on forms and in reports, but you should aim to not modify data from code on forms or in reports. You should be able to replace your forms and reports with another user interface like a web interface without re-coding your business logic.

The use of methods has only been explained briefly here to get an understanding of the ability of methods. In the chapter **Classes**, methods are explained more detailed.

Common methods

As you modify your data by performing inserts, updates or deletes of records, the default methods are executed. If you create a new record from the table browser or a form the table method `initValue()` is executed. `initValue()` is used to set a default value for the fields.

```
public void initValue()
{
    super();

    this.custGroupId = "10";
}
```

Try overriding `initValue()` in `MyTable` and add the above line. Through the table browser, open `MyTable` and press `ctrl+n` to create a new record. The field `custGroupId` will now have the default value 10. The keyword *this* used in `initValue()` is used to refer to the current object. You cannot replace *this* with `MyTable`, as the compiler would expect `MyTable` to be a table variable declared in the method header. Using *this* prevents confusing the object reference with variables, and the notification is used in all parts of MorphX.

Note: To get an overview of which methods are executed, you can override the methods you want to check and add a line printing a text in the InfoLog in each overridden method. Alternatively, override a method and set a breakpoint at `super()`. When the breakpoint is executed the methods called can be seen in the stack trace window in the debugger.

Each time the value of a field is changed the method `modifiedField()` is called. The method is useful to initialize the values of other fields if the value of the current field is changed.

```
public void modifiedField(fieldId _fieldId)
{
    switch (_fieldId)
    {
        case fieldnum(MyTable, custGroupId) :
            this.custCurrencyCode = "";
            break;
        default :
            super(_fieldId);
    }
}
```

As shown in the example above, when the value of the field `custGroupId` is changed, the value of the field `custCurrencyCode` will be set to blank. `ModifiedField()` receives the field number of the active field as parameter. A switch statement is used to check which field is active. If none of the checked fields are active the `super()` call is executed instead. The switch case contains only a validation for one field, so an if-else statement could have been used instead. By using a switch statement it will be easier to later extend the code with additional checks.

This method was first introduced in version 3.0. Until then, modified field checks had to be done from forms. This caused a lot of logic to be put in forms, which ought to be at the table level. For this reason, you will still see modified field checks used widely in forms. The ideal solution is to have all generic checks at table level. However, if the logic is specific for a single form, it is sometimes necessary to put the check on a form.

```
print this.orig().custCurrencyCode;
```

A nice feature when a field value is modified, is that you can re-call the value before the field was modified. The field value retained is the last committed value. The method `orig()` is used to get the stored value. `Orig()` will return an instance of the current table. A single field value from `orig()` is retained by specifying the field. When the record is committed `orig()` will be updated.

The method `validateField()` is similar to `modifiedField()`. Both methods receive the active field number as parameter. Where `modifiedField()` is used to initialize the value of other fields and does not return an value, `validateField()` is used for validation only and will return true or false. If `validateField()` return the value false, the application user will be prevented to continue changing a field value. It is important to use the respective methods for initialization and validation as this makes coding easier.

```

public boolean validateField(fieldId _fieldIdToCheck)
{
    boolean ret;

    ret = super(_fieldIdToCheck);

    if (ret)
    {
        switch (_fieldIdToCheck)
        {
            case fieldnum(MyTable, custName) :
                if (strlen(this.custName) <= 3)
                    ret = checkFailed("Customer name must be longer than 3 characters.");
        }
    }

    return ret;
}

```

The `super()` call will check whether a valid value has been entered if the field has a relation. A mandatory field will also be checked. In the example above, if the checks in `super()` are true, the validation for the field `custName` will be checked. In this case a warning will appear in the Infolog if the length of a customer name is less than or equal to 3 characters. The global method `checkFailed()` is normally used for adding validation messages to the Infolog.

Saving a record will cause the method `validateWrite()` to be executed. Where `validateField()` will check a field entered by the application user, `validateWrite()` will just check mandatory fields. Checks made by `validateWrite()` are the same as the `super()` call in `validateField()`. If a field is not mandatory, the application user may not enter all fields before saving a record. So if your condition is not related to the value an application user enters in a specific field, you should put the validation in `validateWrite()`. Syntax of `validateWrite()` is similar to `validateField()`. After `super()` you should check the return value before processing your validations.

If `validateWrite()` is true, `insert()` or `update()` will be executed. If the record has not been saved previously `insert()` is called. `Update()` is called if the system field `recId` has a value and thereby the record has been save previously. `Insert()` and `update()` are rarely overridden. At this point you should have done your validations and set the value of your fields. If you need to override `insert()` or `update()` you might be on the wrong track. However, if you need to ensure a field has a certain value upon inserting a record, you can initialize your field before calling `super()` in `insert()`. Some special cases might also require overriding these methods; for example, if you need to synchronize the content of a saved record to another table. Examples on synchronization can be found in the tables `CustTable` and `EmplTable`. This is not a recommend solution, but it can be the best choice among your options.

Deleting a record has a similar execution of methods. When deleting a record the method `validateDelete()` is first executed. If true, the method `delete()` will be called. The same rule for inserts and updates goes for deletion. You should, therefore, execute your

validations before delete() is called. ValidateDelete() will also check whether a delete action actually allows the deletion of a record.

Using X++ for entering data requires a bit more than using the user interface like forms. Only the table methods called will be executed. If a validation method is not called, no validation will be performed upon update, insert or delete. When inserting and updating, the only check done is checking unique indexes.

```
static void DataDic_InsertRecord(Args _args)
{
    MyTable    myTable;
;

    ttsbegin;
    myTable.initValue();
    myTable.accountNum          = "100";
    myTable.custName            = "Alt. customer id 100";
    myTable.custCurrencyCode    = "USD";

    if (myTable.validateWrite())
        myTable.insert();
    ttscommit;
}
```

The above example shows how to use the table methods to insert a record in MyTable. InitValue() is called and will set the value of the field custGroupId. The record will only be inserted if validateWrite() is true. As all mandatory fields have a value, the record will be inserted.

Instead of calling insert() you could call write(). This will update an existing record, but if the record does not exist, a new record will be inserted.

Calling the validation methods will cost performance and you might – depending on your case, choose to validate your data via other methods. You can also skip the code on insert(), update() and delete() by using the corresponding table methods prefixed do*. These methods cannot be overridden and can only be called from X++. If you do skip all logic on the table methods to optimize your code, you must assure that your data will not be corrupted.

The select keywords delete_from, insert_recordset and update_recordset make only one call to the database from the client when processing multiple records. Overriding insert, update and delete will affect these performances enhanced select keywords and causing insert, update or delete per record instead.

You will find the methods find() and exist() on most tables. These are not overridden methods. The methods are usually created to fetch a single record using a unique index. It is recommend adding these two methods when creating a new table as sooner or later you will need these methods. All fields which are included in the unique index should be added as parameters. Find() is used to fetch a single record from a table. This could be for setting a value of a field in the table method initValue(). Exist() is used to check

whether a record matching the index fields exist in the table. A check like that is often used before trying to add a new record.

You will find many examples on these methods by browsing the tables in the AOT. Methods used for similar operations, but which do not have a unique index should not just be called `find()` or `exist()`. Consider these name as reserved and use a suffix if the method is not using an unique index like `findVoucherDate()`.

Transaction Tracking System

As the name says the Transaction Tracking System normally referred to as TTS is used for tracking transactions. The idea is to secure the entire transaction to be committed to the database. This is vital for a relational database and even more for an ERP system. Whilst performing an invoice posting you must ensure that this does not result in corrupt data if the system crashes. TTS secures that all database operations within a TTS loop are committed entirely or not.

Every time you are performing an insert, update or delete you should use the TTS. In fact you cannot update a record without doing it in a TTS loop. A TTS loop is initialized by using the keyword *ttsbegin*. The keyword *ttscommit* will write all data operations within the TTS loop to the database. A loop can be aborted by using the keyword *ttsabort*, which will roll back the database to the starting point. If the system crashes, a *ttsabort* will automatically be invoked by the database. You can see an example on the use of TTS from the previous example inserting a record in MyTable. This does not mean that every time you have a database operation, you should put your code in a TTS loop. If your code is called by another method which has already started a TTS loop, you might not need to start another TTS loop. Starting another TTS loop within a TTS loop will add another level. Each level must be committed with a *ttscommit*. Be aware that having nested TTS loops and you do not ensure equal *ttsbegin* and *ttscommit* statements you will have an error from TTS. This is nearly always caused by a missing *ttscommit*. You can execute a job calling *ttscommit* to fix the TTS error. You will of course have to solve the reason for the TTS unbalance, but the alternative is to restart the Axapta client.

```
static void DataDic_UpdateRecord(Args _args)
{
    MyTable    myTable;
;

    ttsbegin;
    select forupdate firstonly myTable;
    myTable.custName = "Customer updated";

    if (myTable.validateWrite())
        myTable.update();
    ttscommit;
}
```

If you need to update a record you must always fetch the record using the keyword *forupdate* in the TTS loop. A common mistake made when selecting *forupdate* is to fetch your data for update before calling *ttsbegin*. If you do this an error will occur. In the example above, the first record in *MyTable* is fetched and the field *custName* is changed before updating via the *ttscommit*.

3.2 Maps

In Axapta several of the features in the Account Payable module and the Account Receivable module are very similar, such as the main tables with either customers or vendors, transactions tables and setup tables. Features like journals for manually entering vouchers also exist in several of the modules. With similar features the business logic is also similar, therefore making it possible to reuse much of the code. However, although the tables and their fields may appear similar in their construction, the names used are likely to be quite different. This is where table maps are used. Maps are often referred to as table maps to differentiate from the foundation class map.

The most common map in MorphX is *AddressMap*. Address information is used in several tables and validation of address information like zip code will not differ whether entered for an employee or a customer. Mapping all tables using addresses to the same map will make it easy to reuse the code of a map as you will not have to worry about the naming of fieldnames in a specific table.

Maps have similar nodes and properties as a table. In maps, typically only fields are created and mapped, as properties are already specified at the mapped table. You can use a map as any table from X++ or objects like forms or reports. Only if using a map from a form or a report, you should consider creating field groups for your map.

Example 6: Creating a table map

Objects used from MORPHXIT_DataDictionary project

- Map, MyMap

A map will be created, mapping the fields in common for the tables *CustTable* and *MyTable*.

1. Locate the node *Maps*, right-click and select *New Map*. Rename the map to “MyMap” using the property sheet.
2. Drag the extended data types *AccountNum*, *CustName*, *CustGroupId* and *CustCurrencyCode* to the *Fields* node of *MyMap*.
3. Save *MyMap*, right-click *MyMap* and select *Restore*.
4. Go to the node *Mappings*, right-click and choose *New Mapping*. Open the property sheet for the new mapping and select the table *CustTable*. A line for each of the

fields in MyMap is now added to the new mapping. Use the property sheet for each of the mapping fields to specify the related field in the property **MapFieldTo**. The related fields from CustTable are accountNum, name, currency and custGroup.

5. Repeat step 5 by adding a map for the table MyTable.
6. Go to the node *Methods* and create the following method:

```
void listRecords(MyMap _myMap)
{
;
while select _myMap
{
info(strFmt("%1, %2", _myMap.accountNum, _myMap.custName));
}
}
```

7. Save the map.
-

In the example, the map was restored after the fields were added. Each time a change is made to the fields in map, you will have to restore the map otherwise changes will not be shown in mappings before restarting the client.

A method was added to the map which is going to be used to test the map. This is a simple method printing a value from the current record the map is initialized with. Try creating a job with the following content:

```
static void DataDic_TestMyMap(Args _args)
{
MyMap myMap;
CustTable custTable;
MyTable myTable;
;
myMap.listRecords(myTable);
}
```

Notice that a map is declared just as a table. In fact only the name shows a map is used. It is recommend suffixing the name of the map with *Map making it easier to read the code. The method on the map is called with MyTable as parameters. This will result in all records of MyTable to be printed in the Infolog. A map contains no records and can be considered as a specialized Common table. The map can be initialized with any of the tables declared in the map.

The methods of a map are similar to those of a table. You can initialize fields, check modified fields and validate before saving or deleting. Calling a default method like insert() on a map will execute the corresponding insert() on the mapped table. You can use this for mapping your own methods. However, you must make sure the name of the method is the same for all mapped tables. For an example on this, take a look at the map method CustVendTrans.existInvoice(). This method calls the corresponding existInvoice() method for the mapped table, thus making it possible to have different validations for different tables in the map.

3.3 Views

In a relational database, data is divided into numerous tables to prevent redundant data and for better performance. This is all very good to manage your data, but extracting data for reports becomes more complex as you will often have to fetch data from several tables. To make extraction of data easier, views can be used. A view is the result of an inner join of two or more tables. Views are read only and support aggregated functions on the fetched data. You can use views as an alternative to data sources in a report. However providing data for OLAP cubes is the main purpose with views. Views are synchronized to the database. This makes views useful if you need to read data from an Axapta table using external tools as you can fetch data directly from the database instead of using the COM interface.

OLAP cubes in Axapta are populated using Microsoft SQL Server Analysis Services. Describing the use of OLAP is out of the scope of this book. You can find more information about OLAP in Axapta by checking the manuals in the standard package.

Example 7: Creating a view

Objects used from MORPHXIT_DataDictionary project

- View, MyView

In this example a view to sum customer transactions and print customer information will be created.

1. Go to the node *Views*, right-click and select *New View*. Rename the view to "Myview" using the property sheet.
2. Locate the node *Metadata/Data Sources*, right-click and select *New Data Source*. Go the property sheet for the new data source and pick the table CustTable using the property **Table**.
3. Expand the data source CustTable and go to the node *CustTable/Data Sources*. Add an additional data source by right-clicking. Select CustTrans as the table for the data source.
4. Open the property sheet for CustTrans and set the property **Relations** to Yes. Expand the node *Relations* for the CustTrans data source and check that a relation has been added.
5. Right-click the node *Metadata* and choose *Open New Window*. This will open a new window with the sub tree metadata from MyView. Drill down the new window to *CustTable/Fields*. Select the fields AccountNum, Name and CustGroup and drag the selected fields to the other window at the node *Fields* at the first level of the view.

6. Repeat step 5 by dragging the field AmountMST from *CustTrans/Fields*. Open the property sheet for the new view field and set the property **Aggregation** to Sum.
 7. Save the new view.
-

When creating a view to fetch data from a table, the view will be defined in the same way as using a standard Axapta query. Fields to be used in the view must be selected from the chosen tables. In this example a single field from the customer transaction table was used for an aggregated function. When using aggregate functions, the records in the related table will be fetched group by. The example showed customer transactions were sum grouped by customer. If the field TransDate from CustTrans was added without any aggregation, the field AmountMST would have been calculated and grouped by customer and transaction's date instead. Notice that aggregated view fieldnames are automatically prefixed with the aggregation function name. You can view the result of MyView by opening the table browser. All customers with transaction are listed and the balance is summed using the field AmountMST from CustTrans.

Views can be used as any table with a few limitations. In relations, delete actions and tables collections views cannot be used. You can use a view as an alternative to a table from X++ or from a data source. If creating a report you could use the view without having to figure out how to do calculations on your report. This would however give the application users some limitations as the application user can normally at runtime make their own settings for calculations.

3.4 Extended Data Types

Extended data types are a central part of MorphX. Whether you are adding fields to a table or declaring variables from X++, extended data types should always be used instead of using base types. The main reason for this is that your modifications will be easier to maintain and you can ensure that a field is presented in the same way, no matter where it is used in the application.

An extended data type is extended from a base type or another extended data type. You can have as many levels as needed. The difference between a base type and an extended data type is that an extended data type has a property sheet where information such as labels, length and left or right adjustment are stored. Relations can also be added to an extended data type.

Before creating a new extended data type you should check whether an existing one will fulfill your needs. If you are creating a new table and, within that table, want to lookup item ids from the table InventTable, you should use the extended data type ItemId. By using ItemId you would not have to specify labels or a relation to InventTable as this is all defined at the extended data type.

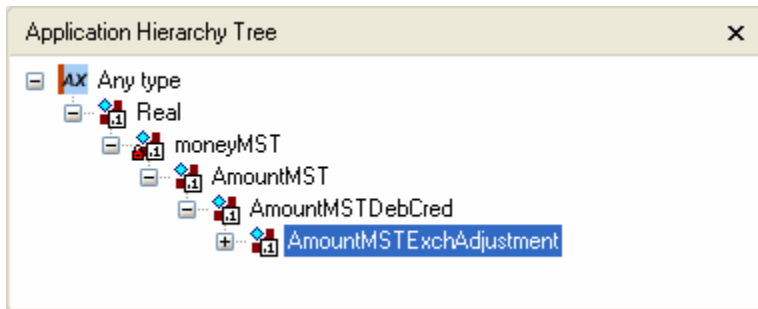


Figure 15: Extended data type lookup in Application Hierarchy Tree

Deciding which extended data type to use is not always an easy task, especially if an extended data type has been extended several times. The Application Hierarchy Tree can be used to get an overview. You can call the Application Hierarchy Tree from the Add-Ins menu by right-clicking an extended data type. Only the base type and the extended levels for the selected extended data type will be shown.

Some extended data types are extended from system extended types. System extended data types are located in the AOT under *System Documentation/Types*. Regional settings like amount and types for system fields are all created as system fields.

Example 8: Creating an extended data type

Objects used from MORPHXIT_DataDictionary project

- Extended Data Type, DataDic_AltCustAccount

An extended data type extending from another extended data type will be created. The new extended data type will make the previously created table relation in CustTable unnecessary.

1. Go to the node *Extended Data Types*, right-click and create a new string. Name the new extended data type "DataDic_AltCustAccount" using the property sheet.
2. Enter the label "Alt. customer" and the help text "Identification for alternative customer account." for the new extended data type.
3. Extend DataDic_AltCustAccount from AccountNum using the property **Extends**.
4. Expand the nodes for DataDic_AltCustAccount and go to the node *Relations*. Add a new relation by right-clicking, choose *New* and select *Normal*. Use the property sheet to select MyTable and the field accountNum as the relation.
5. Save the extended data type and wait until the database has been synchronized.
6. Go to the table CustTable and locate the field altCustAccountNum. Open the property sheet and change the **ExtendedDataType** property to DataDic_AltCustAccount.

7. Locate the relations in CustTable and find the relation MyTable. Press the delete key to remove the relation for MyTable.
 8. Save CustTable.
 9. Change the extended data type for field accountNum in MyTable as done at step 6 and save MyTable.
-

You now have a new extended data type for the alternative customer account. The table relation deleted in CustTable was unnecessary as the extended data type just created will now handle the relation. It is much more flexible having the relations at the extended data type level as you will not have to modify each table where the extended data type is to be used. The extended data type was also changed for the field accountNum in MyTable. No lookup button will be added to accountNum as the related table is the same. This is also called a self relation.

Label and help text was specified for the new extended data type. If no labels were entered for the extended data type, labels from AccountNum would have been used. If you have found an extended data type which suits your needs, but another label or help text is required, you should create a new extended data type. Do not use an existing extended data type and override the label of the extended data type by specifying a label or help text at the field properties. If referring to a field by using X++, the extended data type is used and information about the field label will not be available. This is all explained further when explaining display and edit methods in the chapter **Forms**.

Note: The entire database will be synchronized when saving an extended data type extended from another extended data type. If creating a lot of extended data types press ctrl+break to stop synchronization and synchronize when saving the last extended data type.

Both extended data types and base enums can be used as extension for a field. So why create an extended data type for a base enum? Only extended data types can be used when adding a field to dialog. If a field used in a dialog is of the type enum like NoYes, then the extended data type NoYesId extending the base enum NoYes must be used. A dialog is created using classes, and is described in the chapter **Classes**.

Extended data type array

Defining an extended data type as an array, is one of the powerful features of extended data types. Each array element of an extended data type will be created as a database field. Both from the AOT and from X++ a field based on an extended data type array will look like, and be addressed as, any other field.

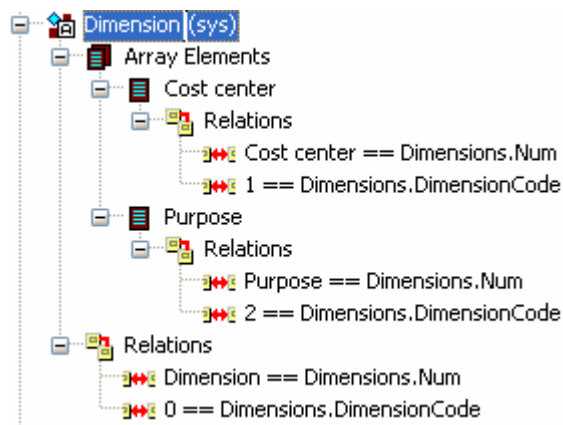


Figure 16: The extended data type Dimension

The node *Array Elements* is used when defining an extended data type as an array. The first entry of the array will be the entry created when creating a standard extended data type. The following array entries are created under the node *Array Elements*. Only label and help text is specified for these subsequent array entries. All other properties are inherited from the first entry in the array. As with the first entry, separate relations can be defined for each array entry.

The most common extended data type using this feature is *Dimension* which is used throughout the application for grouping data. *Dimension* consists of 3 array entries, each having their own relation. If you need another dimension for your application you will only have to add an array entry to the *Dimension* extended data type. All objects like tables, forms and reports will have the new dimension shown without having to add any line of code.

```

static void DataDic_EDTArray(Args _args)
{
    CustTable    custTable;
;

    select firstly custTable;

    info(strfmt("%1, %2, %3", custTable.dimension[1],
                    custTable.dimension[2],
                    custTable.dimension[3]));
}
  
```

The same notification for declaring a variable is used when addressing the single array entries of an extended data type. In the example above, the 3 dimensions from *CustTable* are printed for the first record. When looking up the fields for *CustTable*, only the field *Dimension* will be listed for the array. You must manually specify the array entry to be printed. If array entries are referred directly, such as in this example, a new array entry added will not be printed. A better solution would have been to loop all array entries rather than fix the code to print the first 3 entries.

If looking at a data source in a query, the opposite will be shown. A data source will show a field for each array entry. *Dimension* is always added to a field group when used

in forms and reports so the dimension field group will automatically recognize a new array entry added.

3.5 Base Enums

For categorizing data you have two options. Create a related table, such as item groups which is used for grouping inventory items. However, if you only need a fixed number of categories or the application users are not able to define the categories, you can use a base enum. Item type, as used for defining the type for inventory items, is a good example.

A base enum can have a maximum of 255 entries. A few base enums have a lot of entries such as the base enum `LedgerTransTxt`. However, most enums have only a few entries. The value of a base enum is stored in the database as an integer value. Entry values starts by default from zero and are consecutively numbered. The property **EnumValue** will show you the number stored in the database for an entry. If adding a new entry to a base enum in the standard package, you should make a break in the numbering, specifying a higher value in the property `EnumValue` for your new entry. The base enum `NumberseqModule` uses this practice. An entry made to a base enum in the SYS layer will be created in the current layer. To prevent having the number taken for your new entry if upgrading your application, you should skip some numbers. From X++, base enum names are always used. When using base enums in relations, the integer value is used instead. If you are changing the number of an entry during an upgrade, you will also have to change the relations using the enum. Changing the number of an entry will set the base enum property **UseEnumValues** to Yes. Changing this property to No will result in renumbering all entries starting from zero and numbered consecutively.

```
static void DataDic_BaseEnum(Args _args)
{
    InventTable inventTable;
;

    while select inventTable
        where inventTable.itemType == ItemType::BOM
            || inventTable.itemType == ItemType::Service
    {
        info(inventTable.itemId);
    }
}
```

To refer to a base enums entry, the name of the base enum is entered followed by a double colon. By pressing the second colon lookup will show the available entries. Here all items of the invent types BOM and Service are fetched. You could use the base enum entry numbers instead, but it would make your code more difficult to read and maintain.

```
while select inventTable  
  where inventTable.itemType >= ItemType::BOM
```

Instead of checking whether the field `itemType` is equal to one of the enum entries specified, 'greater than' or 'equal to' the item type `BOM` would have fetched the same records. However, this is not recommended as it makes the code more difficult to maintain. If a new entry is created for the base enum `ItemType`, the new entry would *also* have been fetch, which may not have been the intention.

Note: The first entry of a base enum normally has the value zero - which will return false, if the first entry is validated in an if-expression. This is also the reason why fields of the type enum should not be mandatory as the first entry of an enum would be considered as not valid.

System enums are located in the AOT under *System Documentation/Enums*. These are all enums used for selections in property sheets. If you need to address properties using X++ you should consider checking the system enums. As an example, the system enum `TableGroup` represent the entries of the table property `TableGroup`.

3.6 Feature Keys

Prior to version 3.0 of Axapta, feature keys were used to configure user restrictions and to determine which tables to be synchronized to the database and handling licenses. This was not without issue when configuring the Axapta system and could take quite some. By version 3.0 security settings are handled by using configuration and security keys and licenses by licenses keys. Feature keys cannot be used in version 3.0. The reason for having the feature keys in version 3.0 is for backward compatibility. If upgrading from version 2.5 you will be able to see you features keys making it easier to manually replace the feature keys with configuration keys and security keys.

3.7 Licenses Codes

When purchasing Axapta you will have to decide on system settings such as number of users, number of servers, access to MorphX and X++. Which application modules you are going to use must also be decided. For each system setting and module you will receive a license code. All license codes will be compiled in a code letter. These license codes are used for controlling which part of Axapta you will have access to. Only modules with a valid license code will be available in the main menu. Trying to execute an object without a valid license code from AOT will result in an error.

Partners usually have a code letter with license to all features in Axapta. A customer will buy part of the features fitting for the customers business. When creating modifications for a customer you can easily have a different setup. This can cause errors as your modifications might only be working in your application as you have access to all features. Before shipping your modifications to a live system, you should consider testing your modifications with the same features enabled as in the live application.

You can create new license codes and attach the license codes to your modifications. However to use your own license codes you will have to contact Microsoft as they will have to generate license codes on your behalf. Creating new license codes is used by companies creating modules for the GLS layer or by partners who want to sell their own modules.

Configuration keys have a property called **LicenseCode**, which is used for attaching a license code to your modifications. You do not need license codes for setting user permissions. This is done using security keys. License codes are solely used if you want a license code like most software packages have.

3.8 Configuration Keys

You have two levels of security settings in Axapta. Configuration keys are the highest level, and security keys are the second level. If a configuration key is disabled, the related objects will not show up in menus, forms or reports and no one will have access to those related objects. Configuration keys are defined in a tree hierarchy where the top configuration key is related to a license code. The form SysConfiguration shows the hierarchy of configuration keys. Only configuration keys where the related license code has been entered can be enabled and the top level configuration key can only be disabled by removing the license code. If you have all license codes entered not all configuration keys are enabled by default. Some configuration keys are, by default, disabled. This goes for advanced features and country specific features.

When changing the settings of configuration keys, the database must be synchronized. Configuration keys determine whether a table should be synchronized to the database or not. You should be careful when changing configuration keys as data will be lost for tables where the configuration keys are disabled.

Only a few configuration keys should be created for each module. You might have a main configuration key for a module and sub configuration key for each sub module. Normal practice would be to attach a configuration key to each table, map, view and menu item for the modifications. This will ensure that an object where the configuration key is disabled will not show up in the menu. In addition, an error will occur if a user tries to activate an object from the AOT as there will not be access to the database for that object.

Most of the objects in the AOT have a property for defining a configuration key. You might have seen that some of the extended data types and base enums in the standard package have a related configuration key. Having configuration keys at all levels would be difficult to maintain. However, it makes sense adding configuration keys to some extended data types and base enums. In the table SalesTable there is a field called ProjId. The extended data type for ProjId has a configuration key. If you have a valid license to the modules Account Receivable and Project you will be able to enter a

project id for a sales order. If you only have a license to Account Receivable, you will not have this option as the field ProjId will not show up.

Adding a configuration key to an entry of base enum is used on occasion in the standard package. Take a look at base enum FormTextType. Each of the entries added to this base enum from the GLS layer have configuration keys. If you do not have the related modules enabled it will not make sense to select the base enum entries. You can use this for your own modifications as well. If you want to prevent the application users from selecting an entry from a base enum, you can create your own configuration key and attach it to a base enum entry.

```
static void DataDic_ConfigurationKey(Args _args)
{
    SalesTable salesTable;
    ProjTable projTable;
;

    select firstly salesTable;

    info(salesTable.salesId);
    info(salesTable.custAccount);

    if (isConfigurationKeyEnabled(configurationkeynum(ProjBasic)))
    {
        info(ProjTable::find(salesTable.projId).name);
    }
}
```

From X++ you can make a check for whether a configuration key is enabled. The global method `isConfigurationKeyEnabled()` is used to validate configuration keys. There is no reason for executing code for a feature which is disabled. In the above example, the name of the project related to a sales order will only be printed if the top level configuration key for the project module is enabled. If the check was left out, a blank value would have been printed. But what if the code was to update or delete a record? Having left out the check it could have modified the wrong record.

Note: Several tables and fields are prefixed with `DEL_`. These are objects not used anymore and will remain until the release of the next version. The tables and fields are only renamed as if they were deleted, data from the previous version would have been lost during an upgrade. All are having the configuration key `SysDeletedObjects30` set. When you have completed an upgrade you can disable this configuration key.

3.9 Security Keys

Where configuration keys are setting access for all users, security keys will set access for a group of users or per user. You would normally set security keys for a group of users as it is far too complex maintaining settings per user. Security keys are like configuration keys linked hierarchically. The standard package uses 9 security keys for each module. One top level security key related to a configuration key, one security key

for tables and the last 7 security keys are used for the groupings of the module objects as seen from the main menu. You can view the security hierarchy from the form SysUserGroupSecurity.

Configuration keys can be related to any security key. However you should only define a configuration key for the top level security keys. If a configuration key is disabled, the security key will not be available and disabling a top level security key will disable all security keys at a lower level.

Security keys must be added to all tables, maps views and menu items. If one of these objects does not have a security key you will not be able to set access rights for the object, and the object will be accessible for all users. Configuration keys can be left out if your modifications are not to be used for a distributed module as modifications created for a single installation would normally not be disabled.

Security keys are not only defined as enabled or disabled when setting up user restrictions. Instead an access level is defined. The choices of access level are no access, view, edit, add or delete. These access levels are set at the data dictionary and the menu items. At tables, maps and views you use the property **MaxAccessMode** to define the access level a user can retain. Tables will, by default, have MaxAccessMode set to Delete, which will allow the users to retain full access to a table. If a table is used for transactions, you should set MaxAccessMode to "View" as transactions are not intended to be modified.

Menu Items has a similar property called **NeededAccesLevel** which acts just the opposite of table as you must specify the required access level to execute the menu item. The default value for NeededAccessLevel is View. You should only change the needed access level for menu items of the type action. Many of the action menu items perform operations which may not be executed by the daily application user. Or at least a menu item requiring a higher access level will be considered twice before just enabling.

Adding security keys should be one of your last tasks when creating your modifications. You should at least have the security keys added before doing your final test. You will of course not be able to verify all combinations for the security keys. But you should try to set up security keys for an application user to give you an idea of how the application user will experience your modifications.

3.10 Table Collections

If using more than one company it will be useful to share data from tables with general information. In the standard package this is done with tables storing data like zip codes and country codes. To share data from a table among all companies set the table property **SaveDataPerCompany** to No. This will merge data for the table and make the data accessible from all companies. In practice, the kernel will delete the system field dataAreaId for the table.

Table collections can be used to share a table for a part of companies. The benefit is that you will not have to change any settings on the existing tables. A new table collection is created by creating a new table collection node and dragging the tables to be share to the new table collection. A table collection is just a template for tables to be shared by any number of companies. Defining which companies to share the tables of a table collection, is defined from the main menu.

The form SysDataAreaVirtual is used to define virtual companies. Table collections are shared using a virtual company. You cannot switch to a virtual company like any normal company. Virtual Company is just the term used to for sharing a table collection among a set for companies. In the form you pick which companies to share specific table collections. Before creating a virtual company you should export data for the tables used in the table collection, as existing data will be deleted from these tables when added to a virtual company.

When using table collection for setup tables such as customer groups, setting up a virtual company will be easy. If you are going to share data from main tables like the inventory table, you should do some more investigation as you cannot only share the table InventTable. You must include all tables which are related to InventTable.

Sharing data should be used with precaution. Before commencing the sharing of data, you should carefully try out your settings in a test environment, either using the property SaveDataPerCompany or using table collections. You should make sure that your modifications will not cause errors on any objects using the tables to be shared.

3.11 Special Table Use

All the basic steps using the data dictionary have been explained throughout this chapter. This section will show examples on how to use some of the advanced features in MorphX in relation to tables.

Using System Classes

In the AOT under *System Documentation/Classes* you will find some classes prefixed with Dict*. These are classes which can be used to address any data dictionary node or property sheet. If writing generic code where you are not aware of the called table until runtime, you will be able to use the system classes. Your case might be to loop the fields of a table, build a lookup for the user to pick a field.

Two example of using system classes for the data dictionary are shown here. The first example will loop all the fields of a table. In the second example methods of a table will be looped.

Elements used from MORPHXIT_DataDictionary project

- Job, DataDic_SystemClassesFields

➤ Job, DataDic_SystemClassesMethods

```

static void DataDic_SystemClassesFields(Args _args)
{
    SysDictTable    dictTable;
    DictField       dictField;
    Counter         counter;
;

    dictTable = new SysDictTable(tableNum(MyTable));

    for (counter=1; counter<=dictTable.fieldCnt(); counter++)
    {
        dictField = new DictField(dictTable.id(), dictTable.fieldCnt2Id(counter));

        if (dictField.isSystem())
            info(strfmt("System field: %1", dictField.label()));
        else
            info(strfmt("User field: %1", dictField.label()));
    }
}

```

Fields from the table MyTable are printed. A text specifies whether the field is a system field or a normal.

The class SysDictTable is used to initialize the table. With this class you can get a handle to any table properties or child nodes. Notice that SysDictTable is an application class inherited for the system classes DictTable. Several system classes are inherited as application classes, and if so you should consider using those instead of the base classes as the inherited class will have additional validations.

```

static void DataDic_SystemClassesMethods(Args _args)
{
    SysDictTable    dictTable;
    MethodInfo      methodInfo;
    Counter         counter;
    CustTable       custTable;
;

    select firstly custTable;

    dictTable = new SysDictTable(tableNum(CustTable));

    for (counter=1; counter<=dictTable.objectMethodCnt(); counter++)
    {
        methodInfo = dictTable.objectMethodObject(counter);

        if (methodInfo.returnType() == Types::UserType)
        {
            info(strfmt("Method: %1, return value: %2",
                        methodInfo.name(),
                        dictTable.callObject(methodInfo.name(), custTable)));
        }
    }
}

```

}

This example shows how to get a handle to the methods of a table and have certain methods executed. This could also be achieved by calling the methods directly from the CustTable variable, but what if the table was not known till runtime and the table variable was the system table Common.

The methods of CustTable are looped. The first record of CustTable is fetched to have a cursor for the called methods. If a method is returning an extended data type or base enum which the kernel refers to as user types, the method is executed and the returned value from the method is printed together with the method name.

External databases

Integration with external systems where data has to be transferred is never an easy task. The problem is, that modification for integrating an external system is often made by two parts. To secure that everything works well, a common platform for the integration must be chosen. One option is using the business connector, also referred to as COM from the external system as you can call any business logic within Axapta. Consider the business connector as an Axapta client without user interface. This will require a connection always open or at least always accessible.

Another option is using an external database for storing data to be exchanged. Each system will then access the shared database. From Axapta, a batch job could be configured to schedule processing data in the external database. In this way you will be able to integrate two systems without having to require skills on how to communicate directly with the external system.

Elements used from MORPHXIT_DataDictionary project

- Job, DataDic_ExternalDatabase

```
static void DataDic_ExternalDatabase(Args _args)
{
    LoginProperty      loginProperty;
    ODBCConnection     odbccConnection;
    Statement           statement;
    ResultSet           resultSet;
    ResultSetMetaData   resultSetMetaData;
    Counter             counter;
;
    loginProperty = new LoginProperty();
    loginProperty.setDatabase("Northwind");
    loginProperty.setDSN("AX30SP4"); // Datasource name for the Axapta database
    loginProperty.setUsername("sa"); // Database login name
    loginProperty.setPassword(""); // Database password

    odbccConnection = new ODBCConnection(loginProperty);

    statement = odbccConnection.createStatement();
```

```
resultSet = statement.executeQuery("select * from Employees");
resultSet.next();
resultSetMetaData = resultSet.getMetaData();

for (counter=1; counter <= resultSetMetaData.getColumnCount(); counter++)
{
    switch (resultSetMetaData.getColumnType(counter))
    {
        case 0,1 :
            info(resultSet.getString(counter));
            break;
        case 3 :
            info(date2StrUsr(resultSet.getDate(counter)));
            break;
    }
}
```

This example will only function with Microsoft SQL Server. The demo database "Northwind" which is installed by default when installing Microsoft SQL Server is used. You will have to specify the data source, login name and password for your Microsoft SQL Server database. The first record from the table Employees from the Northwind database will be read and each field will be printed to the InfoLog. If you want to process all records from the table this can be achieved by adding a WHILE loop to resultSet.next(). First the connection to the external database is established, and then the table is fetched. Before printed, a field from the external table the type of the field is checked, as the type must be known before being able to choose the right method converting the field value.

3.12 Summary

Throughout this chapter designing changes to the data dictionary have been explained. You should by now have learned how to arrange data in tables for optimal performance and to use extended data types to make modifications easier to maintain. Using relations and delete actions to secure consistency of your data and how to use configuration and security keys to make it possible to set user permissions. The table browser has been used for checking the data entered to the database. This is a tool normally not available for the application users. In the following chapters you will learn how to create the user interface for the data dictionary.

4 Macros

In Axapta's predecessor, macros were widely used. The predecessor did not support classes, so macros were used instead. This might be the reason that macros are a part of MorphX today. In MorphX macros are not commonly used. A few places make use of macros such as keeping track of the list of fields stored when using dialogs. It is recommended only to use macros to define constants. Macros are not supposed to be used for code, as reusing code from macros is not flexible as using methods.

Macros can be created under the Macro node in the AOT, as a local macro in a method or a single line defining a constant. The main difference between a macro and a method is that a macro has no variable declaration part, and the code in a macro is not validated for errors before executed from a method. This is one of the main reasons not to put code in macros, as it makes the code more difficult to read.

When using macros in your code, the macros must be declared after the variable declaration. The common place to put the definition is in the ClassDeclaration of a class, form or a report. This will make the macro definition available for all parts of the object.

4.1 Macro commands

For writing macros a set of simple macro commands are used. You have commands for setting the start and the end of a macro like in a method. An if-statement can be set to control the flow of a macro. Macro if statements are used to validate whether a parameter is specified for a macro or not. For a list of the macro commands, see **figure 17: Macro commands overview**.

Command	Description
#define	Used to define a constant. See macro HRMConstants. Example #define.myConstant100('100')
#endif	Ends a #if.empty or a #if.notempty statement.
#endmacro	Ends a #LOCALMACRO or a #GLOBALMACRO.
#globalmacro	No difference whether declaring the start of a macro with #localmacro or #globalmacro. #globalmacro is not used in the standard package, consider using #localmacro instead.
#if.empty	Will return true if the macro has not been called with the parameter validated in the statement. Example #if.empty(%3) %3 = %2; #endif
#if.notempty	Will return true if the macro has been called with the parameter validated in the statement. See macro InventDimJoin. Example #if.notempty(%3) print %3; #endif
#linenumber	Returns the current line number of the macro. Can be used while debugging, but not of must use.
#localmacro	Specify the start of a local macro. See classDeclaration for class SalesReport_Heading. Example #localmacro.MyLocalMacro print %1; #endmacro
#macrolib	Used to load an AOT macro from code. See class method BOMHierarchy.searchDownBOM(). Example #macrolib.MyMacro

#undef	<p>Undefine a constant declared with #DEFINE. A defined constant cannot be used if #undef is called with the define name.</p> <p>Example</p> <pre>#define.MyConstant(100) print #MyConstant; #undef.MyConstant print #MyConstant; // will fail, as #MyConstant is not defined.</pre>
--------	---

Figure 17: Macro commands overview

4.2 Defining constants

This is macros in the simplest form. Instead of using text in your code it is strongly recommend defining your text as constants. Often you will need an integer or a text for setting a value. If you are going to set RGB color it is not easy to read the following:

```
myStringColor(255, 255, 255)
```

Instead you should consider defining a constant with a descriptive name:

```
myStringColor(#RBGColorWhite)
```

Using a macro as a constant rather than entering the value in code makes it easier to maintain. If, at a later time, you need to change the value, you will only have to modify the macro. Best practice will catch integer and text values used in code making it easy to change integer and text values to constants. A practice way of organize the constants you are using in your modifications is by creating a macro in the AOT for keeping all your constants in one place. Take a look at the macros located in the AOT, and you will see that some of the macros are prefixed with a module name followed by the word Constants.

To define a macro the macro command #define is used. The above mentioned RGB value is defined as follows:

```
#define.RBGColorWhite(255, 255, 255)
```

4.3 Creating macros

If browsing classes in the AOT you will see a macro called CurrentList is used a lot in the application. This is the most common macro used. The macro CurrentList is used for the list of fields to be stored in dialogs.

```
#define.CurrentVersion(2)
#localmacro.CurrentList
    FromDate,
    ToDate,
    Interest,
    CategoryA,
    CategoryB,
    CategoryC,
    Model
#endmacro
```

This is a snip of ClassDeclaration from the class InventReport_ABC. The constant CurrentVersion is used to keep track of the version number of the CurrentList macro. If the list of fields is changed in the macro CurrentList, the constant CurrentVersion is manually increased. This is a part of the interface used to store values used in dialog, which are explained further in the chapter **Classes**.

```
static void Macros_LocalMacro(Args _args)
{
    CustTable custTable;
;
    #localmacro.selectCustTable
        #ifnot.empty(%1)
            while select %1
                #ifnot.empty(%3)
                    order by %3
                #endif
                {
                    info(queryValue(%1.%2));
                }
            #endif
        #if.empty(%1)
            info("No table specified.");
        #endif
    #endmacro

    #selectCustTable(CustTable, accountNum)
    #selectCustTable
}
```

Macros are either created directly in the code, or put in an AOT macro and then the AOT macro is declared in the code. The above example is showing a #localmacro created in the code. The macro will select records from a table and print a field from the chosen table. The table to be fetched, and the field to be printed must be specified in the parameters. If no table has been specified when the macro is called a text will be printed to the Infolog. As you cannot declare variables in a macro, integer values prefixed with

a percentage sign such as %1 are used instead. The integer values are consecutively numbered and refer to the parameter position when calling the macro. Here the macro has 3 parameters. The sorting order of the fetched data is optional as a #ifnot.empty condition is checking whether the third parameter is specified. Of course, there ought to be more validations, as more could go wrong, but this is a common way of using the macros. The validations must be before calling the macro. Notice that you can call a macro without entering the parentheses after the macro name.

Note: If changing an AOT macro, you must recompile all AOT objects which are using the AOT macro. Changes to an AOT macro are first recognized by the objects using the macro when compiling.

If your macro is going to be used in several places it would make sense creating the macro in the AOT as you will then be able to reuse the macro. To create a new macro in the AOT unfold the Macro node, right-click and choose *New Macro*. A new empty macro node will be created. You can name the new macro by opening the property sheet. Now paste the code of the #localmacro to your new macro.

```
static void Macros_MacroLib(Args _args)
{
    CustTable custTable;
;
    #macrolib.Macros_MyMacro

    #selectCustTable(CustTable, accountNum)
    #selectCustTable
}
```

To use the macro created in the AOT the macro command #macrolib is used. Here the AOT macro is name Macros_MyMacro. Note that an AOT macro can contain as many #localmacro as required. You can also add constant definition and #localmacro to the same AOT macro. Still, it might be better dividing your constants and #localmacro into two macros, making your code easier to read.

Note: If an AOT macro name is equal to the name of a #localmacro in the same AOT macro, then the #localmacro can be addressed without declaring the AOT macro using #macrolib. This will not work properly as the parameters for the #localmacro will not be recognized.

4.4 Summary

In this chapter the use of macros in MorphX has been explained. You should now be able to define constants using macro commands and to create macros either in your code or added to an AOT macro, also called a macro library.

Also, you should by now know what macros can be used for, and when to use a macro, and more importantly why to put your code in methods rather than using macros.

5 Classes

X++ is an Object Oriented Programming language, also called OOP language. This means that code is encapsulated in objects. By using the object's parameter profile, an object can communicate with other objects. One of the powerful features is inheriting of code. A class can be inherited in a subclass making it possible to reuse the code of the parent class, also called the super class. This makes it easier to maintain modifications, as you can extend existing functionality by creating a subclass for your modifications. If you want to read more about OOP you can find several useful resource sites by browsing the internet. Basic knowledge on OOP will help you when designing your classes. You will not need to have knowledge at OOP reading this chapter however it will help clarify some of the terms used.

A class is a collection of methods. Compared to table methods, using a class makes reusing your code easier. The main difference is that classes can be inherited, and a class method can refer to other methods within a class. By that, you cannot say it is better to put your code in classes as table methods serve some purposes and classes others. The default tables methods can be used to execute your own table methods. If the code is relevant for other objects than a single table, you should consider using a class instead. Another option would be to declare a class and execute a class method from a default table method.

Classes are one of the complex data types in MorphX. A base data type stores a single value whereas a class object can store several values. Like base types, classes use variables to store values. Variables within a class can only be referenced by the methods of the class. From outside a class, the class methods are used to reference variables in a class. This is also called encapsulate. Consider a class as a custom designed object where the class methods are the handles to use the class. The class has no values set as only a class object can contain values. When declaring a class an object of the class is created. This is just like declaring a table variable as only a table variable can have a cursor.

5.1 Classes Basics

Axapta has two main categories of classes, application classes and system classes. You use application classes for constructing your application. Only application classes can be created and modified. System classes are primary used for doing runtime changes to the user interface. You will find an explanation of system classes later in this chapter. Application classes is located in the AOT under the node *Classes*.

Methods

One of the ideas with an OOP language is to split code up in small bricks, where each brick provides a certain operation. A class is such a brick. Classes are divided into

methods, where each method is doing or should be doing a single task. To make your code reusable you should bear this in mind when designing your methods. You could write most of your code for a class in a single method, but it would make the class useless for other purposes. Some use a rule of thumb to only have a certain number of code lines in a method. This can be a good solution, as long you bear in mind that your methods should be easy to reuse for other purposes.

Methods components

The code in a method consists of 4 blocks: Identification, variable declaration, code lines and return value. In the top of a method is the identification of the method which has the following syntax:

< modifiers> <return type> <method name>(parameter profile)

Modifiers are optional keywords defining the behavior for the method. An explanation of these keywords can be found in the section **Modifiers**.

The return type must be specified. This can be any base type or complex type. If you do not want a method to return a value you should use the keyword *void* as return type. Void is also used by some of the default methods on objects such as tables and forms which have no return type.

The name of a method should start with lower case letter. Use a descriptive name for the method which specifies what the method does. Names like `calcInventQty()` or `isFormDatasource()` will explain much more than just `calcQty()` or `formDatasource()`. After the method name the parameters are specified. Parameters are declared like any variables. It is a common practice to prefix parameter variable names with "_" to avoid confusing parameter variables with variables used in the method. It is optional whether a method should have parameters. You can however also set parameters to be optional when the method is called. This is done by initializing a parameter variable. Note that optional parameter variables should be put as the last parameters in the parameter profile.

If you are overriding a method, your method will have the same parameter profile.

The second block is variable declaration. You can find more information on variable declarations in the chapter **Intro to X++**.

Code lines are added after the variable declarations. If the method has a return type the keyword *return* is used for returning the value. Calling return will end executing of the method and return the value for the type specified. You can use return any number of times in a method. However it is considered best practice to only have one return in a method, as having several returns makes the code more difficult to read. Create a variable to store the value to be returned and call return as the last line of your method.

```
public AmountMst sumCustTrans(CustAccount _custAccount,
                             TransDate _transDate = systemdateget())
{
    CustTrans    custTrans;
```

```

AmountMst      sumAmountMst;
;

if (!_custAccount)
    throw error("Customer account not specified.");






sumAmountMst = (select sum(amountMst) from custTrans
                  where custTrans.accountNum == _custAccount
                  && custTrans.transDate   >= _transDate).amountMst;

return sumAmountMst;
}

```

This method calculates the sum of the customer transaction and returns the sum using the variable `sumAmountMst`. The method has two parameters where one is optional. If the first parameter customer account is not specified an error will occur, and if the method is called with only customer account, the second parameter transaction data will be initiated to current date.

