

Professional Expertise Distilled

Learning Dynamics NAV Patterns

Create solutions that are easy to maintain, are quick to upgrade, and follow proven concepts and design

Foreword by Thomas Hejlsberg, CTO of Dynamics NAV

Marije Brummel

[PACKT] PUBLISHING enterprise professional expertise distilled

Learning Dynamics NAV Patterns

Create solutions that are easy to maintain, are quick to upgrade, and follow proven concepts and design

Marije Brummel

[PACKT] PUBLISHING enterprise professional expertise distilled

BIRMINGHAM - MUMBAI

Learning Dynamics NAV Patterns

Copyright © 2015 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: September 2015

Production reference: 2180722

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78528-419-9

www.packtpub.com

Credits

Author
Marije Brummel

Reviewers
Alex Chow
Luc Van Dyck
Alain Krikilion
Rakesh Raul
Daniel Rimmelzwaan
Bill Warnke

Commissioning Editor
Kunal Parikh

Acquisition Editor
Tushar Gupta

Content Development Editor
Priyanka Mehta

Technical Editor
Rohith Rajan

Copy Editor
Yesha Gangani

Project Coordinator
Izzat Contractor

Proofreader
Safis Editing

Indexer
Tejal Soni

Graphics
Sheetal Aute

Production Coordinator
Aparna Bhagat

Cover Work
Aparna Bhagat

Foreword

One of the many things that make Microsoft Dynamics NAV unique is that it is fully customizable and extendable. The ERP application, shipped by Microsoft as a part of NAV, is completely customizable and extendable using the traditional raw source code modification, or using the events as of Microsoft Dynamics NAV 2016. This makes the product extremely flexible and adored by the SMB market.

NAV has been built to focus on simplicity. It has been built with a strong belief that the application developer need not know all about the technical domain, like transaction isolation, advanced error recovery, database locks, memory allocations, and so on, but instead, should be able to unleash his or her domain skills. In short, write the happy path code and leave the rest to the NAV—no more, no less. I still believe that these principles hold their value and lately, we can add web-services, cloud, devices, and other new terms to the list of challenges that Microsoft Dynamics NAV handles behind the scenes almost like magic.

The application that is shipped with Microsoft Dynamics NAV has been designed by keeping simplicity and extendability in mind. The simple yet powerful syntax of the the application language, the ability to copy and paste parts of the application and reuse it in a different way, has always been a part of the Microsoft Dynamics NAV DNA. Although copying and pasting, from a purely engineering perspective, could be considered bad practice, it has proven itself very productive over the years.

Over a period of time, many Patterns that are found in today's application have evolved naturally. Many of them were first used in the character-based version, upgraded to the Windows client, and then later to the current three-tier architecture. While the adherence to and use of such Patterns is purely optional for developers, multiple reasons exist to follow the recommended Patterns and guidelines—the code will typically be easier to test and easier to upgrade, just to mention two.

By following some guidelines and principles, the sharing of application code in a large and growing community will also benefit tremendously. It might be as simple as consequent variable naming when writing code, or consistent naming of fields and tables when designing the data model.

I believe that small things like identifying recurring Patterns and assigning names to these can enhance communication, and create a common syntax between developers. It allows the community to discuss best practices, and improve and share design. This is very important, since downstream it allows the design of extensions that are easy to merge, have a lower maintenance cost, and allow easier implementation of additional changes.

Using common guidelines and best practices make it easier to work together and share code in an ecosystem that is dynamic and spans across the globe.

This is what makes Patterns unique and powerful—just like NAV itself.

Thomas Hejlsberg
Chief Architect
Microsoft Dynamics NAV
August 2015

About the Author

Marije Brummel - Author, Programmer, Consultant, Project Manager, Presenter, Evangelist, Sales person, and Trainer. She has been working for Partners, ISV's, End Users, Master VARs, and Microsoft across the globe in more than 25 countries. It's next to impossible to find someone as widely and deeply experienced as Marije in the Business Central community.

She received numerous awards including the Microsoft MVP award and the NAVUG All-Star. She was chair of the Dynamics Credentialling committee and authored the official Microsoft Exam materials. She did both onsite and online readiness sessions for Microsoft when the product went through major changes such as moving from two to three tier and the introduction of extensions.

Her biggest passion is changing the world for the better in every way possible. One of her biggest achievements was the introduction of Design Patterns into the Business Central community. She is the go-to-girl when it comes to performance troubleshooting and upgrade challenges. Technologies such as the Azure stack and Web frameworks don't hold secrets in regards to Business Central. She wrote a number of books and countless blog articles and YouTube videos, which influenced almost every person and project involved with Business Central. At home, she enjoys the outdoors with her dog and tries to spend as much time as possible with her kids and family. She likes shopping and traveling.

Acknowledgement

This book is the result of working with Microsoft Dynamics NAV® for almost 20 years, and having had many discussions, debates, and even arguments with other people who are as passionate about the product as I am.

Being an early adopter of the software, I was very fortunate to be able to make many mistakes during my career and learn from them each time. Fundamental questions like variable naming, the user interface, and where to write the code, were all discussed many times, and have changed over the years.

Like many developers, I feel like a creative artist that needs to put his signature on his work. Unfortunately, this does not go well when writing solutions for Microsoft Dynamics NAV. When you adopt the style of the base application, you will find that collaboration becomes easier, and people will understand your efforts better than when you create your own style.

During the years that I have worked with the product, I've come to appreciate the structure of the base application. At a certain point in my career, I made an assumption that many of you might recognize. I started to treat what was done in the core product as "always correct".