Icons

The icon for a method is determined by the modifiers. Overridden methods have an icon with an arrow , methods created in the current class has the icon , protected methods has an icon with a key , private methods are represented with an icon with a keyhole  and static methods has an icon with a red cross .

The icons make it easy to get an overview of the methods of a table or class. Take a look at the class `CustBillExchangeClose`. It is easy to spot the type of methods. Note that overridden methods and instance methods are sorted alphabetically and static methods are listed right after. There is a good reason for this sorting, as static methods are referred in a different way.

Note, that an icon has a "traffic light" at the right side of the icon. The colors red and yellow indicate whether there are errors or warnings in the method. A green light indicates the method has been compiled without any errors.

Class Components

An application class has 3 default nodes: *ClassDeclaration*, *new* and *finalize*. See **figure 18: Default application class nodes**. In *ClassDeclaration* global variables for the class is defined. Variables can only be declared in *ClassDeclaration*, you cannot initialize variables. Only variables declared locally in methods can be initialized at the same time. Macros to be global for the class are also declared in *ClassDeclaration*.



Figure 18: Default application class nodes

When declaring a class you are declaring a class object. Before being able to reference the class object, you must initiate the class object. The compiler will throw an error if you are trying to reference a class object not initiated. The keyword `new` is used to initiate the class. The syntax is `new <Class name>()`. The class object and the class initiating the class object must be of the same type. This means that the class object and the class used to initiate the class object must be equal or a parent class to the class object.

```

MyClass      myClassObject;
;

myClassObject = new MyClass();
  
```

Initiating a class will invoke the method `new()`. As any other methods `new()` can also have parameters. Variables declared in `ClassDeclaration` are often initialized with the parameter values from `new()`.

Invoking `new()` will call the constructor for the class. A constructor is used to initialize a class. Several programming languages have a method by the same name of the class. This method is referred to as the constructor. This practice is not used in MorphX as `new()` is used as the constructor.

It is common practice in MorphX to create a method called `constructor()` if you need to initialize your class using different subclasses. Normally the `constructor()` method has a single parameter which is used in `constructor()` to determine which subclass to initialize using `new()`. The class `NumberSeqReference` which is part of the number sequence system in Axapta uses such a `construct()` method.

The method `finalize()` is used to remove the class from memory. After `finalize()` is called you will not be able to reference the class object. This is not a method which is called automatically as the garbage collector will automatically remove objects not used anymore from memory. It is not common practice to call `finalize()` each time an object is not used anymore. Even though you have a loop initiating the same class for each loop, you would not have to call `finalize()` as the garbage collector is set to run when a specific number of objects are no longer used.

It will make sense using `finalize()` if you object is not intended to be used any more, and to prevent other objects using your class object.

Example 1: Creating a class

Objects used from MORPHXIT_Classes project

- Class, MyClass

In this example a class with a single method will be created. This is a simple example to show how to create and use a class.


1. Go to the node *Classes*, right-click and chooses *New Class*. A new class called "Class1" will be created. Open the class by double-clicking the new class.
2. Check the left window to make sure that *ClassDeclaration* is selected. Rename the class by changing "Class1" to "MyClass".
3. In *ClassDeclaration*, declare a variable of the extended data type *ItemId*. *ClassDeclaration* should look like the following:

```
class MyClass
{
    ItemId itemId;
}
```

4. Now add an new method to the class by pressing ctrl+n. The new method called "Method1" will automatically be opened in the editor. Change the name of the method to "parmItemId".
5. Add the extended data type *ItemId* as parameter to the new method *parmItemId()*. The parameters should be initialized with the global class variable *itemId*. Set the parameter variable equal to the global class variable *itemId*. The method must return *itemId*.

```
ItemId parmItemId(ItemId _itemId = itemId)
{
    ;
    itemId = _itemId;

    return itemId;
}
```

6. Click the save icon  in the editor tool bar to save all changes to the class.

As class variables cannot be referenced from outside a class you can create a method to set and get the value of a class variable. Such methods are often prefixed with *parm**. The class *MyClass* contains one method called *parmItemId()* which will return the value of the global class variable *ItemId* if called without any parameters. The optional parameters in *parmItemId()* can be used to set the value of *ItemId*. These types of methods are especially used in dialogs for transferring the value of a dialog field to a calling object. The ability to set a value makes it possible using the class without using the user interface of the dialog.

```
static void Classes_TestMyClass(Args _args)
{
    MyClass    myClass;
```

```

;
myClass = new MyClass();
myClass.parmItemId("Item100");

info(myClass.parmItemId());
}

```

The easiest way of testing a class is by creating a job. Try creating a job which looks like the job shown here for testing MyClass. The job initiates MyClass and executed the method `parmItemId()` with a parameter. The method `parmItemId()` is called again without a parameter, printing the previously set value to the Infolog.

```

void methodWithFunction()
{
    void methodWithFunction(CustAccount _custAccount)
    {
        ;
        info(_custAccount);
    }

    methodWithFunction("4000");
}

```

Functions can be written inside a method. A function can only be used by the method and cannot be referenced outside the method. The syntax of a function in a method is similar to the one of a method. Here a function is declared of the same name as the method. This is allowed as long as the parameter profile is not equal to the methods parameter profile. A function is called without qualifying the object name. Note that no validation will be done whether the right number of parameters is entered for a function. You should consider functions in methods as an option in X++. This is not a recommended feature as it does not make code easier to reuse. Instead you should consider creating another method.

Modifiers

It is optional to specify modifiers for a class or a method. Modifiers are however quite useful as you can set restrictions on the use of a method or a whole class and how a class is inherited.

Access Modifiers

You can add an access modifier to a class or a method to restrict the use of a method. Both instance and static methods can use access modifiers and restrictions can be set at different levels. You may want a method only to be used within the class or only to be used for the class hierarchy. The default level is public which will give full access to a class and all methods of the class. The access modifiers are shown in **figure 19:**

Overview of access modifiers.

MorphX support setting access modifiers for methods and classes only. Variables declared in ClassDeclaration will always be accessible for subclasses.

Keyword	Description
Public	Default behavior for classes and methods. A public class can be inherited and class methods can be overridden in subclasses and called outside the class.
Protected	Only methods can be protected. A protected method can overridden in subclasses, but can only be used inside the class hierarchy.
Private	Both classes and methods can be set as private. However this will only affects methods. A private method can only be used within the current class.

Figure 19: Overview of access modifiers

Access modifiers in MorphX are often forgotten as the default behavior is public. This let to that every part of a class can be accessed from anywhere. Having no restriction for a class might cause the class is not used as intended to. You should consider specifying access modifiers to make the use of your class easier to understand. A class often has several method used for calculations done internally in the class. Such methods should be restricted from being overridden by subclasses.

Example 2: Access Modifiers

Objects used from MORPHXIT_Classes project

- Class, MyClass_AccessModifiers
- Class, MyClass_AccessModierers_sub

A super class and sub class will be created to show the use of methods access modifiers.

1. Create a new class and rename the class “MyClass_AccessModifiers”.
2. Add a method called publicMethod() to the class. Set the access modifier to public. Add a line printing the access modifier keyword to the Infolog.

```
public void publicMethod()
{
;
    info("public");
}
```

3. Repeat step 3 by adding similar methods for the access modifiers protected and private. Remember to set the access modifier for each method.
4. Save the class MyClass_AccessModifiers.

5. Create a sub class called `MyClass_AccessModifiers_sub`. ClassDeclaration for the sub class must look like the following:

```
class MyClass_AccessModifiers_sub extends MyClass_AccessModifiers
{
}
}
```

6. Right-click the node `MyClass_AccessModifiers_sub` and choose *Override Method*. Select the method `protectedMethod()`. This will create a new method in the sub class.
7. Save the class `MyClass_Modifiers_sub`.

The super class contains 3 methods, but when overriding the methods in the subclass only the methods public and protected are shown. MorphX will validate the access modifiers and only show methods which can be overridden. Even though methods by default are declared public it makes sense using the keyword. Methods with access modifier qualified cannot have the access modifier changed in a subclass. Note that, an overridden method will only contain a call to `super()`. The `super()` call will execute the code written for the method in the super class.

Note: The tools Visual MorphXplorer and Application Hierarchy Tree called from the add-ins menu will help you get an overview of the class hierarchy. Remember to update the cross-reference.

When creating a sub class it is considered good practice to prefix the subclass with the name of the super class followed by "_". If the super class is `SalesFromLetter`, the name of a sub class could be `SalesFormLetter_Confirm`.

```
static void Classes_Modifieres(Args _args)
{
    MyClass_Modifiers_sub modifiers_sub;

;
    modifiers_sub = new MyClass_Modifiers_sub();
    modifiers_sub.publicMethod();
}
```

This job is testing the class `MyClass_AccessModifiers_sub`. Only the method with the access modifiers public is executed, as the private and protected methods cannot be executed from outside the class hierarchy. When pressing the dot after the class name all methods from the super class and the subclass are shown. The compiler will, however throw an error if trying to use a method without the proper access level.

Note: Prior to version 3.0 access modifiers was not validated by the compiler. Access modifiers could still be specified but had no effect.

Using the modifier private for a class will not have any effect. If you want to restrict how a class is declared you could instead set `new()` as private. This will prevent

declaring the class in the normal way using `new()`. Instead you could create a `constructor()` method to control how the class is declared. A `constructor()` method is always created as a static method and will be accessible depending on the access modifier for the `construct()` method.

Access modifiers can also be used in forms and reports to prevent methods being called outside the object. In fact you can use most type of modifiers in forms and reports however it only makes sense using access modifiers.

Static Modifier

By default methods are created as instance methods. This means that you must declare a class object before being able to access the method. If the keyword *static* is added to method you can access the method without declaring the class object.

```
static EmplTable find(EmplId _emplId,
                    boolean _forUpdate = false)
{
    EmplTable emplTable;

    if (_emplId)
    {
        emplTable.selectForUpdate(_forUpdate);

        select firstonly emplTable
            index hint EmplIdx
            where emplTable.emplId == _emplId;
    }

    return emplTable;
}
```

The above code block shows the `find()` method from `EmplTable`. The method will return an employee record based on the employee id given as a parameter. Table methods such as `find()` and `exist()` are always created as static. When using a method for finding a table record it will not make sense to first declare a table variable as the method should return the table variable.

The following block of code shows how to call the static `find()` method on `EmplTable`. Note that the double colon syntax used for static methods. This is the same notification used to refer an enum entry.

```
static void Class_CallStaticMethod(Args _args)
{
    ;
    info(EmplTable::find("AMO").name);
}
```

Static modifiers are often used in classes for methods which need to be accessed frequently. The class `WinAPI` is a good example on using static methods. `WinAPI`

contains a list of static method such as methods to access the file system for doing file operations. Typically only a single method is required at the time from WinAPI such as checking whether a file exists.

Methods created for the purpose to declare an object of a class use the static modifier. The most common examples are the methods `main()` and `constructor()`. Note that `main()` is the only user defined method of a class invoked by the kernel.

Final Modifier

Setting a class to final will prevent the class to be overridden. The reason could be that you would either force the use of the class as it is. The class might be intended to be used by another class or the class may be used to provide data to another class. The final class `InventOnhand` is an example of this.

A final method cannot be overridden by a subclass. Only instance methods can be qualified as final, as a static method cannot be overridden. Defining a method as final will only prevent inheriting. The method will have the same access level as a public method. You can however use final together with one of the access level modifiers like:

```
final protected void protectedMethod()
{
;
    info("protected");
}
```

Display and Edit Modifiers

MorphX has two special modifiers, display and edit which are used in the user interface. These are used for form and report controls which are not related to a data source field. Both display and edit can be qualified with the modifiers static and final. You will find a detailed explanation on the display and edit modifiers in the chapter **Forms**.

Abstract Modifier

An abstract class or method is the exact opposite of a final. The use of abstract classes is a way of planning inheritance as it forces creating a subclass for using the class, as an abstract class cannot be declared. This is often used in the standard package for super classes to control that the super class is not declared by mistake. The class `SalesFormLetter` which is used for creating documents such as sales confirmations and sales invoices uses this practice. The class has an `construct()` method which should be used, and to prevent the class being declared using `new()` `SalesFormLetter` is qualified as abstract.

```
abstract CustAccount myAbstractMethod()
{
    // The code in this method is never executed
}
```

```

    #if.never
        select firstly custTable
        where custTable;
    #endif
}

```

Methods can be declared as abstract, but only if the class is abstract. An abstract method must be overridden as an abstract method cannot have a code block. Abstract methods contains only a parameter profile. You can however use the macro call `#if.never` to add "code" to an abstract method. This will help clarify why this method must be overridden in the subclass. You might wonder why you should not simply add comments in the abstract method. The point is that code within the macro call is shown in color as with any other code in the editor making it easier to differentiate comments and code. Note that no validation is done on code written in the `#if.never` macro so anything could be written. It is optional to qualify methods of an abstract class as abstract. For this reason you will still be able to add variables in the `ClassDeclaration` of an abstract class..

The access modifiers `protected` can be use with abstract methods. An abstract method cannot be static, as the static methods only exist in the super class.

Interface Class

MorphX can only handle single inheritance, meaning that only one super class is allowed. This is not bad at all, as in languages supporting multi inheritance it can be tricky getting an overview of the class hierarchy.

Another option exists as you can create interface classes and implement the interface classes in your sub class. Like abstract classes and methods an interface class cannot be declared and the methods of an interface class cannot contain code. So what is the difference between an abstract class and an interface? Well an abstract class is used to give instructions on the content of a subclass. An interface does not have to be part of the hierarchy. You can implement more than one interface for a class. Interfaces are used for common tasks like the interface `SysPackable` which is controlling that the methods for storing the last values of a dialog are created.

```

interface MyClass_Interface
{
}

```

An interface is in fact not a real class. When creating an interface you are replacing the *class* keyword with *interface*. Interfaces can be inherited by another interface, but a class cannot inherit an interface, a class implements an interface.

```

public abstract class Runbase extends Object implements sysSaveable, sysRunnable
{
}

```

When implementing an interface class in a class the keyword *implements* is used. Here the class header from the class Runbase is shown. Note that implements are put after the inherited class.

Note: By calling the global method pickInterface(true) from a job all interface classes will be listed in a lookup window. The global class has several methods prefixed with pick* which can be used for lookups of AOT nodes.

Any access modifiers can be used in an interface class. Remember an interface class is not a super class, but a sort of template for your class.

Passing Values

As seen, a methods parameter profile can contain any number of parameters. Both base types and complex types can be passed on as parameters. The use of passing and returning variables to and from a method is the main purpose of methods. It makes your methods acts like bricks which are easy to fit together.

Call by

In MorphX, variables are called by value. This means variables passed to a method will not be changed by altering the parameter variable inside a method. Using a temporary table as a parameter differs from that rule as temporary tables are called by reference. If a temporary table is passed as a parameter to a method, and the temporary table parameter is modified in the method, the calling variable will also be changed.

No matter which type of parameter variables you are using it is recommended to create a local variable for a parameter variable, instead of modifying the parameter variable as this will make your code easier to understand.

Example 3: Call by

Objects used from MORPHXIT_Classes project

- Class, MyClass_PassingValues

A class containing two methods will be created. The purpose is to show how MorphX acts when passing variables to a method. A parameter in the method callByValue() is changed inside in the method. This is not considered best practice.

1. Create a new class called MyClass_ParameterValues.
2. Add a method called callByValue() with one parameter of the extended data type Counter. Increase the value of Counter by 10 and have the method returning the result.

```
Counter callByValue(Counter _counter)
```



```
{
    _counter += 10;

    return _counter;
}
```

3. Add a method called `callByReference()` with the temporary table `TmpAccountSum` as parameter. The method should fetch the last record from the temporary table and print a field to the Infolog.

```
void callByReference(TmpAccountSum _tmpAccountSum)
{
    TmpAccountSum tmpAccountSum;
;
    tmpAccountSum = _tmpAccountSum;

    select firstonly tmpAccountSum order by accountNum desc;
    {
        info(tmpAccountSum.accountNum);
    }
}
```

4. Save the class.
-

To test the class `MyClass_PassingValues` create the following job:

```
Static void Classes_CallByValue(Args _args)
{
    MyClass_PassingValues    passingValues    = new MyClass_PassingValues();
    Counter                  counter          = 100;
;

    info(strFmt("%1", counter));
    info(strFmt("%1", passingValues.callByValue(counter)));
    info(strFmt("%1", counter));
}
```

The job is initializing the counter variable and printing the value to the Infolog before calling `callByValue()`. The return value of `callByValue()` is printed to the Infolog, and at last the counter variable is printed again showing that `callByValue()` has not modified the initialized value of the counter variable.

```
static void Classes_CallByReference(Args _args)
{
    CustTable                custTable;
    TmpAccountSum            tmpAccountSum;
    MyClass_PassingValues    passingValues = new MyClass_PassingValues();
    Counter                  counter;
;

    while select custTable
    {
```

```

    counter++;

    if (counter > 5)
        break;

    tmpAccountSum.accountNum = custTable.accountNum;
    tmpAccountSum.insert();
}

select tmpAccountSum;

info(tmpAccountSum.accountNum);
passingValues.callByReference(tmpAccountSum);
info(tmpAccountSum.accountNum);
}

```

A similar test is done with the method `callByReference()`. First the temporary table must have some data inserted. The first 5 records from `CustTable` are looped and inserted. The first record of the temporary table is printed to the Infolog. `CallByReference()` is called with the temporary table as a parameter, and will print the last record from the temporary table. Note, when printing the temporary table again after calling `CallByReference()`, the fetched record has been changed to the record fetched by `CallByReference()`.

Call by reference can be confusing and this might not be appropriate for your case. To avoid having your temporary table variable changed in the calling code you should modify `callByReference()` to look like the following.

```

void callByReference(TmpAccountSum _tmpAccountSum)
{
    TmpAccountSum tmpAccountSum;
;
    tmpAccountSum.setTmpData(_tmpAccountSum);

    select firstly tmpAccountSum order by accountNum desc;
    {
        info(tmpAccountSum.accountNum);
    }
}

```

The method `setTmpData()` must be used to initialize the local `TmpAccountSum` as setting `TmpAccountSum` equal to the parameter variable `_tmpAccountSum` will still result in a call by reference. Note that only temporary tables are called by reference. A normal table like `CustTable` would be called by value.

Recursive calls

A recursive method is a method calling itself. Make sure when creating a recursive method that your code will not result in a never ending loop. With that said, recursive methods are excellent for reusing your code and make your code simple.

Try adding the following method to the class MyClass_PassingValues:

```
void recursiveCall(ProjId _projId = "")
{
    ProjTable projTable;
    ;

    while select projTable
        where projTable.parentId == _projId
    {
        setprefix(projTable.parentId);
        info(projTable.projId);

        this.recursiveCall(projTable.projId);
    }
}
```

The table ProjTable which is the main table for projects is loop in the method recursiveCall(). Projects are built in hierarchy in Axapta, so the method is sorting the records by the tree hierarchy. A lot of code should have been written, or at least two methods would have been needed if it was not for the recursive call. For each loop in the method recursiveCall() a line is printed in the Infolog. The setprefix() method is used to show the level of the projects in the Infolog. Write the following job to test the method:

```
static void Classes_RecursiveCall(Args _args)
{
    MyClass_PassingValues passingValues = new MyClass_PassingValues();
    ;

    passingValues.recursiveCall();
}
```

Returning values

A method can only return one value, or rather one variable. When return is called the method is ended. Sometime you might have a situation where it would be neat to return more than one variable. In this case you could reconsider your design as you might try to accomplish too much in your method.

```
Container returnContainer()
{
    CustAccount custAccount;
    custName custName;

    return [custAccount, custName];
}
```

As the return type can be a base type or a complex type you could also use a container. You do not have to declare a container variable in the method as you can return the values separated by comma in brackets [] like shown in the above example. MorphX have set of foundation classes which also can solve this issue. A foundation class can contain a number of variables. You will find more information on foundation classes in the section **Foundation Classes**.

5.2 AOS

An Axapta installation can be configured to run as a 2-tier or 3-tier installation. Where a 2-tier installation consists of a client and a server, a 3-tier installation also has an Application Object Server, also called AOS. Without an AOS server the workload is on the client instead which will result in more calls between the database server and the client. An AOS will reduce the traffic between database and client as the AOS will communicate with the database instead, making it possible only to have calls to client when communicating with the user interface.

You can have any number of AOS servers configured in a clustering environment to spread the workload. For more information about AOS see the manuals in the standard package. An explanation of the AOS is out of the scope for this book. The focus is how to optimize your code using AOS.

Note: Optimizing your code for using AOS is an option. You should not optimize every part of your code for using AOS as you might end up spending too much time on optimization. If you are having some heavy modifications or limited bandwidth you will get a much better performance optimizing for AOS, and you should.

Setting Tier

When a table method or a class is executed you can specify whether the code should be executed from the AOS or from the client. The default table methods accessing the database for inserts, updates and deletes cannot be overruled. These methods will always run server side. Forms should always run on the client. By default an object will be executed on the tier the object is called from. A decision on changing the default behavior on where an object is executed should be caused by optimization only.

Your code will still run in both 2-tier environments and 3-tier environments as the settings running an object on a specific tier is only validated in a 3-tier environment. This also means that AOS optimization should only be made running 3-tier. Changing a form to run server side will work in a 2-tier environment, but running the form in 3-tier will cause the form never to be shown as the form has been specified to be executed on the server.

To define the tier for executing an object the property **RunOn** and the modifiers *client* and *server* are used. Menu items and classes have the property RunOn. The default setting for menu items is running on the client. Classes are by default set to be called

from. The property RunOn can be set to client, server or called from. Setting RunOn to called from means the object will be executed on the tier used by the calling object. The keywords client and server are used to specify the tier in methods. A method is set to be called from by adding both client and server as modifiers. Note that, it is considered good practice to put AOS modifiers in front of any other modifiers.

```
server AmountMST balancePerDate(TransDate _transactionDate = systemdateGet())
{
    return this.CustVendTable::balancePerDate(_transactionDate);
}
```

The property RunOn in the property sheet for a class is used to define the tier for which the class is executed. All instance methods in the class will be executed on the tier specified by the RunOn property. Only static methods of the class can have their own AOS modifier. The method shown above is set to run on AOS. Tables differ slightly as both instance methods and static methods in tables can have AOS modifiers.

If you set the RunOn tier for a menu item calling a class, then the class will default run on the same tier if the class property RunOn is set to called from. The setting of a menu item can however be overruled by the class. The tier is determined by where the class is declared, where new() is called. This also means that the RunOn property of a super class will decide on which tier the subclasses will be executed.

Objects to Optimize

So which object should be AOS optimized and when should this be done? A rule of thumb is that table methods and classes with a lot of database calls should be candidates for objects running on AOS. Reports using the runbase framework can also be set to run on a server. It might sound wrong having a report running on server, while a form cannot. The runbase framework will handle calls between client and the AOS so the application user will still have the report dialog shown even though the report is executed on the AOS.

The MorphX tools System Monitoring and Code Profile can be used to track calls between the tiers. You should familiarize yourself with these tools before trying to optimize your code for AOS.

5.3 Runbase Framework

A batch job is a task set to be executed at a specific time or to be repeated with a specified interval. The batch server processing the batch jobs is typically set to run on a dedicated server. An Axapta client is started to run the batch server, alternatively the batch server can be set to run as a Microsoft Windows Service. You can find detailed

information on configuring batch servers and batch jobs by checking the manuals in the standard package.

Heavy tasks like classes or complex reports processing a lot of records are typically scheduled as batch jobs. This could be a daily task like transferring sales orders to an external system or periodic printing of customer balances.

Batch jobs have no interaction with the application users as this will keep the batch job on hold. When a class is scheduled as a batch job, the values entered by the application users are used as the batch server will skip any dialogs.

Using Runbase Framework

The runbase framework has two primary functions. To create a similar layout for dialogs presented to the application users and to make it possible to schedule a process to be batch able. The classes prefixed with RunBase* are referred to as the runbase framework. Several of these runbase classes are only called by the framework. In daily use you will only need to know the purpose of a few of them. See **figure 20: Common used runbase framework classes**.

Classname	Description
RunBase	The class RunBase is used for tasks which should not be batch able.
RunBaseBatch	RunBaseBatch is used to give the application user the option to schedule a batch job for the task.
RunBaseReport	Only used for heavy reports and is therefore a subclass of RunBaseBatch.

Figure 20: Common used runbase framework classes

The part of the runbase framework used for reports is explained in more detail in the chapter **Reports**.

Example 4: Creating batch able class

Objects used from MORPHXIT_Classes project

- Class, MyClass_RunBaseBatch
- Menu item action, MyClass_RunBaseBatch

In this example a class using the runbase framework will be created. The class will have an option to be executed as a batch job. Focus is on how to construct a class using the runbase framework.

The class will print all customers where customer transactions are posted after a specified date. The last entered from date used for searching customer transactions will be stored.

Create a new class called “MyClass_RunBaseBatch”.

```
class MyClass_RunBaseBatch extends RunBaseBatch
{
    FromDate    fromDate;
    DialogField dialogFromDate;

    #define.CurrentVersion(1)

    #localmacro.CurrentList
        fromDate
    #endmacro
}
```

The new class must be a subclass of the class RunBaseBatch. A variable of the extended data type FromDate is created for storing the last entered date. The variable dialogFromField is a variable of the class DialogField which is used to add a field to the dialog shown to the application users. The macro constant CurrentVersion is keeping track of the last version of the dialog. CurrentList is a macro containing the variables to be stored from the last run of the class.

```
public container pack()
{
    return [#CurrentVersion, #CurrentList];
}
```

Pack() is overridden. The method is returning the two macros created in ClassDeclaration.

```
public boolean unpack(container packedClass)
{
    container    base;
    boolean      ret;
    Integer      version = conPeek(packedClass,1);

    switch (version)
    {
        case #CurrentVersion:
            [version, #CurrentList, base] = packedClass;
            ret = true;
            break;
        default:
            ret = false;
    }
    return ret;
}
```

Unpack() is also a overridden method(). The variables in the macro CurrentList are initialized when unpack() is executed.

```
static void main(Args _args)
{
    MyClass_RunBaseBatch runBaseBatch = new MyClass_RunBaseBatch();
};
```

```

    if (runBaseBatch.prompt())
        runBaseBatch.run();
}

```

This is a static method declaring the class. Main() is used to execute the class. If the dialog is not canceled by the application user, the method run() will be executed.

```

protected Object dialog()
{
    DialogRunBase dialog = super();
    ;

    dialog.addGroup("Date");
    dialogFromDate = dialog.addFieldValue(typeId(FromDate), fromDate);

    return dialog;
}

```

In this overridden method the dialog is declared. A field for entering from date is added to the dialog.

```

public boolean getFromDialog()
{
    boolean ret;

    ret = super();

    if (ret)
    {
        fromDate = dialogFromDate.value();
    }

    return ret;
}

```

GetFromDialog() is called if the button Ok is pressed in the dialog. The value entered in the from date field in the dialog is stored in the variable fromDate.

```

client server static ClassDescription description()
{
    return "Testing batch able class";
}

```

All classes inherited from the runbase framework should have this method. As the method is static you cannot override the method from the super class, so description() must be created manually. The method will add a caption to the top of the dialog window.

```

public void run()
{
    CustTable      custTable;
    CustTrans      custTrans;
    Counter        totalRecords;
    ;

    select count(recId) from custTable
    exists join custTrans

```



```

        where custTrans.accountNum == custTable.accountNum
        && custTrans.transDate    >= fromDate;

totalRecords = custTable.recId;

startLengthyOperation();
this.progressInit("List customers with transactions", totalRecords, #AviSearch);

while select custTable
exists join custTrans
    where custTrans.accountNum == custTable.accountNum
    && custTrans.transDate    >= fromDate
{
    progress.incCount();
    progress.setText(sprintf("%1, %2", custTable.accountNum, custTable.name));
    sleep(500);
}
endLengthyOperation();
}

```

Run() is processing the task for the class based on the dialog settings. The first select statement is counting the number of customers to be processed using an aggregate function. The counted numbers are used for a progress bar which will be incremented for each loop where looping the customers. The global methods startLengthyOperation() endLengthyOperation() is used to set the scope to invoke the hour glass. A macro call is used as parameter when initializing the progress bar. This is macro constant from the macro library AviFiles which is declared in the runbase framework. Instead of printed information to the Infolog, the progress bar is updated for each loop. The function sleep() is setting a delay of 1/2 second making it possible to see the progress bar.

Finally, create a menu item for the class by dragging the class to the menu item node *Output*.

When executing MyClass_Runable a dialog will appear. The dialog has two tab pages. The first tab page has a field group with the field for specifying from date. The value added will be stored as the default value next time the class is executed. The prompt() method in main() is calling the dialog() method. The method run() is called if the button Ok is pressed in the dialog and the method getFromDialog() is validated true. You should always start by overriding the methods pack() and unpack() when creating a runbase class. If you try to create the static main() method before overriding these methods you will have an error as pack() and unpack() are part of an interface implemented by the runbase framework.

The second tab page is for the batch settings. Even though a class may not be executed as a batch job you can still use the class RunBaseBatch. If the method canGoBatch() is overridden and return false, the batch tab page will not appear.

Notice, if you need to validate the values keyed in by the application user in the dialog, you should override the method validate(). The dialog cannot be closed using the Ok button before the method validate() returns true.

You can execute the class `MyClass_Runnable` directly by right-clicking the class and choose *Open*. As the class has a static method called `main()` with the parameter `Args`, the class can be executed without any coding. This is also called a run able class. All classes with a static `main()` method are runnable and can be executed from a menu item. Note that the runbase framework is not required to create a runnable class.

You should execute the class from a menu item. It is recommended always to create a menu item for your runnable class as you will then be testing your class as the application users will execute the class. Runnable classes are always created as action menu items, if not calling a form or a report.

A progress bar was used in the example. For a task which takes more than a few minutes to process you should consider adding a progress bar. This will make your class more user friendly as the application user can see when the job is finished. As progress bars are a part of the runbase framework you can easily add a progress bar to your subclass. You can however use progress bars for any block of code. The classes prefixed with `SysOperationProgress*` are used for building progress bars from scratch. The form `Tutorial_Progress` will show you a lot of the features available when using progress bars.

For an overview of the available animations in progress bars check the class `Tutorial_ShowAviFiles`.

The parameter `Args` in `main()` is used to get a handle on the calling object. This is often used when calling and runnable class from a form as you by using `Args` will be able to get the cursor record or just identify the calling object. The chapter **Forms** shows the use of `Args`.

Dialog

A dialog is the user interface of a class. When you need a dialog for your class you should use the runbase framework as dialogs is an integrated part of the framework. The classes prefixed with `Dialog*` are used by the framework. Simple form features like grouping fields, adding lookups to related fields are provided by the dialog classes. You can add additional buttons to your dialog like calling a query. If your class is inherited from the `RunBaseReport` class your report query and printer settings will automatically be wrapped in the dialog.

Example 4: Class dialog

Objects used from MORPHXIT_Classes project

- Class, `MyClass_RunBaseDialog`
- Menu item action, `MyClass_RunBaseDialog`

In this example some of the common features in the class dialog is shown. To simplify the code, the methods `description()` and `run()` has by purpose been left out.

Start duplicating the class `MyClass_RunBaseBatch` and rename the class to “`MyClass_RunBaseDialog`”. `Pack()` and `unpack()` will remain unchanged. The `run()` method must be deleted. The other methods will be modified as follows:

```
class MyClass_RunBaseDialog extends RunBaseBatch
{
    NoYesId          selectDate;
    FromDate          fromDate;
    ToDate            toDate;
    NoYesId          selectCustGroup;
    CustGroupId       custGroupId;
    DialogField        dialogFromDate, dialogToDate, dialogSelectCustGroup, dialogCustGroup;
    DialogGroup        dateGroup, countryGroup;

    #define.CurrentVersion(1)

    #localmacro.CurrentList
        fromDate,
        toDate,
        selectCustGroup,
        custGroupId
    #endmacro
}
```

Some additional dialog fields have been declared. A variable is created to store each of the dialog fields. The macro `CurrentList` is extended to store the new variables.

```
protected Object dialog()
{
    DialogRunbase dialog = super();
    ;
    dialog.allowUpdateOnSelectCtrl(true);

    dateGroup = dialog.addGroup("Date");
    dateGroup.frameOptionButton(FormFrameOptionButton::Check);
    dateGroup.columns(2);

    dialogFromDate = dialog.addFieldValue(typeId(FromDate), fromDate);
    dialogToDate    = dialog.addFieldValue(typeId(ToDate), toDate);

    countryGroup = dialog.addGroup("Customer");
    countryGroup.columns(2);

    dialogSelectCustGroup = dialog.addFieldValue(typeId(NoYesId), selectCustGroup,
        "Select customer group");
    dialogCustGroup       = dialog.addFieldValue(typeId(CustGroupId), custGroupId);

    return dialog;
}
```

The dialog group date now contains from date and to date. By default all dialog fields will be added to a single column. The method `columns()` is used to set from date and to date at one line. The dialog method `frameOptionButton()` is used to have a checkbox in the field group header. The other field group Country has two fields. If the field

dialogSelectCustGroup is unmarked, the field dialogCustGroup cannot be edited. This feature can only be used if the dialog method allowUpdateOnSelectCtrl() is set to true.

```
public void dialogSelectCtrl()
{
;
    if (dialogSelectCustGroup.value())
    {
        dialogCustGroup.allowEdit(true);
    }
    else
    {
        dialogCustGroup.allowEdit(false);
    }
}
```

Each time a dialog method is entered this method is executed if the dialog method allowUpdateOnSelectCtrl() is set to true.

```
public boolean getFromDialog()
{
    boolean ret;

    ret = super();

    if (ret)
    {
        fromDate      = dialogFromDate.value() ? dialogFromDate.value() : systemdateget();
        toDate        = dialogToDate.value() ? dialogToDate.value() : systemdateget();
        selectCustGroup = dialogSelectCustGroup.value();
        custGroupId    = dialogCustGroup.value();
    }

    return ret;
}
```

The values are retrieved from the dialog fields and initialized with the variables declared in ClassDeclaration. The check field in the header of the dialog group date is on purpose not added.

```
static void main(Args _args)
{
    MyClass_RunBaseDialog runBaseDialog = new MyClass_RunBaseDialog();
;

    if (runBaseDialog.prompt())
        runBaseDialog.run();
}
```

Main() has just been changed to executed the new class.

All that is left is to create a new action menu item for the class.

The class will not process anything as `run()` has been left out. As `run()` is a member of the super class this will not give any error. Running the class will show the dialog seen in **figure 21: Dialog example**.

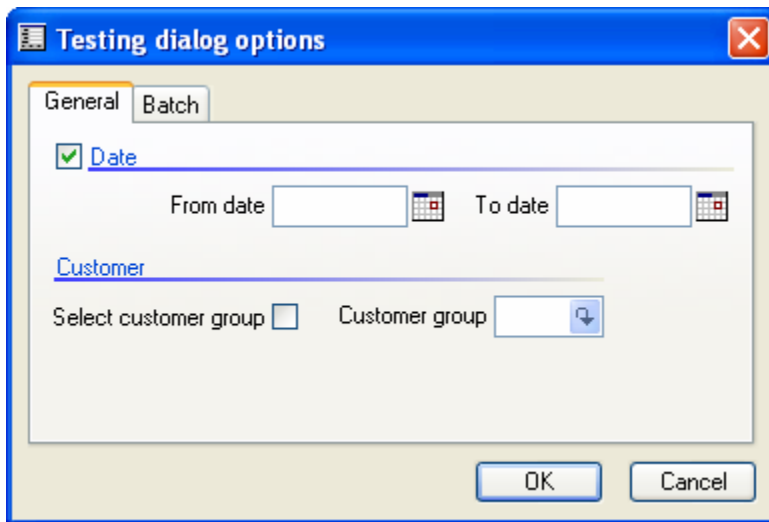


Figure 21: Dialog example

The dialog shows two ways of making a value of a field value depended on another field value. If unmarking the check field in the date field group header, the fields from date and to date cannot be edited. Your case could be that a date range is entered and if disabled all records are looped.

The Customer field group uses a similar functionality. The field Customer group can only be specified if the Select customer group is marked. As the method `dialogSelectCtrl()` can only be executed when using fields of the type `DialogField` this is a often used solution. However `dialogSelectCtrl()` cannot refresh the dialog so the dialog will not be update with the new settings before leaving the field select customer group.

An alternative to use the dialog classes for building dialogs is to create a form. The `runbase` framework supports the using standard forms as dialogs. A form will be presented just as a dialog. This is often a preferable solution for complex dialogs. If you have to build a complex dialog it can often save time to use a form. You will have an easier overview of your dialog and it will be easier later to extend the dialog. You will find more information on using forms as dialogs in the chapter **Forms**.

In the dialog example the dialog fields could just as well have been added to a query instead. By using the dialog method `addMenuItemButton()` a button called Select will be added to the dialog. The query loaded when the select button is pressed must be specified by overriding the method `queryRun()`. Add the following code to your dialog to have a button for the query shown in the dialog:

```
dialog.addMenuItemButton(MenuItemType::Display,
                        menuItemDisplayStr(RunBaseQueryDialog),
                        dialogMenuItemGroup::BottomGrp);
```

By overriding `queryRun()` with the code below, a query containing `CustTable` joined by `CustTrans` will be loaded.

```
public QueryRun queryRun()
{
    QueryRun ret;

    ret = new QueryRun(QueryStr(Cust));

    return ret;
}
```

Details on how to build and retrieve information from a query is explained in the chapter **Queries**.

A useful feature of using the runbase framework is that the last entered value of a field can easily be stored for the next run. Typically this is used for storing values of dialog fields, however any variables can be stored. You will only have to add the variable to the `CurrentList` macro. When changing `CurrentList` on an existing class, you should increment the macro constant `CurrentVersion` by one, as errors might occur if using the same version when changing the list of saved variables. The last values are stored in the system table `SysLastValue` per user and company. Incrementing `CurrentVersion` will cause a new record to be created in `SysLastValue`.

`Pack()` and `unpack()` must always be overridden as they are part of an interface in the runbase framework. Also the two macros in `ClassDeclaration` are required. You will have situations where the last values are not needed. To skip using the last values `pack()` must return an empty container like `connull()` and `unpack()` must return `false`.

If you just need a dialog to appear with the choice to continue or not, there will be a more simple solution than using the runbase framework. The class `Box` has a collection of static method which supports all the common combinations. You can wrap the call of a method from the class `box` in an if statement:

```
static void Classes_Box(Args _args)
{
    if (Box::yesNo("Continue", DialogButton::Yes,"Test of Box") == DialogButton::Yes)
        info("Here goes.");
}
```

5.4 Fundamental Classes

The last classes listed in the AOT under the node *Classes* are a special type of classes. These classes are referred to as fundamental classes and are not ordinary classes which should be inherited or modified. You can recognize the fundamental classes by the icon



You will be using the fundamental classes for your modifications even though you might not be aware of it, as these classes are normally not declared as a normal class. The most important fundamental classes are declared by the kernel when starting the client. Having a basic knowledge about fundamental class will help you understand some of the elementary processes in MorphX. In the following you will find an explanation of the most important fundamental classes.

ClassFactory

Whenever an object of the user interface is invoked like a form, report or a dialog the class ClassFactory is activated, or at least ClassFactory should be called when executing an object. When calling a form, report or a query from X++, the object should be initialized using ClassFactory. This is automatically done if calling an object using a menu item or when using the runbase framework.

The purpose by using ClassFactory is to have the same fundament. This can be used to have a handle for making general changes to forms or reports such as adding headers or controls to forms at runtime. For examples on how to use ClassFactory for forms and reports, see the chapters **Forms** and **Reports**.

Note that running a 3-tier environment two instances of ClassFactory will exist. One declared on the AOS and the other declared on the client. When an object is executed, the ClassFactory instance on the current tier will be used. If you reference ClassFactory from a form which will run on the client, referencing ClassFactory again from a class running on AOS may not have the same result as the two instances are operating separately. The purpose by this construction is to lower the number of calls between client and server.

Global

All methods in the Global class are static methods. You can create instance methods in the Global class. However using an instance method would require declaring the class. The idea with Global is to have a collection of function create in X++ which can be referenced without specifying the class name. Methods in global are referred in the same way as a system function.

Several of the methods are a supplement to the functions used for base type operations like the Global method `date2StrUstr()` which convert a variable of the type date to a string formatted by the user default date settings.

You can add your own methods to the Global class. This can be useful if you have a piece of code often used. Before adding your own Global method, you should check the existing Global methods and the system functions located under *System Documentation/Functions* as you might find a function already solving your needs.

Info

The Infolog system can be addressed using the class Info. An instance called infoLog of the Info class is declared on startup of Axapta. You use the Info class when printing to the Infolog. You should never reference the Info class directly as this is done by using the global methods info(), warning() and error().

The Info class has other purposes as the class is being used by the Document handling system and for calculation of user licenses.

You can find more information on how to use the Infolog in the chapter **Intro to MorphX**.

5.5 System classes

System classes are used as application classes. A system classes can be inherited as any application class. Some of the application classes in the standard package are subclasses of a system class. As you cannot modify system classes inheriting a system class makes it possible to add additional logic to a system class. Classes like SysDictTable or SysReportRun are subclasses of system classes.

The types of often used system classes are described in the following sections.

Object

The system class Object is the base for all class in MorphX. You can compare Object with the system table Common. Any class can be declared using the class Object. As MorphX will validate methods of an object at compile time using the class Object it is quite useful as the type of the object will not have to be known until runtime. This can be used to execute your own method of a form or a report, where the name of the form or report is not known until runtime. For an example on how to accomplish this, see the chapter **Forms**.

Runtime changes

One of the primary uses of system classes is the ability to address any node in the AOT. This is very useful, especially for forms and reports as you can override any property or method at runtime. Throughout the book you will find examples on how to use system classes for addressing any object in the AOT making your code more dynamic. System classes used for addressing AOT nodes are usually prefixed with the corresponding AOT node like all system classes for forms are prefixed with Forms*.

Args

You might have noticed the system class Args have been used as a parameter variable. A run able class must have Args as parameter in the main() method. The purpose of Args is to transfer parameters between objects. If you are calling a run able class from a form you can use Args to pass a formRun object making it possible to query data in the form, or even execute methods in the calling form.

Example 5: Args

Objects used from MORPHXIT_Classes project

- Class, MyClass_Args
- Menu item action, MyClass_Args
- Form, CustTable

A new menu item will be added to the customer form. The class called by the new menu item will print the selected record in the form.

1. Create a new class and rename the class “MyClass_Args”.
2. The class has only the main() method making the class run able. Args is used to make a check whether the calling object has the table CustTable. The caller could be any form as long as the calling form uses CustTable. If validated true, the table variable CustTable is initialized with the calling record and the result is printed.

```
static void main(Args _args)
{
    MyClass_Args    myClassArgs = new MyClass_Args();
    CustTable       custTable;
;

    if (_args.dataset() == tablenum(CustTable))
    {
        custTable = _args.record();
        info(strfmt("Customer selected: %1", custTable.name));
    }
}
```

3. Save the class and create a menu item for MyClass_Args.
 4. Go to the *Forms* node in the AOT and locate the form CustTable. Expand the form and drill down the design. Drag the menu item for MyClass_Args to the node *Design/ButtonGroup:ButtonGroup* and save the form.
-

When running the form CustTable you will see the form has a button with the same name as the class. No label was specified for the menu item, so the class name is shown instead. Execute the class by clicking the button, and the name of the customer selected is printed in the Infolog.

This was just a simple example on the use of Args. You could have used Args for calling a report and add filters to your report only to print selected records. Args is often used for passing parameters to a class to have the heavy tasks executed by a class. As a form always run on client, this will make it possible to put the workload of a form on the AOS. By browsing the AOT, you will find plenty of examples using Args.

Foundation Classes

MorphX have a group of system classes which can be used as an alternative to complex data type. These are called foundation class. Characteristics for foundation classes are that they can store a dynamic list of variables. Values are stored in memory so they are very fast to use. Compared to a temporary table which swaps data on the disk, a foundation class will perform significantly better.

Five different foundation classes are implemented in MorphX: Array, List, Map, Set and Struct. The foundation classes are designed to use any type. A useful thing is that you can use a foundation class as a parameter for a method or having a method returning a foundation class.

You can find useful examples on the use of the foundation class by locating them in the AOT and press F1 to see the online help.

Optimized Record Operations

If you have a task which fetches the same records several times you should consider using a record sorted list for better performance. The case could be that you are printing a lot of records, but before printing you have a class updating the same records.

Normally you would have to fetch the records in your class and when calling your report, the reports query would fetch the records second time. To prevent fetching the records the second time, you can use the system class RecordSortedList. Posting and printing sales invoice is one of the places in the standard package using RecordSortedList.

If you do not need your records to be sorted, or if the fetched data are already sorted in the preferred order you would be better off using RecordLinkList instead.

The system class RecordInsertList is an alternative to the select keyword `insert_recordset`. However `insert_recordset` is preferable.

For examples of the use of optimized record operations, take a look at the online help.

File Handling

Importing data from a simple text file is pretty simple as MorphX have a set of system class handling files. The base class for file handling is the system class Io. The system classes AciiIo and CommaIo are subclass of Io used for text files.

```
static void Classes_CommaIo(Args _args)
{
    CommaIo      fileOut;
    FileName     fileName = "c:\\Customers.csv";
    CustTable    custTable;
;
    #File

    fileOut = new CommaIo(fileName, #io_write);

    if (fileOut)
    {
        while select custTable
        {
            fileOut.write(custTable.accountNum,
                          custTable.name,
                          custTable.custGroup,
                          custTable.currency);
        }
    }
}
```

The example shows the use of CommaIo. CustTable is looped and written to a comma separated file. A macro constant from the macro library File is used for setting write access when declaring the CommaIo file. The method write() in the CommaIo class will take any number of fields as parameters. Note that double backlash must be used when referring to a path in the file system. In MorphX you will not have to close the file, as this is automatically done, when the code initializing the file is out of scope.

Note: If you have to export complex data types such as containers, this can be done using the system class BinaryIo. Alternatively you will have to export your data as XML.

5.6 Special Use of Classes

In this section examples on how you can use classes for more special cases are shown. Several of the chapters in this book have examples like this, also using classes for integrating to external products or showing some of the classes you might not notice as they have a special purpose.

Using COM

The Business Connector also referred to as the COM connector can be used to integrate Axapta with an external product. In this example Microsoft Outlook is accessed from Axapta using COM. To use this example you must have at least one licensed COM user.

Elements used from MorphxIt_Classes project

- Job, Classes_ReadFromOutlook

```
static void Classes_ReadFromOutlook(Args _args)
{
    SysOutlookApplication sysOutlookApplication;
    SysOutlook_NameSpace sysOutlookNameSpace;
    SysOutlookMapiFolder sysOutlookMapiFolder;
    SysOutlook_Folders sysOutlookFolders;
    SysOutlook_Items collection;
    COM message;
    Notes messagebody;
;

    #sysOutLookComDef

    sysOutlookApplication = new sysOutlookApplication();
    sysOutlookNameSpace = sysOutlookApplication.getNameSpace("MAPI");

    sysOutlookNameSpace.logon();
    sysOutlookFolders = sysOutlookNameSpace.Folders();
    sysOutlookMapiFolder =
sysOutlookNameSpace.getDefaultFolder(#OlDefaultFolders_olFolderInbox);

    collection = sysOutlookMapiFolder.Items();
    message = collection.GetFirst();

    while (message)
    {
        info(message.subject());

        message = collection.GetNext();
    }
}
```

Application classes prefixed with SysOutlook* are used to access Microsoft Outlook. The class SysOutlookApplication will open a COM connection and SysOutlookNamesSpace will logon to Microsoft Outlook. A warning will appear in Axapta and you will have to accept to access your mail client. The macro library SysOutlookComDef contains a list of macro constants use for Microsoft Outlook. The default folder Inbox is selected in Microsoft Outlook and subject for all mails in the Inbox are printed to the Infolog.

This example is only reading from Microsoft Outlook. However you could as well be writing or synchronizing your mail client with data from Axapta. The form

HRMInterviewTable is using Microsoft Outlook for creating appointments in the calendar based on data keyed in from Axapta.

Note: If you are going to interface an external system you can use the COM Class Wrapper Wizard to create a COM wrapper for your external system. The SysOutlook* class are made using this wizard.