In the past ten years, ever since I received my first MVP award, I have become closer to the development team and have been fascinated by the way the product was created and how issues are prioritized. I heard many stories from people who worked on the product in the early '90s, and started to realize that the core product is written by developers just like me and that many of the core components were acquired from partners who had their own development styles.

Over the years, I felt the core NAV product became "polluted", and the code was written in a less structured way, both by Microsoft and NAV partners. Between 2006 and 2010, my core business was performance troubleshooting and upgrades. The code that I saw during this period motivated me to write my first book about Application Design. This was my first effort to document the way Microsoft Dynamics NAV is designed, and how application areas work together.

Eric Wauters, who many of you know as Waldo, was a reviewer of this book, and his reaction was very positive. The book became mandatory reading for his team of developers and consultants. Together, we met Gary Winter at an event called Directions. I had worked with Gary many years before this, and remembered having conversations about designing components that can be reused. I had written a generic piece of software for weighing bridges, and I demonstrated this to him. After that event, Partner Ready Software was born—a think tank about Microsoft Dynamics NAV, trying to solve the bigger issues of life.

Design Patterns were first discussed at NAVTechDays. Many thanks to Luc Van Dyck for trusting the three of us, and giving us 90 minutes of presentation time without really knowing what we would do. At the same event, I also talked about tips and tricks for developers sharing ideas about variable naming and source control.

After NAVTechDays, we had the opportunity to present our ideas at MDCC in Denmark where our code product was being developed. The talks we had with Bardur Knudsen and Lars Hammer were inspiring. Lars was then working on DevBase, an internal prototype that we now know as the Delta file, using PowerShell.

With Partner Ready Software, we started to realize that we were reinventing the wheel. Many great minds had preceded us and the saying, "Great minds think alike" was spot on. As you will read in this book, we use many best practices from object-oriented programming, though C/AL is not object-oriented. We don't have inheritance or super classes, but with discipline and guidelines we can get very close.

Life became very interesting when the MVPs were introduced to a project that was happening internally at Microsoft. Apparently, the developers who were working on the core product were also asking questions about why things were done in a certain way, and if that way was good, why was it not documented? The Microsoft Design Patterns project was born, and I was first in line to participate.

The most that I learned about most in this project was how to define Patterns. How abstract should a Pattern be, and when does a Pattern become a functional description or a tip, or a "how do I" description?

I do believe that Patterns should be abstract. They relate very closely to the objects that we have in the product, and could be the properties of these objects and elements. The Patterns in this book have a hierarchy and dependencies.

I'd like to thank everyone in the Microsoft Design Patterns team for their efforts and hope that the project will continue for many years to come.

Special thanks to the two very important mentors that I've had in my professional career—David Studebaker and Michael Nielsen. Dave and his wife Karen are very special friends, and their knowledge and wisdom have been very important. Very often when I discuss an idea with Dave, he says things like, "Yes we did that in 1973 with Cobol." Michael has been my guardian angel at Microsoft for a decade. Being the most senior person on staff, it is very interesting to use him as a sounding board for ideas and vice versa.

I hope this book will inspire you, as it did me, to write software in a generic style that others will recognize and find easy to work with.

I'd also like to thank the reviewers as mentioned in the book, but also Dave Studebaker and Luc van Vugt, who unofficially helped me with this book. Also, thanks to Soren Klemmensen, Andrew Good, and Rick Williams for listening to me for many hours on Skype.

Last, but definitely not least, I want to thank my wife, Dionel, and my kids Josefien, Wesley, Saskia, and Daan. Thank you for allowing me to spend time writing these books, and travelling to the events where I can talk to others about this.

About the Reviewers

Alex Chow has been working with Microsoft Dynamics NAV, formerly Navision, since 1999. Over the years he has conducted hundreds of implementations across multiple industries. His customers range from $2 million-a-year small enterprises to $500 million-a-year multinational corporations.

Over the course of his Dynamics NAV career, Alex has often been designated as the primary person responsible for the success and failure of a Dynamics NAV implementation. The fact that he is still in the Dynamics NAV business means that he's been pretty lucky so far. His extensive career in the Dynamics NAV business is evidence of his success rate and expertise.

With his background in implementing all the functions and modules in and outside of Microsoft Dynamics NAV, Alex has encountered and resolved from the most practical to the most complex requirements and business rules. Through these experiences, he has learned that sometimes you have to be a little crazy to have a competitive edge.

Believing that sharing these experiences and knowledge would benefit the Dynamics NAV community, Alex writes about his journey at `http://www.dynamicsnavconsultant.com/`. He also founded AP Commerce, Inc. (`http://www.apcommerce.com/`) in 2005, which is a full service Dynamics NAV service center. In addition, he has written a book about Dynamics NAV titled *Getting Started with Dynamics NAV 2013 Application Development*, published by *Packt Publishing*

Alex lives in Southern California with his beautiful wife and two lovely daughters. He considers himself the luckiest man in the world.

Luc Van Dyck is active as a software consultant and works for a Belgian Microsoft partner. He started working with Microsoft Dynamics NAV in 1997 (at that time, it was called Navision Financials 1.10).

In 1999, he started the website `http://mynavision.net/` to provide a forum and downloads for users of the Dynamics NAV ERP system. When Microsoft bought Navision Software A/S in 2002, the site was renamed `http://mibuso.com/`.

Mibuso is one of the largest online communities for Microsoft Dynamics professionals. This community gives users and the developers of products from the Microsoft Dynamics family (NAV, AX, CRM, and so on) a place to exchange ideas and tools, and to find business partners and products. The website provides a forum where you can ask questions about the different Dynamics products, and also contains a large selection of downloads. Microsoft partners can submit their company details to the Business Directory, and publish their add-ons or factsheets in the Product Directory.

Since 2011, Luc has organizes a two-day technical conference called NAV TechDays. This annual conference takes place in Antwerp (Belgium), and is targeted towards all Microsoft Dynamics NAV developers.

Alain Krikilion has been working with Dynamics NAV (formerly "Navision Financials", "Navision Attain", and "Microsoft Business Solutions Navision") since 1999, mostly solving technical problems in both small local projects and big international projects. Besides specializing in NAV, he also specializes in optimizing Microsoft SQL Server for Dynamics NAV.

He started working with NAV in rainy Belgium, and in June 2002 he moved to sunnier Italy. In April 2015, he moved to less sunny London.

Alain is also a moderator of the `http://mibuso.com/` forum where, aside from moderating, he also helps to solve (mostly technical) problems. He also assists in the organization of the NAV Techdays event.

Alain Krikilion has been a Microsoft Dynamics NAV MVP (Most Valued Professional) since 2006.

Rakesh Raul is a Dynamics NAV consultant with more than 11 years of NAV and consulting experience. His 10 year journey with Microsoft Dynamics NAV includes more than 40 implementations—1 horizontal and 3 vertical solution designs and development.

Rakesh is also the author of another book in Packt's Dynamics NAV Series: *Microsoft Dynamics NAV 7 Programming Cookbook*, published by *Packt Publishing*. It is a collection of short, easy-to-read recipes or tutorials about writing the code for NAV.

Currently, Rakesh works in Cognatio Systems, Dubai-UAE, as a Team Lead. Cognatio Systems is a provider of Enterprise Solutions. The Microsoft Dynamics family for Enterprise Resource Planning (ERP) and CRM is the company's primary focus along with various other complementary products and solutions.

Daniel Rimmelzwaan was born and raised in The Netherlands, and moved to the USA at the end of 1999 to be with his new American wife. In Holland, he worked as a Microsoft Access and VBA developer. When looking for a job as a VB developer in the USA, he was introduced to Navision by a "VB Recruiter", and was intrigued by the simplicity of its development tools. He decided to accept the job offer to be a Navision Developer, with the firm intention of continuing to look for a "real" developer job.

More than 15 years later, with a couple of stints in the Microsoft partner channel and a few years as a freelancer, Daniel currently works as the chief quality officer for KCP Dynamics Group, an international partner servicing customers all over the world, and he is enjoying his career more than ever.

Ever since he started working with NAV, Daniel has been an active member of the online communities for NAV, such as `http://mibuso.com/`, `http://dynamicsuser.net/`, and the online forums managed by Microsoft. For his contributions to these online communities, he received his first of eleven consecutive Microsoft Most Valuable Professional Awards in July 2005, which was only the second year that the MVP Award was given out for NAV. The MVP Award is given out by Microsoft to the independent members of technology communities around the world, and recognizes people that share their knowledge with other members of the community.

Daniel lives with his wife and two kids in Arizona in the USA.

Bill Warnke has been working with Microsoft Dynamics NAV since 2006, when he was tasked with creating an integration between NAV and an existing mobile invoicing solution during an ERP migration. He was hooked on rapid development, all-in-one environment, and integration capabilities. Since then, he's worked exclusively with Dynamics NAV.

Currently, he works for ArcherPoint (`http://www.archerpoint.com/`) as a Dynamics NAV developer.

Bill has also worked as a reviewer for *Getting Started with Dynamics NAV 2013 Application Development*, published by *Packt Publishing*

He lives in Wisconsin with his wife, two young daughters, and his dog. You can read his blog at `http://billwarnke.com/`.

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit `www.PacktPub.com`.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



`https://www2.packtpub.com/books/subscription/packtlib`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Free access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

Instant updates on new Packt books

Get notified! Find out when new books are published by following `@PacktEnterprise` on Twitter or the *Packt Enterprise* Facebook page.

Table of Contents

Preface **ix**

Chapter 1: Introducing Dynamics NAV Patterns and Software Architecture **1**

What do we mean by 'patterns'? **2**
Software design patterns 2
The Procedural software design patterns 2
Microsoft Dynamics NAV Design Patterns **3**
The object-oriented patterns 3
Non-object-oriented patterns 3
The functional oriented patterns 4
Patterns elements and hierarchy **4**
Architectural patterns 4
Design patterns 4
Structuring code 4
The coding standards 5
Anti-patterns 5
Code cloning 5
Legacy code 5
Upgradability 6
The upgrade frequency 7
Design patterns and upgrades 8
Delta files 8
Repeatability 8
Design patterns and repeatability 8
Lifecycle of a design pattern 9
Technology changes 9
Old habits die hard 10
New design patterns 10

Software architecture in Microsoft Dynamics NAV 11
Metadata 11
Building blocks 11
Table 12
Page 12
Report 13
Codeunit 13
Query 13
XMLPort 13
MenuSuite 14
Metadata 14
Flowfields 14
Flowfilters 15
Base application 15
Functional processes 15
Summary 16
Chapter 2: Architectural Patterns 17
The Singleton Pattern 18
Technical description 18
Company-Initialize 19
Implementations 19
Examples 19
Enhancements 19
The Supplemental and Subsidiary Pattern 20
Technical description 20
Implementations 21
Examples 21
The Compound Pattern 21
Technical description 21
Implementation 22
Examples 22
The Rules Pattern 22
Technical description 22
Implementations 23
Examples 23
The Master Data Pattern 24
Technical description 24
Implementations 25
Examples 26