X++ Compiler

MorphX has a system class called XppCompiler which can be used to compile X++ code written as text. You might wonder what is the use of this as you have everything integrated in MorphX, so why bother? This could be a way of giving administrators of a system the option to write their own validations using X++ code without going to the AOT making modifications. In a controlled environment you could make your modifications more flexible by allowing the user to handle simple adjustments.

There are places in Axapta where the user has some hardcode variables which can be used in a text field like the form TransactionsTexts. What if instead of using the hardcode variables you could specify your own variables written in X++? This could be pretty neat, and by using the class XppCompiler you can achieve this.

Elements used from MorphxIt_Classes project

- Class, Classes_XppCompiler
- Menu item output, Classes_XppCompiler

A class with two methods is created. The class must be runnable.

```
Notes buildFunction()
{
    TextBuffer    textBuffer = new TextBuffer();
;

    textBuffer.appendText("static void test(Counter _counter, CustTable _custTable)");
    textBuffer.appendText("{}");
    textBuffer.appendText("while select _custTable");
    textBuffer.appendText("{}");
    textBuffer.appendText("info(_custTable.name);");
    textBuffer.appendText("_counter++;");
    textBuffer.appendText("{}");
    textBuffer.appendText("info(strfmt(\"Customers printed: %1\\", _counter));");
    textBuffer.appendText("{}");

    return textBuffer.getText();
}
```

First a method creating the function code to be used by the class XppCompiler is build. This function will take two parameters, a counter and the table CustTable. All records in CustTable will be looped. Customer name and total number of customers will be printed to the Infolog.

The system class `TextBuffer` is used to build the string containing the function. Adding strings using `+` sign is very slow. `TextBuffer` should always be used when adding several strings.

```
static void main(Args _args)
{
    MyClass_XppCompiler myClassXppCompiler = new MyClass_XppCompiler();
    xppCompiler          compiler = new xppCompiler();
    Notes                codeString;
    CustTable            custTable;
;

    codeString = strfmt(myClassXppCompiler.buildFunction());

    if (compiler.compile(codeString))
    {
        runbuf(codeString, 0, custTable);
    }
    else
    {
        info(compiler.errorText());
    }
}
```

The `main()` method will declare the compiler class `XppCompiler`. If the code built in the `buildFunction()` is compiled without errors the function `runbuf()` will execute the code. The first parameter of `runbuf()` is the code string and the following parameters are the parameters specified for the build function. All variables used in the built function must be specified as parameters as you cannot declare any variables.

5.7 Summary

Classes are fundamental in MorphX. The application classes are used for creating the business logic. System classes bind the kernel with the application. This chapter should have clarified the differences and explained how classes are used. You should by now have required knowledge of the options when using classes in MorphX. In the following chapter you will see how you can combine the use of classes for designing the user interface.

6 Forms

Forms are the most important part of the user interface as a form is the link between the application user and the database. When an application user insert, update or delete data, forms are used. The database can be accessed using the table browser, but you should never consider this as an option in a live system as you might mess up your data. A form will typically do additional validations compared to the table browser, besides a form will structure the data fetched making it easy to use.

A form has a lot of features useable for the application user like sorting, filtering and personalizing the appearance of a form. Some of those features available for the application users will be mentioned in this chapter. However keep in mind the focus is how to use MorphX to construct forms.

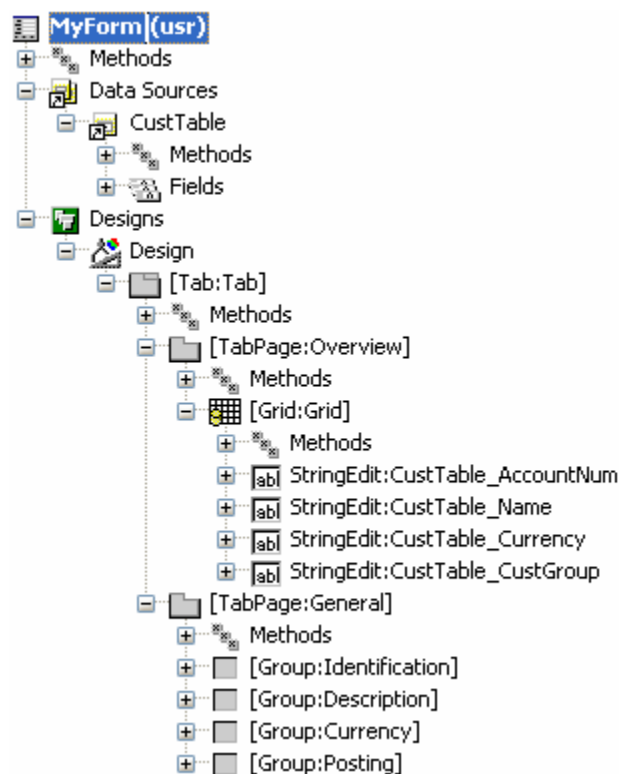


Figure 22: Form overview

6.1 Creating Forms

An Axapta form is created in the AOT using the node *Forms*. A form consists of two parts, the query fetching data and the design which is used for defining the layout of the form. The design will normally be used to present the tables fetched in the query.

Example 1: My first formElements used from MORPHXIT_Forms project

- Form, MyForm
- Menu item display, MyForm

To get experience on how to use forms start creating a form as shown in **Figure 22: Form Overview**. This is a simple form which will learn you the basic steps creating a form. In the following examples details will be explained and more features will be added to this example. For simplicity no labels are used in this example, however labels should always be created for your modifications.

1. Right-click the AOT node *Forms*, and select *New Form*. Rename the form to "MyForm" using the property sheet.
2. Expand the nodes for the new form so the nodes *Data Sources* and *Designs* are visible. Open another windows of the AOT, located the table SalesTable and drag SalesTable to the form node *Data Sources*.
3. Go to the node *Designs/Design* and open the property sheet. In the property **Caption** enter the text "Sales orders". Select the data source SalesTable in the property **TitleDatasource**.
4. Right-click the node *Designs/Design* and select *New Control/Tab*. Go to the property sheet for the tab control and set **Width** to "Column width" and **Height** to "Column height".
5. Now add a tab page by right-clicking the node *Designs/Design/[Tab:Tab]* and select *New Control/TabPage*. Rename the new tabpage "Overview" by using the property sheet. Enter a label for the tab page by typing "Overview" in the property **Caption**.
6. Add a grid control to the tab page Overview by right-clicking *Designs/Design/[Tab:Tab]/[TabPage:Overview]* and select *New Control/Grid*.
7. Go to the new grid control and set the properties Width and Height to "Column width" and "Column height".
8. Expand the data source SalesTable and pick the fields SalesId, CustAccount, CurrencyCode and SalesStatus. Drag the fields to the grid control in the mentioned order. This is easiest done by right-clicking the data source SalesTable, select *Open New Window* and drag the fields from the new AOT window.
9. Add a second tab page by right-clicking the node *[Tab:Tab]*. Rename the new tab page to "General" and set the Caption property to "General".

10. Go to the data sources SalesTable and locate the *Fields* node of the data source. The field groups defined in the data dictionary for SalesTable are listed right after the fields. Select the field groups Identification, Currency, Customer and Status. Drag the field groups in the mentioned order to the design node *Designs/Design/[Tab:Tab]/[TabPage:General]*.
 11. Save the form. You have now created your first form!
 12. Create a menu item for the form by dragging the form MyFrom to the node *Menu Items/Display*.
-

The form just created is an example of a common form in Axapta. No data connection needs to be configured. Just pick the tables to be used and drag the tables to the query part of the form. Creating the layout of the forms is really simple as MorphX will auto positioning controls. When dragging a field to the design a control is automatically created and the control is auto arranged according to the other control.

The best part is that not a single line of code was written to create this form, only a few properties were set. You will often have to add code to your forms, but the example shows how easy it is to presents the data of table. Designs are always auto positioned, even for complex forms and this is really timesaving.

The properties set in the design for this example are properties always set for a standard form. At the node Design the text for the caption bar shown in top of a form was specified. The property Caption specifies the form name and the TitleDatasource property will shown the value of the fields specified in the tables' properties TitleField1 and TitleField2 positioned after the caption name.

Setting the properties Width and Height to Column width and Column height will allow the application users to resize the form. Width and Height are set for both the tab control and the grid control to make it possible to resize the grid control too.

Fields should always be added to forms using field groups as it makes it easier changing which fields to be shown on forms using the data dictionary. A field group control has a property called **AutoDataGroup**. Changes made to the data dictionary field groups will automatically be reflected by form and report field groups with the property AutoDataGroup set to Yes. This makes it easy to add a new field to an existing field group as only the data dictionary has to be changed.

To simplify this example fields groups was not used for the grid control. You can either drag field groups to a grid control or specify a field group in the property **DataGroup** on the grid control. A table will typically have a field group called Overview, which contains the fields to be used by the grid control.

Note: If you want to discover more of the features available with forms you can try running the forms prefixed with tutorial*.

6.2 Form Query

A query contains the tables to be used by a form. It is not mandatory to have a query defined for your form. You could just fetch the data for your form using selects. However a query should always be the first choice for objects accessed by the application users as the query provided several features for the application users such as filtering, sorting and printing data.

The nodes used for constructing a form query in the AOT differs a bit from queries built other places in the AOT such as queries in reports. Where a report query is logic built with tables presented at their respective join level in the tree, a form query is listing all tables at the same level in the tree under the node *Data Sources*. When adding a table to the data source node of a form, you will be adding a form query data source. This can be a bit confusing when switching between developing forms and reports. You will soon adopt these differences as properties are similar. From X++ queries are addressed in the same way no matter which kind of object used.

For more information about the basics of queries, see the chapter **Queries**.

Joining Data Sources

The most common way of filtering the records to be listed in a form is by joining the data sources. This could either be accomplished by joining the forms data sources or if the form is called from a related form which join the forms data sources with the calling forms data sources.

Example 2: Outer joined form

Elements used from MORPHXIT Forms project

- Form, MyForm_OuterJoin
- Menu item display, MyForm_OuterJoin

In this example the form created in example 1 will be extended to show data from the employee table. No labels are created in the example.

1. Duplicate the form MyForm, and rename the form to “MyForm_OuterJoin”.
2. Expand the *Data Sources* node of the new form, open another window of the AOT and find the table EmplTable. Drag EmplTable to the *Data Sources* node.
3. Open the property sheet for *Data Sources/EmplTable*. Pick SalesTable in the property **JoinSource**. Set the property **LinkType** to OuterJoin.
4. Select the field SalesTaker from SalesTable and drag the field to *Designs/Design/[Tab:Tab]/[TabPage:Overview]/[Grid:Grid]*.
5. Repeat step 4 by adding the field Name from EmplTable to the grid.

6. Select the field group Name from EmplTable and drag the field group to the tabpage control General.
 7. Go to the tabpage General and set the property **Columns** to 2.
 8. Save the form and create a display menu item for the form.
-

The form MyForm_OuterJoin has two additional fields in the grid showing the employee id and name of the sales receipts. The employee name is fetched from the employee table. Still the same numbers of records are fetched, as the data sources are joined using an outer join. Using an inner join would have resulted in only sales order with sales receipts would have been shown. Try changing the property LinkType for the data source EmplTable to see how data are fetched.

When adding data sources to your form by dragging a table from the data dictionary, the name of the data source will be equal to the table name. You can however change the name of the data source. If you are joining the same table more than once in a query, the data source must have difference names. This is just like declaring two table variables of the same table from X++.

The property Columns changed for the tabpage named General will divide the field groups into two columns. By default all field groups will be listed in one column, so the number of columns should be adjusted if you form contain several field groups.

Note: If you are unsure about how two tables are related try creating an AOT query with the two tables to be joined. In the AOT query the relations will automatically be shown by setting the property Relations to true at the joined data source.

Joining tables to present data from several tables is very performance friendly, and should be considered when joined data where a 1-1 relation exist. For the application users there will be no differences whether fields are from one or more tables. The application user might even not be aware of that data are fetched from several tables. As data from both SalesTable and EmplTable were presented in the grid control in the example, the join mode must either be inner join or outer join. An exist join or not exist join would have caused data not to be shown from the joined table.

Form Link Types

Join mode of a form is set by using the data source property **LinkType**. Besides the standard join modes, forms have 3 additional join modes which are used to makes forms more users friendly: Passive, Delay and Active. When using an inner join or an outer join all records from the joined data source is always fetch immediately. This is necessary when joining several data sources in a grid. If data from the joined data source is not in the same grid control, you should consider using one of the special form join modes.

The default value of LinkType is Delayed. This is the most used join mode when joined data sources are not listed in the same grid control. Joining delayed will make a delay before fetching data from the joined data source. This is quite useful and will speed up scrolling through the records in a grid control, making the form more users friendly. It is only a matter of milliseconds and the application user will hardly know that data is fetched delayed. The form SalesTable makes use of this technique.

Note: Using the join modes Delayed, Active and Passive will always cause data sources to be joined as an outer join. All records will be fetched from the joined data source.

The LinkType Active is similar to Delayed. The only difference is that Active has no delay before fetching data from the joined data source. Active is not used very often as it will not give the application users the same flexibility as when joining delayed. Joining Passive is the opposite of Active. Where Active will always fetch data from the joined data source, using Passive will not fetch any data from the joined data source. Passive can be used if you want to control when data is fetched using X++.

Linking Forms

Relations defined in the data dictionary are automatically used for linking forms. Say you are creating a new form to be called from the customer form, showing customer data. If the table for your new form has a field using the extended data type CustAccount, your new form will automatically be linked to the customer form. The extended data type CustAccount has a relation to the table CustTable used by the customer form. When calling your new form from the customer form, the extended data type is used by MorphX to create a link between the two forms. This type of linking is referred to as dynalinks.

Try opening the customer form CustTable and open the customer transactions form by clicking the button Transactions in the customer form. When stepping the customer records in the customer form, the customer transaction form will automatically fetch data for the current customer. This is done without any code and is the default behavior when creating forms using related tables.

Some cases might require disabling dynalinks. You might need to show all records in the related form, or make an alternative linking between the two forms.

```
void init()
{
    super();

    this.query().dataSourceNo(1).clearDynalinks();

    criteriaOpen = this.query().dataSourceNo(1).addRange(fieldnum(CustTrans,Closed));

    custTransDetails = new CustTransDetails(custTrans);
}
```

To disable dynalinks for the form customer transactions locate the form CustTrans in AOT and edit the method init() located under *Data Sources/CustTrans*. After the super() call in init() add a call to clear dynalinks as shown in the above code block. Now when opening the customer transaction form from the customer form no filtering is done and all records are shown.

The reason for disabling a dynalink could be that you need to show all records in the related form as the related form is used to lookup all records. This could be the case if calling a form for a lookup button.

Instead of using the dynalink based on the data dictionary relation, you can create your own link. Try extending the init() method so it looks like the following:

```
void init()
{
    super();

    this.query().dataSourceNo(1).clearDynalinks();
    this.query().dataSourceTable(tablenum(CustTrans)).addDynalink(
        fieldnum(CustTrans, CurrencyCode),
        element.args().record(),
        fieldnum(CustTable, Currency));

    criteriaOpen = this.query().dataSourceNo(1).addRange(fieldnum(CustTrans,Closed));

    custTransDetails = new CustTransDetails(custTrans);
}
```

After the call of clearDynaLinks() a new dynalink is built using the addDynalink() method. You must specify the field in the current data source to be linked with the calling data source, and the calling record. Here element.args().record() is used to return the calling record.

By adding this dynalink the customer transactions will now be filtered based on the customer currency code instead of the customer account. This will of course not make sense in a live application. The point is to show how to override default dynalinks.

Note: If you are unsure of which fields the dynalink uses for filtering, check the caption bar of the form. The dynalink fields are listed as the last part of the caption text after the form name and the title1 and title2 fields.

Setting Access

General restrictions for tables and fields are defined in the data dictionary. Restrictions specific for a form should be defined at the form data source. You cannot override the settings made in the data dictionary, but you can add additional access restrictions.

Access restrictions can be set at both data source and field level of a data source. For a table you specify the max access mode which corresponds to the properties AllowEdit, AllowCreate and AllowDelete for the data source. At the data source field level you can set restrictions for single fields, such as defining whether the field should be editable or

visible. Restrictions can also be set at controls in the form design. However restrictions should always be set at the data sources is possible as the same field can be used for more than one control in the design. It is preferred not to not have code in design, or modify default settings in the design. Programming in MorphX tends to not have any code in forms, making it easy to switch the user interface. You cannot prevent modifying your design as if a control in the design is not bound to a data source field, you will have to set access restrictions in the design.

Some access restrictions are typically set using the properties. A form like CustTrans showing customer transactions should not be editable. Often you will need to set restrictions depending on the caller or depending on the single records in the form.

Example 3: Setting AccessElements used from MORPHXIT_Forms project

- Form, MyForm_SettingAccess
- Menu item display, MyForm_SettingAccess

The example will show to set access restriction for a form data sources from X++. Changing restrictions from X++ will normally be based on a condition. To simplify the code no conditions has been added.

Duplicated the form MyForm, and rename the new form to “MyForm_SettingAccess”. Override the form init() method and add the following:

```
public void init()
{
    super();

    salesTable_ds.allowCreate(false);

    salesTable_ds.object(fieldnum(salesTable, CurrencyCode)).allowEdit(false);
    salesTable_ds.object(fieldnum(salesTable, SalesStatus)).visible(false);
}
```

Save the form and create a display menu item for the form.

When executing the form you will not be able to add new records. The field CurrencyCode cannot be edited and the field SalesStatus is not shown in the design.

Any properties and methods of a data source can from X++ be reference using the name of the data source suffixed with _ds. When referring to the data source fields, you must use the data source method object(). By using the table field id as parameter for object() you will be able to reference the data source fields. In the standard package you will find several examples where access restrictions are set using controls in the design rather than using the data sources fields. This is not optimal, but the reason for not using

data source fields is that this option was first introduced in one of the latest versions of Axapta.

6.3 Design

Some development environments are really time wasters when doing the layout of forms, as you will have to adjust controls manually. This is not the case with MorphX. In fact doing the layout of a form in MorphX is pretty straight forward. Best practice recommends not having code in design and if possible auto position controls. This speeds up development. When the query part of a form has been created, you can do most of the layout of the design by dragging field groups from the form data sources to the design.

You will have cases where code is needed in your design. And some basic adjustments should be made to the properties for the controls of a design. This is though often minor settings required.

Auto positioning controls in the design benefits in several ways. When adding a field or a field group to the middle of a row of fields, the following controls will be positioned accordingly. If a field is removed or added to a field group on a table, the related field group in the design will automatically be updated. Setting a field to invisible will also caused the following fields for field controls to be adjusted. If no fields of a field group are to be shown, the field group will automatically be invisible.

Using auto settings will make it more difficult to do a special design for a form. You can however disable any auto settings and positioning every control manually. The idea with auto settings are that it should be quick and easy designing forms, and make forms appears with a standardized look.

Creating Design

Some basic rules should be followed to have a standardized look and feel of your forms. A typical form is shown in **figure 23: Standard form**. A form should be divided into tab pages, where the first tab page normally called Overview should have a list of available records. The fields shown at the first tab page should be defined in a table field group. If a field is set to be mandatory, meaning that the field must be filled out, the field should be shown at the first tab page. At the following tab pages, the remaining fields from the data sources should be shown grouped by tab pages with logical names. Fields which logical do not fit into a separate tab page are usually put at the second tab page which is then named General. Fields should if possible be added using the table field groups. Common practice is to add fields shown at the first tab page on the following tab pages too. The first tab page is often a list of the most important fields of a table, and these fields are typically part of another field group, so these fields will appear twice.

Even if you create a simple form with only a few fields, you should consider following this practice. You will save a little time just adding a single control to your form to show the fields from the data source. This will however have you form looking different and will break from the standard layout of forms.

Buttons are typically added to the right side of the form. It is preferable to use menu items for a button as menu items are automatically validate by the setting of security keys and configuration keys. For more information on creating menu items, see the chapter **Menus and Menu Items**.

Item number	Item name	Search name	Item group	Item type
B-R14	Battery Baby R14	BatteryBabyR14	Parts	Item
B-R6	Battery Penlight R6	BatteryPenlightR6	Parts	Item
ESB-005	Energy Saving Bulb 5 Watt	EnergySavingBulb5Wat	Bulbs	Item
ESB-007	Energy Saving Bulb 7 Watt	EnergySavingBulb7Wat	Bulbs	Item
ESB-009	Energy Saving Bulb 9 Watt	EnergySavingBulb9Wat	Bulbs	Item
ESB-011	Energy Saving Bulb 11 Watt	EnergySavingBulb11Wa	Bulbs	Item
ESB-013	Energy Saving Bulb 13 Watt	EnergySavingBulb13Wa	Bulbs	Item
ESB-015	Energy Saving Bulb 15 Watt	EnergySavingBulb15Wa	Bulbs	Item
FL-Penlight	Flash Light Penlight	FlashLightPenlight	Parts	Item
FL-Standard	Flash Light Standard	FlashLightStandard	Parts	Item
FLL-2500	Floor Lamp 2500 Color	Floorlamp2500Color	Lamps	BOM
FLL-MeasureConfig	Floor Lamp w MeasurementConfig	FLL-MeasureConfig	Lamps	BOM
FT-018	Fluorescent Tube 18 Watt	FluorescentTube18Wat	Bulbs	Item
FT-036	Fluorescent Tube 36 Watt	FluorescentTube36Wat	Bulbs	Item

Figure 23: Standard form

The design of a form is created under the node *Designs*. Do not get confused by the name of the node, as a form can only have a single design.

You can view the design of a form by double clicking the design node. This will open the form in design mode as seen in **figure 24: Form edit mode**. The form edit mode is intended to be used for creating and editing the form design. However you should just consider using this graphical presentation for getting an overview of your design, as the tool is pretty basic. Use the form tree nodes instead when creating and modifying your form design.

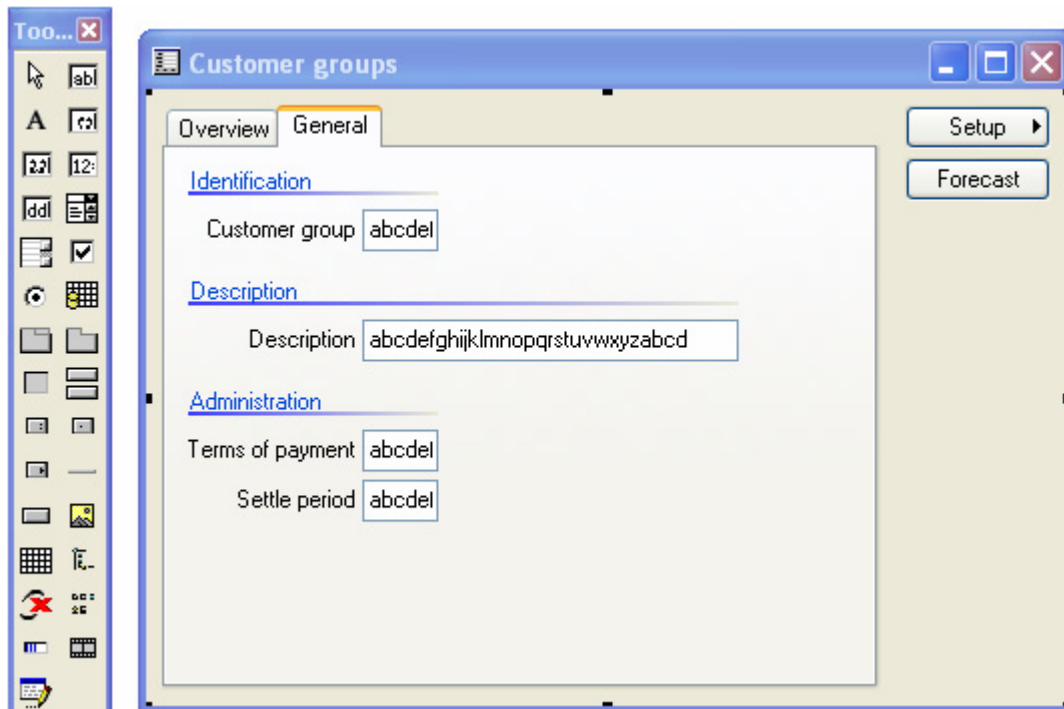


Figure 24: Form edit mode

When opening the design of a form in edit mode a toolbar will be opened next to your form. You can use the toolbar to add controls by selecting a control in the toolbar and click on the form design to position the selected control. To remove a control from the design, selected the control and press the delete key.

Viewing a form design in edit mode is useful when getting to know the single parts of a form design, as you can navigate around your form design. Especially to get an overview of a form with a lot of controls like the form SalesTable. When clicking a control in the design, the property sheet will be active for the selected control. Until you get familiar with the form controls this can be a quick way to located the properties of a certain control.

Controls in Design

Creating the design is done by adding the controls to be used in the design, either by dragging data source fields and field groups to the design or by manually adding a control. When dragging from the data sources, a control of the corresponding type will automatically be created with properties set to the related data source. This is the fastest way to built you design and it will assures that your controls are created correctly. Controls can be created manually. This is typically done for controls which are not bound to a data source. Notice that fields and field groups cannot be dragged from the table, as in a report design.

You should consider using a descriptive name for your controls. Creating a control manually will name the control of the control type with a consecutively number. Modifying a design where controls are created with the default names will make the design more difficult to understand as you will have to expand all the nodes to get an overview. Take a look at **figure 25: Bad naming of controls**. And this is just a simple form.

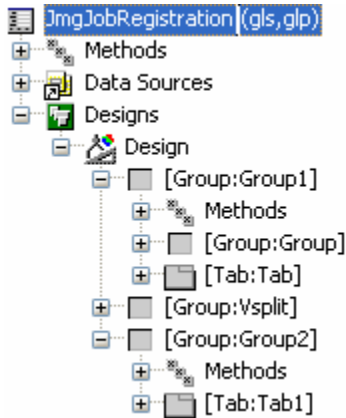


Figure 25: Bad naming of controls

Labels used for fields and field groups are specified at the extended data types, base enums or at the table. You will only have to specify labels for controls organizing fields and field groups such as tab pages and grid controls. If you need another label for a field or a field group you can override the default label by specifying another label for the control. This is not recommended as you will disable auto settings and this will make your form more time consuming to maintain.

For an overview of the available controls for a form design, see **figure 26: Form controls**.

Name	Description
ActiveX	Used to integrate an ActiveX control in a form. See form KMKnowledgeAnalogMeter.
Animate	Used for playing movie clips.
Button	A simple button control. Use the MenuItemButton control where possible, as the method clicked() on this control must be overridden to have a job executed.
ButtonGroup	A buttonGroup control is used for grouping any kind of button controls.
CheckBox	Returns a true or false value. Often used to set the value for a NoYes enum.
ComboBox	The most common control used to show the entries of a enums. Presents the enum with the available values shown in a drop down list.
CommandButton	A special type of button control. The control has a property called Command, where the action for the button is specified. Normally the control would be used for adding Ok, Cancel and Apply buttons to a form. See form ReqTransPoCreate.
DateEdit	Used for showing dates. Dates will be formatted accordingly to the Windows regional settings.
Grid	Used to list available records in the form. A grid control is typically added to the first tab page. The properties Width and Height should always be set to Column width and Column Height making it possible to resize the grid.
Group	Group control is commonly used for showing table field groups. A Group control can also be used to arrange other group controls to present groups in a specific number of columns.
HTML	Can be used to append a HTML content to a form. See form ForecastItemAllocationDefaultDataWizard.
IntEdit	Used for integer values.
ListBox	A simple edition of the ListView control. Can be used for enums. This control is rarely used.
ListView	Often used where a fixed number of values can be selected, where one ListView control contains all possible values and another ListView control contains chosen values. Data must manually be filled in a ListView. See form KMAcctionType.

Name	Description
MenuButton	A MenuButton control is used for organizing buttons in sub menus. Validations for enabling buttons in a sub menu are often placed at the MenuButton clicked() method.
MenuItemButton	Used for creating a button for a menu item. MenuItemButton is the preferred button type as no code is needed for executing the related menu item. The properties MenuItemType and MenuItemName defines the object to be executed. Normally created by dragging a menu item to the form design.
Progress	Normally progress bars are created using the application classes SysOperationProgress. This control can be used to integrate the progress bar in the form. Use the progress control method pos() to update the counter.
RadioButton	Alternative to the ComboBox control for showing enums. Each entry of the enum will be shown as a select able button.
RealEdit	Used for real values.
Separator	A control used by the menuButton control. Will separate buttons in a sub menu by adding a horizontal line.
StaticEdit	Shows a non editable text. The control can be bound to a data source field. Normally the text for the control is set in the property sheet or from X++. Notice, the control cannot be used in a grid control.
StringEdit	Used for string values. If used for memo fields, the properties Width and Height should be set to Column width and Column height to allow resizing the memo field.
Tab	This is the top level control of a standard form design. A tab control is used for adding tab pages. The properties Width and Height should always be set to Column width and Column Height making it possible to resize tab pages of the tab control.
Table	Fairly not used. Can show multiple records as a grid control. Data must be filled in the control manually.
TabPage	Tabpages are added to a tab control. A tabpage is used for storing any control containing data. The design of a form will typically consist of one or more tabpages.
TimeEdit	Control for showing the time. Time will be formatted accordingly to the Windows regional settings.

Name	Description
Tree	Used for showing records in a tree view. The logic for handling the tree is put in an application class. See form ProjTable.
Window	Can be used when adding a bitmap to a form. See form KMAction.

Figure 26: Form controls

Grid Control

The grid control has several features which are nice to know as it can make your modifications more users friendly. You can mark a number of records in a grid control in the same way as marking files in the Windows file system. This is useful if you need to do a certain task on a number of records like calling a run able class updating the selected records. A checkbox control is often added to the grid control for controlling which records are marked. If you do not need to save information about which records are marked using the mark feature is much better.

Example 4: Multi mark

Elements used from MORPHXIT_Foms project

- Form, MyForm_MultiMark
- Menu item display. MyForm_MultiMark

This example will show how to loop the marked records in a grid control.

1. Duplicate the form MyForm, and rename the form to “MyForm_MultiMark”.
2. Go to the node *Designs/Design*, right-click and select *New Control/ButtonGroup*.
3. Add a button to the control ButtonGroup by right-clicking *ButtonGroup* and select *New Control/Button*. Go to the property sheet for the new button control and change the name of the control by entering “MultiMarkTestButton” in the property **Name**. Set the property **MultiMark** to Yes. In the property **Text** enter “Show marked”.
4. Now override the method clicked() on the button control *[ButtonGroup:ButtonGroup]/Button:MultiMarkTestButton/Methods* and add the following code:

```
void clicked()
{
    salesTable salesTableMarked;
;
    super();

    if (salesTable_ds.anyMarked())
```

```
{
    salesTableMarked = salesTable_ds.getFirst(1,false);

    while (salesTableMarked)
    {
        info(salesTableMarked.salesId);
        salesTableMarked = salesTable_ds.getNext();
    }
}
```

5. Save the form, and create a display menu item.

Try opening the new form MyForm_MultiMark. Mark some records in the grid and press the button Show marked. The value of sales id for the selected records will be printed in the Infolog. In a real case you would have used a menu item button calling a run base class and have the class processing the records. Only buttons with the property MultiSelect set to Yes can be used when marking more than one records. This is nice, as you will not have to validate each single menu item whether it is called with more than one record. The form SalesTable in the standard package makes use of multi marking for posting.

Note: A grid control only loads part of the records available in the form. This means that the vertical scroll bar in a grid is positioned based on the number of cached records. This is done for better performance.

Another nice feature using a grid control is that marked records can be copied and pasted directly to Microsoft Excel. However only records bounded to a data source field can be copied this way.

Both records and single fields can be colored in grid. Using colors can make your forms more users friendly as it can be used to monitoring values of certain fields. You can find examples on the use of colors in forms in the sections **Special Forms**.

Tree Control

If records in your data source are related hierarchically you should consider using a tree controls. This will allow the application user to have a logical view of the records. Forms like ProjTable and HRMOrganization makes use of the tree control.

Using a tree control can be a bit tricky, especially if using tree control for viewing data as you will have to add some code to make it work. It is far easier to duplicate and existing form using a tree control instead of trying to built your own from scratch. A class is used to control the data in the tree control. You will find two versions of this class in the standard package. The simplest class called FormTreeDatasource is used by the form ProjTable. The form HRMOrganization is using a more advanced class called CCFormTreeDatasource. The advanced version supports among others drag and drop.

A form using a tree control is often large and when the application user changes the size of such a form the tree control might not be shown properly. This can be solved by adding a 'splitter' between the tree control and the other field controls in the form. You will then be able to click the splitter and expand the size for either the tree control or the other field controls. A splitter is not an ordinary control, as it is built using a class. If you want to make use of a splitter, copy the code from a form already using a splitter.

Figure 27: Splitter control is showing where a splitter control is added to the form HRMOrganization. You use a field group control for creating a splitter and overrides the methods `mouseUp()`, `mouseMove()` and `mouseDown()`. Make sure to check the properties of the field group `SplitControl` as you will have to change several properties.

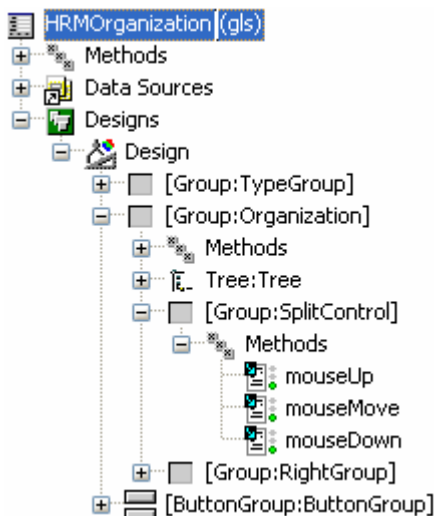


Figure 27: Splitter control

Display and Edit Modifiers

MorphX has two types of special method modifiers used by the user interface. These are used if you need to show a value from a field which is not part of the query. You might have a case which requires that you will have to build a complex query to join a certain table field, or there might not be any relation to the required table field at all. To get the field value in a simple way the modifiers display and edit can be used.

These two types of methods are normally referred to as display methods and edit methods. Display and edit methods can be used in any content for returning the value of a type. The modifier display and edit just tells that the method is meant to be used by the user interface.

Before making your choice on using a display or edit method you should be aware that you will not be able to sort a grid by controls using a display or edit method. Sorting fields in a grid by clicking the label name of a field in a grid is commonly used by

application users. An application user will not know whether a control is bound to a data source or the control is based on a display or edit method.

Example 5: Display and edit

Elements used from MORPHXIT_Forms project

- Table, SalesTable
- Form, MyForm_DisplayEdit
- Menu item display, MyForm_DisplayEdit

This example will show how to use the user interface modifiers, display and edit. MyForm will be extended to have controls using both display and edit modifiers.

1. Duplicate the form MyForm, and rename the form to “MyForm_DisplayEdit”.
2. Open another window of the AOT and drill down to the methods of the table SalesTable.
3. Drag the method customerName() to the table field group Customer.
4. Create a new method in SalesTable with the following code:

```
display LineAmount salesTotal(SalesTable _salesTable)
{
    return (select sum(lineAmount) from salesLine
            where salesLine.salesId == _salesTable.salesId).lineAmount;
}
```

5. Make sure the changes are save to SalesTable and drag the new method salesTotal() to the form MyFrom_DisplayEdit to the design at *Designs/Design/[Tab:Tab]/[TabPage:Overview]/[Grid:Grid]*. Position the control last in the grid. Open the property sheet for the new control and select SalesTable as value for the property **DataSource**.
6. Go back to the table SalesTable and add a new method with the following code:

```
edit CustName editCustomerName(boolean _set,
                                CustName _name)
{
    CustName    name = _name;
    CustTable    custTable;

    if(_set)
    {
        if(name)
        {
            ttsbegin;
            custTable = CustTable::find(this.custAccount, true);
            custTable.name = name;
        }
    }
}
```



```

        custTable.update();
        ttscommit;
    }
}
else
{
    name = CustTable::find(this.custAccount).name;
}

return name;
}

```

7. Save changes to SalesTable, and drag the new method to the form design at *Designs/Design/[Tab:Tab]/[TabPage:Overview]/[Grid:Grid]*. Position the new control in the grid right after the control SalesTable_CustAccount. Open the property sheet for the new control and select SalesTable as DataSource for the control.
8. Go to the node *Data Sources/SalesTable/Methods* and override the init() method with the following code:

```

public void init()
{
    super();

    this.cacheAddMethod(tablemethodstr(SalesTable, salesTotal));
}

```

9. Save the form, and create a menu item for the form.
-

Two different display methods were added to the form. The SalesTable method customerName() is a display method returning the name of the customer for the current selected sales order. Both display methods and edit methods can be added to table field groups. This is a nice feature as in this case you will not have to modify the form to have the name of the customer shown. Using display and edit methods in table fields groups has a disadvantage. When adding or deleting a table method, the method ids are renumbered. This means that you will have to check your table field groups using display and edit methods, each time the number of table methods is changed, as your field groups might be used a wrong display or edit method.

Note: If your display method is shown on the form without a label and a value, you might have forgotten to set the data source name for the control. When dragging a display or edit method from a table method this must be done manually.

The second display method used in the form, salesTotals() is doing a simple summarization of the sales order total. When using a display method in a grid control, you will have to put the current record as parameter. To use the current record as parameter, the display method must be created as a table method or as a method for the data source.

Display and edit methods cause an overhead, especially when used in a grid control, as display methods are executed several time when using the form. To optimize performance display methods can be cached. By overriding the `init()` methods of the data source the data source method `cacheAddMethod()` can be used to cache table display methods. Only table methods can be cached, and caching must be done after `super()` in the `init()` method of the data source. Display methods are normally created as table methods so this is normally not a problem. Edit methods cannot be cached, however edit methods are not as common as display methods so caching of display methods will surely improve performance.

In the example only the display method used in the grid were cached. There are not much gain caching the display method showing the customer name, as this will result in customer name will be lookup up for all records.

An edit method is an extension of display methods. Display methods are normally used for displaying addition information, or to do calculations, where edit methods are used where a join cannot be used and a value in a related table must be editable. Edit methods has two parameters, the first will be true if the value of the control is changed, and the second parameter contains the value of the control. You will have to add code to the edit method to have the value in the related table updated. As you might guess this has an extra cost and you should always consider whether your design ought to be changed before using an edit method.

The extended data type used as return type in display and edit methods is used for setting labels and formatting for the control. If you need another label for your control, you should consider creating another extended data type to be used for your method, rather than changing labels in the design.

Note: An alternative to edit methods if you only need to display a single record is to just specify an extended data type in the property sheet for the control and set the `AutoDeclaration` property. You will then be able to both set a get values for the control. The form *KMAction* makes use of this for filtering in top of the form.

6.4 Methods on a Form

Basic forms can be written without adding any line of code. Even complex forms can be written without writing any code in the form. In fact you should try keeping code out of forms and instead put the logic to be used in tables or classes, as this will make it easier using another user interface. If using an external system and executing the logic through COM you will have to rewrite forms code as forms cannot be executed through COM.

You cannot avoid having code in your forms. Just have in mind when constructing forms that if possible you should add your code elsewhere. Say you need to have certain tasks done after the value of a control is changed. If the control is bound to data source

field it will be far more better to add the code to the table method `modifiedField()` rather than to the data source field method `modified()`.

In the following sections you will find an overview of methods which can be overridden on forms.

Form Methods

When executing a form an instance of the system class `FormRun` is initiated. From the node *MyForm/Methods* you can override the methods of the `FormRun` object. These methods are normally referred to as the form methods. When right-clicking the form methods node and select *Override Method* not all methods from the `FormRun` class are shown. Only the methods which can be overridden are shown so methods qualified as `final` will not show up in the list.

Form methods can be referenced using the keywords *this* and *element*. The keyword *this* can only be used within another form method, whereas the keyword *element* is in scope for the entire form object and can be used when reference form methods from data sources or controls.

Name	Parameters	Description
activate		Called each time a form is in focus. If switching between two forms the method will be called when selecting the form as the active form. See form PBATable.
canClose		Called by the methods closeCancel() or closeOk(). Often used to validate whether the form may be closed. See form InventTransPick.
close		This is the last method executed when a form is closed. See form ProdParameters.
closeCancel		When clicking a CommandButton control with the Command property set to Cancel, the method closeCancel() is called. If closeCancel() is true, the method canClose() is called. See form AssetChangeGroup.
closed		Closed() will return true after the super() call in close() has been called. Can be used to check whether close() has been called. Not often used.
closedCancel		Will return true if closeCancel() has been called. Often used from close() to check whether the form has been closed cancel or closed ok. See form CustParameters.
closedOk		Will return true if closeOk() has been called. The return value can be used for checking whether the data source method write() should be executed. See form SalesCreateOrder.
closeOk		When clicking a CommandButton control with the Command property set to OK, the method closeOk() is called. If closeOk() is true, the method canClose() is called. See form AssetChangeGroup.
closeSelect	str_selectString	Can be used to overrule the value returned by the lookup form. Often this is not necessary. Instead the method is used for adding code to be executed before the lookup form is closed. The FormRun method selectMode() will set a form in lookup mode and specify the form control used as return value. See form KMGamePlanLookup.

controlMethodOverloadObject	Object _val	By overriding this method, the methods for controls added to a form at runtime can be created in a class rather than being created on the form. See class BOMChangeItem.
copy		Executed when the application user presses ctrl+c to copy the content of a form control or an entire record.
cut		Executed when the application user presses ctrl+x to cut the content of a form control.
doApply		When clicking a CommandButton control with the Command property set to Apply, the method doApply() is called. Used to commit changes made in the form. See form ProdSetupRelease.
docCursor		As Document handling relates documents to the current active data source table, the method docCursor() can be used to determine which table to related documents to, if a form uses more than one data source. See form InventTable.
Finalize		Called when the form is closed. Will remove the form object from memory. This method is normally not overridden.
firstField	int _flags = 1	Set focus to the first field in the form design. Overridden if focus should be set to another form control. See form CustOpenTrans.
init		This is the first method called which can be overridden. The method is initializing the form. Entities used in the form are typically initialized here. Validation on whether the form is intended to be called by another form is also done using init(). See the form CustOpenBalance.
lastField	int _flags = 1	This method has no effect. Should set the last field in the design in focus.
loadUserSettings		Customization made by the application user to a form is loaded by loadUserSettings().
new	Args _args = NULL	New() is the first method called when

		executing a form. This method should never be overridden as this will cause the form to crash when opening.
nextField		Executed when stepping to the next field by pressing either tab or enter.
nextGroup		Executed when stepping to the next field group by pressing ctrl+pgdn.
paste		Executed when the application user presses ctrl+v to paste a value to a form control.
prevField		Executed when stepping to the previous field by pressing shift+tab.
prevGroup		Executed when stepping to the previous field group by pressing ctrl+pgup.
print		Executed when printing a form. Will print the fields from the table field group AutoReport. See the form ReqItemJournalSafetyStock.
printPreview		PrintPreview() is called if File Print Preview is selected from the top menu. See form SysLicenseAgreement.
reload	Args _args = NULL	Not used.
resize	int _width, int _height	If the size of the form is changed, the method resize() is called.
run		Run() is loading the form. The super() call in the method will call the form data source query. Initialization of the form should if possible be done in the form init() method and the data sources init() methods, rather than using run(). See form ProdTable.
saveUserSettings		Application users' customization of the form is saved by saveUserSettings.
selectControl	FormControl _control	Called when focus is moved to another control. See form LedgerTransSettlement.
send		Send() is called if File Send is selected from the top menu.
setApply	Object _object, Object _parm	Can be used in combination with a

		Command button control if calling the form from a class. See the class DocuActionTrans.
task		Called when the application user hits a key. Can be used to define hot keys. See form JmgSelectJob.

Figure 28: Form methods

Form Data Source Method

The form data source methods are often overridden when making changes to the form query or if you need to insert, update or delete records from a data source using X++. A form data source is an instance of the system class FormDataSource.

Name	Parameters	Description
Active		Active() is called each time a new record is selected. The method is often used for setting control access. See form SalesTable.
create	boolean _append = false	When pressing ctrl+n the method create() is called. Initialization of field values should be done using initValue(). See form SalesTable.
cursorNotify	int _event	Called when the cursor is set to another record. The method is also called when the data source cache is being updated. Can be used for advanced caching. See form projPeriodEmpl.
defaultMark	int _value	Will return the value 1 if all records in the grid is marked pressing ctrl+a. See class ReqTransFormPO.
Delete		If the application user press alt+F9, the method delete() is called. Delete() will call validateDelete() before trying to delete the record. See form ProjJournalTable.
deleteMarked		The method is called if the application user has marked more than one record and presses alt+F9. DeleteMarked() will call delete() for each record to be deleted.
displayOption		Used for coloring columns or rows. See form EmplTable.
executeQuery		Fetch data based on the settings in the form query. The method is often overridden to set values of ranges added from X++. See form CustTrans.
filter	fieldId _field, str _value	If right-clicking a field and select Filter, the method executeQuery() is called. Not often overridden as executeQuery() is used instead.
findRecord	Common _record	Can be used to set the cursor to a specific record by using a Common object. Notice, this is a very slow way to position the cursor. See form SalesTable.

findValue	fieldId _field, str _value	Can be used to set the cursor to a record matching the value of a specified field. This is a very slow way to position the cursor. See form LedgerJournalTrans.
first		Called when pressing ctrl+home for selecting the first record in a data source. See form ContactPerson.
forceWrite	boolean _value	Mark a record as “dirty”. If set to true the current record will be saved before leaving. Must be called from X++. See form LedgerJournalTransApprove.
init		Init() is the first data source method called. Runtime changes made to the query are normally added here. See the form CustTrans.
initValue		When creating a new record in a form initValue() will be called. The method is used for initializing field values. Use the corresponding table method instead, unless the settings are specific for the form. See form BankAccountStatement.
last		Called when pressing ctrl+end for selecting the last record in a data source. See form HRMApPLICANTTable.
leave		The method is called when the cursor is moved from the current selected record.
leaveRecord		Can be called from X++ to check whether the cursor may be moved to another record. See form InventPosting.
linkActive		If calling a related sub form, linkActive() will be called each time a new record is selected in the main form. LinkActive() will call executeQuery(). See form MarkupTrans.
mark	int _value	Will return 1 if the current record is marked. Can be used to mark a record by calling the method with the value 1.
next		Called when the next record is selected.

		See form EmplTable.
nextPage	int _pageSize	When the application user press the page down key in a grid control, nextPage() is called. The parameter _pageSize contains the number of shown records in the grid.
prev		Called when the previous record is selected. See form EmplTable.
prevPage	int _pageSize	When the application user press the page up key in a grid control, prevPage() is called. The parameter _pageSize contains the number of shown records in the grid.
print	PrintMedium _target	Called if ctrl+p is pressed for printing the auto report. Can be used to change the behavior of an auto printed report such as doing runtime changes to the layout. See form CCCount.
prompt		Activated when the application user calls the form query dialog. Prompt() is only called for the current data source. Can be used to prevent calling the query dialog on forms using a tree control. See form HRMOrganization.
refresh		Often used together with reread().Where reread() update the form data source for the current record with the values from the table, refresh() update the form design for the current record with the values from the form data source. See form WMSShipment.
refreshEx	anytype _pos	Extended version of refresh(). If the method is called with the parameter -1, all records for the data source will be updated. See form CustOpenTransReverse.
removeFilter		Called when a filter previous set is removed. Normally overridden on data sources which only contain a single record such as parameter forms. See from InventParameters.
reread		Must be called from X++. Fetch the current record from the data source

		table. See form SalesCreateOrderForm.
research		Must be called from X++. Fetch the records specified by the query. Similar to calling executeQuery(), except that research() will keep settings made to the query. See form LedgerTable.
validateDelete		Delete() will call validateDelete() to check whether the record may be deleted. See form ProdJournalTable.
validateWrite		ValidateWrite is called by write(). Will return true if the record may be inserted or updated. See form InventTable.
write		Called when pressing ctrl+s. Will call insert() on a new record, otherwise update() will be called. Validations should if possible be made in validateWrite(). See form CustOpenTrans.

Figure 29: Form data source methods

Form Data Source Fields Methods

All form data source fields have the same set of methods as they are all declared using the system class FormDataObject.