The Journal Template-Batch-Line Pattern **26**
Technical description 26
The Journal Template table 28
The Journal Batch table 28
The Journal Line table 28
Codeunits 29
Implementations 30
Hierarchy 30
Journals as contracts 31
Examples 31
The Entry Pattern **31**
Technical description 32
The Entry Table types 32
Historical entries 33
Temporary entries 33
Entries that can be deleted 34
Temporary tracking buckets 34
Implementations 35
Examples 35
Document Pattern **36**
Technical description 36
The Header table 36
The Line table 37
Posting and archiving 37
Implementations 38
Examples 38
The Argument table **38**
Technical description 38
Implementations 39
Summary **39**
Chapter 3: Design Patterns **41**
The Entity State Pattern **41**
Technical description 42
Setting the State 42
Testing the State 43
Implementations 43
Examples 44
The Select Pattern **44**
Technical description 44
Implementations 45
Examples 45

The Number Series Pattern **45**
Technical description 46
The Table reference field 46
Setup reference 46
C/AL functions and variables 47
Implementations 48
Examples 48
Address Integration Pattern **48**
Technical description 49
Fields 49
Copying the address 50
Printing addresses 51
Implementations 52
Examples 52
Multi-Level Setup **52**
Technical description 53
Implementations 54
Examples 54
Document History **54**
Technical description 54
The naming conventions 54
Codeunit Structure 55
Test Near, Test Far 55
Do it 55
Clean up 55
An Example 56
TRANSFERFIELDS 56
Implementations 57
Examples 57
Copy Document Pattern **57**
Technical description 58
Validation logic 58
Implementations 58
Examples 59
Job Queue Entry Pattern **59**
Technical description 59
Preprocessing 59
Getting records 60
The Main function 60
Post-processing 60
Error handling 60
Implementations 61
Examples 61

Dismissible dialogs and save preferences **61**
Technical description 62
Dismissible dialog 62
Implementations 62
Examples 63
Setup Update Pattern **63**
Technical description 63
Restrictions 64
Implementations 65
Example 65
Model-View ViewModel Pattern **65**
Technical description 65
Implementations 66
Examples 66
Summary **66**
Chapter 4: Building an Example Application Using Patterns **67**
The design elements **67**
The Pattern dependencies **67**
Step one – Foundation **68**
Example setup 68
Which Patterns to use 68
The implementation 68
The Multi-Level Setup Pattern 71
Step two – Master Data **72**
Example Person 72
The Example Product 74
Number Series 74
Adding Setup fields 75
Adding variables 76
Adding the No. Series field 76
Code snippets 77
Entity State 78
Implementing the Address Integration Pattern 79
Step three – Journals & Entries **81**
Creating your own Journal and Entry 81
Validation in Journals 84
Step four – Documents **85**
Implementing a Document 85
Document history 87
Selecting a Pattern 88
Other Patterns 88

Posting 88
The ThrowErrorIfNoSelection function 89
The TestNear function 89
The TestFar function 89
The PostExampleOne/PostExampleTwo function 89
The PostExJnlLine function 90
The DeleteExampeDocument function 90
The layered model 90
Summary 92
Chapter 5: Coding Best Practices 93
Coding guidelines 93
Code design 94
Variable capacity 96
The Cyclomatic Complexity 96
The function size 96
Using complex data types 97
Localizability 97
Date formulas 98
Code readability 98
Variable naming 100
Self-explaining 101
Suffix versus prefix 101
Approved abbreviations 102
Class-Method-Property 102
The Table as a class 102
A Codeunit as a method 104
Function libraries and interfaces 106
Natural Language Programming 107
Pitfalls 112
Bonus – debugging 112
Encapsulation 113
Override Functions 116
Façade 116
Hooks 118
Disabling the original source code 119
Predefined hooks 120
Variant variables 121
Managing DLL 121
Summary 122
Chapter 6: Anti-patterns and Handling Legacy Code 123
Anti-patterns 124
Code cloning 124
Forking 124

Templating 124
Customizing 125
Exact match 125
Examples 125
Transaction mirroring 125
Code normalization and writing clean code 126
The Document type normalization 127
The small business application 127
The VAT and Sales Tax calculation 130
Avoiding code cloning 131
Boat anchors 132
Other anti-patterns and context 134
Programming language capabilities 135
Legacy code 135
Refactoring the C/AL Code 136
Refactoring tables 136
Renaming 137
Normalization 138
Code refactoring 139
Finding the process owner 139
Moving to Codeunit 139
Breaking down the code into functions 139
Encapsulating functions 139
Normalizing the code 140
The boy scout rule 140
Summary 140
Chapter 7: Building Solutions Using Patterns 141
Building blocks 142
Applying Architectural Patterns 142
The application setup 143
Guests 143
Room 143
Season 143
Price 143
Reservation 144
Stay 144
Check-in 144
Check-out 144
Clean 144
Evaluation 144
Invoice 145
Creating the Table objects 145
Applying Design Patterns 145
The Bed and Breakfast setup 146
Bed and Breakfast season 146

Bed and Breakfast room 147
Bed and Breakfast price 148
Bed and Breakfast clean entry 148
Bed and Breakfast stay 149
Bed and Breakfast journal and entry 150
The Bed and Breakfast evaluation header and evaluation line 150
Defining the methods 151
Writing the Code and Link methods 152
Creating the cleaning entries 152
Refactoring 153
The new class members 154
The new function library 154
Room members 155
The room price calculation 155
Testing the application 157
Test isolation 158
Analyzing the results 159
Upgrading the application 159
Creating a Delta file 160
The steps of creating a Delta file 160
Inside the Delta file 162
Building our application 163
Building the result 165
Managing the source code 166
Creating the workspace 166
Setting up Visual Studio 167
Uploading the files 168
Summary 171
Index 173

Preface

Hi, and welcome to the world of Design Patterns. This is how I open my short introductory video about this subject that I made for Microsoft. You can find it on my YouTube page together with more than 15 hours of Patterns content.

This book is about Patterns and clean coding practices. It is about creating software that users will recognize as Microsoft Dynamics NAV, that other developers can maintain, and that is as easy to merge and upgrade as possible with the technology we have today.

Microsoft Dynamics NAV, and before that, Navision Software, always had a specific footprint. I often call it Navision DNA. It is what makes our beloved product unique and successful.

But what is "it"? What is this Navision DNA? This is the question that we will try to answer in this book. You will find that many answers can be found in the product, dating back to 30 years ago. We will also find some answers in the object-oriented programming best practices and the clean coding concepts.

Together, these form a recipe for writing good software in C/AL—our programming language. Despite the fact that we are not object oriented, we can learn from other initiatives around Patterns and practices.

After reading this book, I hope that you have some fresh ideas, or maybe even recognition that you're already doing a lot of these things in the correct way.

What this book covers

Chapter 1, *Introducing Dynamics NAV Patterns and Software Architecture*, will introduce you to the concept of good software architecture and Design Patterns. Dive into the history and learn some characteristics that are specific to our product, which makes our Patterns unique.

Chapter 2, *Architectural Patterns*, will discuss various ways of designing the tables that are used in the product and have proven to be successful. You will find that up to 99 percent of all the tables can be easily categorized. By following these Patterns, your software will be recognized and proven to be working.

Chapter 3, *Design Patterns*, will discuss some of the most popular and well-known software elements that you can reuse to solve problems, in other words, Design Patterns!

Chapter 4, *Building an Example Application using Patterns*, will show you what happens if we combine the Patterns from *Chapter 2*, *Architectural Patterns*, and *Chapter 3*, *Design Patterns*, together in a solution. We should have something that makes sense and looks like a Microsoft Dynamics NAV application.

Chapter 5, *Coding Best Practices*, will explain that in Microsoft Dynamics NAV we can write code almost anywhere, but should we do this just because we can? Let's learn how to structure our code. We will steal some ideas from object-oriented programming while we're at it.

Chapter 6, *Anti-patterns and Handling Legacy Code*, will show you how you can migrate the existing legacy code into more structured software. We will touch base on refactoring and clean code.

Chapter 7, *Building Solutions Using Patterns*, will help you write an application to run a Bed and Breakfast. We will learn how to divide processes into classes, and select which Patterns to apply. At the end of this chapter, we will write test code and upgrade our software to a new version.

What you need for this book

The Patterns and coding techniques you learn in this book can be applied to any version of Microsoft Dynamics NAV, back to Navision Financials 1.0. The examples that you can download to use with the book were designed in NAV 2015, but should work in NAV2013 and NAV2016 as well.



To work with the objects, you need a valid development license.

The examples are optimized for the worldwide version (W1), but the Delta files can be applied to any localized country version.

Who this book is for

This book is written for programmers, architects, and power users. Knowledge of Design Patterns is not required, and when helpful the book contains hyperlinks to valuable resources.

The book is especially useful for experienced developers who find learning the language easy, but need more time and guidance to find best practices.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "The latter is a new option in Dynamics NAV 2015 where we can for example do `GLAccount.SETCURRENTKEY(Balance)`."

A block of code is set as follows:

```
OnRun(VAR Rec : Record "Table")
Table.COPY(Rec);
Code;
Rec := Table;
Code()
```

New terms and **important** words are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "The screen capture shows the **Object Designer** that allows development access to the main building blocks."



Warnings or important notes appear in a box like this.



Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail `feedback@packtpub.com`, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files from your account at `http://www.packtpub.com` for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from `http://www.packtpub.com/sites/default/files/downloads/1234OT_ColorImages.pdf`.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to `https://www.packtpub.com/books/content/support` and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at `questions@packtpub.com`, and we will do our best to address the problem.

1
Introducing Dynamics NAV Patterns and Software Architecture

"Design patterns have always been the "secret source" of NAV, enabling developers to learn new areas of the application and be productive without a huge ramp up. This makes design patterns key to understanding the architecture of NAV and writing repeatable code."

Michael Nielsen, Director of NAV Development

In this book, we will journey into how to build repeatable and upgradable solutions for **Microsoft Dynamics NAV**. We will do this using a combination of design patterns and coding best practices. We will also cover anti-patterns, and the methodologies of handling the legacy code.

By improving the repeatability and upgradability of your software, the **total cost of ownership** (**TCO**) is reduced, which leads to better margins and better **return on investment** (**ROI**).

The examples and screenshots are based on the current version of the product, Microsoft Dynamics NAV 2015, but all of the concepts can be applied to any older version of the product too.

In this chapter, we will cover:

- Introduction to Software Design Patterns
- Pattern elements and hierarchy

- The Dynamics NAV building blocks
- The Metadata driven development

What do we mean by 'patterns'?

Wherever you look in life, you see patterns. Even life itself follows a pattern. Patterns help us in being organized by giving us structure, which makes things easier for us to understand. Without structure, we start being stressed and disorganized.

As this is true for many things in our lives, this is also true in computer programming. Applying structure in our computer programs make them easier to understand and maintain. Each profession has patterns that have proven to be successful in certain conditions, and unsuccessful in other conditions.

Software design patterns

Good old Wikipedia defines software design patterns succinctly as "*A general reusable solution to a commonly occurring problem within a given context in software design*." By nature, people look for ways to get organized and structured. It is not a surprise that during the 1970s and 1980s people started to look for structures in computer programming, which was a new, fast growing business.