Name	Parameters	Description
context		Called by the application user, when right-clicking a field.
filter	str _filterStr, NoYes _clearPrev	Called when the application user right-click a field and select Filter. Notice, that the super() call in filter() will not work if the method is overridden.
find		Called when the application user right-click a field and select Find.
helpField		Show the help text for the current field in the bottom of the screen. The help text shown is from the current control, related table field or extended data type or base enum.
jumpRef		Is called when press ctrl+alt+F4 for jumping to main table. The method can be overridden to jump to another form where no relation exists. See form InventItemGroup.
lookup	FormControl _formControl, str _filterStr	Used to build a customized lookup form for controls which have no lookup. See form InventTable.
modified		If the value of field bounded to a data source field is changed, modified() is called. If validate() returns true modified() is called. Typically overridden if the value of another field must be set. See form PriceDiscAdm.
performFormLookup	FormRun _form, FormControl _formControl	Can be used to override an existing lookup form. See form HRMApplication.
restore		The method seems not to be used.
toolTip		Used to show the yellow tooltip when position the cursor on a control. Can be overridden to change the default tooltip. See form BOMDesigner.
validate		The method is called when the value is changed in a control bounded to a data source field. Used to add checks whether the control must be changed. See form InventTable.

--	--	--

Figure 30: Form data source fields methods

Validations specific for a single form is done using the form data source fields' methods. If validations made to a control bound to a data source field are of general use, you should use the table methods `validateField()` and `modifiedField()` rather than using data source field methods `validate()` and `modified()`.

Form Controls Methods

Some methods of a form control are similar to the methods of a form data source field. Often you will have alternatives for overriding a form control, as you can either use the methods at the data source level or auto declare a form control. Auto declaring a control is done by setting the control property **AutoDeclaration** to Yes. This is a property in common for all controls. Auto declaring a control will create an instance of the corresponding system class for the form control. Auto declaring a string control will create an instance of the system class `FormStringControl` which can be used from all methods on a form.

Common Form Methods

Only few of the form methods are needed in daily use. Having a basic knowledge of the execution order of these methods will help you a lot when starting making your modifications. The following methods are executed in the listed order when a form is opened and closed:

`init()` ► `ds init()` ► `run()` ► `ds executeQuery()` ► `canClose()` ► `close()`

1. `FormRun.init()` is the first method called. The `super()` call in `FormRun.init()` will call `FormDataSource.init()` for each data source used in the form query.
2. The `super()` call in `FormRun.run()` will call `FormDataSource.executeQuery()` for every data sources.
3. When closing the form `FormRun.canClose()` will validate whether the form may be closed, and if true `FormRun.close()` is called.

These are the most important methods executed when a form is opened and closed. Other methods are executed in the opening and closing sequences such as loading application user settings. However normally you will only need to override the listed methods.

Note: To discover the execution order of the methods, you can set a break point at the `super` call in an overridden method and check the stack trace in the debugger. Another option is to print a line to the Infolog from each method overridden.

For each data source in your form query, the following sequence will be triggered by `executeQuery()` when the form is opened:

`ds.executeQuery() ► refresh() ► active()`

1. The data source method `executeQuery()` will fetch all records for the current data source and afterwards call `refresh()`.
2. `Refresh()` is updating records fetched, and is normally not overridden. If you are inserting, updating or deleting records shown in a form from X++, `refresh()` can be called to have the changed data shown.
3. When opening the form, `active()` is executed for the current record. You can use `active()` for setting access restrictions for controls. `Active()` is called each time a record is selected.

Example 6: Active

Elements used from MORPHXIT Forms project

- Form, `MyForm_Active`
- Menu item display, `MyForm_Active`

The example shows how the data source method `active()` can be used to control access.

1. Duplicate `MyForm` and rename the new form “`MyForm_Active`”.
2. Expand the form and go to *Data Sources/SalesTable/Methods*. Right-click to override the method `active()`. Add the following to `active()`:

```
public int active()
{
    int ret;

    ret = super();

    salesTable_ds.allowEdit(salesTable.salesStatus == SalesStatus::Invoiced ? false : true);

    return ret;
}
```

3. Save the form and create a menu item.
-

Try executing the form `MyForm_Active`. You will not be able to modify sales orders with status invoiced. If you go to the second tab page on the form you will notice that the controls are grayed out for fields which may not be edited. Controls in a grid control will not change the appearance. If you need to change the appearance of a control in a grid, this can be accomplished by coloring rows. How to use colors in forms are explained in the section **Special Forms**.

Data Source Methods

Adding a new record from a form will call the following sequence of data source methods:

`create() ► initValue() ► refresh() ► active() ► refresh()`

1. The method `create()` is called when pressing ctrl+n to add a new record.
2. The data source `initValue()` is called by `create()`. You should initialize fields which must have a specific value in `initValue()`. The table method `initValue()` is called by `super()` in the data source `initValue()`. If the initialized values are not form specific, you should use the corresponding table method.
3. After initialization `refresh()` is called, and when entering a record `active()` is called. Refresh is called again by `active()`.

When saving a record the following methods are called:

`validateWrite() ► write() ► refresh()`

1. `ValidateWrite()` will call the table method `validateWrite()`. Again, always use the corresponding table methods if possible.
2. The data source method `write()` will call either `insert()` or `update()` from the table, depending on, whether the record is already created or not. Checks to be made should be done before this point. If you have conditions to be fulfilled these should be done at `validateWrite()`.
3. `Refresh()` is called to update the form controls.

Deleting a record has a similar flow as insert and delete:

`validateDelete() ► delete() ► active()`

1. `ValidateDelete()` will call the corresponding table method. Condition checks for deletion should be done at this point, either from the data source or from the table.
2. The form data source `delete()` calls the table method `delete()`.
3. `Active()` is called as the cursor will jump to another record after the current record is deleted.

Control Methods

All form controls has methods which can be overridden. You should only in special cases need to override the controls of a form. If a control is not bound to a data source you will have to use the methods of the control. For bounded controls it is considered good practice to use the data source field methods instead.

Some of the special controls such as the tree control and the list control require that some of the control methods are overridden. For controls showing base types you will hardly ever have to use a form control method.

Overriding a Form Query

You will often have situations which require filtering data on a form. As forms usually uses a query for fetching data this can be accomplished by adding ranges to the form query. When using an AOT query or a report query you will be able to add ranges to the query using the nodes in the AOT. A form query does not have the same options for filtering using the AOT nodes. Instead you will have to use X++ for adding ranges.

Example 7: Adding range

Elements used from MORPHXIT_Forms project

- Form, MyForm_QueryFilter
- Menu item display, MyForm_QueryFilter

This example will show how to add a filter to a form query by using a form control.

1. Make a copy of MyForm and rename the form to “MyForm_Query”.
2. Open the property sheet for the node *Designs/Design* and set the property Columns to 1.
3. Right-click the node *Designs/Design* and select *New Control/Group*. Position the new control above the tab control and rename the new control “rangeGroup”. Set the property Caption for the new control to “Filter”.
4. Right-click the control rangeGroup and select a control of the type StringEdit. Rename the control to “filterCurrencyCode”. Open the property sheet for the new control and set AutoDeclaration to “Yes” and ExtendedDataType to “CurrencyCode”.
5. Go to the forms ClassDeclaration and enter the following:

```
public class FormRun extends ObjectRun
{
    QueryBuildRange    rangeCurrencyCode;
}
```

6. Now go to *Data Sources/SalesTable/Methods* and override init():

```
public void init()
{
    QueryBuildDatasource    salesTableDS;
};
```



```

super();

salesTableDS      = SalesTable_ds.query().dataSourceTable(tablenum(SalesTable));
rangeCurrencyCode = salesTableDS.addRange(fieldNum(SalesTable, currencyCode));
}

```

7. Override the executeQuery() method on the data source SalesTable:

```

public void executeQuery()
{
    rangeCurrencyCode.value(queryValue(filterCurrencyCode.valueStr()));

    super();
}

```

8. Located the control filterCurrencyCode and override the modified() method:

```

public boolean modified()
{
    boolean ret;

    ret = super();


    SalesTable_ds.executeQuery();

    return ret;
}

```

9. Save the form and create a menu item for the form.
-

The form MyFrom_QueryFilter has a control above the grid control. This control can be used for only showing sales order of a selected currency. The filter control is not bounded to a data source. The extended data type CurrencyCode related to the control is setting labels and adding the lookup. As the control is not bounded to a data source values keyed in is not validated, so you would be able to key in a currency code not listed in the lookup. A validation should be made to check whether a valid currency code is keyed in. The validation is on purpose left out to simplify the example.

In the init() method on the data source SalesTable a new query range was created. Ranges should always be created in the init() method as this method is only executed once. The value used by the range is defined in the executeQuery() method as this method is triggered each time the form query is changed. A form query can be executed again if the application users enters the query dialog by clicking the query icon  and changes the query values. In the example the executeQuery() method is triggered when the value of the control Currency is changed.

Notice if the application users click the query icon, the ranges for the currency code can be changed. To prevent this the following line must be added to the query init() method:

```
rangeCurrencyCode.status(RangeStatus::Locked);
```

Such a way of adding filter controls to the form design is commonly used in Axapta. The application user could have achieved the same result by accessing the query dialog directly. This example was simple so it would not have been difficult for the application user doing so using the query dialog. If you had several filter controls, and several data sources in your form query this would however not be an easy job for the application users.

Modifying Data Sources from X++

Best practice is to never insert, update or delete records from the methods of a form. Such operations should always be done from a table or a class method to keep data operations out of forms.

A common mistake when inserting, updating or deleting a record from a table used as data source in a form, is to refer directly to the table instead of using the data source. You might have a run able class called from your form inserting a new record in a table used as a form data source. To have the new record shown in your form you call `executeQuery()`. Your new record will be shown alright, but the cursor will be positioned at the first record in the form. This might be fine for your case, but often there is a requirement that the cursor is positioned at the new record. A data source has a method called `findRecord()` which can be used to position the cursor. This is not recommend using `findRecord()` as this might result in that most of the records of your form are traversed to positioning the cursor. Instead you should use the same data source methods as the form would use when inserting new records.

Example 8: Adding records

Elements used from MORPHXIT Forms project

- Class, `MyForm_NewRecord`
- Form, `VendGroup`
- Menu item action, `MyForm_NewRecord`

In this example you will learn how to add a new record to a form data source from X++ by using the same calls as if the records were created from the form user interface.

1. Create a new class, go to the `ClassDeclaration` and rename the class “`MyForm_CreateRecord`”.
2. Make the class run able by adding a static `main()` method:

```
static void main(Args _args)
{
    MyForm_NewRecord newRecord;
;

    newRecord = new MyForm_NewRecord();
    newRecord.run(_args.record());
}
```

```
}

```

3. Create a method called run() and add the following code:

```
void run(VendGroup _vendGroup)
{
    VendGroup      vendGroup;
    VendGroup      callerVendGroup = _vendGroup;
    FormDataSource formDataSource;
;

    formDataSource = callerVendGroup.dataSource();
    vendGroup.data(callerVendGroup);

    formDataSource.create(true);
    callerVendGroup.data(vendGroup);
    callerVendGroup.recId      = 0;
    callerVendGroup.vendGroup  = "G1";
    callerVendGroup.name       = "Group 1";
    callerVendGroup.write();

    formDataSource.reread();
    formDataSource.refresh();
}
```

4. Save the class and create a menu item for the class. Go to the property sheet for the new menu item and set Label to "New record".
5. Locate the form VendGroup and drill down the form design to *Designs/Design/[ButtonGroup:ButtonGroup]*. Drag the menu item for the new class to the button group control.
6. Save the form VendGroup.

When opening the form VendGroup you will have a new button called New Record. By clicking the button a new record will be inserted into the table VendGroup, and the cursor will be positioned on the new record. The record will be a copy of the current selected record. Only account number and name is changed.

In the class a handle is created to the VendGroup data source. The current selected record is passed on as a parameter in the main() method. A table variable has a method called datasource() which is initialized when called from a form. This method is used to call the data sources methods create() and write() for adding a new record and inserting the new record. As the current record is copied the system field recId is set to 0 to force a new record to be inserted.

The call of the data source methods refresh() and reread() is very important as they will update the record in the form and read the new record from the table. Forgetting these calls will result in an error if using AOS since the cache will not be updated.

Using the form data source methods for inserting, updating and deleting records makes your form more users friendly. This is a preferable solution when adding single records. However if you have to insert a lot of records in a form you would be better off using the table methods instead as calling the form data source method for inserting a large number of records will be very slow.

The system class `Args` was in this example used to get a handle to the calling object. `Args` is often used for validating the calling object. Some forms require to be called from another form or class. In such a case `Args` can be used to validate the calling object, and depending on the calling object different ranges could be added, or restrictions on controls could be set. You will find plenty of examples on the use of `Args` in forms by checking the forms in the standard package.

Building Lookups

Adding a relation to an extended data type will cause a lookup button automatically to be added to form controls using the extended data. Often this will be just fine. However you might have cases where no relation exists and a lookup is required. The case could also be that you need to filter the records listed in the lookup, or you might need a more advanced layout such as several tab pages in the lookup which different records listed. A lookup form can be overridden either by building a new lookup from X++ or by replacing the default lookup with a customized lookup created using a form.

Take a look at the following code which is taken from the form `HRMApplication`. This is the method `performFormLookup()` overridden for the data source field `hrmRecruitingId`:

```
public void performFormLookup(FormRun _p1, FormControl _formControl)
{
    super(hrmApplication::hrmRecruitingIdLookup(_p1), _formControl);
}
```

To override an existing lookup from X++, you can use the method `performFormLookup()` from the data source field. Both data fields and form controls have a method called `lookup()`. These should only be used if your control does not already have a lookup, or if you need a lookup for a control which is not bounded to a data source field. As the method `performFormLookup()` was first introduced by Axapta 3.0 you might not be able to find many examples on the use of this method in the standard package. `performFormLookup()` has the `FormRun` object used by the lookup as the first parameter and the control for the lookup field as the second parameter. Here the method `hrmRecruitingIdLookup()` on the table `HRMApplication` is called to override the `FormRun` object passed on the `super()` call.

```
static formRun hrmRecruitingIdLookup(FormRun lookupFormRun)
{
    formDataSource formDataSource;
    query formquery;
```

```

;
formDataSource = lookupFormRun.objectSet();
formQuery      = formDataSource.query();

formQuery.dataSourceNo(1).addRange(fieldNum(HRMRecruitingTable,
                                         status)).value(queryValue(HRMRecruitingStatus::Started));

return lookupFormRun;
}

```

If looking up the table method `hrmRecruitingIdLookup()` you will see that the `FormRun` object is used to get a handle on the default lookup form. In that way you can access the query used by the lookup. A new range is here added for filtering the records listed in the lookup. The query used in the lookup is an ordinary query, which can be joined with any data sources.

If you need to create a lookup for a control which has no lookup, you will not be able to use the method `performFormLookup`. Instead the class `SysTableLookup` is used to build a query for your lookup. The form `InventTable` makes use of this class in several methods located in the table `InventTable`. The lookup methods on the table `InventTable` are prefixed with `lookup*`.

A form created in the AOT can also be used as a lookup. Forms used as lookup forms are easily recognized in the AOT as they have the suffix `*lookup`. To have a form look like a default lookup form generated by the kernel, you should create the form without tab pages. Just add a grid control with the fields to be listed. By setting the property **Frame** on the node *Design* to “Border” the form will appear as a default lookup form. The `init()` method on the lookup form should return the control to be selected:

```

void init()
{
    super();
    element.selectMode(CustTable_AccountNum);
}

```

The above code is taken from the form `CustTableLookup`. The form method `selectMode()` will set the form in ‘lookup mode’. When the application user press enter or click on a record, the form is closed and the value of the form control specified as parameter is returned.

```

client static void lookupCustomer(FormStringControl _formStringControl,
                                CompanyId _companyId = curExt())
{
    Args args;
    FormRun formRun;

    if (! CompanyTmpCheck::exist(_companyId))
    {
        throw error(strFmt("@SYS10666", _companyId));
    }
}

```

```
changecompany(_companyId)
{
    args = new Args();
    args.name(formstr(CustTableLookup));
    formRun = classFactory.formRunClass(args);
    formRun.init();
    _formStringControl.performFormLookup(formRun);
}
}
```

Here the form CustTableLookup is called from the method lookupCustomer on the table CustTable. Notice that the data source method performFormLookup is used to link the lookup form with the calling form. A better solution than calling your lookup method directly from X++ is to relate your lookup form directly to an extended data type. Just enter the form name at the property **FormHelp** on extended data type and your lookup form will automatically be used when using the extended data type in a form. This is really neat, as this does not require any line of code to be written from the calling form. If necessary you can make restrictions in your lookup form on how the lookup form should behave depending on the caller. The extended data type ProjId is using a customized lookup form. A lot of the extended data types using special lookups are also using customized lookup forms such as the extended data type TransDate.

Note: If you need to create a lookup on AOT objects such as tables or classes, take a look at the Global classes prefixed with pick*. Check the form sysDatabaseLogTableSetup for an example on how to use the Global class method pickTable().

Form Dialog

A common way to create a dialog for a runbase class is using the Dialog* classes. A dialog is often simple and you will just have to add a few controls and categorized the controls in field groups.

However if you have created a complex dialog with several controls, where controls are being enabled based on runtime settings, you will soon realize that the dialog will be time consuming to maintain. Before coming so far, you should consider using a form as dialog instead. The runbase framework supports using an ordinary form as a dialog. All needed is to follow a few rules when constructing the dialog form.

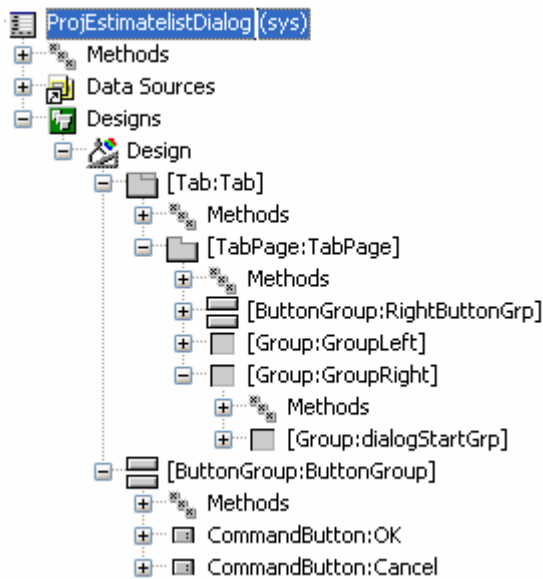


Figure 31: Form dialog

Take a look at **figure 31: Form dialog**. The form is an example of a dialog form used by a report in the Project module. The runbase framework will check that certain controls are created and named correctly in the design. These mandatory controls are used by the runbase framework to locate the different part of the design. If you have decided on using a form dialog you will still be able to add additional controls to your form dialog from the runbase class using the dialog classes. The mandatory controls in the form dialog is used to positioning controls added from the class.

When creating a form dialog it is much faster to copy an existing form dialog rather than creating your own from scratch. By using a form dialog as the one shown here you will be certain that you have all the required controls needed. Just replace the controls in the field group named GroupLeft with the controls needed for your dialog. All controls used for keying in data in your form dialog should be created using edit methods. This will make it easier to transfer the values to the runbase class.

The above mentioned form dialog is called from the class ProjReport_EstimateCheck. The dialog() method in the class looks like the following:

```
public Object dialog()
{
    DialogRunbase dialog = Dialog::newFormnameRunbase(formstr(ProjEstimatelistDialog),this);
    ;
    dialog = super(dialog);

    // Add extra fields below

    return dialog;
}
```

Notice that additional dialog fields are also added in the dialog method. A parameter method is created for each field in the dialog to get and set the values for the dialog. The method `getFromDialog()` will only be used if you have added additional fields to the form dialog from your class.

6.5 Special Forms

This section shows examples on how to create more advanced forms. The examples should give you an idea of the opportunities with Axapta forms. Even though forms are standardized and the layout is normally auto positioned you still have several options for making your forms more user friendly without making you code more complex.

Calling User Defined Method

You might have noticed when executing a form from X++, you will not be able to call your own methods created on the form. If you have created a new method on a form, your method will not be known when declaring a `FormRun` object outside the form as only members of the `FormRun` object are known.

Sometime it can be useful to call your own methods from outside a form. This can be achieved by using the super class for the system class `FormRun`.

Elements used from MORPHXIT_Forms project

- Form, `MyForm_CallingUserMethod`
- Menu item display, `MyForm_CallingUserMethod`
- Job, `Forms_CallingUserMethod`

Duplicate `MyForm`, rename the form to “`MyForm_CallingUserMethod`” and create a menu item for the new form.

```
void setReadOnly()
{
    salesTable_ds.allowEdit(false);
    salesTable_ds.allowCreate(false);
    salesTable_ds.allowDelete(false);
    salesTable_ds.insertAtEnd(false);
    salesTable_ds.insertIfEmpty(false);

    element.design().caption(strfmt("READ ONLY - %1", element.design().caption()));
}
```

Add the above method to `MyForm_CallingUserMethod`. Calling this method will prevent modifying data in the form and prefix the caption text of the form with a read only message. The method will be used to show how to call a new form method from outside a form.


```

static void Forms_CallingUserMethod(Args _args)
{
    Args      args;
    Object     formRun;
;

    args = new Args();

    formRun = new menuFunction(menuItemDisplayStr(MyForm_CallingUserMethod),
                               MenuItemType::Display).create(args);

    formRun.init();
    formRun.setReadOnly();
    formRun.run();
}

```

Create a new job with the above code. The form `MyForm_CallingUserMethod` will be called when executing the job. Instead of using the system class `FormRun`, the system class `Object` is used for declaring a `FormRun` object. `Object` will not validate for valid `FormRun` methods. This gives the options for calling any new methods added to the form. You must assure that the method exists, as no validation is made until runtime and no lookup information will be shown when entering a dot after a instance of the class `Object`.

The method added to the form `MyForm_CallingUserMethod` did not have any functionality which could not be called by referring the form and form data source methods. Still using such a method can be useful as you would only have to add a single line of code to the job calling the form. As the form might be called from several places you be able to have the code in one place.

Overload Methods

If you have added controls at runtime to your forms, you might have situations where it would be useful to override the methods of the runtime control. Overriding the methods of a dynamic control is possible, you will just have to enable this feature on the form and create a method on the form for each method to be overridden.

Elements used from MORPHXIT_Forms project

- Form, `MyForm_OverloadMethod`
- Menu item display, `MyForm_OverloadMethod`
- Job, `Forms_OverloadMethod`

Duplicate `MyForm`, rename the form to “`MyForm_OverloadMethod`” and create a menu item for the new form.

```

public void init()
{
    super();

    element.controlMethodOverload(true);
}

```

```
}
```

In the `init()` method of the new form you must enable the option to override methods for runtime added controls. The code for the overridden methods must be added to the form. The syntax is: `<controlname>_<controlmethod to be overridden>`. This means that you must know the name of the new control added before runtime. In this case the modified method of a new control called `showOnlyOpenOrders` will be overridden:

```
Boolean showOnlyOpenOrders_modified()
{
    FormCheckBoxControl checkBoxControl = this.controlCallingMethod();
    QueryBuildRange rangeSalesStatus;
    Boolean ret;
;

    ret = checkBoxControl.modified();

    rangeSalesStatus =
SalesTable_ds.query().dataSourceTable(tablenum(SalesTable)).findRange(fieldnum(SalesTable,
salesStatus));

    if (!rangeSalesStatus)
    {
        rangeSalesStatus =
SalesTable_ds.query().dataSourceTable(tablenum(SalesTable)).addRange(fieldnum(SalesTable,
salesStatus));
    }

    if (checkBoxControl.value() == NoYes::Yes)
    {
        rangeSalesStatus.enabled(true);
        rangeSalesStatus.value(queryValue(SalesStatus::Backorder));
    }
    else
    {
        rangeSalesStatus.enabled(false);
    }

    SalesTable_ds.executeQuery();

    return ret;
}
```

A new check box control is going to be added to the form. The new control will be used to show only open sales orders. The code in this method will be executed when the value of the new control is modified. By calling `this.controlCallingMethod()` a handle to the new control is made. This is important as you will have to make the `super()` call of the modified method manually. Also your method must have the same return value as the overridden method.

Note: You can put the methods for the controls added runtime in a class by using the `FormRun` method `controlMethodOverloadObject()`.

In the method a new range is added to filter the sales order status. As this method might be called several times the rangeSalesStatus object is first tried initialize by searching the query for an existing range on the field salesStatus. If no range is found the method is executed for the first time, and a new range is added. By marking the check box, the range will be enabled and filtering on open sales orders. If unmarking the check box, the range will be disabled.

```
static void Forms_OverloadMethod(Args _args)
{
    Args                args;
    FormRun              formRun;
    FormCheckBoxControl  ctrlShowOnlyOpenOrders;
;

    args = new Args();

    formRun = new menuFunction(menuItemDisplayStr(MyForm_OverloadMethod),
                                MenuItemType::Display).create(args);
    formRun.init();
    formRun.design().columns(1);

    ctrlShowOnlyOpenOrders = formRun.design().addControl(FormControlType::CheckBox,
                                                            "showOnlyOpenOrders");
    ctrlShowOnlyOpenOrders.label("Only open orders");
    formRun.design().moveControl(ctrlShowOnlyOpenOrders.id());

    formRun.run();
}
```

The form is called using a job. After initializing the form and activating the option to override the methods, the check box control is added. Remember that the name of the new control must match the prefix for the form method used to override modified(). A label is added for the control. The label system is not use to simplify the case. To have the new control appear in top of the form the columns number on the form design has be set to 1. As a new control will always be appended as the last control, the design method moveControl() is used to set the check box control as the first control. MoveControl() takes two parameters, the second parameter specifies which control to insert right after. As the check box should be the first control, the second parameter is just left blank.

Overriding methods of a dynamic control is not optimal as you will have to create a method on the form for each method to be overridden. Often when adding dynamic controls you might not be aware of the number of controls to be added till runtime, or you might need to have a certain number of controls added to several forms. An alternative is adding a menu item at runtime instead. By adding a menu item calling a run able class, you will be able to add all your logic in the calling class. This will not give you the exact same options as overriding methods of a form control, but in most cases this should fulfill your requirements.

General Form Changes

Each time a form is executed the class SysSetupFormRun is called by the ClassFactory class. SysSetupFormRun is a sub class of the system class FormRun. This means that you have the option to get a handle on FormRun when a form is executed. By modifying SysSetupFormRun you will be able to make runtime changes to any form without having to modify the form. This is an awesome feature which really can save a lot of coding.

Keep in mind that such modifications will cause an overhead as the code is called each time a form is loaded. You should be caution when modifying SysSetupFormRun. Always try out you changes in a local installation of Axapta. If you make any mistakes in you code, you might have the entire application to crash, and this might not be appreciated if other people are using the application. Not just normal application forms will be affected. All MorphX tools created by the AOT such as the compiler form and the search form will be affected.

Elements used from MORPHXIT_Forms project

- Class, SysSetupFormRun

Add a new method called showDatasourceInfo to the class SysSetupFormRun:

```
void showDatasourceInfo()
{
    FormDatasource          formDatasource;
    FormGroupControl        formGroupControl;
    FormStaticTextControl    formStaticTextControl;
    Counter                 counter;
    Counter                 noOfDatasources;
;

    noOfDatasources = this.form().dataSourceCount();

    if (noOfDatasources)
    {
        formGroupControl = this.form().design().addControl(FormControlType::Group,
                                                            "formGroupControl");
        formGroupControl.caption("Tables used by form");
        formGroupControl.frameType(3);
    }

    for (counter=1; counter <= noOfDatasources; counter++)
    {
        formStaticTextControl = formGroupControl.addControl(FormControlType::StaticText,
                                                            "formStaticTextControl" + int2str(counter));
        formStaticTextControl.text(tableid2name(this.form().dataSource(counter).table()));
    }
}
```

The method will print the names of tables used as data source by the form. A field group will be added to the form, and for each data source a static text control will show the data source table names. The field group will only be added if the form has any data sources.

```
void new(Args args)
{
    KMActionMenuButtonAuto          KMActionMenuButtonAuto;
    CCDatalinkMenuButtonAuto        CCDatalinkMenuButtonAuto;
    KMKnowledgeCollectorMenuButtonAuto KMKnowledgeCollectorMenuButtonAuto;
    ;

    super(Args);
    // CC Start
    if (this.name() != formStr(SysLicenseCode))
    {
        KMActionMenuButtonAuto = new KMActionMenuButtonAuto(this);
        if (KMActionMenuButtonAuto.check())
            KMActionMenuButtonAuto.create();

        CCDatalinkMenuButtonAuto= new CCDatalinkMenuButtonAuto(this);
        if (CCDatalinkMenuButtonAuto.check())
            CCDatalinkMenuButtonAuto.create();

        KMKnowledgeCollectorMenuButtonAuto= new KMKnowledgeCollectorMenuButtonAuto(this);
        if (KMKnowledgeCollectorMenuButtonAuto.check())
            KMKnowledgeCollectorMenuButtonAuto.create();
    }
    // CC End

    this.showDatasourceInfo();
}
```

The methods is called from the new() method in the class SysSetupFormRun. When executing a form, a field group with a frame will automatically be added showing tables used by the form.

This is just a simple example on how to make general changes to forms. Try take a look at the other code in the new() method. The code is used by the HRM modules to add buttons dynamically to forms. The class CCDataLinkMenuButtonAuto is used to add buttons to a form at runtime. This HRM feature is called Data links and is used to link an Axapta record to an external database. If a form uses a data source configured as a data link, a button to call a form showing the related records from the external database is added at runtime to the form.

Colors

A typical form in Axapta uses a standard set of form colors. Only a few places in the standard package form controls are colored in other colors. By changing the default colors of a form you will in a simple way make a form more user friendly. Coloring a

control is an alternative to sort or filter data as it makes it easier to spot a control with a certain value.

Elements used from MORPHXIT Forms project

- Form, MyForm_ColorRow
- Form, MyForm_ColorColumn
- Menu item display, MyForm_ColorRow
- Menu item display, MyForm_ColorColumn

Make two duplicates of MyForm called “MyForm_ColorRow” and “MyForm_ColorColumn”.

Go to the form MyForm_ColorRow and override the SalesTable data source method displayOption() with the following code:

```
public void displayOption(Common _record, FormRowDisplayOption _options)
{
    SalesTable salesTableLocal = _record;
    ;

    if (salesTableLocal.currencyCode == CompanyInfo::find().currencyCode)
    {
        _options.backColor(WinApi::RGB2Int(255, 255, 255)); // White
    }
    else
    {
        _options.backColor(WinApi::RGB2Int(255, 0, 0)); // Red
    }

    super(_record, _options);
}
```

The method displayOption is executed for each record in the form when the form is loaded. You use the method for changing the default colors of a control. In this example records where the currency of the sales order is different than the company currency will be colored red. Notice, colors must be set for all records and not only the records to be colored red.

The example just shown was coloring all controls of a record. Try changing the displayOption() method in the form MyForm_ColorRow to look as the following:

```
public void displayOption(Common _record, FormRowDisplayOption _options)
{
    SalesTable salesTableLocal = _record;
    ;

    if (salesTableLocal.currencyCode == CompanyInfo::find().currencyCode)
    {
        _options.backColor(WinApi::RGB2Int(255, 255, 255)); // White
    }
}
```

```
else
{
    _options.backColor(WinApi::RGB2Int(255, 0, 0)); // Red
    _options.affectedElementsByControl(SalesTable_SalesId.id(), SalesTable_CurrencyCode.id());
}

super(_record, _options);
}
```

The only changes compared to the previous example is that the method `affectedElementsByControl()` is called. This will cause only the controls listed as parameters for `affectedElementsByControl()` will have the color changed. You can add any number of controls to be colored. Just remember to set the controls added to be auto declared.

The examples showed where hard coding the criteria's and the colors used. In a real case you should let such options be available for the application users. When using the HRM modules the form `EmplTable` make uses of coloring rows. The colors used by `EmplTable` are defined by the application user in the parameter form `HRMParamters`.

6.6 Summary

This chapter has shown how to use the single parts of a form such as data sources and the controls in the form design. You should know be aware of how to get data from database and have the data maintained in a form. Also you should have acquired knowledge of how to create forms with a standardized look. As forms make out the main part of the user interface, having a standardized look and feel of your forms is essential as this makes the user experience better.

7 Reports

Reports in Axapta are based on a query and a design stored in the AOT. A report is just a definition, it contains no data, and when run, will fetch the data it needs from the database. When executing the report, you have the option of printing the report immediately, or defining a batch job for executing the report at a later time on a separate batch server. Batch processing is normally used for lengthy reports such as printing monthly customer balance lists.

Axapta reports are very flexible since MorphX provides tools to override a report's definition without the need for complex programming. By using the MorphX environment, you can filter data, or even modify the layout of the printed job at runtime.

There are two ways of creating reports in Axapta. You can use the built-in report wizard, or the report generator, located in the AOT under *Reports*.

This chapter focuses on the technical aspects of creating reports, and is not intended to provide an explanation of the various options contained in the report dialog. While it is always helpful to understand Axapta's user interface, you do not need to understand the report user interface to benefit from the information contained this chapter. If you are not familiar with the end user interface for reports, you can get more detailed information by checking the manuals in the standard package.

7.1 Report Wizard

The Report wizard is a tool designed so that non-technical persons can create reports. The wizard can be accessed in the toolbar menu **Tools | Development tools | Wizards | Report Wizard**. This is a good place to start learning about Axapta reports. The Report wizard is an end user tool, which guides you through the steps of creating reports. You have the option of storing a report created by the wizard in the AOT. It is a good idea for new Axapta programmers to review these wizard-generated objects in order to become familiar with the standard elements contained in a report. The Report Wizard is also helpful for experienced Axapta programmers who can use it to create the basic structure of a report and then use the report generator for final modifications.

For a step-by-step guide showing how to use the Report wizard, see the Appendix: **Reports Wizard**.

7.2 Creating Reports

By reviewing the output generated by the report wizard, you can see what will be required to create reports from scratch using the report generator. When you are first learning how to create a new report, it is useful to begin by looking at some of the standard reports in the AOT to see if you can find an existing report that already meets

some of your needs. Duplicate this report and start making modifications to your copy. For help in locating a report from the menu, see the chapter **Menus and Menu Items**.

When starting out with the report generator, the tutorial reports in the standard package can also be useful. Take a look in the AOT at the reports prefixed with **tutorial_**.

Axapta reports are divided into two parts, the *Data Sources* node which defines the data to be fetched and the *Designs* node which is used to define the report's layout and presentation. **Figure 32: Report overview** shows an overview of a typical report.

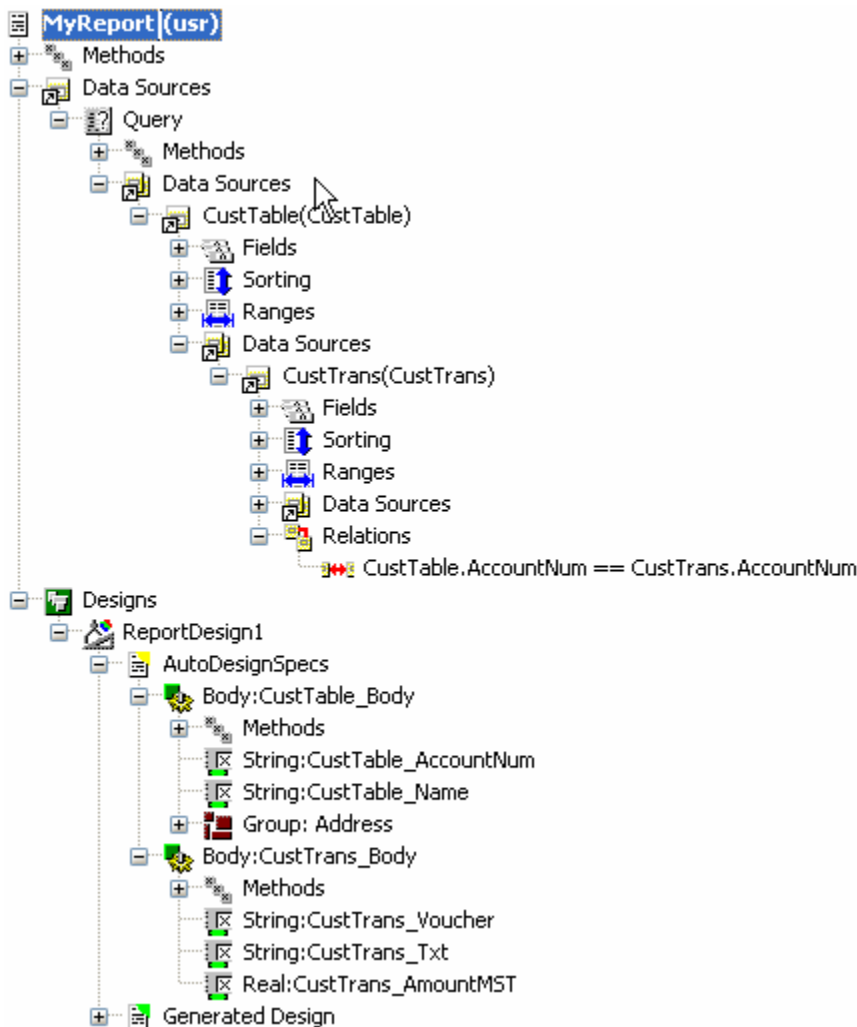


Figure 32: Report overview

Example 1: My first report

Elements used from MORPHXIT_Reports project

- Report, MyReport
- Menu item output, MyReport

As a preliminary exercise, start creating a report as shown in **figure 32: Report overview**. The report prints customer transactions grouped by customer. The example is kept simple since the purpose is to learn the basic steps required to create a report. Later in this chapter, the details will be explained and more features will be added to the example.

1. Create a new report by right-clicking the *Reports* node in the AOT and selecting *New Report*. A new report called "Report1" will be created. Open the property sheet and rename the report "MyReport".
2. For this report, you will be retrieving information from the Customer main table along with each customer's related transaction data. This is accomplished by specifying two levels of data sources for the query. For this example, the customer tables are the first level and the customer transaction table is the second level. MorphX development often involves dragging and dropping elements. In this case you will drag tables from one section of the AOT to the data source of the report you are developing. To make the task of dragging and dropping as simple as possible, Axapta allows you open multiple instances of the AOT. In this case, you will open another instance of the AOT and drill down the *Data Dictionary/Tables* node. Select the table CustTable. Expand the *Data Sources/Query* node in your report, and drag CustTable to the *Data Sources/Query/Data Sources* node. Now the first level of the query has been added. Expand the CustTable data sources node in your report. Drag the table CustTrans to the *CustTable/Data Sources* node.
3. The two data sources of the query must be linked, otherwise all of the transactions contained in the Customer Transaction table will be printed repeatedly for each customer. Go to the CustTrans node in the *Query* and set the property **Relations** to "Yes". The node *CustTrans/Relations* will now contain an entry linking the two tables. Now the report will only print the transactions that belong to the customer being processed.
4. The query part is now ready to fetch data for the report, and the presentation of data must now be done. Navigate to the *Designs* node, right-click and choose *New Report Design*. A new design called "ReportDesign1" will be created. Navigate to the *ReportDesign1* node and enter the text "Customer transactions list" in the **Caption** property.
5. Go to the *ReportDesign1/AutoDesignSpecs* node. Right-click the node and select *Generate Specs From Query*. Your design will now contain two body sections, one for each table in the query.
6. The last step is to select the fields to be printed. Right-click the *Query* node of your report, and choose *Open New Window*. This will ease up dragging the fields to be printed to the design. Pick the fields AccountNum and Name from the CustTable data source and drag the fields one at the time to the body section node

CustTable_Body. Go to the CustTrans data source and drag the fields Voucher, TransDate and AmountMST to the body section CustTrans_Body.

7. You now have a report as shown in **figure 32**. Execute the report by right-clicking the report name and selecting Open. Axapta will display a dialog for filtering, sorting and other print options. For now just click OK. The next dialog is the printer dialog. Check to ensure that the printout is set to “Screen” and click OK. Your report will now be printed to the screen.
 8. As you might have noticed, the layout leaves something to be desired. Much of the formatting tasks may be left to MorphX by specifying that the report should use a predefined report template. Templates instruct MorphX to create standard report features such as headings. To add a template to your report go to the *Designs/reportDesign1* node and locate the property **ReportTemplate**. Click the arrow and choose the template InternalList.
 9. Run the report again by following step 7. Now the report has heading information like name of the report, page number, date and time. You have created your first report!
-

In the MyReport example, you did not have to write a single line of code. When creating reports in Axapta, you will not have to worry about writing code for data connections and position controls in the layout. MorphX will handle this for you. Simply create a query and select the column order of your controls in the design. You should only have to use X++ when creating more advanced reports where fetching data is too complex for query, or when you need a special layout.

When you executed the report, two dialog windows were shown. Try creating a menu item for MyReport by dragging MyReport to *Menu Items/Output* node. You can now execute your report by right clicking the new menu items and selecting Open. The information that was previously contained in the two separate dialog windows is now presented in a single window. Running the report using a menu item automatically activates a more sophisticated runtime report framework. This is the dialog that users will see. Additional details can be found later in the chapter.

7.3 Report Query

While there are situations where a query cannot fulfill your needs and you have to fetch your data using X++, most of the time you will fetch data for a report using a query that specifies the data sources used for the report and how they are related. The report generator uses a standard Axapta query. For more information on building a query, see the chapter **Queries**.

Before building your query, you will have to decide which tables are needed. It is often the case, as in the MyReport example, that you need to print data from a single form or

a set of related forms. In this case, you can examine the forms themselves to determine the names of the required tables. For help on how to locate tables and fields from the forms, see the chapter **Menus and Menu Items**.

After you have identified the required tables, you need to determine how the data should be sorted and how it needs to be filtered in order to exclude any records that should not appear on the report. The decision made here can have a significant impact on system performance. A little additional planning at this stage can significantly reduce the report's execution time. For example, query selection criteria are more efficient when placed on the highest level table defined in the data source. As a general rule, you should attempt to limit the use of selection criteria to the tables in the first two levels of your data source. If your report is filtering data at the third level of the query, you should revisit your design and attempt to identify a more efficient approach. Can the same result be achieved using two separate reports? If not, can a temporary table be used to achieve the desired result?

For most reports, you will add all of the required tables to the report's *Query* node. If data from related tables are used, the tables must be joined in the query as you did with *CustTable* and *CustTrans* in the *MyReport* example. Sometimes it is necessary to fetch data from two tables that have no relation set up between them that can serve as a link. In this case it may be possible to use another table that shares a relationship with each of the tables from which you want to report. For example, if you want to print sales invoice lines grouped by customer, there is no direct relation between the *Customer* table and the *Sales Invoice Lines* table. Therefore, the *Customer Invoice Journal* table must be used to create the report. See **figure 33: Relation between CustTable and CustInvoiceTrans**. You will either have to add all three tables to the query, or just add the first level table, **CustTable**, as a data source and use X++ to fetch the other two tables. In general, the preferred approach is to use a query instead of X++ since this will give the user the full benefit of the report dialog.

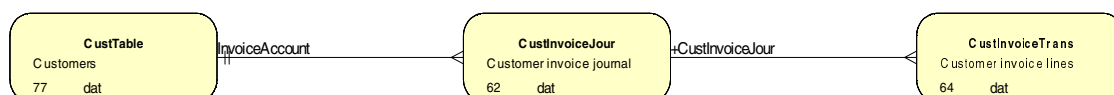


Figure 33: Relation between CustTable and CustInvoiceTrans

Joining data sources can be done in two ways. If the data sources already have a relation, the property **Relation** on the lower level data source must be set to True, just as in the *MyReport* example. The relation will then be visible under the *Relations* node for the joined data source. If no relation shows up, you must manually create the relation under the *Relations* node, and the property **Relation** must then be set to False. The best practice is to use an already existing relation, rather than manually creating your own, as changes to the data dictionary would then automatically be reflected in the report. By default, the data sources will be joined using an inner join, but the join mode can be changed on the properties for the joined data source. Inner joins are often used in business reporting where you have data in a main table and want to print the related transactions. However, if you want to print all records from the main table even if there

are no transactions, you will have to change the join mode to OuterJoin on the transaction table.

By right-clicking the *Fields* node, you can choose to add a field or an aggregation function. By default, all of the fields from the current table are listed and so it makes no sense to add additional fields from the table. If you add an aggregate function, all of the fields will be removed because you cannot use both. To remove the aggregate functions and restore the field list, change the *Fields* node **Dynamic** property to Yes. The aggregate functions can be used if you want to count the number of customers by customer group. You will have to select a table, choose an aggregate function and the fields to be used.

Example 2: Aggregate function

Elements used from MORPHXIT_Reports project

- Report, MyReport_aggregate
- Menu item output, MyReport_aggregate

The following example will produce a count of customers by customer group. The design has been simplified to focus on the aggregate functions.

1. Add the Customer table to the report query. Then, navigate to the *Fields* node, right-click it and select the aggregate function **Count**. The count field must be AccountNum.
2. On the CustTable data source, set the property **OrderMode** to Group by. The last step is to specify how the information should be sorted. Go to the *Sorting* node and add the field CustGroup. You will now have a query as shown in **figure 34:**

Aggregate function.

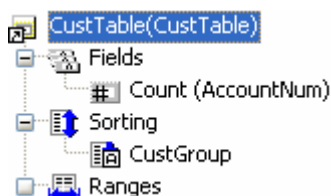


Figure 34: Aggregate function

3. The next step is to create a design to print the result. Create an auto design and choose *Generate Specs From Query* as done in the MyReport example. You will now have a body section for CustTable with one control printing the field CustGroup. Add the field AccountNum.
 4. Run the report. A row will be printed for each customer group. The AccountNum field will count the number of customers in each customer group.
-

When using aggregate functions data must be selected with the group by parameter set to **OrderMode**. The compiler will give an error if trying to select order by. This makes sense as information is retrieved record by record when using order by. When the OrderMode is set the group by MorphX will retrieve a single record for each group based on the sorting fields. This means only fields added as sorting fields will contain a value when using group by. You can add as many aggregate functions as needed. The case could be that you want to print a transaction list with an aggregation for min, max and average amounts.

The *Sorting* node under the *Data Sources* node is used for specifying how the output of the report should be sorted. This can be done either by using indexes or choosing fields. At least one index or sorting field should be defined. At runtime, the user may change the fields chosen for sorting. Keep in mind that the use of a field for sorting rather than an index may slow down your report.

The sorting fields added, have a property called **AutoSum**, this is used if you want MorphX to print subtotals when the value of the field changes. Auto Sums are explained in greater detail during the discussion of **Auto design**.

Ranges are used to filter the records processed by the report. The default ranges are specified using the *Range* node under the data source node. At runtime, the user may be allowed to add additional ranges or remove the default ranges depending on the property settings for the range. You can specify a default value for a range, and whether the user may change the specified values. Properties are may be set which specify that the range should be locked or hidden. If no ranges have been specified, the first element of each index for the table will be used as default ranges at runtime. Try executing the report MyReport. You will see a default set of ranges has been added. Now go back and add the fields AccountNum to *Data sources/CustTable/Ranges* node. When executing MyReport only the range AccountNum will be listed.

7.4 Templates

In Axapta you have two different types of templates, report templates and section templates. The templates are located as the two first entries under the *Report* node in the AOT.

Report template

Report templates are used to specify a report's basic formatting such as header and footer information. You can create templates for more advanced cases like using data from specific tables, but this will tend to limit where the template can be used. Report templates are usually used for information not related to a specific table, like caption, page numbering and lines. The template **InternalList** used in the MyReport example is a commonly used report template, which is formatting the caption name, setting

company name, page number, initials, date and time. To view the template, locate the template in the AOT, open the visual editor by right-clicking the template node and choose *Edit*. Should you decide to create a report template with controls from a specific table then any report using the template still must explicitly declare the table and fetch the required records.

Example 3: Report template

Elements used from MORPHXIT_Reports project

- Report template, MyInternalList
- Report, MyReport_MyInternalList

In this example you will create a new template based on the InternalList template. InternalList contains basic header formatting. A prolog and an epilog section will be added to the new template. The new report template will be used to extend the MyReport example.

1. Start duplicating MyReport and rename the new report to "MyReport_MyInternalList".
2. Go and locate the report template **InternalList** in the AOT. Right-click the report template and choose *Duplicate*. Rename the new report template to "MyInternalList". To compile MyInternalList without any errors, Right-click MyInternalList and choose *Restore*.
3. Right-click the report template name, choose *New* and select the report section *Prolog*. The prolog will contain a text and a new page feed. First the text to be printed must be defined. Go to *Prolog/Methods*, right-click and choose *New Method*. Open the new method and enter the following:

```
display description prologDescription()
{
    return sprintf("Start of report: %1", element.design().lookupCaption());
}
```

This method will return a "Start of report" string that contains the value of the report design's caption property. This method must then be referenced in the prolog's design. Close the editor and drag the method to the *Prolog* node. Axapta will create a string control which will print the value returned by the display method.

4. Right-click the report template name, choose *New* and select the report section *Epilog*. Now created the following method, and drag the method to the *Epilog* node.

```
display description epilogDescription()
{
    return sprintf("End of report: %1", element.design().lookupCaption());
}
```


- To have the prolog and epilog sections printed on new pages a new page feed must be added. Go to *Prolog/Methods*, right-click and choose *Override Method* and select `executeSection()`. Note that the call to the `newPage` method is placed after the call to `super()`. As a result the page-break will occur after the prolog section has printed:

```
public void executeSection()
{
    super();

    element.newPage();
}
```

- Add a new page feed to the epilog section. The new page feed must be executed before `super()` in the epilog section, as the epilog must be printed on a new page. You will now have a report template as seen in **figure 35: report template**.

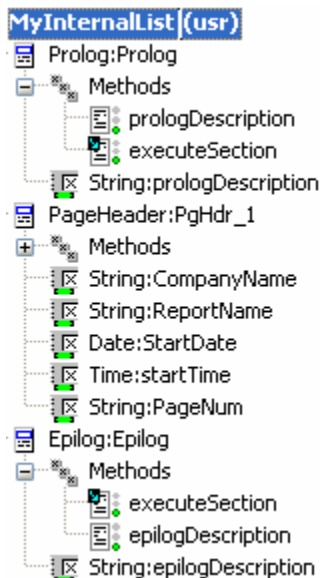


Figure 35: Report template

- The next step is to use the new report template in `MyReport_MyInternalList`. Go to the node *Designs/ReportDesign1*, open the property sheet and select `MyInternalTemplate` as the report template.
- Create a new menu item for the report. When the report is executed a page for the prolog will be printed before the report, and a page for the epilog will be printed after the report.

The `MyInternalList` template uses display methods for returning the values for the controls. Like with forms, you frequently use display methods when creating or modifying reports. This is one of the ways that you can print data which is not easily accessed through a query. In this example you use a display method to return a text

string, in other cases, it could be the result of a calculation. Simple create your display method and drag the method to the design. You do not have to worry about the type of the control to display the value appropriate. MorphX will handle this for you by checking the methods return type.

You ought to consider having one or two report templates, to be used for most of your reports. The advantage of using a report template is that it allows you to easily standardize the basic formatting of your reports. If you later on decide to change a report template, all reports with auto designs using the report template will automatically be changed.

Note: It is often an requirement to print both page number and the total number of pages printed. The template `InternalList` uses `element.page()` to print the current page number. The method `element.pagesTotal()` will return the total number of pages to be printed. `Element.pagesTotal()` can only be used as a return value for a display method with the return type integer. The total number of pages is calculated as runtime so you cannot use the method for any validations. To print something like: `<page>` of `<page total>` you will have to use 3 report controls.

Section template

The section templates were introduced in version 3.0. This could be the reason why they are rarely used and that you might not find any examples of their use in the standard package. A section template must be based on a table *map*. Table maps are explained in the chapter **Data Dictionary**. Fields from the map can then be added as controls. In cases where you have reports with similar section blocks, a section template may allow you to reuse the same piece of code, rather than rebuilding the same block section in multiple reports. However, in practice it may be easier to create one report and use X++ to modify the output, rather than having two reports and using a section template. The `SalesInvoice` report provides an excellent example.

7.5 Designs

Positioning fields and controls in your design is normally handled by MorphX. All controls will by default be auto adjusted as required. This means that the controls will be set to auto positioning and fonts and font size is defaulted from the user options in the toolbar menu **Tools | Options** in the tab page **Fonts**.

When you have chosen the row order MorphX will position the controls based on the information from the extended data types. This is very helpful, if you ever need to add a control in the middle of a row, or you want to hide a control, the following controls will be repositioned accordingly. For most reports, you should allow MorphX to auto position the controls, however in situations where your controls must always have a fixed position you can override the default settings. The disadvantage is that if you set a single control in a row to a fix position, you will have to define fixed positions for all

controls. This is generally not recommended unless your controls must fit the layout of a preprinted form or adhere to customer/vendor/government specifications.

If you need to print controls which must be positioned below each other in the same column you should consider using the property **ModelFieldName** (all report controls have this property). The position of the current control will adjust to the position of the control specified in ModelFieldName, if the current controls positioning is set to auto.

Creating design

A report can have more than one design. Under the Design node, you can create as many designs as needed. This can be useful if you have a form you want to print with different layout for each language or group of customers. Multiple designs within one report are not often used in the standard package. Instead of having several designs the need for different layouts is handled by X++, see the report SalesInvoice. In the SalesInvoice report, the method element.changeDesign() handles whether or not a control should be printed. It is often more time consuming to maintain differences in multiple designs than manipulating a single design using X++. Maintaining header sections across several designs is tedious work as it will take time to locate and verify that your changes are identical in all designs.

Note: If you are creating a report such as a form which must fit in a preprinted layout it may be necessary to do the final adjustment using the specific printer driver that will be used to produce the production output. Variations in printers can cause changes in location of where fields are located on the printed page. Often the layout will be adjusted according to the individual printer driver.

Designs can be created either as auto design or as a generated design. A design can also consist of both an auto design and a generated design. In this case only the generated design will be used. The main differences between auto designs and generated designs are that auto designs take full advantage of MorphX, they allow for dynamic templates, auto headers and auto sums based on criteria established in the query. Generated designs are static, and will not automatically adjust to changes made in the query or report template. It is recommended using auto designs. You should only consider using generated designs in special cases where a fixed layout is needed. Generated designs are generally only required where the layout is fixed by contract or statute, or when you need to use pre-printed forms such as checks and purchase orders.

Generated designs have some extra sections for adding headers and footers to body sections. Beside that auto designs and generated designs use the same type of sections. See **figure 36: Report design sections** for an overview of sections in a report design.

Type	Description
Prolog	This is the first section printed. The prolog is typically used for printing a logo or a title on the first page.
Page Header	The page header is printed at the top of each page. A report can have more than one page header.
Body	The body section is printed after the page header. This is the data section. The report will normally contain a body section for each data source.
Page Footer	Page footer is printed at the bottom of each page. A report can have more than one page footer.
Epilog	This is the last page printed.
Programmable Section	Programmable sections are executed from code. This type of sections can be used in cases, where you need to print data which is not part of the query.
Section Template	Section Templates is used for defining common used data, typically used in body sections. A section is based on a Map.
Header	Header is used in Generated Designs as body header.
Section Group	In Generated Designs, the Body section is added to a Section Group.
Footer	Footer is used in Generated Designs as footer for a body section.

Figure 36: Report design sections

You can also see an approximate image of the report by selecting view. The view option can be close to the printed result, however, if a report has a complex design like the SalesInvoice report, it can be difficult to figure out how the result will look when printed.

You have two options for adding controls to your design, either by using the nodes in the report tree as shown in previous examples, or by using the visual editor. The visual editor gives the option of either viewing or editing the controls in your design. To edit a report using the report tree, double-click the design node; if you want to use the visual editor right-click the design node and choose edit. Like the report view option, the visual editor can be difficult to use for complex reports, but it can be used when

building reports with relatively simple layouts. For creating a design the visual editor offers the same features as the AOT. From the visual editor you can change the properties for an element of the report and add or delete controls. To modify the report from the visual editor, simply position the cursor and right-click for the menu to edit, delete or add an element. To change the unit for the ruler right-click and choose between centimeters, inches or chars.

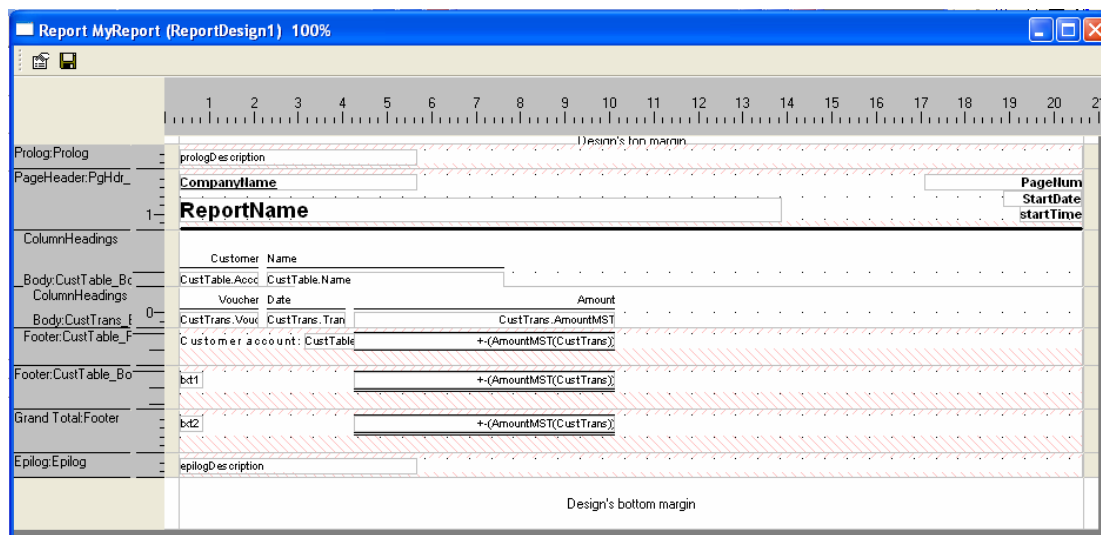


Figure 37: The visual editor

In practice, however, the visual editor is best for getting an overview of your design or to locate and change the properties for a specific control. The visual editor is relatively slow and most things can be done faster when working with the report tree.

Auto design

The most common way of creating the layout for your reports is by using auto designs. When using auto designs, you will only have to choose a report template and the fields to be printed from the query. MorphX will handle formatting the layout. If your report contains integer or real fields, the user will have the option of choosing summarization at runtime.

For a quick start creating your auto design, you can right-click on the auto design node and choose *Generate Specs From Query*, a body section will then be created for each data source in the query, and the sorting fields will be added as controls to the design. To get a visual view of your report, right click the auto design node and select View. Try opening MyReport in the visual editor. Notice that the visual editor will show the sections from the report template, even though the template sections are not a part of the reports nodes. This provides a useful overview of the report. To edit the report right click and choose edit. In edit mode only the nodes which are part of the report are accessible.

Auto sums is a useful feature in auto designs, it allows the user to specify at runtime where subtotals should be calculated. From the report dialog you can set break levels for subtotals for any field or body section as well as set a total for the whole report. From an application users point of view it might be the most important reason for using auto designs. It both makes your report more flexible, and eliminates most of the programming effort that might otherwise be necessary to hardcode these sums.

Example 4: Auto sumElements used from MORPHXIT_Reports project

- Report, MyReport_Sums
- Menu item output, MyReport_Sums

Extend MyReport with totals for the transaction amounts. A subtotal for each customer and a total for all customers will be added.

1. Start duplicating MyReport and rename the new report to “MyReport_Sums”.
 2. As an index does not have an option setting breaks when the value of an index field changes, the index AccountIdx will be removed from the sorting node of the CustTable data source, and instead specify the AccountNum field. Set the property **AutoSum** to Yes for the field AccountNum. You have now defined that each time the value of a customer account changes a subtotal will be printed.
 3. The fields to sum up must be defined. In this example only the fields AmountMST from the table CustTrans will be used. Find the control printing AmountMST in the CustTrans body section, open the property sheet and set **SumAll** to Yes.
 4. Create a new menu item for the report, and run the report. For each customer a subtotal will be printed.
 5. In this case only a subtotal was printed. To add a total for the whole report, close the report and go to the AOT again. Go to body section for CustTable and set the property **GrandTotal** to Yes. The body section CustTrans does not have this property, only CustTable have the property as it is the primary data source.
-

Step 2 specified the control that defined the report’s subtotals, but not which fields would be summed. The design specified which fields should be totaled, here AmountMST. These are the only required settings. The user will be able to do the rest at runtime. The sorting field for breaking the subtotal and the settings for the grand total only specify the report’s default settings and may be changed at runtime.

The *AutoDesignSpecs* node has a property called **GrandTotal**. This property will print a total for the report with the default label ”Super Grand Total” if set to Yes. Super Grand Total and the grand total set from either the report dialog or the body section will always give the same result. Both will print a total for the whole report. So if the user

has the option of setting the grand total from the report dialog, you should not use the super grand total.

The value of an auto sum can be accessed by using the method `element.sumControl()`. To access the auto sum control for `CustTrans.AmountMST` in the above example your code should look like the following:

```
element.sumControl(identifierstr(CustTrans_AmountMST), element.indent());
```

`Element.sumControl` will return the summed value. The first parameter is the control name of the field summed. The system function `identifierstr()` is used to prevent the best practice check from giving a warning. Always use `element.indent()` as the second parameter to set the correct indent level.

As you might have noticed there is a property called auto header. This is used in the same way as auto sum. Instead of printing totals, a header will be printed each time a sorting field breaks. The user can control auto header at runtime, but if needed you can default auto headers to be visible. Both auto sums and auto headers are features only available in auto designs.

All sections mentioned so far are triggered by either the reports framework or the reports query. You will have situations where you need to trigger a report section manually. In this situation, programmable sections which are executed from X++ are used.

Example 5: Programmable section

Elements used from MORPHXIT_Reports project

- Report, `MyReport_ProgSec`
- Menu item output, `MyReport_ProgSec`

You will add a programmable section to `MyReport`. For simplicity the section will just print a text control.

1. Start duplicating `MyReport`, and rename the new report “`MyReport_ProgSec`”. Go to the report design node *AutoDesignSpecs*, right-click the node and choose new *ProgrammableSection*.
2. Open the property sheet for the new programmable section and locate the property **ControlNumber**. The control number is used to reference the section from X++. Set the **ControlNumber** to 10.
3. Now add a control to the programmable section. Right click the programmable section and choose *New Control* to add a text control. Go to the new text control, located the property **Text** and enter “Header for customers”.

4. Now define the execution of the programmable section. Go to *MyReport_ProgSec/Methods*, right-click and choose *Override Method* and select *init()*. The *init()* method must look like:

```
Public void init()
{
    super();

    element.execute(10);
}
```

5. Create a menu item for the report *MyReport_ProgSec*. When executing *MyReport_ProgSec* the text 'Header for customers' will be printed before the query is traversed.
-

In the example you changed the programmable section's number to 10. It is recommended to leave gaps in the sequence of numbers that you assigned to programmable sections. This way, if at a later time, you need to add a new programmable section you will be able to preserve a logical numbering sequence.

The execution of a programmable section can be called from X++ where needed. However when used in combination with auto sums there are some things that you need to be aware of: Say you want to print a programmable section before a body section is printed, then the logical place to add your code will be in the *executeSection()* method just before *super()* in the body section. This will print your programmable section before the body section is printed, but the programmable section will also be executed before each auto sum. At runtime MorphX treats an auto sums as a footer, and this causes the body section to be executed again. The solution is to use the report methods *header()* or *footer()* instead. From here you can control which body section is being executed. When a body section is executed, the parameters *_tableId* and *_fieldId* will contain a value.

```
public void header(ReportSection _headerSection, tableId _tableId, fieldId _fieldId)
{
    if (_tableId == tableNum(custTable) && _fieldId)
        element.execute(10);

    super(_headerSection, _tableId, _fieldId);
}
```

Here the *header()* method is used. A check is made to ensure that it is the body section for the customer table which is printed. If so the programmable section is executed. Another way of using the *header()* and *footer()* method could be to add a new page after a sum if you are printing a batch of vendor or customer transactions.

Note: Programmable sections are often used to insert separators like blank rows or lines. Doing so can be done by setting the properties of the programmable section. You will need to add a "dummy" control to your programmable section as if the programmable section does not have any controls the section will not be printed.

Generated design

Using generated designs can at first glance seem easier to use than auto designs, as you have more sections which you can use to construct your report and all sections are visible. However generated designs are more static, the design is based on the settings from the query and property settings from the design node. The disadvantage is that if you have chosen a report template for your report and later on decide to change the report template or simply choose another template, it will not automatically update your design. Furthermore, the user will not be able to choose summarization when executing the report.

To have a closer look at a generated design MyReport will be used to create a generated design. Right click on the *Designs/AutoDesign1* node in MyReport and choose Generate Design. A new node called *Generated Design* has now been created below *Designs/AutoDesign1*. If you unfold the generated design you will notice that a design similar to the auto design is created. The difference is that the sections from the report template and the sections calculating totals have been added.

If you need to see an overview of your report when using auto designs, the option to create a generated design based on your auto design can come in handy, as all sections based on your templates and auto sums will be populated.

Controls in design

The most common way to add controls to your design is by dragging fields or display methods from a data source or direct from a table. When dragging a basic type field or display method such as string, enum, integer, real date and time, MorphX will automatically create a control of the same type. Controls like prompt, shape, sum and field group are used for more special purposes and must be added manually. You can of course add all type of controls manually, but it speeds up just dragging the controls as MorphX will auto position the control and fill out the properties with reference to either the field or the display method.

Auto designs and generated designs have the same controls. For an overview of the available controls see **figure 38: Report controls**.

Name	Description
String	Used for string values. If printing memo fields, the property DynamicHeight auto adjust the height of the control according to the number of lines printed.
Enum	Used to print the value of base enums.
Integer	Used for integer values.
Real	Used for real values.
Date	Used for printing dates. Dates will be formatted accordingly to the Windows regional settings.
Time	Used for printing the time. Time will be formatted accordingly to the Windows regional settings.
Text	Used for printing fixed text values. If the text must contain a dynamic value, display methods returning a string is normally used.
Prompt	Prompt will add text with following dots and a colon to the text.
Shape	Will draw a box. Size and position is specified by the properties. Can be used to build the layout for a formula.
Bitmap	Used for printing graphic. Enter path to the bitmap, refer to a container or use resources. For an example of using a resource as a bitmap, see the report HRMApPLICANTSTATUS.
Sum	Used for printing sums with generated designs. With auto designs sums can be used for printing a sum in a programmable section. For an example of the use in auto design, see the report SalesLinesExtended.
Field group	Used for adding a field group from the data dictionary. The field group on the report will automatically be updated if changes are made to the field group in the data dictionary.

Figure 38: Report controls

Bitmap controls can be a bit tricky to use. There are several ways that bitmaps may be configured. You can use icons from Axapta resources. Either you key-in the resource id in the property sheet or you can make a display method returning the resource id. To get an overview of the available resource use the report tutorial_Resources, the report prints the resource id and the corresponding icon. Other options are to enter a path to a bitmap or refer to a bitmap stored in a container. When referring to a path remember to use a double backslash.

The sum control in generated designs is used in footer sections. When used in auto designs the sum control must be put in a programmable section.

Note: When printing a report the infolog may say that the report is scaled to fit the page. This is caused by too many columns in your design. Set the property FitToPage to No on the *design* node to disable scaling.

When adding controls to your design you must assure that controls referring to a data source are fetched at the time the section is printed. For the MyReport example, you cannot print a control from the body sections CustTable and CustTrans in a page header section, because when the page header is processed MorphX has not fetched the information associated with the body sections. The same goes for adding a control to the body section CustTable with reference to a field from CustTrans. This will also give an error as CustTable is fetched before CustTrans.

7.6 Methods on a Report

You can create simple reports, such as a list of inventory items, without having to write X++ code by simply using the facilities provide by the report generator. For more advanced reports like those that filter data based on a caller or which require specific sorting, you will need to override the report methods with your own X++ code. For an overview of the methods on a report see **figure 39: Report methods**. The query methods are described in the query chapter, see **Queries**.

The report system classes are often used when modifying reports at runtime. They are the fundamental reports components and they allow the programmer to redefine all aspects of a report during execution. In fact, you can create a report from scratch using the system classes. For more information about the system classes, see the chapter **Classes**.

Name	Parameters	Description
CallMenuFunction	MenuFunction _menuFunction	Web method.
Caption	str _reportSpelling, str _reportName, str _designCaption, str _designName	Used for setting the caption for the report. The parameters _reportSpelling and _designCaption set the captions in the Print to Screen window.
CreateProgressForm		Overrides the standard progress form executed when the report pages are created. The method provides the option to create your own progress form.
Dialog	Object _dialog	Dialog() is used when adding fields to the reports dialog. The report runbase framework will call the dialog when the report is executed. See the report KMAAction.
Fetch		This method is the engine of the report. Fetch() opens the user dialog, selects the records from the database by processing the query and sending the records to be printed. This method is generally overridden, when an expression cannot be specified in a query. An example could be printing detail information as in the report HRMCourseSkills.
Footer	ReportSection _footerSection, tableId _tableId, fieldId _fieldId	The method is triggered each time a section in the design is executed. Since auto sums is not a part of the design, this gives the option to execute code before or after auto sums is printed.
GetTarget		Returns the selected print medium.
Header	ReportSection _headerSection, tableId _tableId, fieldId _fieldId	The method is triggered each time a section in the design is executed. As auto sums is not a part of the design, this gives the option to execute code before or after auto sums is printed.
Init		This is the first method called. The method is initializing the report. Entities used in the report are typically initialized here. See the report salesFreightSlip.

New	anytype _argsOrReportOrContainer, str _designName, boolean _isWebReport=FALSE	Used to initialize a reportRun object. This is normally not done from the report generator. The common case would be if a report is initialized from X++.
Pack		This method is used for storing last values. It is used in conjunction with unpack(), which loads the last value stored. However unpack() is not a base method. If a new dialog field has been added, pack() is overridden to store the values from the dialog. See the report KMAAction.
PageFormatting		The method is not used anymore. As of version 3.0 the method PrintJobSettings.PageFormatting() is used instead.
Print		Print() returns the number of pages to be printed. The method can be used to check whether or not there are any pages to print. See the report projTimeSheetEmpl.
PrinterSettings	int _showWhat=-1	Used to activate the different parts of the printer dialog. The method is only called if the report runbase framework is not in use, as it is called from the prompt() method.
ProgressInfo	int _pageNo, int _lineNo	ProgressInfo is executed for each line printed on the report.
Prompt	boolean _enableCopy=TRUE, boolean _enablePages=TRUE, boolean _enableDevice=TRUE, boolean _enableProperties=TRUE, boolean _enablePrintTo=TRUE	Prior to version 3.0 prompt() handled the report dialog. Now the dialog() method is used instead. The method cannot be used in combination with the report runbase framework since the framework will overrule the settings. The class PrintJobSettings must be used instead.
Run		Run() is called when the OK button is pressed in the dialog. Run() performs the following steps: If no generated design exists, a design is created on the fly based on the auto design. Call fetch() Call print()

		The method can be used for adding ranges to the query after the Based On settings in the dialog. See the report ReqPO.
Send		Send() is related to fetch(). Fetch() iterates through the query records, and send() sends the records to the design. The method can be overridden to validate whether or not the record should be printed. See the report CustTransList.
SetTarget	PrintMedium _target	Sets the target media for the report
ShowMenuFunction	MenuFunction _menuFunction	Web method.
ShowMenuReference	WebMenu _menuReference	Web method.
Title	str_title=""	Not of much use anymore. It can override the caption bar when printed to screen, if executed from fetch().
ExecuteSection		Each section in the design has the executeSection method, which is used to print the section. The method can be used to validate whether or not the section must be printed. See the report CustCollectionJour.

Figure 39: Report methods

The previous parts of this chapter have focused on the individual elements that make up reports. Now it is time to dig into how to use X++ to modify your reports.

Report Runbase Framework

You might have wondered why you sometimes get different dialogs when executing reports within Axapta. If a report is executed from the AOT, you will first get the query dialog and then the printer dialog. When a report is executed from a menu item you will get only one dialog. In version 3.0 of Axapta a new runbase class RunbaseReportStd was introduced. If a report is called from a menu item RunbaseReportStd will be called from the class SysReportRun.

This Report Runbase Framework is often a source of confusion for new Axapta programmers. It is important to understand that a report may be invoked in any of four ways:

1. Directly from the reports node in the AOT.

2. Through a menu item either on a user menu or directly from the AOT.
3. Invoked through a class that inherits from RunBaseReport.
4. Called directly though X++.

The class RunbaseReportStd is called by the Report Runbase Framework only if your report is not called from a class inherited from the RunBaseReport class. However when a report is executed directly from the *Reports* node, the Runbase Report Framework is not executed, and the two dialogs are shown.

Note: Reports are often used to check the integrity of data in the system. While it might be useful to have the report update data, the best practice is not to have reports write to the database. If your report must update or insert records, you should consider creating a class to perform the data manipulation and have the class called from a class inherited from RunBaseReport .

It is good practice to always create a menu item that executes your report so that you will have the same dialog shown as that shown to users when they launch the report from one of their menus. Keep in mind that a report executed from a class must always be inherited from the class RunBaseReport. The class RunbaseReportStd is only used internally by the runbase framework. For more information about the runbase classes, see the **Classes** chapter.



Figure 40: Runbase report classes

When creating a report in prior versions of Axapta, it was a common rule that the report had to be called from an inherited RunBaseReport class. This was done to wrap the two above mentioned report dialogs, and make the report capable of running in batch. It also facilitated better performance since the class could be set to run on the server. With the introduction of the RunBaseReportStd class, only reports with a heavy database load should be inherited from the RunBaseReport class.

Example 6: Report runbase

Elements used from MORPHXIT Reports project

- Class, SalesReport_DailyEntries
- Report, SalesDailyEntries
- Menu item output, SalesReport_DailyEntries

To explore the Report Runbase Framework, navigate to the *Classes* node and locate the class SalesReport_DailyEntries. This class is inherited from the RunBaseReport class and is therefore used to invoke a report. This is a common report class named with a prefix for the module. You will see many similar report classes when traversing the application classes.

SalesReport_DailyEntries calls the report SalesDailyEntries. Actually this class is not needed since the introduction of Axapta 3.0, since the logic of the class can be handled by the RunBaseReportStd class. However, the class SalesReport_DailyEntries is a good example of how to construct a report class. The class has the following methods:

```
public identifiername lastValueElementName()
{
    return reportstr(SalesDailyEntries);
}
```

Here, the name of the report is specified so that the method must be overridden. The function reportstr() ensures that the report name entered is a valid report name.

```
client server public static ClassDescription description()
{
    return "@SYS77491";
}
```

This optional method defines the caption name for the report dialog.

```
static void main(Args args)
{
    SalesReport_DailyEntries salesReport_DailyEntries;
    ;
    salesReport_DailyEntries = new salesReport_DailyEntries();

    if (salesReport_DailyEntries.prompt())
    {
        salesReport_DailyEntries.run();
    }
}
```

The main() method is a static method which initializes the class. This allows the class to be run so it can be executed from a menu item. A check is made to validate whether the dialog is called. If OK is pressed in the dialog, the report is executed.

In order to use the class in the report, a variable for the salesReport_DailyEntries class is declared in the class declaration for the SalesDailyEntries report:

```
public class ReportRun extends ObjectRun
{
    SalesReport_DailyEntries salesReport_DailyEntries;
}
```

```
public void init()
{
    super();

    salesReport_DailyEntries = element.args().caller();

    if (!salesReport_DailyEntries)
    {
```



```

        throw error(Error::missingRecord(funcName()));
    }
}

```

An instance of the SalesReport_DailyEntries class is passed through args().caller() into the report. A check is made to validate whether the report is called from a class. This is done to prevent the report being executed directly from X++ or the AOT. In this example, it would not matter which report source was called, but in some cases the class may be filtering the data to be printed.

As mentioned earlier, the SalesReport_DailyEntries class is not needed since the internal RunBaseReportStd class will handle the logic required to display the query/print dialog and automatically facilitate batch processing. To change the report so that it does not use the SalesReport_DailyEntries class, modify the init() method of the report. In this case you can simply delete init(). The menu item still refers to the class, so you will have to go to the output menu item SalesReport_DailyEntries and change the properties for the menu item so it calls the report instead. When executing the report directly from the AOT, you will get the same result as if you were using the class SalesReport_DailyEntries. If you executed the report from a menu, you will see the consolidated dialog screen.

Example 7: Report dialog

Elements used from MORPHXIT_Reports project

- Report, SalesDailyEntries
- Menu item output, SalesDailyEntries_Without_Class

Now it is time to add some more features to the SalesDailyEntries report. A dialog field for specifying whether details must be printed will be added. The value of the new field will be stored so that the last value is loaded when executing the report. The following must be added to the ClassDeclaration of the report:

```

public class ReportRun extends ObjectRun
{
    DialogField dialogPrintDetails;

    Boolean    printDetails;

    #DEFINE.CurrentVersion(1)
    #LOCALMACRO.CurrentList
        printDetails
    #ENDMACRO
}

```

DialogPrintDetails is a variable of the class DialogField and is used for the new dialog field that will be displayed to the user when the report is run. The variable printDetails stores the value of the dialog field. The macro CurrentList is a list of variables to be stored. The list will typically contain a variable for each field in the dialog. In this example CurrentList only contains a single variable. To add more, simply separate the

variables by a comma. CurrentVersion keeps track of the stored version of CurrentList. Axapta allows the parameters of batch jobs and reports to be saved from one execution to the next. The system will then preload the parameters that the user last used to run this job. If changes are made to CurrentList, then CurrentVersion must be incremented by one. It is also possible to reset the usage data which will have the effect of the stored values pertaining to prior job executions, but this will also reset other values such as any selection ranges that set up by the user.

```
public Object dialog(DialogRunbase _dialog = null)
{
    DialogRunBase dialog;
;

    dialog                = super(_dialog);
    dialogPrintDetails    = dialog.addFieldValue(typeId(NoYesId), printDetails, "Print details",
                                                "Print additional information for the transactions.");

    return dialog;
}
```

The dialog is initiated from super() and contains the default dialog for the report. The only thing needed is to add the new field for printing details. The new field will automatically be put in a default field group called Parameters.

```
public boolean getFromDialog()
{
    boolean ret;

    printDetails          = dialogPrintDetails.value();
    ret                   = true;

    return ret;
}
```

When OK is pressed in the dialog screen, the system calls the getFromDialog method. The value from the new dialog field is stored in the printDetails variable.

```
public container pack()
{
    return [#CurrentVersion, #CurrentList];
}
```

As shown here, the last value from the new dialog field is stored. Pack() saves the current value of CurrentVersion and CurrentList specified in the ClassDeclaration.

```
public boolean unpack(container packedClass)
{
    boolean    ret;
    Integer    version = conpeek(packedClass,1);

    switch (version)
    {
        case #CurrentVersion:
            [version, #CurrentList] = packedClass;
            ret = true;
    }
}
```

```

        break;
    default:
        ret = false;
    }
    return ret;
}

```

This method loads the last value stored using `CurrentVersion` and `CurrentList`.

```

public void run()
{
    if (printDetails)
    {
        SalesLine_Name.visible(true);
    }
    else
    {
        SalesLine_Name.visible(false);
    }

    super();
}

```

The final step is to check whether or not details must be printed. In this example, the difference is whether the `SalesLine.name` is printed or not. All report controls must be declared before they can be referenced through `X++`. You have the ability to change properties at runtime. A report control is declared by setting its **AutoDeclaration** property to Yes. In the above example you must first set the `AutoDeclaration` property for the `SalesLine_Name` control to Yes.

In the report dialog example, the methods `pack()` and `unpack()` were used. These are the methods used to store the last value of a dialog and transfer the user parameters specified in the dialog from the client to the server when the report will be executed in batch mode. If you need this functionality for your dialog, you can just copy the two methods from an existing class and add `CurrentVersion` and `CurrentList` to the `ClassDeclaration`.

Dynamic Reports

The ability to make changes at runtime is very useful, since it provides the option of changing properties or adding new elements at runtime. Thus, you can create one report, instead of having several reports with similar designs. The report `SalesInvoice` is an example of this. Depending upon the sales parameter settings, `SalesInvoice` is printed with different controls visible.

As Axapta is a multi-language system, it has to be able to print reports in multiple languages. In most cases, Axapta will handle this correctly without the need for additional programming. By default, the report will be printed in the language of the Axapta user. However exceptions may occur in certain circumstances. For example, a sales invoice report must be printed in the customer's preferred language. This is done by setting the **Language** property on the *ReportDesign* node. You can set the property

to a fixed value, but the best way would be to set the language from X++. Here `myTable.languageId` is used to set the language for the design:

```
public void init()
{
    super();

    element.design().languageId(myTable.languageId);
}
```

The keyword `element` is used within the report to reference all of the report's objects. Here, the `element` is referring to the method `languageId()` in the object design. In this way, you can get a handle on each element of the report. Using `element` to reference a report object might result in a long path, as shown here:

```
element.design().sectionGroup(tablenum(CustInterestTrans)).section(ReportBlockType::BODY).control
Name('custInterestTrans_custInterestJourInterestAmount');
```

The above line is from the method `init()` in the report *CustInterestNote*. A better way to handle this situation is to use the property **AutoDeclaration**. When using the `AutoDeclaration` property you are actually declaring an instance of a system class. `SalesLine_Name` used in the report dialog example is an instance of the system class `ReportStringControl`.

Note: When browsing the standard reports in the AOT, you may see reports where system classes for sections and controls have been declared manually from code, rather than using the `AutoDeclaration` property. This is because the `AutoDeclaration` property for report controls was only added in v3.0 of Axapta.

The system classes are meant to be used when you need to add elements such as sections and controls to a report at runtime. For example, based on where your report is called from, you could decide to print additional detailed information or even print an extra section from a different table.

Instead of using the system classes to create the controls at runtime, another approach is to add all of the sections and controls needed for the different combinations and then use the **Visible** property to determine whether the control should be displayed or hidden. In some cases using the system classes is preferable. By using system classes for creating controls, you can wait until runtime to decide the type of control needed. This is especially useful if the controls you need to add are dependent on the user settings or parameters.

Example 8: Report system classes

Elements used from MORPHXIT_Reports project

- Report, `MyReport_SystemClasses`
- Menu item output, `MyReport_SystemClasses`

To try out the report system classes, create a new report as shown in **figure 41: Test of report system classes**. At runtime, the report will create a body section for CustTable. The body section will have 10 controls, printing the value of the first 10 fields from CustTable. The table CustTable is added as a data source and the node for the auto design has been created.

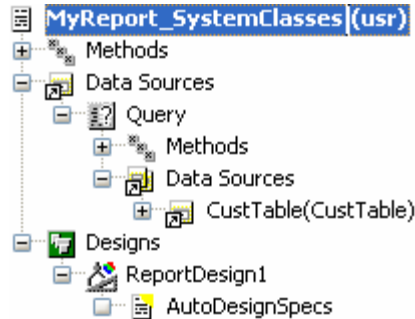


Figure 41: Test of report system classes

Override the `init()` method of the report as shown here. No other code will be required:

```
public void init()
{
    ReportSection    reportSection;
    DictTable        dictTable;
    DictField        dictField;
    Counter          fieldCounter;

    super();

    reportSection = element.design().autoDesignSpecs().addSection(ReportBlockType::Body);
    reportSection.table(tableNum(custTable));

    dictTable = new DictTable(tableNum(custTable));

    while (fieldCounter < 10)
    {
        fieldCounter++;
        dictField = new DictField(dictTable.id(), dictTable.fieldCnt2Id(fieldCounter));
        reportSection.addControl(dictTable.id(), dictTable.fieldCnt2Id(fieldCounter));
    }
}
```

The instance of the system classes `reportSection` and `reportControl` is used to create the body section and its associated controls. A new report section of the type Body Section is added to the *AutoDesignSpecs* node and specified to use the table `CustTable`. `DictTable` is also an instance of a system class. `DictTable` is often used when a handle is needed for table and field properties. Similarly, `DictField` is a system class that provides a handle to a specific field within a specified table. Here `dictField` and `DictTable` are used to loop the first 10 fields in `CustTable`. For each loop, a control is added to the body section.

MorphX will handle adding the proper control type and auto adjust the controls, so that when you run the report, you will have the first 10 fields of the table CustTable printed in a row. If you are creating a module where the user must have the option of defining his/her own layout of a report, system classes could be the answer.

Common Report Methods

When making modifications to a report, you are either overriding existing methods, or adding new methods which are called from the overridden methods. The following methods are executed in listed order when a report is loaded as shown here:

init() ► dialog() ► run() ► fetch() ► send() ► print()

1. Init() and dialog() are triggered when the report is loaded.
2. Run() is triggered when the OK button is pressed in the dialog.
3. Fetch() is looping through the query and for each record found send() is triggered.
4. Finally, print() is triggered.

These methods are the most important within a report, and are the ones you will override most often. Typical customizations include adding controls to a dialog, manipulating the output from the query before it is printed, or adjusting the output by executing a programmable section which must be printed within the report body section.

The above listed execution order is used when the report runbase framework is in effect. If you call your report directly from the AOT without using a menu item, the execution order of the methods is slightly different as shown here:

init() ► run() ► prompt() ► fetch() ► send() ► print()

Notice that dialog() will not be triggered. RunBaseReportStd controls the dialogs of the report and when the “runbase framework is not active” prompt() is used.

Example 9: Overriding fetch()

Elements used from MORPHXIT Reports project

- Report, MyReport_Fetch
- Menu item output, MyReport_Fetch

A new report will be created printing customer transactions for each customer, filtered from n-days until the current system date. The report will have a dialog where the n-days number is keyed in by the user. Start by duplicating the MyReport example. The report will end up as shown in **figure 42: Report for overriding fetch()**. The example will focus on overriding the methods.

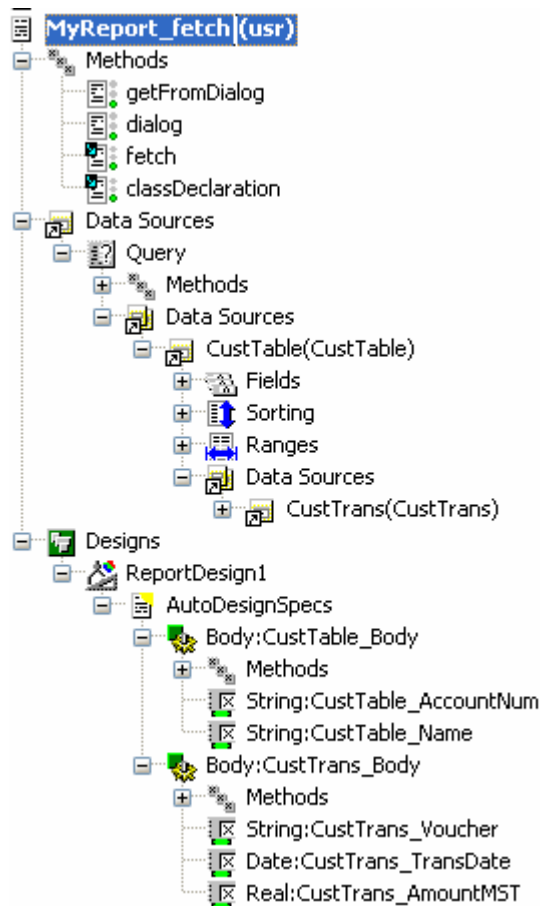


Figure 42: Report for overriding fetch()

Now that you have created the query and the design for the report, you are ready to create the methods for the report.

```
public class ReportRun extends ObjectRun
{
    DialogField    dialogDaysBack;
    NumberOf      daysBack;
}
```

The variable `dialogDaysBack` is needed for the dialog. The value will be stored in the variable `daysBack`.

```
public Object dialog(Object _dialog)
{
    DialogRunBase dialog;
    ;

    dialog                = super(_dialog);
    dialogDaysBack = dialog.addFieldValue(typeId(NumberOf), daysBack, "Number of days",
                                         "Number of days back to be printed.");

    return dialog;
}
```

A field is added to the dialog to enter the n-1 days.

```
public boolean getFromDialog()
{
    boolean ret;

    daysBack = dialogDaysBack.value();
    ret      = true;

    return ret;
}
```

The variable daysBack is set to store the value from the dialog.

```
public boolean fetch()
{
    QueryRun          qr;
    QueryBuildRange    rangeTransDate;
    Boolean            ret;

    qr = new QueryRun(element);

    rangeTransDate =
        element.query().dataSourceTable(tablenum(CustTrans)).addRange(fieldnum(CustTrans, transDate));
    rangeTransDate.value(queryRange(systemdateGet()-daysBack, systemDateGet()));
    rangeTransDate.status(RangeStatus::LOCKED);

    element.design().caption(strfmt("%1, %2", element.design().caption(), rangeTransDate.value()));

    if (qr.prompt() && element.prompt())
    {
        while (qr.next())
        {
            custTable = qr.get(tablenum(CustTable));
            custTrans  = qr.get(tablenum(CustTrans));

            if (!custTable)
            {
                ret = false;
                break;
            }

            if (qr.changed(tablenum(custTable)))
            {
                element.send(custTable, 1);
            }

            if (qr.changed(tablenum(custTrans)))
            {
                element.send(custTrans, 2);
            }
        }
        ret = true;
    }
    else
```



```

    ret = false;

    return ret;
}

```

The variable `daysBack` contains the value that the user has keyed in. A range must be added to the query to filter the transaction date with `n-1` days to system date. A `QueryRun` object called `qr` is initialized with the active query of the report, and then a range for the customer transaction date is added to `qr`. The range is locked, so the user cannot change it. The transaction date range is added to the caption of the report.

At this point, the query is looped. The standard loop of the query and the printout of the record is what the `super()` call in `fetch()` handles. Before the query is looped, there is a check to see whether the dialogs for the query and the report are called. These are the two dialogs which are wrapped by `RunBaseReportStd`. Within each loop, the tables `CustTable` and `CustTrans` are initialized. If no records are found, the loop breaks and the report ends. If a data source has changed, a new record is found and the record is printed using the `send()` method. Note the second parameter in the `send()` method. The second parameter defines the level of the record. `CustTable` is on the first level of the query and `CustTrans` is on the second level. This is important since, if it is not set correctly, auto sums will not be printed.

In the `fetch` example, the query of the report was looped. The case could also be looping a `WHILE` statement, `SELECT` statement, or a combination of both. For each record looped in the query, you might want to select a record from a table which is not part of the query, or even build a temporary table to be printed. For each record to be printed, all you have to do is call the `send()` method.

If your objective is to validate which records should be printed, override the `send()` method instead of the `fetch()` method:

```

public boolean send(Common _cursor, int _level=1, boolean _triggerOffBody=TRUE, boolean
    _newPageBeforeBody=FALSE)
{
    boolean    ret;
    CustTrans  custTrans;

    if (_cursor.tableId == custTrans.tableId)
    {
        custTrans = _cursor;
    }

    if (custTrans.transDate == systemDateGet())
    {
        ret = super(_cursor, _level, _triggerOffBody, _newPageBeforeBody);
    }

    return ret;
}

```

The `send()` method has the record to be printed as a parameter. All you need to do is initialize the appropriate table. In this case, the table `CustTrans` is initialized if the

cursor is a CustTrans record. Only customer transactions with a transaction date equal to the system date will be printed.

Adding query values can often be handled by simply overriding the `init()` method, which is much easier as only a few lines of code are needed. The range added to the query in the fetch example was dependent upon user interaction. The modification had to be done after the dialog was closed, so the code had to be put in `fetch()`. Before adding a range to a query, you should always check to see whether or not the query already contains a range for the field, by using the `QueryBuildDataSource.findRange()` method. If two ranges are created for the same field, the ranges will be OR'ed which may give you unexpected results. The user has an option in the report dialog to print ranges for a query, when sending to a printer. Ranges added to the query data sources as well as the ones added from X++ will be printed. However, if `fetch()` is overridden, this option will be disabled.

7.7 Special Reports

Until this point, this chapter focused on the basic steps in creating reports. To get an idea of the options within the MorphX languages this section will provide examples showing how to handle special reports and how you can make your reports more user-friendly.

Execute report from X++

Reports are normally activated from a menu item, called from either the main menu or from a form. Sometimes it is necessary to call the report directly from X++ since the user may not call the report directly from a menu item.

Elements used from MORPHXIT_Reports project

- Job, Reports_ExecuteReport
- Job, Reports_ExecuteReportSilent

```
static void Reports_ExecuteReport(Args _args)
{
    Args          args;
    SysReportRun  reportRun;
;

    args = new Args();

    reportRun = new menuFunction(menuItemOutputStr(MyReport),
                                MenuItemType::Output).create(args);
    reportRun.run();
}
```

The job shows how `MyReport` is called from X++. Notice that the application class `SysReportRun` is used. `SysReportRun` is inherited from the system class `ReportRun`. The benefit of using the application class is that you have the option of overriding the

code in the SysReportRun class. You could also create your own extended class inherited from the SysReportRun class. This could be the case if you have a series of reports where checks must be made when printing, rather than modifying each single report.

MyReport is called using the menu item since this will trigger the runbase report framework and load the correct dialog for the report. If you do not want to use the runbase framework or you want to print the report without user interaction, you must instead trigger the report without using the runbase report framework.

```
static void Reports_ExecuteReportSilent(Args _args)
{
    Args          args;
    SysReportRun  reportRun;
;
    args = new Args();
    args.name(reportstr(MyReport));

    reportRun = classFactory.reportRunClass(args);
    reportRun.query().interactive(false);
    reportRun.report().interactive(false);
    reportRun.setTarget(PrintMedium::Printer);
    reportRun.run();
}
```

In this job, the report is executed without using the menu item and thereby the runbase report framework is not used. The example prints the report direct to the default printer without user interaction as both the query and the report dialog are set to inactive. If dialog() is overridden in your report, you must ensure that this does not create any problems since dialog() will not be executed.

If your report consists of more than one design, you must specify which design to use. If not specified, or if an invalid design name is entered, the first design for the report will be used.

```
reportRun.design("MyDesign");
reportRun.run();
```

Using temporary tables

If a special sorting is needed, or you must select data from several tables which cannot be joined, using a temporary table may be the answer. Using temporary tables is fairly simple. The temporary table must be filled and passed to the report. There may be performance issues when using temporary tables, since the report will have two runs. First the temporary table is built. Next it is looped in the report. The use of temporary tables should not be your first choice. You might be better off reconsidering your design. For more on temporary tables see the chapter **Data Dictionary**.

Elements used from MORPHXIT_Reports project

- Class, Reports_TempTable
- Report, Reports_TempTable

When using temporary tables the report should be called from a class. This will give you the option building the temporary table on the server. The following example shows how to create a report using a temporary table. For simplicity, the example adds 10 records to a temporary table and prints the result.

```
class Reports_TempTable extends runBaseReport
{
}
```

The class is inherited from runBaseReport.

```
public identifiername lastValueElementName()
{
    return reportstr(Reports_TempTable);
}
```

The name of the report is specified.

```
tmpAccountSum tmpTable()
{
    CustTrans          custTrans;
    TmpAccountSum      tmpAccountSum;
    Counter            counter;
;

    while select custTrans
    {
        counter++;

        if (counter == 10)
        {
            break;
        }

        tmpAccountSum.accountNum      = custTrans.accountNum;
        tmpAccountSum.currencyCode    = custTrans.currencyCode;
        tmpAccountSum.balance01       = custTrans.amountMST;
        tmpAccountSum.insert();
    }

    return tmpAccountSum;
}
```

The temporary table tmpAccountSum is used. The first ten records from the table CustTrans are inserted into tmpAccountSum. This method is used by the report to pass the buffer for the temporary table to the report.

```
static void main(Args args)
{
    Reports_TempTable reports_TempTable = new reports_TempTable();

    if (reports_TempTable.prompt())
```

```

{
    reports_TempTable.run();
}
}

```

The class is initialized and the report is executed.

The last step is to create the report. A report with the temporary table TmpAccountSum as data source must be created. The three fields filled out with values from CustTrans will be printed. Your report must look like **figure 43: Report using temporary table**.