You can read more about the software Design Patterns on Wikipedia at `https://en.wikipedia.org/wiki/Software_design_pattern`. Software design patterns, especially in **object-oriented programming** (**OOP**), gained popularity in the 1990s when after *Design Patterns: Elements of Reusable Object-Oriented Software* was published by the *Gang of Four*, *Erich Gamma*, *Chard Helm*, *Ralph Johnson*, and *John Vlissides*. They published 23 classic software design patterns. Since then, many other patterns have been documented. Many of these were based on object-oriented programming, but people also started to document patterns outside this context.

The Procedural software design patterns

The product that we work with, Microsoft Dynamics NAV, has a programming language that people refer to as **Procedural**. It was invented in the 1980s and is based on Pascal, which is a very popular programming language that is still loved by many.



In this article, Paul Ford explains the history of code, procedural languages, and object oriented programming at `http://www.bloomberg.com/graphics/2015-paul-ford-what-is-code/#b06g24t20w15`.

As with everything in life, we seek structure, hence we can apply software design patterns to Procedural languages. We just need to look carefully at all the object-oriented content that is available, and cherry-pick what applies. We can then complete this content with specifics of our language.

Procedural languages preceded the more modern object-oriented languages. One of the challenges that object-oriented programming is trying to solve is grouping things together that belong together. For this, **classes** are used. In this book, we will attempt to do this as well by treating tables and **codeunits** as a class, and then grouping functions together accordingly.

Microsoft Dynamics NAV Design Patterns

The base application that is shipped with Microsoft Dynamics NAV is full of design patterns, general **reusable** solutions to a commonly occurring problem within a given context in software design. Using these patterns when applicable, not only do you use solutions that have proven to work, but you also create software that people recognize as Dynamics NAV, since it will probably behave the same as the standard software.

The object-oriented patterns

The most well-known object-oriented patterns are those that are documented in the book called *Elements of Reusable Object-Oriented Software*, which we mentioned earlier in this chapter. Examples of these patterns are **Builder**, **Prototype**, **Bridge**, **Singleton**, **Façade**, and **Adapter**. We reuse some of these names in this book when they resemble the original pattern. Another object-oriented pattern is the **Model-View-ViewModel** (**MVVM**) pattern, which also applies to Dynamics NAV.

Non-object-oriented patterns

The base design patterns from object-oriented programming are pretty much targeted at structuring applications, running as Windows services or applications. With Microsoft Dynamics NAV, we get all of that out of the box. We don't have to think of how to get the data from the database, or how to handle concurrency.



For a better understanding of object-oriented patterns and why they don't apply to Dynamics NAV and **C/AL**, please read and watch the information at `http://www.newthinktank.com/2012/08/design-patterns-video-tutorial/`.

The functional oriented patterns

The patterns that we use in Dynamics NAV typically are functionally oriented. They are sometimes even focused on the typical ERP problems, such as storage and formatting of address information. They still solve commonly occurring problems.

Patterns elements and hierarchy

Many books about patterns use different ways to categorise them. Patterns are closely related to the application building blocks, such as tables, fields, functions, and codeunits. Patterns will help us categorize these elements, so that we can learn different table types, and codeunit types.

Architectural patterns

When a pattern describes table definitions or a methodology to move data from one set of tables to another, we refer to it as an **Architectural Pattern**.

By learning Architectural patterns, we understand how the application's large building blocks are constructed. There are only a dozen of these patterns that are most commonly used. Examples are Master Data, Journals, and Rules tables.

This does not mean that any other way of designing tables and moving data is wrong by definition. It might just not be a common way of doing it, making your code harder to understand for others, and less easy to maintain.

Design patterns

Any pattern that is not related to the table definitions, or moving data in the database, is a **Design Pattern**. There are dozens of patterns, and in this book we'll only focus on the ones that are most commonly used, such as number series and entity state.

Structuring code

Even if you follow the patterns in this book step-by-step, your solution will be hard to maintain if you don't bring **structure** to your code. There are standard ways and best practices for doing this. You can find a lot of them online, mostly related to object-oriented programming, but also outside of object-oriented programming.

We can translate these best practices so that they can work with C/AL. This will help us implement best practices in Dynamics NAV, which are widely accepted in computer science.

The coding standards

The readability of your code depends in large part on how you handle **variable naming** and code punctuation.

For Microsoft Dynamics NAV, a wide range of documented standards are available that we will discuss in detail. Following this standard will enable other developers to read your code with ease, and seamlessly integrate your code into standard code. Remember, Microsoft Dynamics NAV offers access to the entire source of the application, so it makes sense to follow standards that are already applied throughout the standard code.

This makes it easier to **clone** (copy) parts of the application.

Anti-patterns

Where there are best practices, there can also be ways that are proven to be less productive. These are referred to as anti-patterns. Anti-patterns are bound to context, which means that a good pattern applied to the wrong problem can be an anti-pattern.

Code cloning

A typical example of an anti-pattern in computer programming is code cloning. This means copying and pasting similar code in your application rather than normalizing it into classes and functions. Code cloning is applied in Dynamics NAV in large proportions.

When working with C/AL, being able to copy a part of the application and making it specific to certain needs is considered powerful.

This does not yet make code cloning a good thing. We will explain good and bad examples of code cloning, and help you with a set of criteria about when cloning can be applied, and when it should be avoided.

There are many good and bad examples available in the standard application. We will discuss them in *Chapter 6*, *Anti-patterns and Handling Legacy Code*.

Legacy code

The Microsoft Dynamics NAV application code started being developed in 1980s, and grew organically.

Despite the fact that the original architect applied many patterns, we still know today from the very early days we can safely say that many parts of the application are not optimized in a way they could be if one would have the opportunity to start over.