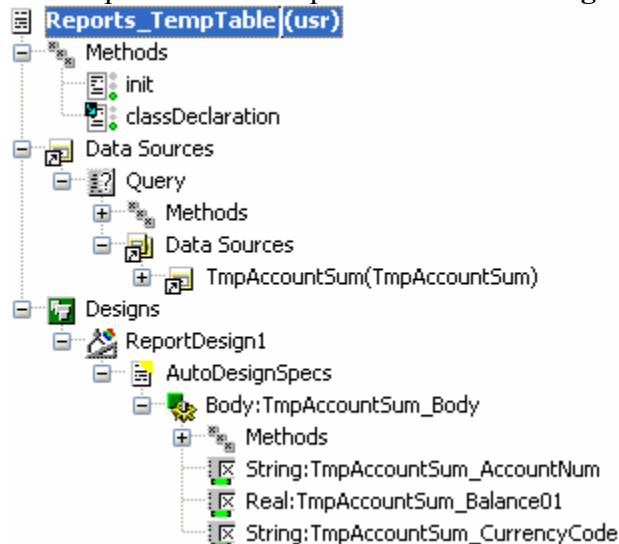


Figure 43: Report using temporary table

Init() must be overridden. The runbase class is initialized and the query is set with a buffer to the temporary table. Notice that you will have to set a reference to the buffer. The query will then have the full scope of the temporary table, and loop through all of the records in the temporary table.

```

public void init()
{
    Reports_TempTable    reports_tempTable;
;
    super();

    reports_TempTable = element.args().caller();

    if (!reports_TempTable)
    {
        throw error(Error::missingRecord(funcName()));
    }

    reports_TempTable.queryRun().setRecord(reports_TempTable.tempTable());
}

```

Coloring rows

Colors are rare in the standard packaged reports. You will need to fiddle a bit with the report to obtain the desired result. However, the use of colors can give your report the final touch and even make the printout easier to read.

Elements used from MORPHXIT_Reports project

- Report, MyReport_Color
- Menu item output, MyReport_Color

This example shows how to color a single column based on a condition. The report will set the background color to control the printing of CustTrans.amountMST. To simplify coding, the condition check is done from X++. In real life, the conditions could be set up in a dialog or based on data in a form. You should end up with a design as shown in **figure 44: Report coloring rows**.

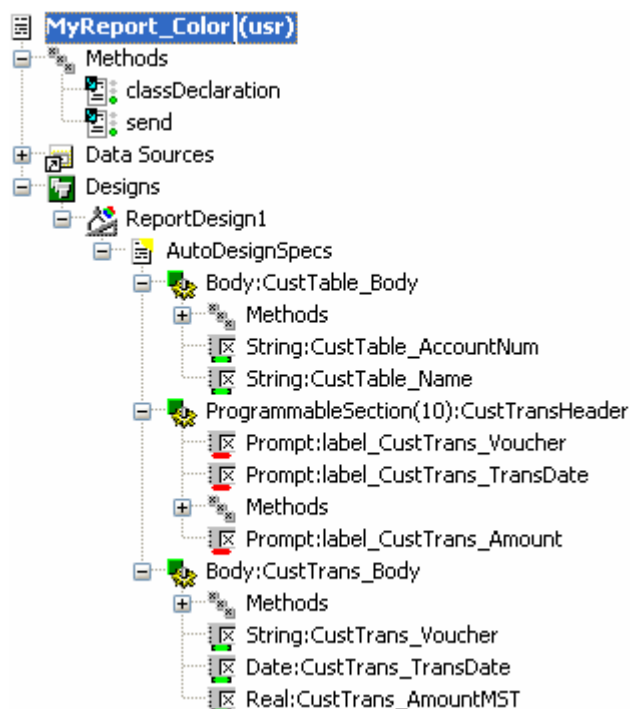


Figure 44: Report coloring rows

1. Start out by duplicating the report MyReport and rename it “MyReport_Color”.
2. The goal is to color the control CustTrans_AmountMST. By using the MyReport example as is, the header label will also change color. Instead of the standard header for the body section CustTrans_Body, a new one must be created. To skip the standard header, set the property **NoOfHeadingLines** to “0” in the body section CustTrans_Body.

3. Create a new header by adding a programmable section and a prompt control for each of the three fields in the body section. The property ModelFieldName on each prompt control must be set to the corresponding body section control name. Add a label for each prompt control.
4. Declare a variable to keep track of when the programmable section used for the header must be printed.

```
public class ReportRun extends ObjectRun
{
    Boolean    printCustTransHeader;
}
```

5. Now override send(). If the current record is a CustTrans record, the header is printed for the first CustTrans in a row. A condition is set up for CustTrans.AmountMST. If an amount larger than 500 is printed, the background color is set to yellow. Otherwise the background color will be neutral.

```
public boolean send(Common _cursor, int _level=1, boolean _triggerOffBody=TRUE, boolean
    _newPageBeforeBody=FALSE)
{
    boolean ret;
    ;

    if (_cursor.tableId == tableNum(custTable))
        printCustTransHeader = true;

    if (_cursor.tableId == tableNum(custTrans))
    {
        if (printCustTransHeader)
        {
            element.execute(10);
            printCustTransHeader = false;
        }

        if (custTrans.amountMST > 500)
        {
            CustTrans_AmountMST.backgroundColor(Winapi::RGB2int(255, 255, 0));
        }
        else
        {
            CustTrans_AmountMST.backgroundColor(Winapi::RGB2int(255, 255, 255));
        }
    }

    ret = super(_cursor, _level, _triggerOffBody, _newPageBeforeBody);

    return ret;
}
```

Print using Microsoft Word

Creating reports in Axapta with a complex design such as a formula with boxes, tables and graphic can be a challenge. By using the COM interface you can connect to external applications like Microsoft Word. To use Microsoft Word for printing data, you must first create a Microsoft Word template with bookmarks. The bookmarks are used to position the data from Axapta. Notice, that you must have a license code for at least one COM client to use the COM interface.

Note: The Document handling in the standard package uses the COM interface to attach files from Microsoft Excel and Microsoft Word. The classes used by Document handling are prefixed with DocuActionCOM.

Elements used from MORPHXIT_Reports project

- Class, Reports_PrintUsingWord
- Job, Reports_PrintUsingWord

Additional

- Microsoft Word template, Reports_WordTemplate.dot

This example will show how to connect to Microsoft Word and create a new document which prints data from the table InventTable. The first 10 records from InventTable will be printed. Labels will be printed for report and column headers.

You will have to create a Microsoft Word template, with the following bookmarks: label_header, label_itemid, label_itemname, and label_itemdesc. Label_header will print a heading text for the columns. Create a table and add the remaining 3 bookmarks as headers for the table. The next step is to create the following class:

```
void run()
{
    COM      COMAppl, COMDocuments, COMDocument;
;

    COMAppl      = new COM('Word.Application');
    COMDocuments = COMAppl.documents();

    // enter path to the template Reports_wordtemplate.dot
    COMDocument = COMdocuments.add('d:\\Reports_WordTemplate.dot');

    if (COMDocument)
    {
        this.setLabels(COMDocument);
        this.sendInventTable(COMDocument);
        this.showDocument(COMAppl);
    }
}
```

Run() will initiate the COM connection, and open a new Microsoft Word document based on the template Reports_WordTemplate.dot. Remember to check that the path for the template is valid. If the document is created, labels and data will be added. Finally

the document will be shown. Note that the document shown is not saved. If you want to save the document you must add: `COMdocument.saveAs(<filename>,0,false,"",false);`.

```
void setLabels(COM _COMDocument)
{
    COM          COMBookmarks, COMBookmark, COMrange;
    DictField    dictField;
    Label        label;
;

    COMBookmarks = _COMDocument.bookmarks();

    if (COMbookmarks.exists('label_header'))
    {
        COMbookmark = COMbookmarks.item('label_header');
        COMrange    = COMbookmark.range();
        COMrange.InsertAfter("Inventory list");
    }

    if (COMbookmarks.exists('label_itemId'))
    {
        COMbookmark = COMbookmarks.item('label_itemId');
        COMrange    = COMbookmark.range();
        DictField    = new dictField(tableNum(inventTable), fieldNum(inventTable, itemId));
        COMrange.InsertAfter(dictField.label());
    }

    if (COMbookmarks.exists('label_itemName'))
    {
        COMbookmark = COMbookmarks.item('label_itemName');
        COMrange    = COMbookmark.range();
        DictField    = new dictField(tableNum(inventTable),
                                     fieldNum(inventTable, itemName));

        COMrange.insertAfter(dictField.label());
    }

    if (COMbookmarks.exists('label_itemDesc'))
    {
        COMbookmark = COMbookmarks.item('label_itemDesc');
        COMrange    = COMbookmark.range();
        label       = new Label(CompanyInfo::languageId());
        COMrange.InsertAfter(label.extractString(literalstr("@SYS58702"))));
    }
}
```

A check is made to see whether the bookmark can be found. If found, the labels are set. The header label is set with the static text “Inventory list.” The labels for the bookmarks `label_itemId` and `label_itemName` are set with the label for the corresponding table fields. The label for the bookmark `label_itemDesc` is set using the method `label.extractString()` to fetch the label for the default company language.

```
void sendInventTable(COM _COMDocument)
{
    COM          COMTable, COMRows, COMRow;
```

```

COM      COMCells, COMCell, COMRange;
InventTable  inventTable;
Counter      counter;
;

//init tabel
COMTable      = COMDocument.Tables();
COMTable      = COMTable.Item(1);
COMRows       = COMTable.Rows();

while select inventTable
{
    counter++;

    if (counter == 10)
    {
        break;
    }

    // add new row
    COMRow      = COMRows.Add();
    COMCells    = COMRow.Cells();

    // item id
    COMCell      = COMCells.Item(1);
    COMRange     = COMCell.Range();
    COMRange.InsertAfter(inventTable.itemId);

    // item name
    COMCell      = COMCells.Item(2);
    COMRange     = COMCell.Range();
    COMRange.InsertAfter(inventTable.itemName);

    // item description
    COMCell      = COMCells.Item(3);
    COMRange     = COMCell.Range();
    COMRange.InsertAfter(inventTable.itemDescription());
}
}

```

This code loops through the inventTable. The table in Microsoft Word is initiated first. For each loop, a new row is added to the table in Microsoft Word. The fields itemId, itemName and the display method itemDescription() from InventTable is added to the three rows in the Microsoft Word table. Notice that no bookmarks are needed.

```

void showDocument(COM _COMAppl)
{
    _COMAppl.visible(TRUE);
}

```

The created Microsoft Word document is shown.

```

static void main(Args _args)
{
    Reports_PrintUsingWord  printUsingWord = new Reports_PrintUsingWord();
;

```

```
printUsingWord.run();  
}
```

This code initiates and executes the class.

7.8 Summary

This chapter introduced you to reports in Axapta. It covers the basics involved in creating reports. By now you should be familiar with the various report elements, such as data sources, designs, sections and controls in designs and the common methods used when creating a report. You should also have acquired knowledge about how to create reports using the report generator, and hopefully gained insight into the power offered by reports in Axapta and the MorphX development environment.

8 Queries

Select statements and queries are the two options for fetching data from the database when using MorphX. Where a select statement is a static expression written in X++, a query can either be written in X++ or created by using the AOT node *Queries*. A query can at runtime be changed by the application user by using the query dialog. From X++ the query can be change using the query system classes. This makes queries quite useful when working with objects like forms and reports as data can be filtered and sorting changed either by using the query dialog or from X++.

Settings made by the application user to the query dialog are saved per user in each company, also referred to as last values. Storing last values makes the use easier for the application users as their preferable settings only needed to be set at the first run.

Whether using a query or a select statement will not matter regarding performance. Selects and queries are both part of MorphX and both are executed by the kernel. Using a select or a query must depend on your needs. Queries should be used if you want the application user to be able to filter data or if you need to do runtime changes like filtering data in a form or a report depending on the calling object. Selects are typically used if the data fetched are used without user interference.

This chapter will focus on creating and using queries from MorphX. You can find information on the use of the query user interface by check the manuals in the standard package.

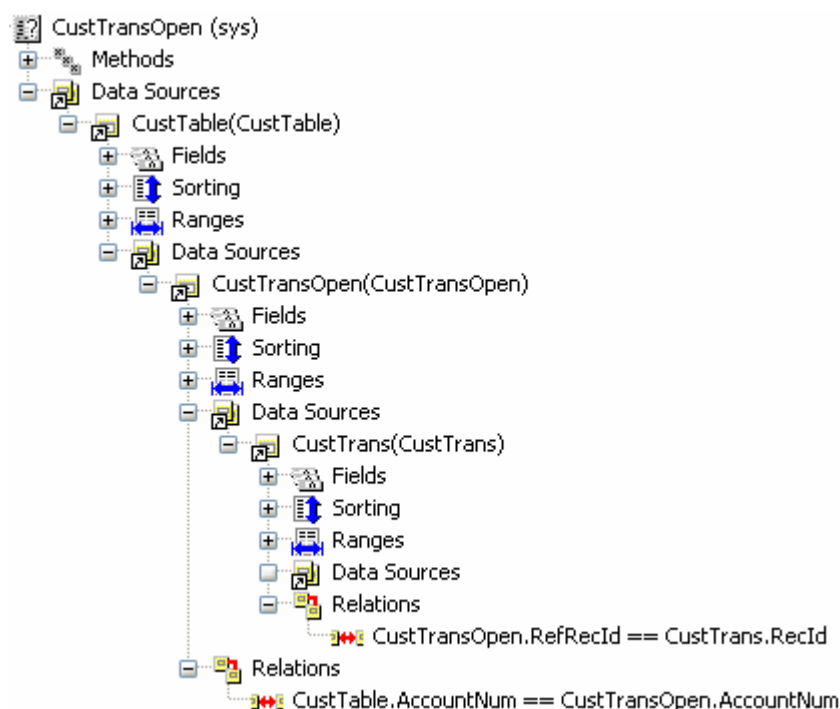


Figure 45: Query stored in the AOT

8.1 Building Queries

Complex queries and queries where the default methods are overridden should be built using the *Queries* node. A query stored as a node in the AOT is constructed using a tree making it easy to get an overview of single components of the query. Especially when start learning about queries you might find it far easier, as you will not have to worry about which system classes to use and how to use them.

AOT Query

Queries stored in the AOT can be used in any part of your code. An AOT query cannot be declared like a type or a table. You will have to use a system classes to execute your query. Still it makes an AOT query very flexible as only a few code lines is required to integrate your query in any part of code. If you later on decide to change your AOT query like changing the way data is filtered, you changes will reflect all places the query is used. The disadvantage is that it can be unclear which query to use. If you find a query in the AOT fitting your needs you will not now where the query is used, unless using the cross reference system. Making changes to such a query could have fatal consequences. So you will probably ending up creating your own query in the AOT.

Basic Components

The first step creating a query is to localize the tables to be used for the query. A table used in a query is referred to as a data source. You can compare a data source with a table variable. Normally a data source has the same name as the related table. Only if you need a table to be used more than once in a query, you should consider changing the data source name. The name of the data source is from X++ referred like a table variable. Tables, table maps and views can be used as data sources. As from X++ these are all used in the same way. Only when using temporary tables you will have to do some additional coding.

Note: If you are unsure what the outcome of your query will be, you should try creating your query in a report. The report will print the result of your query. See the chapter **Reports** on how to use the report query.

If a query is fetching data from more than one table you will like in a select statement have to define the joined order of the tables. Each level in the query tree defines a join of two tables.

Example 1: Creating an AOT query

Elements used from MORPHXIT_Queries project

- Query, MyQuery

This example will show how to create an AOT query joining two tables and setting default ranges for filtering. You should end up with a query as shown in **figure 46: MyQuery**.

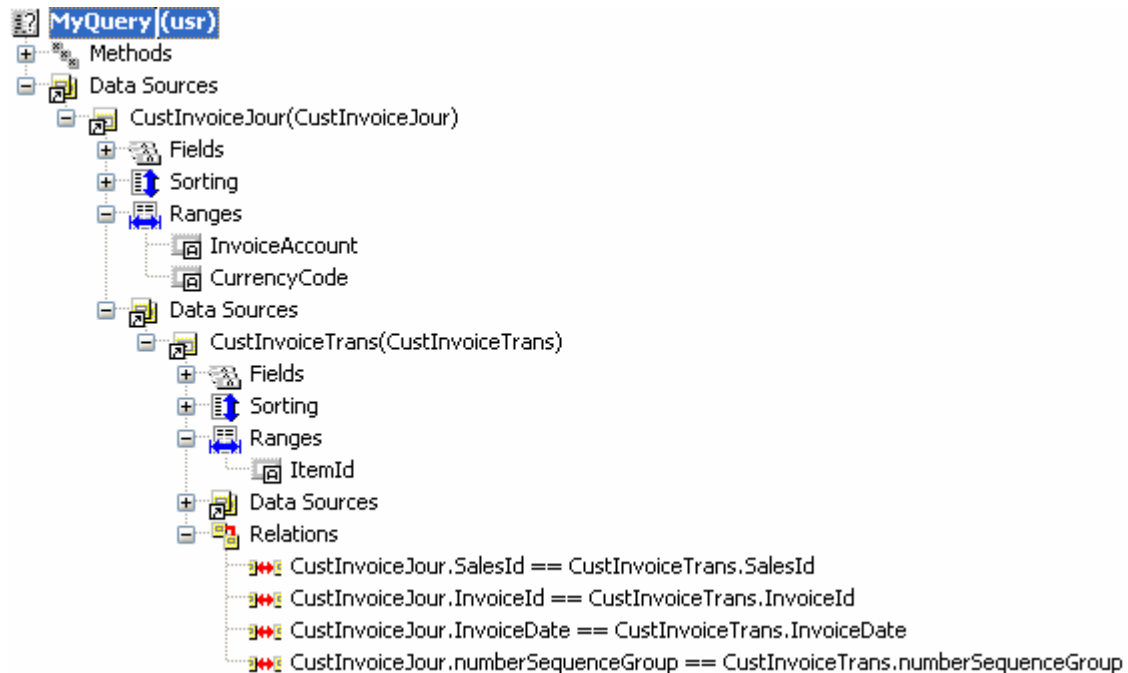


Figure 46: MyQuery

1. Go to the *Queries* node in the AOT, right-click and select *New Query*. Rename the query to "MyQuery" and unfold the new query.
2. Open another window of the AOT and locate the table *CustInvoiceJour* and drag the table to the query node *Data Sources*. The data source will get the suffix "_1". This is done prevent unique names. Remove the suffix.
3. Unfold the new data source node, and drag the table *CustInvoiceTrans* to the node *Data Sources/CustInvoiceJour/Data Sources* and remove the suffix.
4. Go to the property sheet for the data source *CustInvoiceTrans* and set the property **Relations** to "Yes". Unfold the new data source and check that relations fields are listed under *CustInvoiceTrans/Relations*.
5. Go to *Ranges* node located under the data source *CustInvoiceJour*, right click and select *New Range*. A new range will be added showing the first field from the data source table. Change the field by opening the property sheet for the new range and pick the field *InvoiceAccount* using the property **Field**.
6. Repeat step 5 by adding a range for the field *CustGroup*.
7. Go to the data source *CustInvoiceTrans* and add a range for the field *ItemId*.

The query created in this example is a join of the tables CustInvoiceTable and CustInvoiceTrans which contains the customer invoice journals. The tables are by default joined using an inner join. You can change the join mode of two data source by setting the **JoinMode** property at the lower level data source. The property **Relations** set at the lower level data source is used to define the relation fields used for the join. These are the relation fields specified in the data dictionary. You can specify relations fields manually, but it is preferable to use the data dictionary settings as this will keep maintain easier. If a relations is change in the data dictionary all queries using the data source will automatically be updated.

Now try creating a job to test the query:

```
static void Queries_TestMyQuery(Args _args)
{
    SysQueryRun      queryRun = new SysQueryRun(querystr(MyQuery));
    CustInvoiceJour   custInvoiceJour;
    CustInvoiceTrans  custInvoiceTrans;
;

    if (queryRun.prompt())
    {
        while (queryRun.next())
        {
            custInvoiceJour   = queryRun.get(tableNum(CustInvoiceJour));
            custInvoiceTrans  = queryRun.get(tableNum(CustInvoiceTrans));

            if (queryRun.changed(tableNum(CustInvoiceJour)))
            {
                info(strfmt("Account: %1", custInvoiceJour.invoiceAccount));
            }

            if (queryRun.changed(tableNum(CustInvoiceTrans)))
            {
                info(strfmt("Item: %1, Qty: %2, ",custInvoiceTrans.itemId, custInvoiceTrans.qty));
            }
        }
    }
}
```

The subclass SysQueryRun of the system class QueryRun is used to execute a query. The name of the query initialized is specified by the function querystr(). The name of the query could be entered just as a text, but querystr() will validate that the query is created in the AOT. QueryRun.prompt() will call the query dialog and if OK is pressed in the query dialog, the records matching the ranges specified in the query will be fetched. A table variable for each data source must be initialized to get the record values from the query. When a query loops the records you will need to validate which data source the present record is fetched from. This is done using queryRun.changed().

Note: By dragging an AOT query to the editor you can create the basics structure for looping the query.

When MyQuery is executed, the query dialog appears. The 3 fields specified under the *Ranges* node in the query are the default fields used for filtering. The application user can add or remove any of the default range. You can however set restrictions for a range by using the range property **Status**. By default ranges are created with status Open. You can enter a default value for a range using the property **Value**. If the Status property is set to Lock the application user will not be allowed changing the range. Locking a range can be useful if filtering data in a form and you would inform the application user about the mandatory filter. You can set the range property Status to Hidden if you do not want to show the mandatory filter to the application user.

The node *Sorting* is used to specify an index or fields used for sorting. Specifying fields for sorting will cause the data to be fetch order by or group by. In the MyQuery example no sorting was defined. Like in selects, sorting is not specified unless you must be certain about the sorting order. Sorting is typically used for queries in reports. A report query has additional properties for sorting making it possible to do summarization based on the sorting fields. For more information on using queries in reports, see the chapter **Reports**.

```
static void Queries_MyQueryAsSelect(Args _args)
{
    CustInvoiceJour      custInvoiceJour;
    CustInvoiceTrans     custInvoiceTrans;
    ;

    while select custInvoiceJour
    join custInvoiceTrans
        where custInvoiceJour.salesId == custInvoiceTrans.salesId
            && custInvoiceJour.invoiceId == custInvoiceTrans.invoiceId
            && custInvoiceJour.invoiceDate == custInvoiceTrans.invoiceDate
            && custInvoiceJour.numberSequenceGroup == custInvoiceTrans.numberSequenceGroup
    {
        // print fetch data
    }
}
```

If MyQuery were to be written as a select statement, the code would look like the above. Notice that all fields used in the where clause are and'ed. A query will always select by and'ing field ranges. Only a select can use both "and" and "or". However workaround exists for using the same options in a query. This will be explained in the following sections.

Aggregated Functions

By default a query is fetching all fields of a record like a select. If changing the property **Dynamic** on the *Fields* node to No you can change the query to only fetch values of the fields specified. It recommended only using this features if you have to optimize a query as you might not be aware of this setting when using the query from X++.

The property *Dynamic* also serves another purpose. When right-clicking the *Fields* node you can pick one of the aggregated functions AVG, SUM, COUNT, MIN or MAX. Choosing an aggregate function will set the property *Dynamic* to No as only the aggregated fields will contain a value. An aggregate function is used to do a calculating on the fetched data. The most common use of aggregate function is for summarizing a report.

Example 2: Aggregate function

Elements used from MorphxIt_Queries project

- Query, MyQuery_Aggregate

The example will show how to use the aggregate function SUM to summarize sold quantities per item using the customer invoice journal.

1. Create a new query and rename the query to "MyQuery_Aggregate".
 2. Drag the table CustInvoiceTrans as data source and remove the suffix.
 3. Unfold the data source CustInvoiceTrans, right-click the node *Fields* and select *New/SUM*. Open the property sheet for the sum field and choose the field Qty.
 4. Go to the *Sorting* node, right-click and select *New/Field*. Use the property sheet to select ItemId as the sorting field.
 5. The last step is to set sorting order. Go to the data source CustInvoiceTrans and set the property **OrderMode** to "Group by".
-

You can add any number of aggregated functions to a query. To simplify the example only a single aggregated function is used. When using aggregated functions you use the sorting fields to specify the grouping for the function. Together with the selected aggregated functions only sorting fields will have a value. The order mode should always be group by when using aggregate functions. It would not make sense using order by as this would cause all records to be printed, and in this example there would not be anything to sum.

```
static void Queries_TestMyQuery_Aggregate(Args _args)
{
    SysQueryRun      queryRun = new SysQueryRun(querystr(MyQuery_Aggregate));
    CustInvoiceTrans custInvoiceTrans;
;

    if (queryRun.prompt())
    {
        while (queryRun.next())
        {
            custInvoiceTrans = queryRun.get(tableNum(CustInvoiceTrans));
        }
    }
}
```


```

        if (queryRun.changed(tableNum(CustInvoiceTrans)))
        {
            info(strfmt("Item: %1, Qty: %2, ",custInvoiceTrans.itemId, custInvoiceTrans.qty));
        }
    }
}
}

```

This job can be used for testing the query. When executed you will see the sorting field added to the query dialog at the sorting tab page. As data are fetched group by you cannot change the sorting fields.

Advanced Ranges

A limitation in using a query is that ranges are and'ed. Say you wanted to find transaction in the customer invoice journal belonging to the customer group "40" or transactions which have the currency code "USD". If using a query you will end up fetching records where only both conditions are true. A work around exists for handling this. Try opening the form CustInvoiceJournal and press the query icon . Now enter the following in any range: ((custgroup == "20") || (currency == "USD")). It does not matter which query range is used as the code entered are not translated as a value for the chosen query field. When pressing the OK button in the query dialog, the records in the customer journal transaction form will be fetched using an OR expression.

Note: Wildcards used by the query dialog, such as '*' and '.' can also be used from X++ when building query ranges. See the online help for the query dialog for a list of all the wildcard options.

The parentheses make it possible to write a boolean expressions in a query range by using the data source fields. There are no validations when keying in such an expression. In fact it can be a bit tricky to write an error free expression. For this reason you would always add such expressions to your query by using X++. For testing such an expression it is far easier to try out the expression using the query dialog, and then paste the query range text to X++.

You might wonder, why bother when this can be done using a select statement. The case could be that you would have to modify an existing object like a class or a form using a query. If you were to replace an existing query with a select you might have to change a lot of code. Besides it is always preferable using queries for objects used by the user interface.

Methods on a Query

Methods on an AOT query are fairly never overridden. Changes are always made to the data source methods. Only forms have AOT nodes for overwriting the data source methods. See the chapter **Forms** on how to use the form data source methods.

X++ Query

As seen a query can be executed from X++, but a query can both be built and executed from X++ using the system classes prefixed with Query*. Typically simple queries or queries only to be use for a specific purpose are built using X++.

Note: some of the query system classes are inherited as application classes. You should use the application sub classes as they contain additional logic. A subclass of a system class is typically prefixed with Sys* like SysQuery.

When you get to know the single part of a query you will find it pretty easy constructing queries using X++.

```
static void Queries_SystemClasses(Args _args)
{
    SysQuery          query;
    SysQueryRun        queryRun;
    QueryBuildDataSource custInvoiceJourDS, custInvoiceTransDS;
;

    query = new Query();
    custInvoiceJourDS = query.addDataSource(tablename(CustInvoiceJour));
    custInvoiceJourDS.addRange(fieldnum(CustInvoiceJour, InvoiceAccount));
    custInvoiceJourDS.addRange(fieldnum(CustInvoiceJour, CurrencyCode));
    custInvoiceTransDS = custInvoiceJourDS.addDataSource(tablename(CustInvoiceTrans));
    custInvoiceTransDS.addRange(fieldnum(CustInvoiceTrans, ItemId));
    custInvoiceTransDS.relations(true);

    queryRun = new SysQueryRun(query);
    queryRun.prompt();
}
```

Here the query used in the MyQuery example is built using the query system classes. This is just straight forward as using the AOT. First a new query is declared and a data source is added to the query node. The class QueryBuildDataSource gives a handle to the data source CustInvoiceJour which is used to add the data source at the next level. Like in the AOT query relations are set to true, using the data dictionary relations. The default range fields are added using the addRange() method. To get an overview of the query, the prompt() method is called when executing the job. Having the query dialog shown makes it easy to get an overview of the query built.

If you are using a query for looping a temporary table, you must add the following line after initialized your QueryRun object. This is a common mistake leaving this line out. You might wonder that you will not get an output of your temporary table and end up using a select instead.

```
queryRun.setRecord(MyTable);
```

Adding ranges to a query is often done from X++. A typical case is to filter data fetched from a query using the values of some variables.

```

static void Queries_SystemClassesRanges(Args _args)
{
    SysQuery          query;
    SysQueryRun        queryRun;
    QueryBuildDataSource custInvoiceJourDS;
    QueryBuildRange    rangeInvoiceAccount, rangeInvoiceDate, rangeDimensionDepartment;
;

    query = new Query();
    custInvoiceJourDS = query.addDataSource(tablenum(CustInvoiceJour));
    rangeInvoiceAccount = custInvoiceJourDS.addRange(fieldnum(CustInvoiceJour, InvoiceAccount));
    rangeInvoiceAccount.value(queryValue("4000"));

    rangeInvoiceDate = custInvoiceJourDS.addRange(fieldnum(CustInvoiceJour, InvoiceDate));
    rangeInvoiceDate.value(queryRange(datetime(), systemdateget()));

    rangeDimensionDepartment = custInvoiceJourDS.addRange(fieldId2Ext(
                                                fieldnum(CustInvoiceJour, Dimension), 1));
    rangeDimensionDepartment.value(queryValue("Sales"));

    queryRun = new SysQueryRun(query);
    queryRun.prompt();
}

```

In the above example, 3 ranges are added to a query containing a single data source. The method `addRange()` returns an instance of the class `QueryBuildRange` which is used to set a value for each of the ranges.

Notice the global methods `queryValue()` and `queryRange()`. You should consider using these methods when adding a value to a query as they takes the variable anytype as parameter and formats the value to be presented in a range. `QueryRange()` is used to enter a from and to value, whereas `queryValue()` is used for a single value. If you need more values to be put in a range you can instead use the function `strfmt()` for formatting your expression. Just remember that the range value must be separated by commas.

Using an entry of an array field as a query range requires some additional coding.

Dimension is the most common array field. When adding Dimension as a range from the AOT query you can either pick the field Dimension which will add each entry of the array as a range. You can also pick each entry of the array from the field list. It is a bit different from X++ as your cannot just qualify the array entry by entering `Dimension[1]` for the first array entry of Dimension. Instead the global method `fieldId2Ext()` is used. The first parameter is the field id and the second parameter is the array entry number. Here the first dimension from `CustInvoiceJour` is added as a range.

A query range has a limited size. Though the max range size is high and you will have to concatenate several hundreds of ranges to reach the limit. You might think that this is plenty. If you criteria's for fetching data cannot be fulfilled using a query an often used practice is to use selects for build a string storing the values used by the query range. This could be a list of item ids to be fetched from the inventory table. However this is not considered good design as you will not know whether you are going to break the

limit. Adding an additional range for item ids when the first is filled will only result in the two ranges are AND'ed. Reaching the limit will result in an error from the database. By using the extended data type Range, you will now go that far. The extended data type range has a string length of 250 chars. You can change the length to 1000 chars which is the max length of a string. Before going that far, you should instead consider using a temporary table or a records sorted list.

You will be using query ranges a lot when constructing forms and reports, as ranges are typically used from X++ to filter the fetched data like when a form or a report must be filtered based on a calling form.

8.2 Queries in Forms and Reports

Both forms and reports make use of queries for fetching data. As Forms and reports have their own nodes for building a query they are not using the AOT queries. Compared to an AOT query forms and reports have additional properties. Using X++ you can specify another query or event built the entire query from X++ to be used for a form or report. The normal case would however be using the built in nodes in the AOT.

Select could do the same job as a query in a form or report, but using a select in a form or a report will be less user friendly.

In forms the options for filtering and sorting data are also used for printing auto reports based on the fields specified in the table field group AutoReport. Forms which have no query have not got this feature.

Reports without a query will have no filter options or options for automatically printing totals. If using a select on a report, filter options would have to be built manually from code.

8.3 Summary

Queries have a central role in MorphX. It is important to understand the concept of queries as you will be using queries a lot. Knowing how a query is constructed and how to use the query system classes will make it easier for you to modify objects using a query and thereby make your modifications more user friendly.

You should now have the basic knowledge of how to build and how to use queries. By reading about queries in the chapters **Forms** and **Reports** you should get the final picture on how MorphX uses queries.

9 Jobs

You might have noticed, that a lot of the examples throughout this book have been written using jobs. This is in fact one of the main purpose of jobs, writing test scripts. Jobs should be used for testing your modifications such as if you need to try out a complex code block, figuring out what a specific system function does, or where a single run is needed to update data. A job cannot be inherited or called from code in the same way as methods.

9.1 Creating jobs

A job is created as static to make it runnable. Without the static keyword the compiler would consider a job as any method. When creating a new job an optional parameter `Args` is automatically added to the parameter profile. The profile of a job is exactly the same as a class method being runnable. Jobs cannot return a value like a method. This is why a job is created with the void keyword.

```
static void Jobs_MyJob(Args _args)
{
;
    info("Test of job.");
}
```

Jobs can be called from a menu item making it possible to execute a job from the menu. This is useful if you have created a job fixing data, which must be executed in an Axapta installation without license code to MorphX. A job cannot be dragged to a menu item like forms, reports and runnable classes. Instead you must create the menu item manually. The reason for this might be that jobs are not intended to be put on menus or at least making you think an extra time before adding a job to a menu. You can drag a job to the node *Classes*. This will create a runnable class with the code from your job. This is really useful. All that is left is to create a menu item for the class. If a job is put on menu, you should ensure that data will not be corrupted by running the job twice.

```
static void Jobs_ExecutingJob(Args _args)
{
    Args          args;
;

    args = new Args();
    args.name(identifierStr(Jobs_MyJob));

    new menuFunction(menuItemActionStr(Jobs_MyJob), MenuItemType::Action).run(args);
}
```

By creating a menu item for a job, you will be able to use the menu item to run your job from code. The example shows how the job `Jobs_MyJob` is executed. Notice the name of the job in the `args.name()` is specified using the general function `identifierStr()` as no

specific function exists for jobs. Args could have been built passing a record or parameter to the called jobs. This could be useful if your job had to be called from a form and the job needed the cursor record in the form to determine whether to be executed the code in the job or not. Passing a cursor record to a called menu item is explained further in the chapter **Forms**.

9.2 Summary

You should by now have an idea about when to use jobs for your code and when to put your code in methods.

10 Menu Items and Menus

Objects such as forms, reports and runnable classes are made available for application users by adding the objects to menus. Often application users will not have permissions to the AOT, so it is a way of controlling which features are available for application users, i.e you would only add an object to menu when it is ready for use.

To have an object called from a menu, two steps must be taken.. First a menu item must be created, and second the menu item must be called from a menu. The reason for having two steps is that more than one menu item can be created for an object. Different properties can set for the menu items, having the object behave according to the calling menu item.

10.1 Menu Items

Menu items are grouped in 3 different nodes. The grouping is only a logical grouping, having a proper icon shown for the object in the menu. However the *display* node should be used for forms, the node *output* for reports and the *action* node for runnable classes. If you are using a class for calling a form or report, you should still be using the nodes *display* and *output* respectively. The menu item and hence the icon shown in the menu is supposed to be of the object type the user will see when activating the menu item.

The easiest way of creating a new menu item is by dragging your object to the desired menu item type. This will create a new menu item with the same name as your object. All that is left is to add a label and set user permission for the menu item. If a label is not specified for a menu item, the menu item will appear in the menu with the menu item name prefixed with a *.

Note: You should always test your modifications by calling your objects using menu items rather than calling an object directly from the AOT as you will then know how your modifications behave when an application user executes your modifications.

Menu items are one of the main areas to control user permission. The properties **ConfigurationKey** and **SecurityKey** are used to determine which user and user groups may use a menu item. If a user is not allowed to execute a menu item, the menu item will not be visible for the user. This powerful feature ensures only accessible menus are shown. The user will therefore see a reduced menu, only showing the parts of the application that are relevant to the user. The second main area handling user permissions is tables. For more information on configuration keys and security keys, see the chapter **Data Dictionary**.

Note: MorphX will automatically select a shortcut key for your menu items. To change the default selected shortcut key add a '&' in front the desired letter to be used as shortcut key: Transacti&ons. This will not work if changing the menu item label. You will have to change the property Text for the MenuItemButton control.

A menu item can be used to call an object from a menu, from a form calling another object or from code. Objects used for the user interface such as forms and reports should always be called using menu items. You can call forms and reports from code without using a menu item, but by using a menu item, user permissions will be automatically verified.

```
static void MenuItem_ExecuteObject(Args _args)
{
;
    new menuFunction(menuItemDisplayStr(CustTable), MenuItemType::Display).run();
}
```

Here the form CustTable is called using the display menu item CustTable. The form will only be loaded if the user has been granted permissions to the CustTable form. The method run() executing the menu item has the class Args as parameter. For example you can use Args to pass a parameter to the called object for filtering data. For more information on the class Args, see the chapter **Classes**.

Another way of passing a value to the called object is by using the menu item's parameter properties. The properties **EnumTypeParameter** and **EnumParameter** can be used to define an enum and an entry for the chosen enum. Setting an enum parameter for a menu item makes it easy to reuse an object like a form for several purposes, rather than creating similar forms. This technique is used in Axapta for calling the journals. Take a look at the display menu items prefixed with LedgerJournalTable*. You will notice these entire menu items are calling the same form LedgerJournalTable with different entries of the enum LedgerJournalType. In the form LedgerJournalTable, the enum is used to determine the behaviour of the form. This is a nice way to arrange your code, as it will make it easy if you need to create a new journal type. You will just have to add an new entry to the enum LedgerJournalType, create a new menu item using the entry and make a check in the form LedgerJournalTable for the new enum entry.

For an overview of all the menu item properties, see **Appendix Properties**.

10.2 Menus

To create a new menu, you can simply drag menu items to your menu. Separators and submenus can be created by right-clicking the menu node and choose new. Instead of creating submenus and adding menu items to the submenu, you should consider using a menu reference instead. A menu reference is a link to another menu. When adding a new object to a menu, adding a menu reference can be chosen. A window will pop up where an existing menu can be dragged to your menu as menu reference. Note that if you drag another menu directly from the AOT to your menu, the selected menu will be created as a submenu and not as a menu reference.

When adding a new menu item to a menu you will often be modifying an existing menu. To make menus easier to upgrade you should consider creating a new menu for

your menu items, and add the new menu as a menu reference to the existing menu. This has two purposes. First you will keep your modifications separated from the standard menus and second you can easily reuse your menu in another menu.

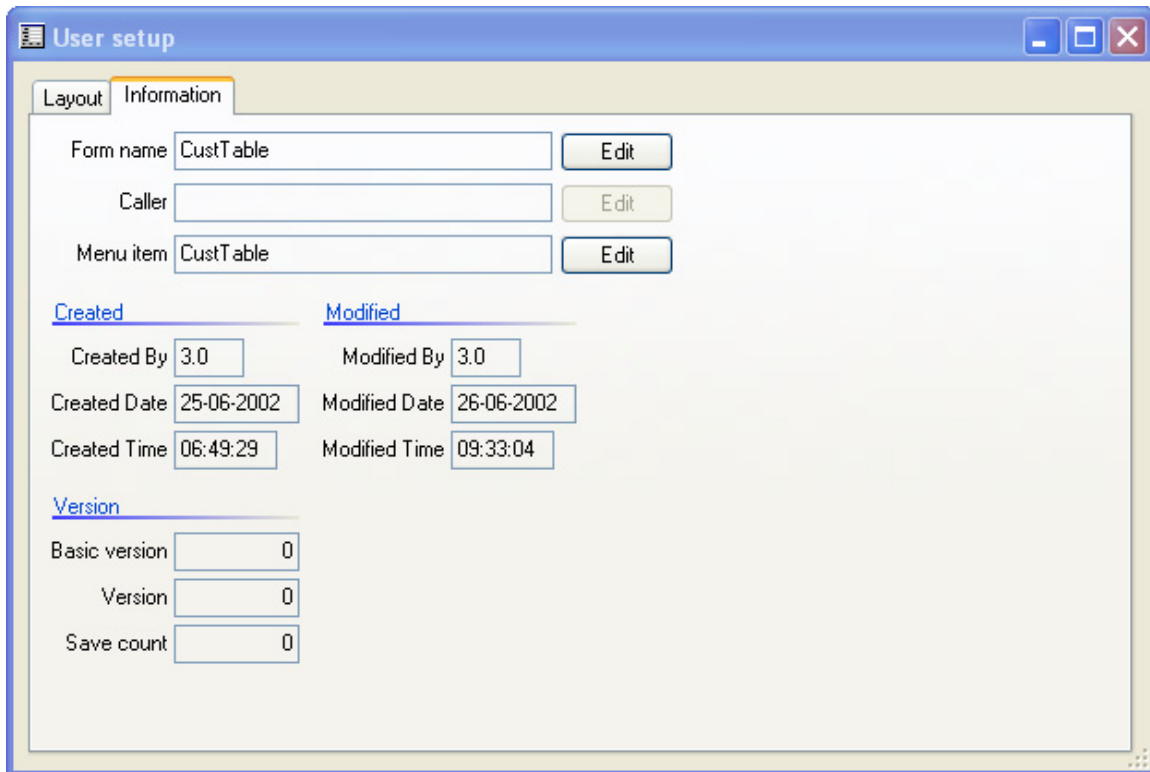
Configuration keys and security keys can be set for an entire menu using the menu properties. However this is not recommended. Instead you should define user permissions for menu items only, as you will assure that regardless where a menu item is called user permissions are verified. Second, setting user permission is easier to maintain only having the settings in one place.

Note: Best practice says that each menu called from the main menu should have a max of 5 menu items in top of the menu. The remaining menu items should be grouped in sub menus named: Journals, Inquiries, Reports, Periodic and Setup.

Locate AOT object from menu

One of the first tasks when modifying an object in the AOT is to locate the name of the object. You probably know where the object is called from the menu. Now you have two options to get the AOT name of the object typically a form, a report, or a class. As you are aware of the menu path to the object, you can find the corresponding menu in the AOT and drill down the menu in the AOT to find the menu item. Check the property sheet for the menu item, and the properties **Class** and **Object** will tell you the name of the AOT object.

The second option is to right-click an open form or dialog and choose *Setup*. This will open the form User setup as shown in **figure 47: User setup form**. Go to the tab page *Information*. The first three fields will show you the AOT name of the called form or class and the menu item calling the object. Click the *Edit* button to open the object in the AOT. Notice, this option is only available for forms and runnable classes and cannot be used for queries and reports.



The screenshot shows a window titled "User setup" with two tabs: "Layout" and "Information". The "Information" tab is active. It contains several input fields and buttons:

- Form name:** CustTable (with an "Edit" button)
- Caller:** (empty field with an "Edit" button)
- Menu item:** CustTable (with an "Edit" button)

Below these are two sections: "Created" and "Modified".

Created		Modified	
Created By	3.0	Modified By	3.0
Created Date	25-06-2002	Modified Date	26-06-2002
Created Time	06:49:29	Modified Time	09:33:04

At the bottom is a "Version" section:

Basic version	0
Version	0
Save count	0

Figure 47: User setup form

As you may conclude, none of these are fast ways to lookup an AOT object name. Sadly these are the only options if you only know the menu path. However as MorphX is open source you can modify the system using the system classes making it easier providing AOT information.

10.3 Summary

You should by now know how to call an object using a menu item, the importance of using menu items as menu items controls user permissions. When adding menu items to a menu, you should know how to arrange your menus making re-use and upgrading easier.

11 Resources

Resources are used to store any type of files. Instead of having bitmaps or any other kind of files used for modifications stored in the file system, the files can be added to the node *Resources*. The node *Resources* are not fully integrated such as objects like forms and reports. When addressing resource nodes some of the general system classes for traversing AOT nodes are used. In some cases you will have to temporarily export a resource node before using the stored file. The resources node was first introduced by version 3.0 of Axapta. This might be the reason that there are only a few methods available when using the resource nodes. Still, resources are useful, as you can skip using paths to the file system and at least you will have all files needed for your modifications in one place.

Resources stored under the *Resources* node in the AOT should not be confused with the resources stored in the kernel file. Bitmaps used for AOT nodes are all stored in the kernel file and referenced using a resource id. For an overview of the resources in the kernel file including the related resource id, see the form *Tutorial_Resources*.

11.1 Using Resources

A resource is added by right-clicking the *Resources* node and choosing Create from file. Browse the file system using the dialog which pops up, and select a file. When pressing the Open button, the chosen file will be stored in the AOT. Information about the added resource can be viewed by right-clicking and choose Open. The type of the resource will be shown in the preview window. If the resource is of the type bitmap, the bitmap will also be shown in the preview window.

When adding a bitmap to a form or a report, you have several options for the source of your bitmap such as the kernel resources, a bitmap stored in a table, specify a file path to a bitmap or referring to a bitmap under the resources node. Depending on your case different solutions will be optimal. If you need to ship a bitmap with your modifications or you want to make sure that an exact bitmap is used, you should consider storing your bitmap under the resources node.

Example 1: Loading bitmap

Objects used from MORPHXIT_Resources project

- Resource, MORPHXIT
- Form, Resources_LoadBitmap

In this example a bitmap stored under the resources node will be loaded and shown in a form. To run the example a resource of the type image with the name MORPHXIT must be created.

1. Create a new form, and rename the form “Resources_LoadBitmap”.
2. Go to the node *Design* and set the properties Width and Height to Column width and Column height.
3. Add a window control to the design. The window control will be used to show the resource node MORPHXIT. Set the property AutoDeclaration to Yes.
4. Create a new form method called showResource(). The method will be used to load resource. The method must look like the following:

```
public void showResource()
{
    ResourceNode    resourceNode;
    Container        imageContainer;
    Image            image;
;

    resourceNode = SysResource::getResourceNode(resourceStr(MORPHXIT));
    resourceNode.AOTload();

    imageContainer  = SysResource::getResourceNodeData(resourceNode);
    image           = new Image(imageContainer);

    imageCtrl.widthValue(image.width());
    imageCtrl.heightValue(image.height());
    imageCtrl.image(image);
}
```

5. The last step is to call the new method showResource() from run(). Overload run() and add the following:

```
public void run()
{
    super();

    element.showResource();
}
```

The class SysResource and the system class ResourceNode are both used for accessing resources. SysResource contains several useful static methods used both for loading and saving resources. Typically SysResource is used to fetch the resource node and return the found node to ResourceNode. If you have a closer look at the methods in the class SysResource, you will notice that the class methods are using the system class TreeNode. TreeNode is a base class which can be used to traverse any AOT nodes.

Note that after the ResourceNode is initialized in showResource() the method AOTLoad() must be called. If not called, the bitmap will not be shown in the form. The window control added to show the bitmap is initialized by using the system class Image.

Both the height and the width of the window control is set so the window control will fit the size of the bitmap.

In this example a bitmap of the type JPG was used. Not all types of bitmaps are supported, for instance you cannot use GIF. An error will occur if you are trying to use a type of bitmap not support like GIF. If you have created a module or an add-on solution, you could make use of resources to add files for your package, even sample data. Demo data or default data are often shipped with the package to make it easier for the user to getting started. You will just have to make an export of the data used and create resources nodes for the data files. This solution is also used in the standard package for setting up the Enterprise Portal and importing roles for the Enterprise Portal users.

Example 2: Default data

Objects used from MORPHXIT_Resources project

- Resource, CustGroup_Data
- Resource, CustGroup_Def
- Job, Resources_DefaultData

This example will show how to make user of data export files added as resources. The data will automatically be imported. You will be prompted if any data exists in the table the data is imported to. For simplicity data is only imported to the table CustGroup.

1. Start exporting the content of the table CustGroup using the export data menu item located in the main menu under Administration | Periodic | Data export/import | Export.
2. Create resource nodes for respectively the data file and the definition file. The resource nodes must be named CustGroup_Data and CustGroup_Def.
3. A job called Resources_DefaultData must be created with the following code:

```
static void Resources_DefaultData(Args _args)
{
    FilePath      tempPath;
    ResourceNode   resourceDefinitionFile;
    ResourceNode   resourceDataFile;
    SysDataImport  sysDataImport;
;
    tempPath = xinfo::directory(directoryType::Temp);

    resourceDefinitionFile = sysResource::getResourceNode(resourceStr(CustGroup_Definition));
    resourceDataFile       = sysResource::getResourceNode(resourceStr(CustGroup_Data));

    sysResource::exportResource(resourceDefinitionFile, tempPath);
    sysResource::exportResource(resourceDataFile, tempPath);

    sysDataImport = sysDataImport::newFilename(tempPath + resourceDefinitionFile.filename());
    sysDataImport.parmLoadAll(true);
}
```

```
sysDataImport.run();  
}
```

The import tool which can be called from the main menu is used for importing the resources data. Before the data files stored as resources can be imported, the files must first be exported. The data files are temporary exported to the temp path for the Axapta installation. This is done using the class SysResource for locating the resource nodes and exporting the files. The export files can now be imported calling the data import tool. When calling the import tool from the main menu you will be prompted for a filename and settings. Here the dialog is skipped as no further information is needed. That is all. This is really a user-friendly way of installing data for your modifications, rather than the application user has to struggle with locating an importing the data manually.

11.2 Summary

The use of resources is not widely used in MorphX. This might be the reason that resources are often forgotten and alternative solutions are made instead. This chapter should shed light on how resources can be used, and how by using resources, you can skip addressing files in the file system.

12 Appendix Properties

In this chapter you will find an overview of all properties accessible from the AOT. The properties are sorted alphabetically in each section. Objects with similar properties are grouped to have a more compact overview. Some sections have a third column called *Type*, which is used to list which object has the current listed property. Where *all* is stated in the column Type, the properties is available for all objects in the section.

12.1 Data Dictionary Properties

The properties for views metadata can be found in the section **Query Properties**.

Tables, Table Maps and Table Views

Property	Description
CacheLookup	Used to specify the caching algorithm used when a specific record is selected by a WHERE statement.
ChangedBy	The user who last modified the table.
ChangedDate	The date the table was last modified.
ChangedTime	The time the table was last modified.
ClusterIndex	Which index should be used as the cluster index. Only unique indexes can be specified.
ConfigurationKey	Used to specify a Configuration Key for the table.
CreatedBy	The user who created the table. If enabled the system field createdBy will be updated when a record is inserted.
CreatedDate	The date of record creation. If enabled the system field createdDate will be updated when a record is inserted.
CreatedTime	The time the record was created. If enabled the system field createdTime will be updated when a record is inserted.
CreatedTransactionId	If set, the transaction id belonging to the transaction which created the record will be stored.
CreateRecIdIndex	If enabled, an index called RecId containing the field recId will be created.
CreationDate	The date the table was created. If enabled the system field createdDate will be updated when a record is inserted.
FormRef	The name of a display menu Item which should be used when going to main table.
ID	The id of the table.

Label	The label of the table. This is the label the application user would normally identify the table with.
LockedBy	The user who has currently locked the table.
MaxAccessMode	Defines the access mode for the table.
ModifiedBy	If set, the user id belonging to the user who modified the record will be stored.
ModifiedDate	If set, the date of record modification will be stored.
ModifiedTime	If set, the time of record modification will be stored.
ModifiedTransactionId	If set, the transaction id belonging to the transaction which modified the record will be stored.
Name	The name of the table, table map or table view.
PrimaryIndex	Which index should be used as the primary index. Only unique indexes can be selected as a primary index.
SaveDataPerCompany	Should data be saved by company.
SecurityKey	Used to specify a Security Key for the table.
Systemtable	If enabled, the table will be considered being a system table.
TableContents	Defines which type of data the table contains.
TableGroup	Defines which table group the table should be a member of. The property is used by the kernel to determine the execution plan for selects using joins.
Temporary	Set for temporary tables.
TitleField1	The first field which is used in the title of forms.
TitleField2	The second field which is used in the title of forms.

Table Field, Map Field

Property	Type	Description
Adjustment	String	Specify the horizontal adjustment if an extended data type is not specified.
AliasFor	All	Which field should this field be alias for.
AllowEdit	All	If set, the field is editable.
AllowEditOnCreate	All	If set, the field is editable when creating the record.

ConfigurationKey	All	Used to specify a Configuration Key for the field.
EnumType	Enum	Specify an enum to be used for the table field.
ExtendedDataType	All	Specify an extended data type to be used for the table field.
FieldUpdate	Integer Real	Default setting is Absolute which will overwrite the current value when the field is changed. Relative will allow several application users to write the same record and have the values entered by the users summed.
GroupPrompt	All	The field label to be used when used in a field group.
HelpText	All	Help text that will be displayed in the status bar.
ID	All	The id of the field.
Label	All	Used to override the label of the ExtendDataType, if specified.
Mandatory	All	If set, a value must be filled in by the application user.
Name	All	The field name.
SaveContents	All	If enabled, the table field value will be stored in the database.
StringSize	String	Used to define the field length in characters if no extended data type is specified.
Type	All	Displays the base field type.
Visible	All	If enabled, the field is visible on forms.

View Fields

Property	Type	Description
Aggregation	All	Choose the aggregation to be performed on the field.
ConfigurationKey	All	Used to specify a Configuration Key for the field.

DataField	All	Select a field from the selected data source.
DataSource	All	Data source which is used in the control. Data will be retrieved from this data source.
EnumType	Enum	Displays the enum used for the table field.
ExtendedDataType	All	Displays the extended data type used for the table field.
GroupPrompt	All	The field label to be used when used in a field group.
HelpText	All	Help text which will be shown in the help text section of Axapta when the field is selected.
ID	All	The id of the view field.
Label	All	Used to override the label of the ExtendDataType.
Name	All	The field name.
StringSize	String	Displays the string size of the table field which is used.
Type	All	Displays the base field type.

Table Field Group, Map field group, View field group

Property	Description
Name	The group name.
Label	The label of the field group.

Table index

Property	Description
Name	The index name.
AllowDuplicates	If enabled, the index can contain two or more records with the same index key.
ConfigurationKey	Used to specify a Configuration Key for the index.

Enabled	If enabled, the index will be enabled.
ID	The id of the index.

Table Relation

Property	Description
Name	The Table Relation name.
Table	The table to be related.
Validate	Determines whether relations must be validated. If disabled the table relation will not be used in objects as forms and reports.

Table Relation Field

Property	Type	Description
Field	Normal Field fixed	One of the fields which will be a part of the field relation.
RelatedField	Normal Related field fixed	One of the fields which will be a part of the field relation.
Value	Field fixed Related field fixed	Specifies a fixed value for a relation. Normally used for setting the value of an enum entry.

Table DeleteAction

Property	Description
DeleteAction	Action to be performed when deleting a record.
Table	Table used for the delete action.

Map Mapping

Property	Description
MappingTable	Defines which table the map should be mapped to.

Map Field Mapping

Property	Description
----------	-------------

MapField	The field in the map.
MapFieldTo	Defines which field the map should be mapped to.

Extended Data Type

Property	Type	Description
Adjustment	String	Specify the horizontal adjustment, if the Extended Data Type does not extend another Extended Data Type.
Alignment	All	Horizontal alignment of the value.
AllowNegative	Integer Real	If disabled, only positive values can be entered.
ArrayLength	All	Displays how many arrays the Extended Data Types consists of.
AutoInsSeparator	Real	Used to have MorphX to set a decimal separator.
ButtonImage	All	Which image should be displayed on the right side of the control when lookup is possible.
ChangeCase	String	Used to set the data to lower case or upper case.
ConfigurationKey	All	Used to specify a Configuration Key for the Extended Data Type.
DateDay	Date	Specify how day is shown. Windows regional settings are used as default.
DateFormat	Date	Set the date format. Windows regional settings are used as default.
DateMonth	Date	Specify how month is shown. Windows regional settings are used as default.
DateSeparator	Date	Set the date separator. Windows regional settings are used as default.
DateYear	Date	Specify how year is shown. Windows regional settings are used as default.
DecimalSeparator	Real	Specify the decimal separator. Windows regional settings are used as default.
DisplaceNegative	Real	Adjust the position of negative values

		printed.
DisplayHeight	String	Sets the maximal number of lines to be shown at one time for the control.
DisplayLength	All	Sets the maximal number of characters to be shown at one time for the control.
EnumType	Enum	Specify an enum to be used.
Extends	All	If set, certain properties will be inherited from this Extended Data Type like Alignment and StringSize.
FormatMST	Real	Format the value using the settings from the standard company currency.
FormHelp	All	Form to be used when performing lookup in the control.
HelpText	All	Help text that will be displayed in the status bar.
Label	All	Set the label to be used.
Name	All	Name of the Extended Data Type.
NoOfDecimals	Real	Set the number of decimals to be shown.
RotateSign	Integer Real	Used to invert negative values.
ShowZero	Integer Real	Defines whether zero values must be shown.
SignDisplay	Integer Real	Set how to display negative values.
StringSize	String	Used to define the length in characters.
Style	Enum	Defines the graphical representation of the Extended Data Type.
ThousandSeparator	Real	Specify the thousand separators. Windows regional settings are used as default.
TimeFormat	Time	Set the time format. Windows regional settings are used as default.
TimeHours	Time	Specify whether to show hours. Windows regional settings are used as default.
TimeMinute	Time	Specify whether to show minutes. Windows regional settings are used as

		default.
TimeSeconds	Time	Specify whether to show seconds. Windows regional settings are used as default.
TimeSeparator	Time	Set the time separator. Windows regional settings are used as default.

Base Enum

Property	Description
ChangedBy	The user who last modified the Base Enum.
ChangedDate	The date the Base Enum was last modified.
ChangedTime	The time the Base Enum was last modified.
ConfigurationKey	Used to specify a Configuration Key for the Base Enum.
CreatedBy	The user who created the Base Enum.
CreatedTime	The time the Base Enum was created.
CreationDate	The date the table was created.
DisplayLength	Sets the maximal number of characters to be shown at one time for the control.
Help	Help text that will be displayed in the status bar.
ID	The id of the Base Enum.
Label	The label of the table. This is the label the user would normally identify the Base Enum with.
LockedBy	The user who has currently locked the table.
Name	The name of the Base Enum.
Style	Defines the graphical representation of the Base Enum.
UsedEnumValue	If disabled, Axapta will number the Base Enum entries and the property EnumValue on Base Enum entries will not be used.

Base Enum Entry

Property	Description
ConfigurationKey	Used to specify a Configuration Key for the Base Enum Item.

EnumValue	This is the Integer value that will be stored in the database.
Label	Name which will be shown.
Name	AOT name of the Base Enum Item.

License Codes

Property	Description
ChangedBy	The user who last modified the License Code.
ChangedDate	The date the License Code was last modified.
ChangedTime	The time the License Code was last modified.
CreatedBy	The user who created the License Code.
CreatedTime	The time the License Code was created.
CreationDate	The date the License Code was created.
Group	Which License Code Group should the License Code be a part of.
ID	The id of the License Code.
Label	The label for the License Code. This is the label the user would normally identify the License Code with.
LockedBy	The user who has currently locked the table.
Type	Most license codes are Boolean values. Only changed when counting the number of licenses such as users or COM users.

Configuration Key, Security Key

Property	Type	Description
ChangedBy	Both	The user who last modified the key.
ChangedDate	Both	The date the key was last modified.
ChangedTime	Both	The time the key was last modified.
ConfigurationKey	Security Key	Used to specify a Configuration Key for the key.
CreatedBy	Both	The user who created the key.
CreatedTime	Both	The time the key was created.

CreationDate	Both	The date the key was created.
ID	Both	The id of the key.
Label	Both	The label of the key. This is the label the user would normally identify the key with.
LicenseCode	Configuration Key	License code used to activate this key.
LockedBy	Both	The user who has currently locked the key.
Name	Both	The key name.
ParentKey	Both	Parent key for this key. If the parent key is disabled, it will influence this key as well.

12.2 Form properties

Form data source

Property	Description
AllowCheck	If set Configuration keys and Security keys will be validated at runtime.
AllowCreate	Enables inserting new record for the data source.
AllowDelete	Enables deletion of table records for the data source.
AllowEdit	Enables editing of table records for the data source.
AutoNotify	Should be disabled if the form query is not used. Used by the form query, but seems as have no effect if only disabling this property.
AutoQuery	If disabled, the application user will not be able to use the query dialog, filter and search options for the data source.
AutoSearch	Should the records be fetched automatically at startup.
Company	If specified, records will be fetched from this company.
CounterField	Used to define a counter for the records inserted using the data source. CounterField can be used if records must be sorted as inserted by the application users. A field of the type real must be created in the table used by the data source. This field must be selected for the property CounterField. MorphX will automatically set the counter value when a record is inserted. The form <i>SalesTable</i> makes use of the CounterField properties when inserting sales order lines.

DelayActive	If set, code execution and linking will be delayed when scrolling through records. This will improve performance.
Index	Index used for sorting and fetching records.
InsertAtEnd	If set, new records will be inserted at the end.
InsertIfEmpty	A new record will be inserted if the data source query does not find any records.
JoinSource	The joined form data source.
LinkType	This property is used in combination with JoinSource. LinkType defines the join mode used when joining two data source.
Name	The name of the form data source.
OnlyFetchActive	This will instruct the query of the form data source only to fetch the values of the fields used in the form.
StartPosition	Should the form data source show the first or the last record.
Table	The table used in the form data source.

Form Data Source Fields

Property	Description
AllowAdd	Allows the user to add this field in the user setup.
AllowEdit	Enables editing of the value in the control.
Enabled	Should the control be enabled.
Mandatory	If set, a value must be specified by the user in the field.
Skip	Should the control be skipped when tab is pressed.
Visible	Used to hide the control. If the follow controls are auto positioned, the controls will be adjusted.

Form Design Group Controls

This section covers properties for the controls ButtonGroup, Group, Tab and TabPage.

Property		Description
AlignChild	All	Should this control be included in the adjustment of the group control it is

		contained in.
AlignChildren	All	If set, controls which are contained in this control will be aligned according to each other.
AlignControl	All	This setting will adjust the controls according to the longest label.
AllowEdit	All	Enables editing of the value in the control.
AllowUserSetup	All	Enables user settings for this control element.
ArrangeMethod	All	Set the orientation for the arranged controls.
ArrangeWhen	All	Specify when the controls in the design must be arranged.
AutoDataGroup	Group	If enabled, the control can only contain fields from the group specified in the property DataGroup.
AutoDeclaration	All	If set to Yes, the form design node can be referred from X++ by using the section name.
BackgroundColor	All	RGB value or name of Windows color scheme item.
BackStyle	All	Set the background for the control to transparent. Used if the background color of bitmaps should not be shown, or to set the color of the background for the control data to the color set with the property BackGroundColor.
Bold	ButtonGroup Group	Set the bold level for control data.
BottomMargin	All	Sets the margin below the control.
Caption	ButtonGroup Group TabPage	Caption for the control.
ColorScheme	All	Specify whether to use RGB colors, or Windows color scheme.
Columns	All	Number of columns in the control. The contained controls will be

		arranged in this number of columns.
Columnspace	All	Set the space between columns.
ConfigurationKey	All	Used to specify a Configuration Key for the control.
DataGroup	Group	Field group name.
DataSource	All	Data source which is used in the control. Data will be retrieved from this data source.
DragDrop	All	Enables drag and drop in the control.
Enabled	All	Should the control be enabled.
Font	ButtonGroup Group	The font to be used for the design. If not specified the default font is used.
FontSize	ButtonGroup Group	The font size to be used for the design. If not specified the default font size is used.
FrameOptionButton	Group	Determines whether the frame should contain a button.
FramePosition	ButtonGroup Group	Sets the placement of the frame.
FrameType	ButtonGroup Group	Which type of frame must surround the control.
Height	All	Set a fixed height for the control.
HelpText	All	Help text that will be displayed in the status bar. Will override help text specified for a field or an extended data type.
HideIfEmpty	All	Hides the control if it is empty.
Italic	ButtonGroup Group	Set the label to italic.
LabelBold	Group	Set the label to bold.
LabelFont	Group	Set the font for the label. If not specified the default font will be used.
LabelFontSize	Group	Set the font size for the label. If not specified the default font size will be used.

LabelItalic	Group	Set the label to Italic.
LabelUnderline	Group	Underline the label text. The property LabelLineBelow is used to set a line below in the full width of the label.
Left	All	Set the control to a fixed position calculated from left. If controls are horizontal aligned and one control is set to a fixed position, all controls must be fixed.
LeftMargin	All	Set a left margin for the control.
Name	All	Name of the control.
NeededAccessLevel	All	Required access level to activate this control.
OptionValue	Group	Value to be used with the property FrameOptionButton. Used to set the default value if FrameOptionButton is set to Check or Radio.
RightMargin	All	Set a right margin for the control.
SecurityKey	All	Used to specify a Security Key for the control.
SelectControl	Tab	Should the first control be activated when changing tab page.
ShowTabs	Tab	If disabled, the tabpages will be hidden.
SizeHeight	ButtonGroup	Should all the buttons in the buttongroup have the same height.
SizeWidth	ButtonGroup	Should all the buttons in the buttongroup have the same width.
Skip	All	Should the control be skipped when tab is pressed.
Tab	Tab	Active tabpage when the form is opened.
TabAppearance	Tab TabPage	Defines how the tabpages are shown.
TabAutoChange	Tab TabPage	Does not work. Should allow using the tab key to go to the following tabpage.
TabLayout	Tab	How the tabpages should be

		arranged. The option Tunnel cannot be used for forms as this is a features used by the web framework.
TabPlacement	Tab	Defines where the tabpages are placed.
Top	All	Set a fixed position for the control calculated from the top of the previous control.
TopMargin	All	Set the margin above the control.
Underline	ButtonGroup Group	Set the label to be underlined.
VerticalSpacing	All	Space above and under the control.
Visible	All	Used to hide the control. If the follow controls are auto positioned, the controls will be adjusted.
Width	All	Set a fixed width for the control. If set to default, the extended data type will set the width.

Form design

Property	Description
AlignChild	Should this control be included in the adjustment of the group control it is contained in.
AlignChildren	If set, controls which are contained in this control will be aligned according to each other.
AllowDocking	Can the window be docked.
AllowUserSetup	Enables user settings for this control element.
AlwaysOnTop	Should the form be displayed as the front most window.
ArrangeMethod	Set the orientation for the arranged controls.
ArrangeWhen	Specify when the controls in the design must be arranged.
BackgroundColor	RGB value or name of Windows color scheme item.
BottomMargin	Sets the margin below the control.
Caption	Caption of the form. This will appear in the title bar of the form.
ColorScheme	Specify whether to use RGB colors, or Windows color scheme.

Columns	Number of columns in the control. The contained controls will be arranged in this numbers of columns.
Columnspace	Set the space between columns.
DataSource	Data source which is used for the control. Data will be retrieved from this data source.
Font	The font to be used for the design. If not specified the default font is used.
Frame	This property defines the appearance of the frame around the form.
Height	Set a fixed height for the form design.
HideIfEmpty	Hides the design and only shows the title bar, if empty.
HideToolbar	Hides the toolbar containing buttons for navigating and viewing documents for the record.
Imagemode	Controls how the background image must be shown. See property ImageName.
ImageName	Name of the image to be used as background in the form.
ImageResource	Name of the resource to be used as background in the form.
LabelFont	Set the font for the label of the control elements. If not specified the default font will be used.
Left	Set a fixed horizontal position for the form.
LeftMargin	Set the left margin for the design.
Mode	Seems to have no effect. This is a property used by the predecessor to Axapta for setting write access on forms.
NeededAccessLevel	Seems to have no effect. NeededAccessLevel is normally defined using the menu items, as the help text for this property also state.
RightMargin	Set the right margin for the design.
SaveSize	Saves the size of the form design and uses this the next time the form is displayed.
SetCompany	Would you like to set the company when the form is activated.
TitleDatasource	If enabled, the title fields of the data source will be shown in the caption of the form.
Top	Set a fixed vertical position for the form.
TopMargin	Set the top margin for the design.

Visible	Used to hide the entire design.
Width	Set a fixed width for the form.
WindowResize	Enables or disables window resizing.
WindowType	Used to change the form to a popup form. A popup form cannot be resized.

Type controls

This section is an overview of form controls for the base types and other controls used for display data such as the grid control and button controls.

Property	Type	Description
ActiveBackColor	Grid	RGB value or name of Windows color scheme item.
ActiveForeColor	Grid	RGB value or name of Windows color scheme item.
AlignChild	Design	Should this control be included in the adjustment of the group control it is contained in.
AlignChildren	Design	If set, controls which are contained in this control will be aligned according to each other.
AlignControl	All	This setting will adjust the controls according to the longest label.
Alignment	DateEdit IntEdit RealEdit StaticText StringEdit TimeEdit	Align the control data. Can be used to left align control data, when controls are positioned vertical.
AllowEdit	All	Allows editing the value in the control.
AllowNegative	IntEdit RealEdit	The property is used to prevent negative values to be entered by the user.
AnimateFile	Animate	The name of the .avi file that should be played in the control.
AppendNew	ComboBox	If set to Yes, the user can manually add new elements.
ArrayIndex	ComboBox DateEdit	If the selected field or method is an array a single element of the array can be

	IntEdit Listbox RadioButton RealEdit StringEdit TimeEdit	specified to be used only.
AutoArrange	ListView	Should icons be arranged automatically.
AutoDataGroup	Grid	If enabled, the control can only contain fields from the group specified in the property DataGroup.
AutoDeclaration	All	If set to Yes the properties for the control can be referenced from X++ by using the control name.
AutoInsSeparator	RealEdit	Seems to have no effect. Should allow disabling auto inserting decimals.
AutoPlay	Animate	Starts playback of the video file automatically.
BackgroundColor	Button CheckBox ComboBox CommandButton DateEdit Grid IntEdit Listbox ListView MenuButton MenuItemButton RadioButton RealEdit StaticText StringEdit Table TimeEdit Tree Window	RGB value or name of Windows color scheme item.
BackStyle	Button CheckBox ComboBox CommandButton DateEdit IntEdit Listbox ListView MenuButton MenuItemButton RadioButton RealEdit	Set the background for the control to transparent. Used if the background color of bitmaps should not be shown, or to set the color of the background for the control data to the color set with the property BackGroundColor.

	StaticText StringEdit TimeEdit Tree Window	
Bold	Button ComboBox CommandButton DateEdit IntEdit Listbox ListView MenuButton MenuItemButton RadioButton RealEdit StaticText StringEdit TimeEdit Tree	Set the bold level for control data.
Border	Animate Button ComboBox CommandButton DateEdit IntEdit Listbox ListView MenuButton MenuItemButton RealEdit StringEdit TimeEdit Tree	Typography of the frame belonging to the control.
BottomMargin	Grid MenuButton RadioButton Table	Set the margin below the controls data.
ButtonDisplay	Button CommandButton MenuButton MenuItemButton	Determines whether text, image or both should be displayed and also the location of it.
CanScroll	ListView Tree	Enables scrolling.
Caption	ActiveX HTML RadioButton	Caption for the control.
Center	Animate	If set to Yes, the video clip will be

		centered in the control.
ChangeCase	StringEdit	Used to set the control data to lower case or upper case.
CheckBox	ListView Tree	If enabled, a checkbox will be shown for each row in the control's data.
ClassName	ActiveX HTML	The name or GUID of the class or object to be used.
ColorScheme	Button CheckBox ComboBox CommandButton DateEdit Grid IntEdit Listbox ListView MenuButton MenuItemButton RadioButton RealEdit StaticText StringEdit Table TimeEdit Tree Window	Specify whether to use RGB colors, or Windows color scheme.
Column	Table	Sets the active column.
ColumnHeader	ListView	Determines whether a header will be shown for the columns.
ColumnHeaderButton	ListView	If enabled, the column headers will function as a button. Only when Viewtype is set to Report.
ColumnImages	ListView	If enabled, each object in a column can contain an image.
Columns	RadioButton Table	Number of columns in the control. The contained controls will be arranged in this numbers of columns.
ComboType	ComboBox	The type of combobox.
Command	CommandButton	Command to be used when clicking on the button.
ConfigurationKey	All	Used to specify a Configuration Key for the control.

Custom	ActiveX Html	This property must be used if a custom ActiveX property editor should be used.
DataField	CheckBox ComboBox DateEdit IntEdit Listbox RadioButton RealEdit StaticText StringEdit TimeEdit Window	Select a field from the selected data source. Instead of selecting a field, a display method can be specified in the property DataMethod.
DataGroup	Grid	Field group name.
DataMethod	CheckBox ComboBox DateEdit IntEdit Listbox RadioButton RealEdit StaticText StringEdit TimeEdit Window	Select a display or edit method to be used. If the method is from a table, the data source must be specified in the property Table and the method must exist either on the data source connected to the table or the table itself.
DataSource	CheckBox ComboBox DateEdit Grid IntEdit Listbox MenuItemButton RadioButton RealEdit StaticText StringEdit TimeEdit Window	Data source which is used in the control. Data will be retrieved from this data source.
DateDay	DateEdit	Specify how day is shown. Windows regional settings are used as default.
DateFormat	DateEdit	Set the date format. Windows regional settings are used as default.
DateMonth	DateEdit	Specify how month is shown. Windows regional settings are used as default.
DateSeparator	DateEdit	Set the date separator. Windows regional settings are used as default.

DateValue	DateEdit	If specified, this date will be used as default in the value of the control.
DateYear	DateEdit	Specify how year is shown. Windows regional settings are used as default.
DecimalSeparator	RealEdit	Specify the decimal separator. Windows regional settings are used as default.
DefaultButton	Button CommandButton MenuButton MenuItemButton	If enabled, this is the standard button.
Direction	Progress	Sets either horizontal or vertical direction.
DisabledImage	Button CommandButton MenuButton MenuItemButton	Will show the bitmap at the specified path if the button is disabled. The property ButtonText must be set to show images.
DisabledResource	Button CommandButton MenuButton MenuItemButton	Will show the specified resource bitmap if the button is disabled. The property ButtonText must be set to show images.
DisplaceNegative	IntEdit RealEdit	Adjust the position of negative values printed.
DisplayHeight	DateEdit IntEdit RealEdit StaticText StringEdit TimeEdit	Sets the maximal number of lines to be shown at one time for the control.
DisplayLength	ComboBox DateEdit IntEdit Listbox RadioButton RealEdit StaticText StringEdit TimeEdit	Sets the maximal number of characters to be shown at one time for the control.
DragDrop	All	Enables drag and drop in the control.
EditLabels	ListView Tree	Enables user editing of labels in the control.
Enabled	All	Should the control be enabled.
EnumType	ComboBox	Specity an enum to be used for the control

	Listbox RadioButton	if a field or a method is not specified.
ExtendedDataType	ComboBox DateEdit IntEdit Listbox RadioButton RealEdit StringEdit TimeEdit	Specify an extended data type for the control if a field or a method is not specified
Font	Button ComboBox CommandButton DateEdit IntEdit Listbox ListView MenuButton MenuItemButton RadioButton RealEdit StaticText StringEdit TimeEdit Tree	The font to be used for the control data. If not specified the default font is used.
FontSize	Button ComboBox CommandButton DateEdit IntEdit Listbox ListView MenuButton MenuItemButton RadioButton RealEdit StaticText StringEdit TimeEdit Tree	The font size to be used for the control data. If not specified the default font size is used.
ForegroundColor	Button CheckBox ComboBox CommandButton DateEdit IntEdit Listbox ListView MenuButton MenuItemButton RadioButton RealEdit	RGB value or name of Windows color scheme item.

	StaticText StringEdit TimeEdit Tree Window	
FormatMST	RealEdit	Format the value using the settings for the standard company currency.
FramePosition	RadioButton	Sets the position of the frame.
FrameType	RadioButton	Which frame type must surround the control.
GridLines	Grid ListView Table	Show lines in the grid. Only valid by control type ListView when the property ViewType is set to Report.
HasButtons	Tree	If enabled, + or – is shown if the control node can be expanded.
HasLines	Tree	This will draw lines for each row in the data of the control.
Headerdragdrop	ListView	Enables drag and drop for the header in the control.
Height	All	Set a fixed height for the control.
HelpText	All	Help text that will be displayed in the status bar. This value will override the label entered at the table field or extended data type.
HideFirstEntry	ComboBox Listbox RadioButton	Hides the first data entry.
HighlightActive	Grid	If enabled, the selected line will be marked with a color.
Imagemode	Window	Controls how the background image must be shown. See property ImageName.
ImageName	Window	Name of the image to be used as background in the control.
ImageResource	Window	Specify an image resource id to be shown.
Italic	ComboBox CommandButton DateEdit IntEdit Listbox	Set the font to italic for the control data.

	ListView MenuButton MenuItemButton RadioButton RealEdit StaticText StringEdit TimeEdit Tree	
Item	ComboBox Listbox RadioButton	Seems to have no effect. The property Selection is used to set the default entry. Is disabled if an enum is specified for the control.
ItemAlign	ListView	Seems to have no effect. Should be aligning the items in a list control to the top or to the left.
Items	ComboBox Listbox RadioButton	Is disabled if an enum is specified for the control. Can be used for manually set the number of entries for a radio button control.
Label	CheckBox ComboBox DateEdit IntEdit Listbox RealEdit StringEdit TimeEdit Window	Used to override the default label from the field or extended data type. The label will not be displayed, if the property ShowLabel is set to false.
LabelAlignment	CheckBox ComboBox DateEdit IntEdit Listbox RealEdit StringEdit TimeEdit Window	Align the label of the control.
LabelBold	CheckBox ComboBox DateEdit IntEdit Listbox RealEdit StringEdit TimeEdit Window	Set the label to bold.
LabelFont	CheckBox	Set the font for the label. If not specified

	ComboBox DateEdit IntEdit Listbox RealEdit StringEdit TimeEdit Window	the default font will be used.
LabelFontSize	CheckBox ComboBox DateEdit IntEdit Listbox RealEdit StringEdit TimeEdit Window	Set the font size for the label. If not specified the default font size will be used.
LabelForegroundColor	CheckBox ComboBox DateEdit IntEdit Listbox RealEdit StringEdit TimeEdit Window	RGB value or name of Windows color scheme item to be used for the label of the control.
LabelHeight	CheckBox ComboBox DateEdit IntEdit Listbox RealEdit StringEdit TimeEdit Window	This property is not working. Should be used to set the height of a label.
LabelItalic	CheckBox ComboBox DateEdit IntEdit Listbox RealEdit StringEdit TimeEdit Window	Set the label to Italic.
LabelPosition	CheckBox ComboBox DateEdit IntEdit Listbox RealEdit	Position the label above or to the left.

	StringEdit TimeEdit Window	
LabelUnderline	CheckBox ComboBox DateEdit IntEdit Listbox RealEdit StringEdit TimeEdit Window	Underline the label text.
LabelWidth	CheckBox ComboBox DateEdit IntEdit Listbox RealEdit StringEdit TimeEdit Window	Used when the property LabelPostion is set to Left. Set a fixed width for the label.
Left	All	Set the control to a fixed position calculated from left.
LeftMargin	Grid MenuButton RadioButton Table	Set a left margin for the control.
LimitText	DateEdit IntEdit RealEdit StringEdit TimeEdit	The maximal number of characters that the user can enter in the control.
LinesAtRoot	Tree	This will draw lines for the root row in the data of the control.
LookupButton	DateEdit IntEdit RealEdit StringEdit TimeEdit	When should the control have a lookup button.
Loops	Animate	Number of times to play the movie clip. Setting to zero will cause the movie clip to be repeated.
Mandatory	DateEdit IntEdit RealEdit StringEdit	If set, the field must be filled out.

	TimeEdit	
MenuItemName	MenuItemButton	Name of the menu item. Only menu items of the type defined in the property MenuItemType can be selected.
MenuItemType	MenuItemButton	Type of the menu item.
MultiLine	StringEdit	Should the value of the data control be able to contain several lines.
MultiSelect	Button CommandButton Grid MenuButton MenuItemButton	In a grid, it determines whether several rows can be selected at one time. In other types of controls, the control will be disabled if several rows are selected and this property is not enabled.
Name	All	The name of the section. This is the name used from X++ to refer to the section, when the property AutoDeclaration is set.
NeededAccessLevel	Button CommandButton MenuButton MenuItemButton	Required access level to activate this control.
NoOfDecimals	RealEdit	Set the number of decimals to be shown.
NormalImage	Button CommandButton MenuButton MenuItemButton	Will show the specified resource bitmap if the button is enabled. The property ButtonText must be set to show images.
NormalResource	Button CommandButton MenuButton MenuItemButton	Will show the specified image if the button is enabled. The property ButtonText must be set to show images.
OneClickActivate	ListView	Activate with just one click.
PasswordStyle	StringEdit	If set, the value of the control will be shown as *.
Pos	Progress	Start position for progress bar.
ProgressType	Progress	Seems as having no effect.
RangeHi	Progress	Maximum value for the progress bar.
RangeLo	Progress	Minimum value for the progress bar.
RealValue	RealEdit	Sets the default value for the control.
ReplaceOnLookup	DateEdit	Should the value entered in the control be

	IntEdit RealEdit StringEdit TimeEdit	replaced when a new value is selected on lookup.
RightMargin	Grid MenuButton RadioButton Table	Set a right margin for the control.
RotateSign	IntEdit RealEdit	Used to invert negative values.
Row	Table	Active row.
Rows	Table	Number of rows in the control.
RowSelect	ListView Tree	Can rows be selected.
SaveRecord	Button CommandButton MenuButton MenuItemButton	Should the record be saved by the system when activating this control.
SearchMode	DateEdit IntEdit RealEdit StringEdit TimeEdit	Sets the search mode on how to find records when typing.
SecurityKey	All	Used to specify a Security Key for the control.
Selection	ComboBox Listbox RadioButton	Sets the initially selected item.
ShowColLabels	Grid Table	Should column labels be shown.
ShowLabel	CheckBox ComboBox DateEdit IntEdit Listbox RealEdit StringEdit TimeEdit Window	If set to No, the label will not be displayed.
ShowRowLabels	Grid Table	Should column labels be shown.
ShowSelAlways	ListView	Should the object selection be maintained

	Tree	when focus is changed.
ShowShortCut	Button CommandButton MenuButton MenuItemButton	If enabled, a shortcut will be reserved and shown in the label of the control.
ShowZero	IntEdit RealEdit	Defines whether zero values must be shown.
SignDisplay	IntEdit RealEdit	Set how to display negative values.
SingleSelection	ListView Tree	Allow more than one object to be selected at a time.
Skip	All	Should the control be skipped when the tab key is pressed.
Sort	ListView	Defines how the sorting of elements is performed.
Step	Progress	Number of steps in each iteration.
Text	Button ComboBox CommandButton Listbox MenuButton MenuItemButton RadioButton StaticText StringEdit	Enter the text to be shown.
ThousandSeparator	RealEdit	Specify the thousand separator. Windows regional settings are used as default.
TimeFormat	TimeEdit	Set the time format. Windows regional settings are used as default.
TimeHours	TimeEdit	Specify whether to show hours. Windows regional settings are used as default.
TimeMinute	TimeEdit	Specify whether to show minutes. Windows regional settings are used as default.
TimeSeconds	TimeEdit	Specify whether to show seconds. Windows regional settings are used as default.
TimeSeparator	TimeEdit	Set the time separator. Windows regional settings are used as default.

Top	All	Set a fixed position for the control calculated from the top of the previous control.
TopMargin	Grid MenuButton RadioButton Table	Set the margin above the control.
TrackSelect	ListView Tree	Should the object be selected when the cursor is moved over the control.
Transparent	Animate	Should a transparent background be used.
TwoClickActivate	ListView	Require activation with double click.
Underline	Button ComboBox CommandButton DateEdit IntEdit Listbox ListView MenuButton MenuItemButton RadioButton RealEdit StaticText StringEdit TimeEdit Tree	Underline the control data.
Value	CheckBox IntEdit MenuItemButton TimeEdit	Specify the initial value.
VerticalSpacing	All	Space above and under the control.
ViewType	ListView	Specify how objects are represented visually.
Visible	All	Used to hide the control. If the following controls are auto positioned, the controls will be adjusted.
VisibleCols	Grid	How many columns should be visible.
VisibleRows	Grid	How many rows should be visible.
Width	All	Set a fixed width for the control. If set to default, the extended data type will set the width.

12.3 Report Properties

The report query properties can be found in the section **Query Properties**.

Report

Property	Description
Name	AOT name of the report.
AllowCheck	If set Configuration keys and Security keys will be validated at runtime.
Autojoin	Joins the reports query with the caller. If the report is called from a form, the report will be joined with the called record.
Interactive	Specify whether the dialog is shown to the user at runtime.

Report design

Property	Description
Name	Name of the design. Used to identify the design, if the report have more than one design.
AutoDeclaration	If set to Yes, the report design node referred from X++ by using the section name.
Caption	Caption for the report. If the standard report template InternalList is used, caption will be used to print the name of the report in the header.
Description	Will be printed in the top bar of the print to screen window. If not specified, the Caption text is printed instead.
JobType	Can be used to identify the design. JobType is not printed.
EmptyReportPrompt	Specify a text to be printed in the Infolog, if the report is empty. If not used, the default text is printed.
ArrangeWhen	Specify when the controls in the design must be arranged.
ColorScheme	Specify whether to use RGB colors, or Windows color scheme.
ForegroundColor	RGB value or name of Windows color scheme item.
ResolutionX	Seems as having no effect.
ResolutionY	Seems as having no effect.
Ruler	Used to set the measure units for the visual designer.

ReportTemplate	Choose a report template to be used for the design.
TopMargin	Set the top margin for the design.
BottomMargin	Set the bottom margin for the design.
LeftMargin	Set the left margin for the design.
RightMargin	Set the right margin for the design.
Language	Set a fixed language for the design. If not set the users default language will be used.
Font	The font to be used for the design. If not specified the default font is used.
FontSize	The font size to be used for the design. If not specified the default font size is used.
Italic	Set the design to italic.
Underline	Underline text in the design.
Bold	Set the design to bold.
PrintFormName	The application form used for the printer dialog. If the report is called using the report runbase framework, this form is not used.
HideBorder	Skip printing labels and lines around controls.
Orientation	Fix orientation to portrait or landscape. A design will default be printed landscape, if the there are too many controls to fit portrait without scaling.
FitToPage	Determine whether to scale the report, if the controls cannot fit in the width of the report.
RemoveRepeatedHeaders	Skip headers where no records are printed for the header.
RemoveRepeatedFooters	Skip footers where no records are printed for the footer.
RemoveRedundantFooters	Skip printing sum footers if only one record is to be summed.

Auto design

Property	Description
GrandHeader	Defines whether to print a header text at sorting breaks. This header text is identified as super grand header, and will be printed before the grand header which can be set at the body section.
GrandTotal	Will print a super grand total for controls where the property SumAll, SumPos or SumNeg is set.
HeaderText	This text will be printed as header text at sorting breaks if GrandHeader is

	set to Yes. If no text has been entered, the default test will be printed.
TotalText	If GrandTotal is set to Yes, this text will be printed as super grand total text, instead of the default text.

Sections controls

This section is an overview of all the different type of report section controls.

Property	Sections	Description
ArrangeMethod	All	Set the orientation for the arranged controls.
ArrangeWhen	All	Specify when the controls in the section must be arranged.
AutoDeclaration	ProgrammableSection Body PageFooter	If set to Yes, the section can be referred from X++ by using the section name.
Bold	All	Set the bold level for headings and data in the section.
Bottom	ProgrammableSection Prolog Body Epilog	Set a fixed position for the section calculated from the bottom of the page.
BottomMargin	All	Set the margin below the section data.
ColorScheme	All	Specify whether to use RGB colors, or Windows color scheme.
ColumnHeadingsStrategy	ProgrammableSection Body PageFooter	Define whether labels must be word wrapped or printed staggered. If set to DisplacedLines, the heading labels will be printed staggered on two lines, if the labels cannot be fit on a single line.
Columns	All	This property has no function. Used on forms to define the number of columns for arranging sub controls.
Columnspace	All	Set the space between columns.
ControlNumber	ProgrammableSection	The identification for a programmable section. Used to execute the section from X++.
Font	All	The font to be used for the section. If not specified the default font is used.

FontSize	All	The font size to be used for the section. If not specified the default font size is used.
FooterText	Body	Only available in auto designs. This text will be printed as grand total text, instead of the default text.
ForegroundColor	All	RGB value or name of Windows color scheme item.
GrandHeader	Body	Only available in auto designs. Defines whether to print a header text at sorting breaks.
GrandTotal	Body	Only available in auto designs. Will print a grand total for controls where the property SumAll, SumPos or SumNeg is set.
HeaderText	Body	Only available in auto designs. This text will be printed as header text at sorting breaks. If not text entered, the default test will be printed.
Height	All	Set a fixed height for the section.
Italic	All	Set the font to italic for headings and data in the section.
LabelBottomMargin	ProgrammableSection Body PageFooter	Set the margin below the heading labels, and before section data.
LabelTopMargin	ProgrammableSection Body PageFooter	Set the margin above the heading labels.
LeftMargin	All	Left margin for the section.
LineAbove	All	Add a line above the section data.
LineBelow	All	Add a line below the section data.
LineLeft	All	Add a line to the left of the section data.
LineRight	All	Add a line to the right of the section data.
Name	All	The name of the section. This is the name used from X++ to refer to the section, when the property AutoDeclaration is set.
NoOfHeadingLines	ProgrammableSection Body PageFooter	Labels are printed as heading lines. Used to set NoOfHeadingLines to a fixed value. Set to zero if no header should be printed.

ResolutionX	All	Seems as having no effect.
ResolutionY	All	Seems as having no effect.
RightMargin	All	Right margin for the section.
Ruler	All	Used to set the measure unit for the current section in the visual designer.
Table	Body	Used by element.send() to determine which body sections to be printed.
Thickness	All	Set thickness of the lines added for the section data.
Top	ProgrammableSection Prolog Body Epilog	Set a fixed position for the section calculated from the top of the page.
TopMargin	All	Set the margin above the section data.
Underline	All	Underline headings and data in the section.

Section Template

The section template is only used by report auto designs.

Property	Description
SectionTemplate	Name of the section template.
Table	The table used for the section template. The table must a part of the map used for the section template.

Section Group

The section group control is only used by report generated design.

Property	Description
DataField	Can be used to identify a section group if a report contains two section group using the same table.
Name	Name of the section group.
Table	The table used in the section group. Used by element.send() to determine when to print the section group.

Type controls

This section covers all type of report controls used by a section control.

Property	Type	Description
Alignment	String Real Integer Enum Date Time Sum Bitmap Prompt	Align the control data. Can be used to left align control data, when controls are positioned vertical.
AllowNegative	Real Integer Sum	This property is of no use on reports. The property is used on forms to prevent negative values to be entered by the user.
ArrayIndex	String Real Integer Enum Date Time Sum Bitmap Prompt	If the selected field or method is an array a single element of the array can be specified to be printed only.
AutoDeclaration	All	If set to Yes the properties for the control can be referenced from X++ by using the control name.
AutoInsSeparator	Real Sum	Used to have MorphX to set a decimal separator.
BackgroundColor	String Text Real Integer Enum Date Time Sum Prompt	RGB value or name of Windows color scheme item.
BackStyle	String Text Real Integer Enum Date	Set the background for the control to transparent. Used if the background color of bitmaps should not be shown, or to set the color of the background for the control data to the color set with the property BackGroundColor.

	Time Sum Bitmap Prompt	
Bold	String Text Real Integer Enum Date Time Sum Prompt	Set the bold level for control data.
BottomMargin	All	Set the margin below the controls data.
ChangeCase	String Text Enum Prompt	Used to set the control data to lower case or upper case.
ChangeLabelCase	String Real Integer Enum Date Time Sum Bitmap Shape	Used to set the label to lower case or upper case.
ColorScheme	All	Specify whether to use RGB colors, or Windows color scheme.
ConfigurationKey	All	Used to specify a Configuration Key for the control.
CssClass	All	Web property.
DataField	String Real Integer Enum Date Time Sum Bitmap Prompt	Select a field from the selected data source. Instead of selecting a field, a display method can be specified in the property DataMethod.
DataFieldName	Sum	Name of the control to be summed.
DataMethod	String Real Integer	Select a display method to be printed. If the display method is from a table, the data source must be specified in the

	Enum Date Time Bitmap	property Table.
DateDay	Date	Specify how day is shown. Windows regional settings are used as default.
DateFormat	Date	Set the date format. Windows regional settings are used as default.
DateMonth	Date	Specify how month is shown. Windows regional settings are used as default.
DateSeparator	Date	Set the date separator. Windows regional settings are used as default.
DateYear	Date	Specify how year is shown. Windows regional settings are used as default.
DecimalSeparator	Real Sum	Specify the decimal separator. Windows regional settings are used as default.
DisplaceNegative	Real Integer Sum	Adjust the position of negative values printed.
DynamicHeight	String	If set to Yes the height will be set according to the text.
ExtendedDataType	String Text Real Integer Enum Date Time Prompt	Specify an extended data type for the control if a DataField or a DataMethod is not specified.
ExtraSumWidth	Real Integer Sum	Defines extra space for the summed value. Useful in currencies with a lot of digits.
Font	String Text Real Integer Enum Date Time Sum Prompt	The font to be used for the control data. If not specified the default font is used.
FontSize	String Text	The font size to be used for the control data. If not specified the default font size

	Real Integer Enum Date Time Sum Prompt	is used.
ForegroundColor	All	RGB value or name of Windows color scheme item.
FormatMST	Real Sum	Format the value using the settings for the standard company currency.
Height	All	Set a fixed height for the control.
ImageName	Bitmap	Path and filename for bitmap.
ImageRessource	Bitmap	Used to specify a resource id for the bitmap. Use the report <i>Tutorial Resources</i> to get an overview of the resource id's.
Italic	String Text Real Integer Enum Date Time Sum Prompt	Set the font to italic for the control data.
Label	All	Used to override the default label or if no DataField or ExtendDataType is specified. The label will not be printed, if the property ShowLabel is set to false.
LabelBold	String Real Integer Enum Date Time Sum Bitmap Shape	Set the label to bold.
LabelCssClass	All	Web property.
LabelFont	String Real Integer Enum	Set the font for the label. If not specified the default font will be used.

	Date Time Sum Bitmap Shape	
LabelFontSize	String Real Integer Enum Date Time Sum Bitmap Shape	Set the font size for the label. If not specified the default font size will be used.
LabelItalic	String Real Integer Enum Date Time Sum Bitmap Shape	Set the label to Italic.
LabelLineBelow	String Real Integer Enum Date Time Sum Bitmap Shape	Print a line with the width of the control below the label.
LabelLineThickness	String Real Integer Enum Date Time Sum Bitmap Shape	Set the thickness of the line below the label.
LabelPosition	String Real Integer Enum Date Time Sum Bitmap Shape	Position the label above or to the left.

LabelTabLeader	String Real Integer Enum Date Time Sum Bitmap Shape	Used when the property LabelPosition is set to Left. Put tabs or dots follow by a colon after the label.
LabelUnderline	String Real Integer Enum Date Time Sum Bitmap Shape	Underline the label text. The property LabelLineBelow is used to set a line below in the full width of the label.
LabelWidth	String Real Integer Enum Date Time Sum Bitmap Shape	Used when the property LabelPosition is set to Left. Set a fixed width for the label.
Left	All	Set the control to a fixed position calculated from left. If controls are horizontal aligned and one control is set to a fixed position, all controls must be fixed.
LeftMargin	All	Set a left margin for the control.
Line	Shape	Specify the line type for the shape.
LineAbove	All	Add a line above the control data.
LineBelow	All	Add a line below the control data.
LineLeft	All	Add a line to the left of the control data.
LineRight	All	Add a line to the right of the control data.
MenuItemName	All	Web property.
MenuItemType	All	Web property.
ModelFieldName	All	Enter the name of a control which this control must be position according to. Is

		often used to position prompt and sums controls to be on line with the data controls.
Name	All	Name of the control.
NoOfDecimals	Real Sum	Set the number of decimals to be shown.
ResizeBitmap	Bitmap	Used to resize the bitmap if a fixed width and height is specified for the bitmap.
RightMargin	All	Set a right margin for the control.
RotateSign	Real Integer Sum	Used to invert negative values.
SecurityKey	All	Used to specify a Security Key for the control.
ShowLabel	String Real Integer Enum Date Time Sum Bitmap Shape	If set to No, the label will not be printed.
ShowPicAsText	Bitmap	Print the image path or the resource id instead of printing the bitmap.
ShowZero	Real Integer Sum	Defines whether zero values must be shown.
SignDisplay	Real Integer Sum	Set how to display negative values.
SumAll	Real Integer	Sum all values. Used by sum controls to define which controls to be summed. If set in auto design, the control will be summed if the user add a group total in the dialog.
SumNeg	Real Integer	Set if only negative values should be summed.
SumPos	Real Integer	Set if only positive values should be summed.
SumType	Sum	Set to sum only positive or negative

		values.
Table	String Real Integer Enum Date Time Sum Bitmap Prompt	Select a data source to be used for the control.
Text	Text	Enter the text to be printed.
Thickness	All	Set thickness of the lines added for the control data.
ThousandSeparator	Real Sum	Specify the thousand separator. Windows regional settings are used as default.
TimeFormat	Time	Set the time format. Windows regional settings are used as default.
TimeHours	Time	Specify whether to show hours. Windows regional settings are used as default.
TimeMinute	Time	Specify whether to show minutes. Windows regional settings are used as default.
TimeSeconds	Time	Specify whether to show seconds. Windows regional settings are used as default.
TimeSeparator	Time	Set the time separator. Windows regional settings are used as default.
Top	All	Set a fixed position for the control calculated from the top of the section. If controls are horizontal aligned and one control is set to a fixed position, all controls must be fixed.
TopMargin	All	Set the margin above the controls data.
Type	Shape	Set the type of shape to be printed.
TypeHeaderPrompt	Text Prompt	Specify whether to put following dots and a colon after the text.
Underline	String Text Real Integer	Underline the control data.