Handling legacy code is highly important for the lifecycle of your application. Thousands of companies have been using Dynamics NAV for more than two decades, upgrading their solution every few years to the latest version.

To provide this upgrade opportunity, one should be careful when redesigning the application.



This makes it even more important to start with the correct patterns from the beginning.

Especially when dealing with limited resources, a situation that most Dynamics NAV development centers face, we want to spend as much time as possible in making new features instead of fixing issues or rewriting existing code.

We will discuss examples of how the legacy code in Microsoft Dynamics NAV was preserved, and how and when it was updated when necessary.

Upgradability

Microsoft Dynamics NAV has a great track record of being upgradable from one version to the next. It is possible to upgrade all the way from the earliest MS-DOS versions of the product to the latest version running on Azure in the cloud.

This requires the application architects to be very careful in making decisions. Changes made to the application may live for many years or even decades.

When upgrading to newer versions, there are two big challenges.

- **Code Upgrade**: each time Microsoft ships a new version of the product, we have to see if the changes that we made to their code still function as intended. This requires not only merging code and testing of the application, but also doing a fit/gap analysis each time new functionality is added to the base application.
- **Data Upgrade**: if there are changes in the way the data is stored in the table objects, we have to provide software programs that convert the data from the old structure to the new structure. This is referred to as the **Upgrade Toolkit** for Microsoft Dynamics NAV. You will learn about this in *Chapter 6, Anti-patterns and Handling Legacy Code*.

The upgrade frequency

Traditionally, customers used to upgrade the software every few years. The initial version is the active release when the implementation of a project starts. After *go-live* of the project, a system would typically not be touched, most often because of the upgrade cost.

Microsoft used to ship major releases of their software every few years. For customers, it was considered best practice to upgrade every other major version, depending on the return on investment.

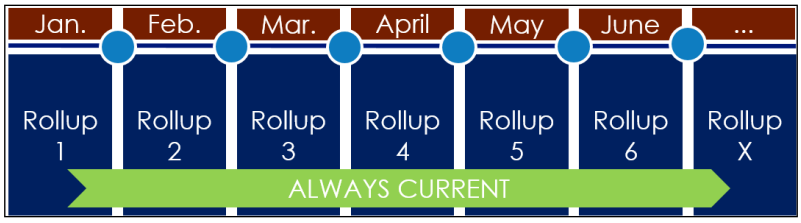
This image illustrates this practice:



However, this best practice of upgrading has shifted dramatically with the introduction of a connected world, internet, and cloud. There is an increasing demand for being up to date with application software, and always being current.

New technologies are introduced in a higher frequency than ever before, and this has impacted the way Microsoft ships the Dynamics NAV product.

We changed to yearly release cycles and frequent rollups, which often contained new features, and new technology. Instead of upgrading every few years, there is a requirement to be (almost) always current on version, as illustrated in the next image:



This methodology is especially important when running in a **Multi-Tenant** environment, where individual customers no longer have a decision in their upgrading strategy.

Design patterns and upgrades

The cost of making the upgrades can be reduced by avoiding conflicts, and/or by automatically resolving conflicts. Implementing design patterns and coding best practices can help us achieve this and make upgrading easier.

Delta files

Another method to reduce the costs of shipping and merging software is using the delta files, which were introduced in 2014 by Microsoft.

Delta files are based on flat text definitions of Dynamics NAV objects, and are handled by PowerShell commandlets. The algorithms can create and apply text files on all the versions of Dynamics NAV—from 2009 and forward.

The Delta file describes, as the name implies, the delta between two objects in such a way that they can be applied to other systems quite easily. This saves time and money in merging the software.

Repeatability

In a cloud-first mobile-first world, repeatability becomes increasingly important. Where barely 20 years ago consultants were driving around in their cars to customers to install software and explain its use, there is an increasing trend towards click-try-buy ERP systems.

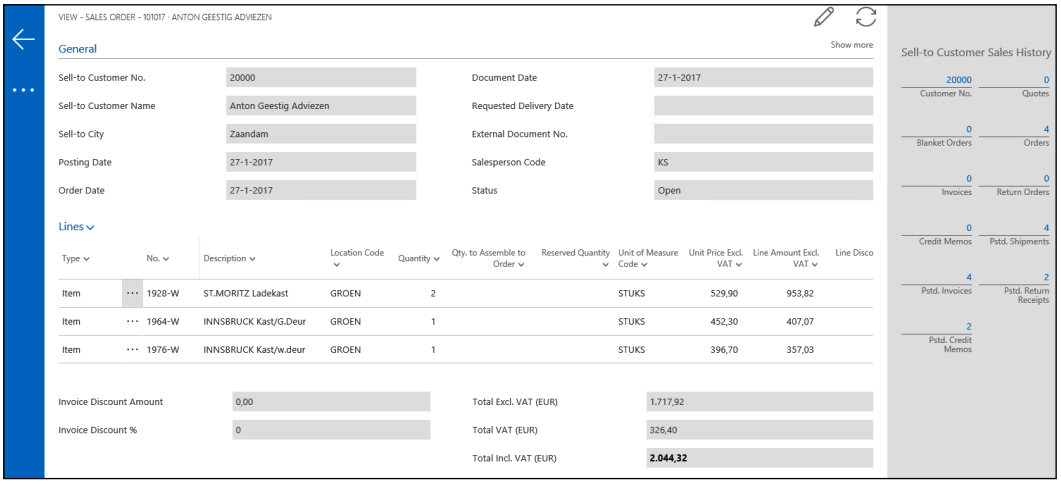
In this ecosystem, intuitive use is very important. Microsoft has invested in a user interface that makes it easier to use Dynamics NAV from a metadata perspective. However, to be really more user friendly, the metadata (UI) needs to be optimized too.