	Enum Date Time Sum Prompt	
Visible	All	Used to hide the control. If the follow controls are auto positioned, the controls will be adjusted.
WarnIfMissing	Bitmap	Print a warning in the Infolog if the bitmap cannot be located from the ImageName, ImageRessource or the DataMethod.
WebTarget	All	Web property.
Width	All	Set a fixed width for the control. If set to default, the extended data type will set the width.

Field Group

Property	Description
AutoFieldGroupOrder	Determine whether properties and field order must be saved. The field group will automatically be updated according to changes made to the field group from the data dictionary.
DataGroup	Field group name.
Table	Table from which the field group is picked.

12.4 Query properties

Query

Property	Description
AllowCheck	If set Configuration keys and Security keys will be validated at runtime.
Form	Specifies the Axapta form used for the query dialog. This is normally not changed.
Interactive	Specify whether the query dialog is shown to the user.
Literals	Choose either default, forcелiterals or forceplaceholders. All statements which are frequently executed should use forceplaceholders and forcелiterals can be used against non-frequent skewed data.

Name	AOT name of the query.
Title	Caption for the query.
UserUpdate	Defines whether user settings to the query are saved.

Data sources

Property	Description
AllowAdd	If disabled, the user can not add or delete table ranges. The same applies for sorting fields.
Company	Fetch data from a specify company. It is normally not used, as the user expect data to be fetched from the active company.
Enabled	Should the data source be enabled or disabled. If disabled, it will be ignored in the sql statement.
FetchMode	Only joined data sources has this property. Can be used to change a one to many relation to one to one relation. This property is not used very often. In the most cases the property JoinMode will be sufficient.
FirstFast	Should the first record be fetched faster than the remaining records.
FirstOnly	Only select the first record.
JoinMode	How should the data sources be joined.
Name	AOT name of the data source.
OrderMode	Choose either order by or group by. This will influence the way data is ordered or grouped from the database.
Relations	If enabled, relations between this data source and the data source one level above will be added using the actual data model.
Update	If set to yes, the record will be fetched forupdate allowing the record to be updated.

Fields

Property	Description
Dynamic	Show default all fields for the table. If the reports order mode is group by, the property will be set to no, and only aggregated values can be chosen for the <i>Fields</i> node.

Sorting fields

Property	Description
AutoHeader	Should a group title be printed each time the value in this field changes. Only used by report queries.
AutoSum	Should a sum be printed each time the value in this field changes. Only used by report queries.
HeaderDetailLevel	Used in report queries by the property AutoHeader. Default a header is printed when the value of the sorting field changes. The property can be used to determine for which part of a field an index break should occur. Fields using the characters dot, space, dash, back slash forward slash consists of several sub fields. If the value of customer account is "CUST100.10 ", the field value consists of two subfields. Setting HeaderDetailLevel to 2 will cause an index break to occur, when the value after the dot is changed.
Ordering	Set the ordering to either ascending or descending.
SumDetailLevel	Used in report queries by the property AutoSum. Default a header is printed when the value of the sorting field changes. The property can be used to determine for which part of a field an index break should occur. Fields using the characters dot, space, dash, back slash forward slash consists of several sub fields. If the value of customer account is "CUST100.10 ", the field value consists of two subfields. Setting SumDetailLevel to 2 will cause an index break to occur, when the value after the dot is changed.

Ranges

Property	Description
Enabled	If disabled, the range will be ignored.
Label	This is the label of the range field.
Name	AOT name of the range.
Status	Choose either open, locked or hide. If open, the range can be modified and deleted. If locked, the range can not be modified or deleted. If hidden, the range will not be shown in the query dialog.
Value	Fixed value for the range.

12.5 Menus Properties

Property	Description
ChangedBy	The user who last modified the Menu.
ChangedDate	The date the Menu was last modified.

ChangedTime	The time the Menu was last modified.
ConfigurationKey	Used to specify a Configuration Key for the menu.
CreatedBy	The user who created the Menu.
CreatedTime	The time the menu was created.
CreationDate	The date the menu was created.
HelpText	Help text that will be displayed in the status bar.
Label	The label of the menu.
LockedBy	The user who has currently locked the menu.
Name	The menu name.
NeededAccessLevel	Required access level to activate this menu.
SecurityKey	Used to specify a Security Key for the control.
SetCompany	Would you like to set the company when the form is activated.

12.6 Menu Items Properties

Property	Description
ChangedBy	The user who last modified the menu item.
ChangedDate	The date the menu item was last modified.
ChangedTime	The time the menu item was last modified.
Class	The object type that will be executed when activating the menu item.
ConfigurationKey	Used to specify a Configuration Key for the menu.
CountryConfigurationKey	Used to specify a country specific Configuration Key for the menu.
CreatedBy	The user who created the menu item.
CreatedTime	The time the menu item was created.
CreationDate	The date the menu item was created.
EnumParameter	Set a base enum entry, see also the parameter EnumTypeParameter.
EnumTypeParameter	Select a base enum to be passed to the args.parmEnum() in the object which is called. See properties Class and Object.

HelpText	Help text that will be displayed in the status bar.
Label	The label of the menu item.
LockedBy	The user who has currently locked the menu item.
MultiSelect	Can this menu item be executed when multiple records are selected in the form.
Name	The name of the menu item.
NeededAccessLevel	Required access level to activate this menu item.
Object	The object that will be executed when activating the menu item.
Parameters	Enter a string value to be passed to the args.parm() in the object which is called. See properties Class and Object.
RunOn	This property will specify whether the object is executed on AOS or not.
SecurityKey	Used to specify a Security Key for the control.
Web	Web property.
WebConfigurationKey	Web property.
WebPage	Web property.
WebSecureTransaction	Web property.

13 Appendix MorphX Development Tools

Located in the top menu at **Tools | Development tools** you have set of tools created using MorphX. These are all tools enabling an overview of the application objects. Most of the development tools can be called from the AOT or from the editor by right-clicking. The debugger is also accessible from the development tools menu, but there is no sense running the debugger from the development tools menu, so the debugger is explained in the chapter **Intro to MorphX**. If you want to have a look at the development tools from the AOT, check the menu **DevelopmentTools** in the AOT.

13.1 Cross-reference

Cross Reference is a tool used to locate where objects, variables and labels are used in the application. If you are about to change the name of an object or a variable, or want to delete a label from the label system, the cross-reference tool helps give you an overview of the consequence of modifying the code.

Before being able to use the cross-reference, you must make a complete re-compilation with cross-reference enabled. You can enable the cross-reference to be built while compiling in the compiler setup form. This will however, increase the compilation time significantly. A single run of the cross-reference can be done from the top menu at **Tools | Development tools | Cross-reference | Periodic | Update**. Select **Update all** to build the cross-reference. Building the cross-reference takes hours and requires a lot of memory. It is not a quick job as the cross-reference will index all objects in the entire AOT. To have the full benefit of the cross-reference, it must be kept up to date. This can be done by setting up a batch job updating the cross-reference at off hours.

The cross-reference can be called from the top menu or by selecting an object in the AOT, right-click and choose **Cross-reference** in the **Add-ins** menu. The result of the cross-reference is stored in the tables prefixed with **xRef**.

Note: If you have done a new standard installation of Axapta, you should consider building the cross-reference and export the cross-reference tables. Whenever you need the cross-reference for a new installation, you can just import the cross-reference data, rather than starting all over again.

The forms *Names* and *Path*, called from Cross-reference in the top menu lists all entries from the cross-reference. The form *Names* list the entries by object type, and the *Path* form will show the AOT path for each entry. If you are going to filter records in the cross-reference, or are searching for a specific object in the AOT, these forms are useful. Otherwise, if you know the AOT path to an object, it is much quicker to drill down to the AOT object, right-click and choose the cross-reference from the **Add-ins** menu. From an AOT object, the *Path* form will only show AOT paths to the location where the selected object is used. Using the cross-reference from, an object gives some additional information like which other objects the selected object uses, and where the selected object is used in the application. All cross-reference forms have the menu item

Edit, used to lookup the X++ code for a method. If the cross-reference entry is a property, the property sheet can be looked up using the Add-ins menu and choosing Properties.

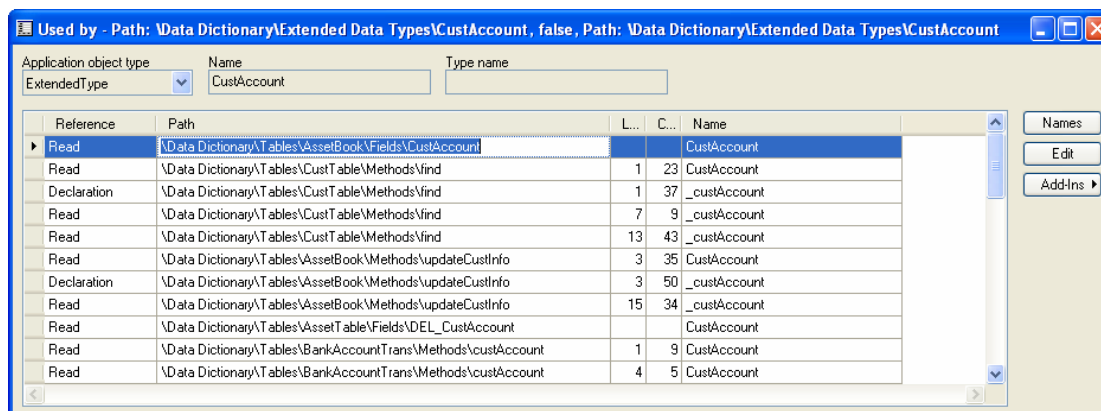


Figure 48: Cross-reference showing where the extended data type CustAccount is used

13.2 Application Objects

The sub menu Application objects can be found in the AOT node *Menus* with the name *ApplicationObjects*. These tools are not ground breaking tools which you will be using a lot, rather the application tools are considered as “nice to have”. As the menu is created in the AOT, you can add your own tools.

Application objects forms

The forms *Application objects* and *Old application objects* list the two system tables *UtilElements* and *UtilElementsOld* which contain information on respectively all object in the AOT, and all objects in the old layers. From the Add-ins submenu in the AOT, you can call the form Application objects filtered on the record for the selected node in the AOT.

You will have to be patient using the form Old application objects, as the form takes ages to load. The form is joining the tables *UtilElements* and *UtillementsOld* to show the differences. This could be a nice feature to compare an upgraded version with the previous version, but the form is so slow it makes the form useless. For more information about the system tables prefixed with *Util**, see the chapter **Data Dictionary**.

Application management

You can use this tool to set people in charge of the single application objects. The Application management form will give you an overview of who is in charge of which objects, and which team the person in charge belongs to. In the related forms, persons

and teams can be defined. The application management tool must have been created for internal use in the Axapta development teams. It is a simple tool, and it might not be fully tested. Just take a look at the overview in the Application management form. The selection of the application object type is placed as the second field to be filled out first, before you can select the application object name. However you can modify the tool to fulfill your needs. All objects are prefixed with `SysUtilMangement*`.

Usage data

The Usage data form shows the content of the system table *SysLastValue* for all users. For information on usage data, see the section **User settings**.

Count of application objects

The form is used to get an overview of the number of objects, like tables and forms used for each module. Only the modules from the standard packaged are counted. If you want to count your own modules, you will have to modify the class *SysUtilCount*. The table *SysCountTable* holds the counted result.

Locked application objects

If you are using the lock application object feature in the AOT, you will find this form useful. The form shows a list of the application objects locked by all users. If you delete an entry in the form, the selected object will be unlocked. However the object will still be shown as locked in the AOT, as the changes made in the Locked application objects form will not be reflected in the AOT unless the Axapta client has been restarted.


Refresh tools

The three menu items, Refresh Dictionary, Refresh Data and Refresh AOD, are used in web development. They are used to flush objects and data so the web interface will be updated with the latest changes.

Re-index

Re-index will re-build the application objects layer index file. You should never re-index the application objects layers if there are other users in the application, as this will corrupt the index file. If you need to do a re-index of the layers, it will be preferable to just delete the index file `AXAPD.AOI`. The file can be deleted when there are no clients logged in, and all AOS servers and the COM connector have been stopped. When starting your Axapta client and logging in, the index file will be built automatically.

13.3 System Monitoring

The system monitoring tool can be called from both the top menu and by double clicking the computer icon  on the status bar. The form System Monitoring shows database calls, and size of the calls. If you are using an AOS server, you will have additional information for client and server calls and an extra tab page showing information on latency. These are all useful pieces of information if you are going to optimize the traffic between client, server and the database.

Database tracing

When starting to trace a sequence, press the *Continue* button and try out your modifications, like opening the sales order form, and create a sales invoice. Press the *Pause* button to stop tracing. This will count the different type of database calls. In many cases managing the count of selects is where you can most effectively optimize an application.

AOS tracing

When using an Axapta Object Server (AOS), objects will either be executed on the client or on the AOS. The calls between client and server will be counted by the system monitoring form. You can use the result of the AOS calls to help minimize the client calls. Objects with user interaction, like forms and dialogs will always be executed on the client. However, it is preferable having classes and database calls executed on the server. For more information on how to set a class to be executed on the server, see the chapter **Classes**.

From the tab page *Remote connection*, you can test the AOS latency. To get the correct latency you should test it a couple of times. If your application is going to be used from clients with limited bandwidth, you can model the user experience using the application by simulating a remote connection. Set the bandwidth and the latency, and press the button *Set as current*. This allows you to test the application's performance under various communication scenarios, and provide good feedback on how you can optimize your code.

13.4 Code Profiler

The *code profiler* is used to calculate the execution time and the database time of the code. Like the system monitoring tools, the code profiler is activated by a start button. When you have completed profiling the features from the menu to be checked, you press the stop button. The trace depth option is used if you only will trace a specific number of levels. When the profile run is stopped, the system prompts for a name under which it will save the collected data. Each profile run is stored in the database and can be accessed by pressing the button *Profiler runs*. The form *Profiler runs* shows profiles

compiled to date. You can drill down the profile and see each code line executed and even edit the code. For a better overview of the code lines use the call tree. Execution time for each code line is calculated, and if chosen, the total time for repeated calls of a code line is available. These total time values are especially important when evaluating code that is repeatedly executed in a loop.

```
static void Intro_CodeProfiler(Args _args)
{
    CustTable      custTable;
    Counter    counter;
;

    #profileBegin("Test of profiler")
    while select custTable
    {
        info(strFmt("Customer name: %1", custTable.name));

        counter++;

        if (counter >= 10)
            break;
    }
    #profileEnd
}
```

Another option for using the code profiler is to specify macro calls which activate the code profile directly in the code. The profile start and stop point must be specified in the code as shown in the example. Setting the profiler calls directly in code is useful when you only want to check a specific part of your code. When you run the profile from the menu, you might be tracing more code than necessary. Notice, you will have to calculate totals manually from the Profiler runs form afterwards.

If you are using an AOS, the AOS time will also be calculated. In the overview of the Profiler runs form the profiles will be listed whether a profile is called from an AOS client or not. You should try running the example from both a 2-tier and a 3-tier client. Notice that fewer lines are generated when the code is executed from an AOS client. The reason for this is that the code is executed on the AOS and fewer calls to the client.

Using the code profile tool is good training as you get to know the execution time of your code, and thereby learn where to optimize. Notice that it is only execution time and not the overall time which is calculated. The code profiler generates a large number of lines just openings a form and calling a few jobs from the form. When the code profiler is running your system it might act slowly, and if you set the code profiler running on a huge batch job the profiling may never finish. If you are going to trace the code of batch job you might be better off limiting the number of runs for the job, or by using the code profiler macro calls.

13.5 Application Hierarchy Tree

Before using the *Application Hierarchy Tree*, the Cross-reference for the type hierarchy must be built. Go to **Tools | Development Tools | Cross-reference | Periodic | Update** in the top menu and chose *Update type hierarchy*.

The best practice in Axapta is to use extended data types, rather than using the base types. Over time, you will find that you create a lot of extended data types, as the extend data types holds information like labels, formatting and relations. The Application Hierarchy Tree comes in handy as it will give you an overview of the extended data types and how the extended data types are inherited. When you are creating a new table and have to select whether to create a new extended data type or figure out if an existing extending data type matches your needs, this tool is quite useful. The names used for the base types are not quite equal to the names used in the AOT. This might be a bit confusing, however knowing that the base type *varstring* is equal to the base type memo helps a lot.

Tables and classes are also listed in the Application Hierarchy Tree. The tables are listed under the top node Common. All tables are at same level as tables are not inherited. You can use the list of tables to browse information about each table such as which methods, fields and indexes a table holds. Both application tables and system tables are shown. Methods and indexes for system tables are also shown which cannot be seen in the AOT. The Object node shows a tree of the classes, and how the classes are inherited. For each class you will have information about the methods and for inherited classes. You can easily see to which class a method belongs.

13.6 Visual MorphXplorer

The *Visual MorphXplorer* is used to build entity relation diagrams showing table relations and class inheritance. Diagrams are built using the first tab page. If you want a title for your diagram, the title can be set from the general tab page. The color tab page is used to change the default colors for the diagram. Diagrams can be saved and printed. A neat feature when printing large diagrams is that you can set the number pages used in width and height.

To add an object, go to the diagram tab page, right-click and choose new table or class. When choosing a table or a class, a list box will be shown where you select an object. Drag the object to the diagram tab page to position the object. Right click the positioned object to see and add related objects. Notice that you cannot combine tables and classes in one diagram. You can use Visual MorphXplorer to build a diagram showing either relations between tables or to show a class hierarchy. If related objects are not shown in the Visual MorphXplorer you will have to build the cross-reference for the data model. Go to **Tools | Development tools | Cross-reference | Periodic | Update** and choose *Update Data Model*.

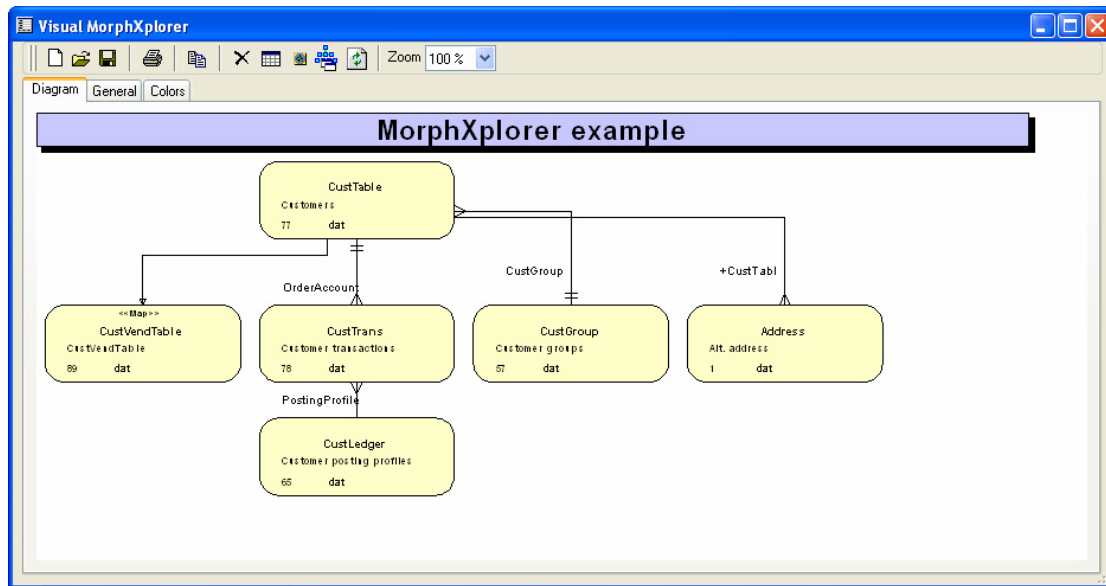


Figure 49: Visual MorphXplorer

Related tables are divided into 1-n and n-1 relation. This is the same way an Axapta query presents related tables. If a table is part of a map, you will also be able to select the map to be related. For an overview of the symbols in the diagram, see **figure 50: Symbols used in Visual MorphXplorer**. When adding a related table, the field making the relation will be printed. If the relation consists of more than one field, the relation fields will not be printed. Instead, the table name will be printed prefixed with a +. This is a limitation of the Visual MorphXplorer, as it would be preferable to see all relation fields, at least in a tool tip box when moving the mouse over the relation.

When building a diagram for a class hierarchy, super and sub class can be chosen. If you have built the cross-reference, you will also be able to select classes using and used by the class.

Symbol	Object	Description
	Table	Indicated 0-n records for the relation.
	Table	Exactly 1 record for the relation.
	Table Class	For tables indicating that the related table is a map. Used for class to indicate a super or a sub class.
	Class	Class using or used by the related class.

Figure 50: Symbols used in Visual MorphXplorer

Visual MorphXplorer can be activated from the Add-ins menu in the AOT. The menu item for the Visual MorphXplorer will only be available if a single table or a single class is chosen. If you import the AOT project MORPHXIT_VisualMorphXplorer you will be able to multi select tables from the AOT and have the diagram shown in the

Visual MorphXplorer with relations. Notice that if there are more than one related field between two tables, like the tables *CustTable* and *CustTrans*, both relations will be drawn. This modification can help you get a quick overview of the data model. You can show a sub module using this modification, but it has some limits. You can only multi select tables and if you choose too many tables, your client will crash.

The form Visual MorphXplorer is using the *VarChart* component to display the diagram. The *VarChart* component is used in several places in the standard package, but it is not well documented, little information is available about this component. However, the Visual MorphXplorer form is a good place to check out some of the features available with *VarChart*.

13.7 Code Explorer

The *Code Explorer* is used to browse the AOT in a HTML style. You can drill down the single nodes and get information on layers, properties and have the code for methods displayed. Cross-reference information will be shown, if updated. Code Explorer uses the help system to display the information. If you need an example on how to use the help system from code, it might be worth checking out the classes prefixed with *SysCodeExplorer**.

13.8 Table Definitions

This menu item will print the content of tables stored in the *UtilElements* table. The report, printing the table definitions, is organized in a nice way showing all the important information on a table like fields, properties and relations. Specify a range for the content you wish to print in the query, otherwise you will keep your printer busy for a while, as the entire content of all tables will make up more than 2000 pages. Use this report only if you need information in print for a few tables or a sub module.

13.9 Number of Records

The *Number of records* form is used to count the records in the current company for each table. The form also lists temporary tables and maps, but only tables will be summed. When setting up caching of tables, the records counts are useful. You might have set entire table cache for a table, but if the cached table contains a lot of records, you might have to reconsider the caching of the table. For more information about caching options for tables, see the chapter **Appendix Properties**.

13.10 Help Texts

The online help is shown in a HTML style. When pressing F1 on a form or an AOT node, the online help for the node is shown. The content of the help system is listed in the form Help Texts.

Three different kinds of online help exist: System Documentation, Application Developer Documentation and Application Documentation. In the AOT you will find the online help as the last three nodes. System documentation is the source for information on kernel objects like class and tables. The Application Developer node is used for creating online help for your own tables and classes. When a new table or class is created, a new entry will be made to the Application Developer node. Application Documentation covers the online help presented to the application users. Editing the online help can be done either by using the form Help texts or by using the three AOT nodes.

Note: Few table and classes have online help. This is supposed to be improved by the release of version 4.0.

13.11 Version Update

As new versions or service packs come out, you will need to upgrade your application. The layer technology in Axapta facilitates the process as modifications are separated from the base code delivered from Microsoft. Still, you will have to check a lot of code manually. If you have modified a form in the VAR layer, and the same form is modified in SYP layer of a new service pack, then you will have to manually check the form. For this the compare tool, described in the section **Compare objects**, can be used. Before using the compare tool, you can get an overview of the changes in the new release by using the Version Update tools.

Renamed application objects

Objects renamed in a new service pack or a new version will be listed in the *Renamed application objects* form. The content of the form must be built manually. To build the list you will have to start the application to be upgraded and click the Update button. This will create a text file containing the list of the renamed objects. The text file can now be imported in the new release by clicking the Update button.

The overview of the renamed objects is useful, as only objects where the name has not been changed need to be compared. You might have done modifications to a SYS object, which is renamed in the new release, so you will only have your own layer of the object. The list with renamed objects will help you to track down the changes, when the object is renamed or deleted.

Create upgrade project

When doing an upgrade of an application, the first step is to create an upgrade project. This will create an AOT project containing all objects where the current layer is modified. An option can be checked to delete changes from the current layer, which now is a part of the lower layer. If you have imported a hot fix in the current layer, you might consider using this option, as hot fixes are normally merged into the released service packs or version updates.

Having created the upgrade project, you will be able to carry on comparing the objects using the compare tool.

Compare layers

To make a complete comparison of the differences in two layers, the *Compare layers* tool can be used. An AOT project will be made with the objects which only exist in the chosen source layer or where the source and the reference layers are not equal.

If you have upgraded to a beta version, or if you have installed a pre-released service pack, using the compare layers tools will give you a quick overview of the changes made between the different versions.

13.12 Wizards

The *wizards* in the Wizards menu will help you get started creating the base of objects like reports and classes. Starting out using a wizard can be a good exercise, as you get to know how the basic of an object is created, like which properties are usually set and which controls are usually added. The report wizard is especially useful, and will help you discover how to use certain properties rather than browsing around the report nodes in the AOT.

If you drill down the menu node in the AOT, you will find the menu Wizards. From here you can add your own wizards.

Report Wizard

To learn how the report generator works, use the report wizard. The wizard will build the base for your report. If you have simple reports, you might not even have to go to the AOT. For a complete walk through of the report wizard, see **Appendix Report Wizard**.

Wizard Wizard

Until Axapta 2.5 it was considered best practice to provide the application users with the option of using wizards for creating records in complex forms. This was changed

with version 3.0 and the introduction of record templates. The Wizard Wizard was created to streamline the development of these wizards.

Even though it is not considered best practice to use wizards for adding data any more, you might have special cases where a wizard will be preferable, especially in cases where the application user will have to add data to an infrequently used form.

Label File Wizard

This is the only wizard not used for creating AOT objects. This wizard is used for creating a new label file. When creating a label file using the wizard, you should start a 2-tier client with no other users logged on. After you have completed the wizard, restart the client before any other users log on the application. Then the new label file will be ready for use. Notice the label file will first be updated when the last user logs off the application, therefore it is recommend that label files be created without any users logged on to Axapta.

For more information on the label system, see section **Label**.

Class Wizard

This is a very simple wizard. The class wizard will create a class in the AOT and add methods from interface classes. A single template is selectable in the wizard. However, you can extend the wizard by adding your own templates, otherwise the wizard is of more of interest for training purposes.

COM Class Wrapper Wizard

If you consider interfacing an ActiveX component using the COM interface, this wizard is the right place to start. This wizard will list all component libraries installed on your computer. Select a library and Axapta classes will be created wrapping all classes and methods for the library. This is awesome, and really speeds up performing a COM interface.

13.13 Label

The label system is one of the powerful features of Axapta making the application handling of multi languages easy. Instead of entering the text for a field or a help text directly in the code, a label id is added. The label id is drawn from the label system which holds information of the corresponding text for label in each language.

The label id has the systax: @<label file id><label number>, like @SYS1002. The label file id is a three character id. Label number is a forth running number, increased when a

label is added. In the Axapta application folder, a set of labels files exist for each label file id. The labels files are named AX<label file id><language id>.<extension> like AXSYSEN-US.ALD. For an overview of the label files, see **figure 51: Label files**.

File extension	Description
*.ALD	A text file containing all labels for the label file id.
*.ALC	Comments added in the label system are stored in this file.
*.ALI	Index file for the label file id. If this file is missing, the file will automatically be created first time the label system is accessed.

Figure 51: Label files

The label fields are updated when the last user logs off the application. If for some reason the last user is not logged off properly, the label files will not be built. As the label files is a part of the modifications, together with your AOT modifications, it is crucial that you assure that the label files are updated with the last labels added. Go and edit the *.ALD file to check that your last labels have been added. If a label is missing in the file, the application user will see the label id instead of the label text.

Note: The reason for using labels is so the application users can run the application in their preferred language. Even if you have no need to translate your modifications to another language, you should still consider using labels as the use of labels will assure that you use consistent naming throughout your application. If you are going to change a term, you will only have to change a single label, rather than traversing your code.

The standard package has a label file for each layer named after the layer. It is not recommended to modify these label files as for each service pack or version released, these label files might have been updated. Instead you should create a new label file for your modifications.

Find label

This form is used to search the label system. This is the same form opened when looking up a label from the property sheet or from the code editor.

The search for labels will be done in the language selected at the top. If you want to search for a label in English, you can have the label shown in other languages by going to the advance tab page and select the desired languages. Your search will still be done in English, but as an addition the selected languages will be listed in the bottom of the form. This can be useful if you want to assure that labels have been added for all the languages you use for formulas like the sales invoice. When searching for a label using < and > will narrow your search. If you want to look up the label 'Customer', it will perform faster by keying in <Customer> as only labels with the exact text 'Customer' will be found. To find all labels starting with 'Customer', you simply enter <Customer.

Normally the label system is called from the property sheet or from the code editor, where you lookup a text to find the appropriate label for the text. If the label is found, you click the *Paste* label button to return the label id. If no label is found, you can add a new label based on the search text by clicking the *New* button. The new label will be created in the default label file id specified in the advanced tab page.

Label log

All changes made according to the labels in the standard package are logged and shown in this form. Labels added, deleted and modified are logged. If you have deleted a label by mistake, the label can be recreated by using the Label log form.

Label file wizard

For a description of the label file wizard, see the section **Wizards**.

Label intervals

The Label intervals form can be used to administrate the label id's in your own label files. If you are working in an environment where modifications are made in more than one application and the same label file is going to be used, you can use this form to define label intervals for each application. Enter the label file id. Interval status must be available to label id numbers. In each application you must specify a label interval to be used, and set the last used label id used in the interval. Next time a label is created, the label id will be taken from within the interval, and the last label used will be incremented in the form.

The preferable solution will always be to create your labels in one application, but this solution can be used as a work-around. You will of course have to merge your label files prefixed with *.ALD manually.

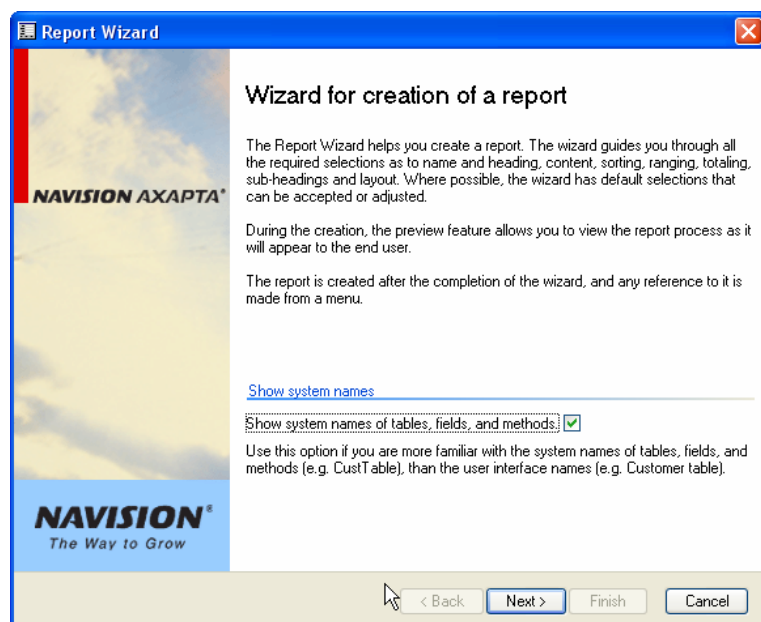
14 Appendix Report Wizard

This appendix contains a step-by-step guide on how to create a report using the report wizard. The report wizard is a tool designed to non-technical skill persons for writing simple reports in Axapta. This is however also a useful tool when learning to write reports in Axapta.

You can either save the result from the wizard as a report in the AOT or just run the report at the end of the wizard. The saved report will be shown as a report in the AOT. Whether using the wizard or the report generator for writing a report, the report will be saved in the AOT. Till you get familiar with the report generator the wizard will be helpful, as you are guided through the basic steps when setting up the structure of your report.

Step 1

Specify whether you want to use System names, or label names. System names are the names used in the AOT. It can be advantages using the system names, as the system name are unique.



Step 2

Enter the name for the report. The name will be used for saving the report in the AOT. Caption is the print name of the report.

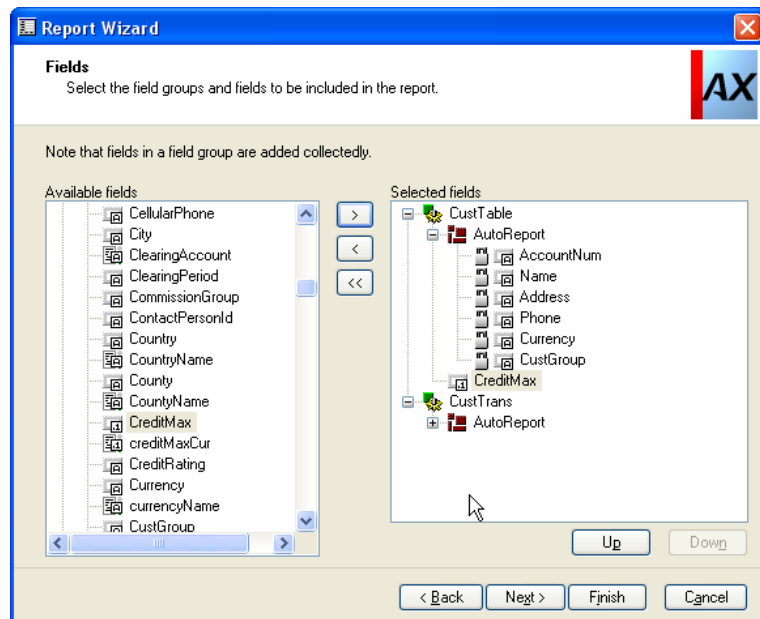
Step 3

Select the tables where data should be fetched from.

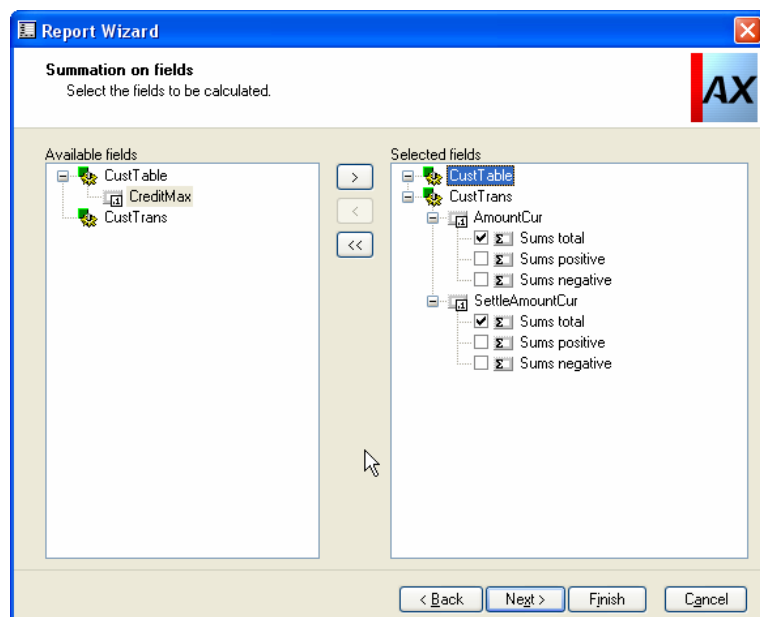
Try select CustTable as the first level table and CustTrans as the second level table. If the related table window is empty, when selecting the table CustTable, you will have to update the cross reference for the data model. This must be done after installing Axapta, or each time changes have been made to the data model. To update the data model go to **Tools | Development tools | Cross-reference | Periodic | Update**. Select Update Data Model only.

Step 4

Select the fields to be printed for each data source. By default the field group *AutoReport* will be selected for each data source. It is possible to select between field groups, table fields and display methods. The *Up* and *Down* buttons are used to set the print order of the fields.

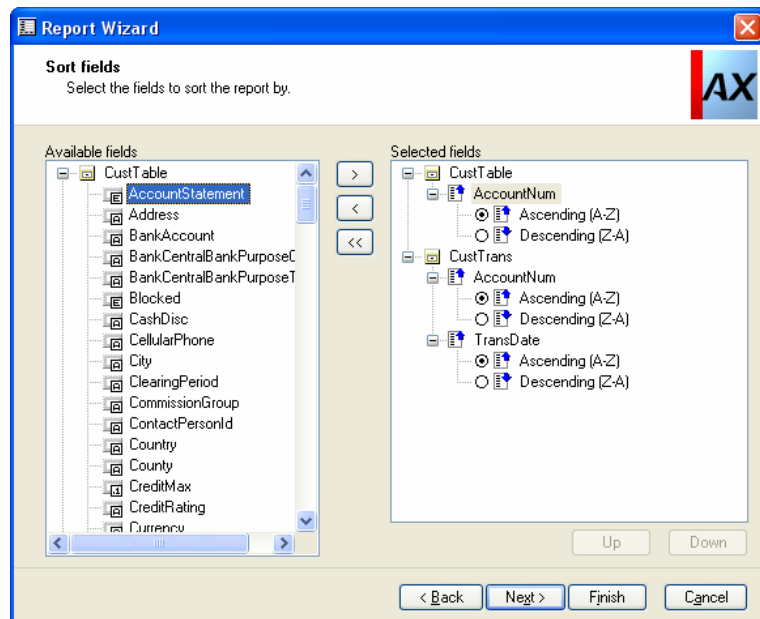
**Step 5**

Select which fields of the base type's integer and real to be summed. By default all integer and real fields will be selected to be calculated.

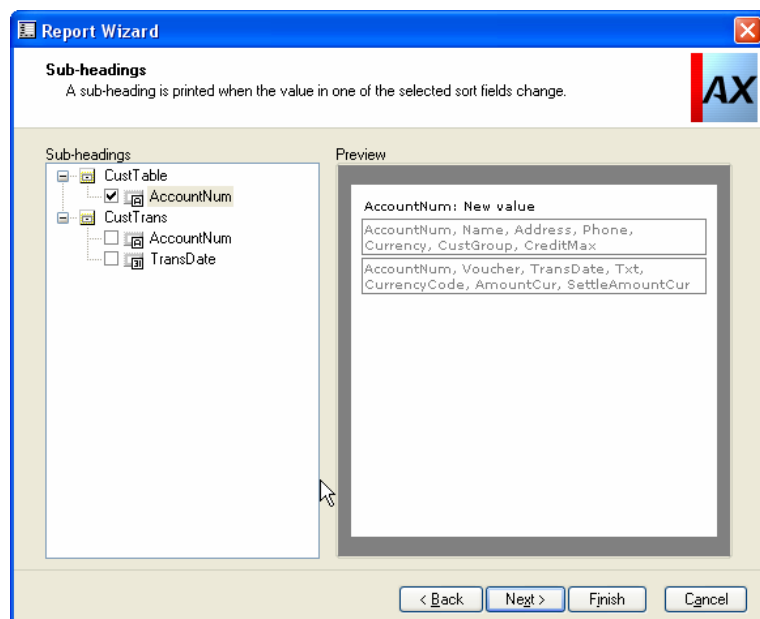


Step 6

Define the sort order for the index fields. By default all fields which are a member of an index will be selected. The *Up* and *Down* buttons are used to set the sort order of the fields.

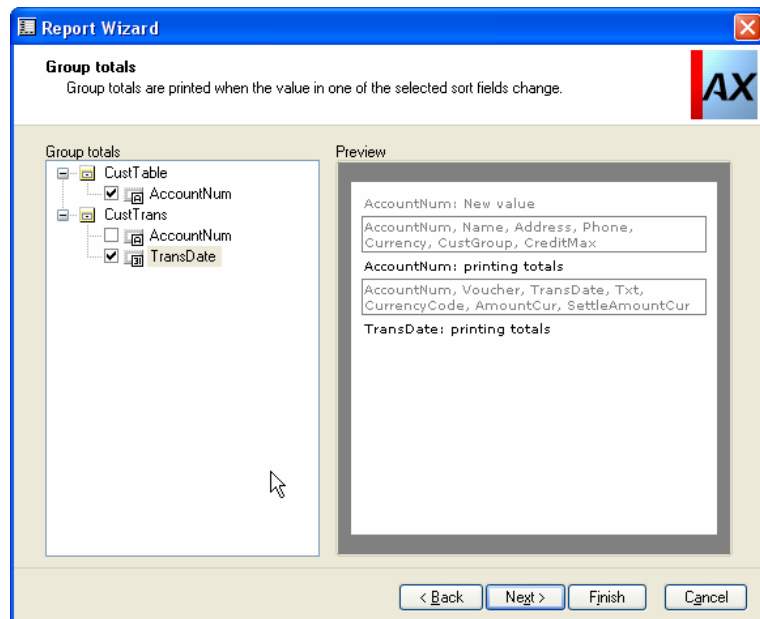
**Step 7**

Specify whether to print a sub-header each time the value in one of the defined sort fields are changed. In the example a sub-header will be printed each time the account number in the customer table is changed.

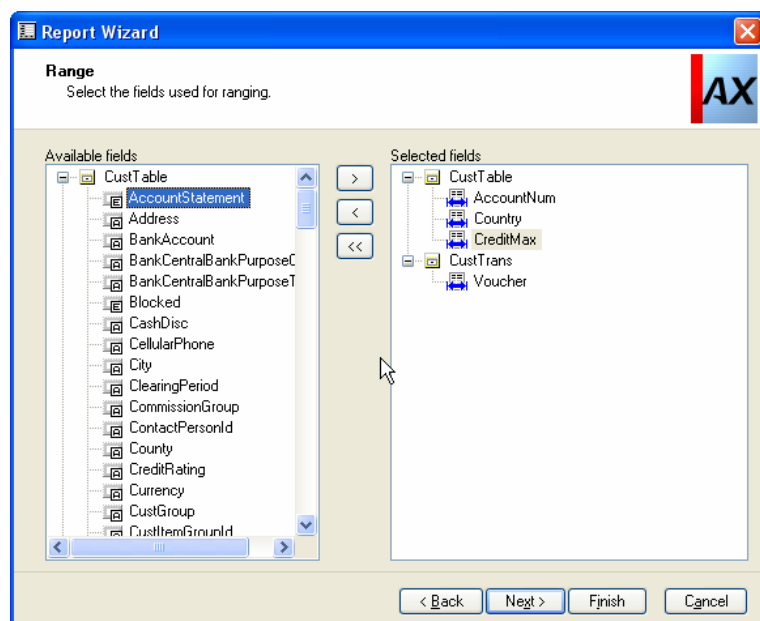


Step 8

Specify whether to print group totals when the value in one of the defined sort fields is changed. In the example group totals will be printed each time the account number in customer table is changed, and when the transaction date in the customer transactions is changed.

**Step 9**

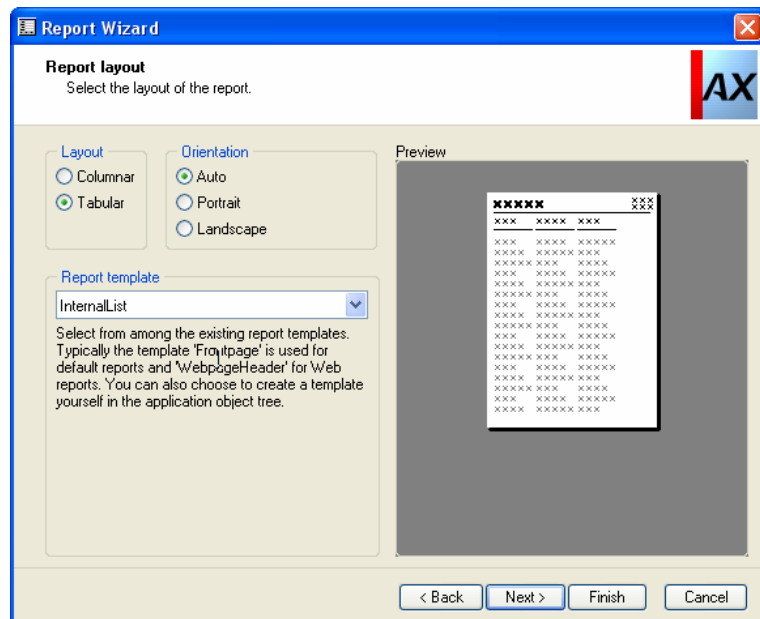
Select the query ranges. By default all fields used by the tables' indexes will be selected.



Step 10

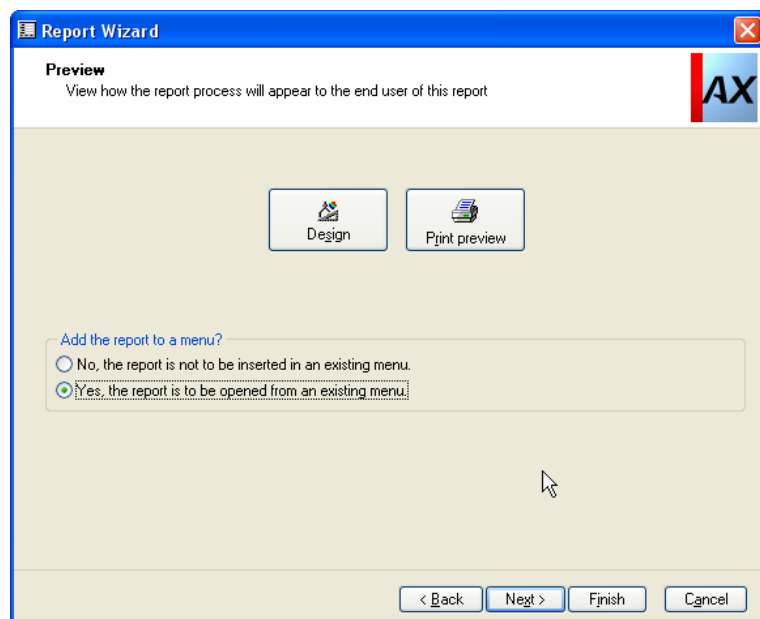
Define the layout and whether or not to use a template for the report.

Report templates are located in the AOT under *Reports*.

**Step 11**

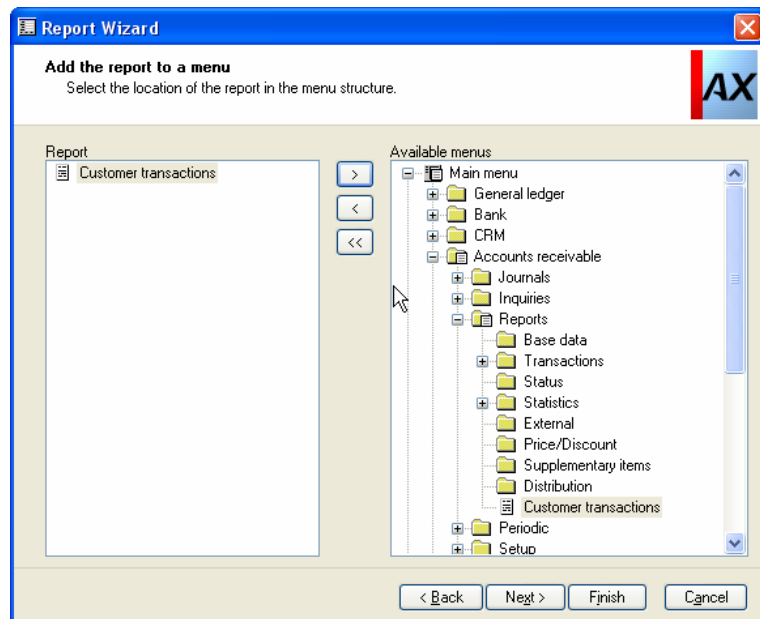
Specify whether to add the report to a menu.

The *Design* button will open the report in the AOT in design view. The *Print preview* button can be used to preview the defined report.



Step 12

Select a menu for the report. When adding the report to a menu, a new menu item of the type *Output* will be created.

**Step 13**

All steps have now been done. Click *Finish* to save the report in the AOT.