Design patterns and repeatability

Repeatable UI has been documented in Design Patterns. This makes it easy for developers to implement this in their own applications, and also makes the software work similarly across the application.

We differentiate between the classic and modern UI patterns. With multiple display targets and new devices, such as tablets being added to the product in an agile way, the UI patterns are a moving target and subject to change, version-by-version. An example of an updated pattern is statistics, wherein the classic UI totals were hidden; they are now visible on documents.

The following image illustrates an example of a modern UI on a tablet:



Lifecycle of a design pattern

Although design patterns are powerful when applied correctly, they can have negative side effects when applied in a situation where they should not be used. When choosing a design pattern to be used, it is important to see if it matches the requirements.

For example, when we need a table to store setup data, we might think of the Singleton pattern, but this only allows us to store a single set of data. If we need multiple sets, we might want to implement the Rules pattern. Both the patterns are documented in this book.

Technology changes

Design patterns can be made obsolete due to technology changes. Some languages have more built-in capabilities than others.

For example, C# has overloading, where the same function has different arguments. This is something that Dynamics NAV does not have. Therefore, we have a design pattern called the **Argument table**. This would be obsolete if we could overrule a function from the design time, or based on business logic.

The patterns in this book have been tested and documented for Microsoft Dynamics NAV 2015, but typically work in older versions of the product too.

Some of the design patterns in Dynamics NAV go as far back as the MS-DOS version of the product.

Old habits die hard

With technology changing, we should also question ourselves each time we design software, if we are still using the most optimal ways.

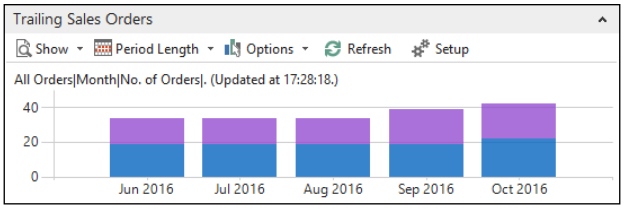
A typical example is this structure that we see a lot in the Dynamics NAV product; for example, in codeunit 81 Sales-Post (Yes/No):

```
OnRun(VAR Rec : Record "Table")
Table.COPY(Rec);
Code;
Rec := Table;
Code()
```

This code pattern started its life in the MS-DOS days, and just stayed in the application without being technically required. Developers apply it to their own objects just because it exists in many places in the standard application. We'll discuss alternative approaches in this book.

New design patterns

Sometimes, new technologies are introduced, such as **charting**. This then requires the new patterns that did not exist before. The following image is an example of a chart:



Other examples are patterns based on Query objects, and patterns for Activity Pages. Neither of these existed in the older versions, hence there was no requirement for Design Patterns.

Sometimes, we can reuse the existing patterns for new elements. The Activity Page uses the Singleton Pattern that we typically see for the Setup data. This adds new functional implementations for patterns, and the pattern documentation needs to be updated.

Developing SolutionsWriting software can, in many ways, be compared to building houses, or even building entire villages or cities. We need to plan ahead, but we also need to adjust to time. Things that we thought were brilliant in the 1990's are now considered old. Also with software, we need to maintain, rebuild, and rearrange as time goes by.

Software architecture in Microsoft Dynamics NAV

One thing that we need to consider when working with Microsoft Dynamics NAV is that both the metadata, and the base application deliver the base architecture for us. Unlike working in other development environments, programmers in Dynamics NAV seldom start from scratch.

Metadata

Much of the program's behavior is determined by **metadata,** which is interpreted at runtime.

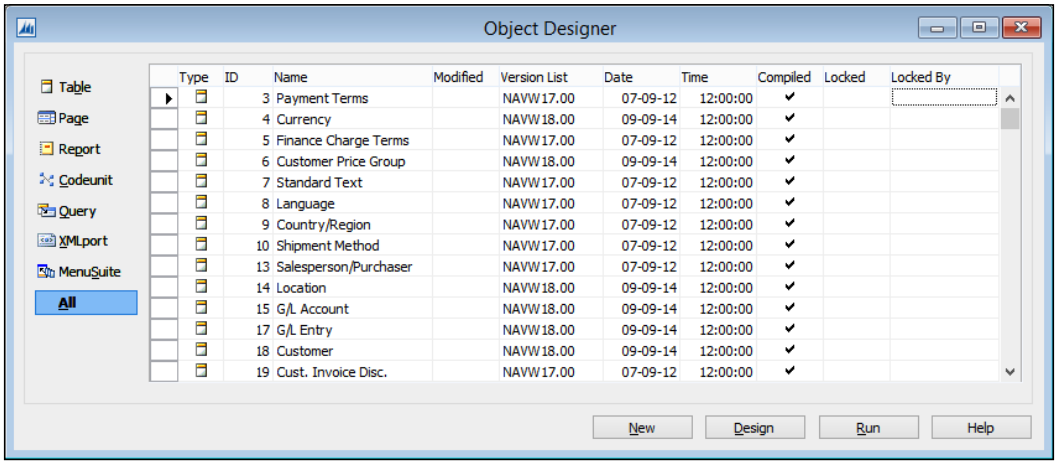
The metadata is a combination of elements, methods, and properties that gives the program its unique behavior. This also limits the possibilities that developers have using the application that enforces a certain level of simplicity.

Understanding the Metadata and base application is core for achieving success in developing your own application. The patterns and best practices in this book will help you with this.

Building blocks

To develop solutions in Microsoft Dynamics NAV, we have seven main building blocks. These blocks determine the behavior of our application together with the programming code.

The following screenshot shows the **Object Designer** that allows the development access to the main building blocks:



Let's go over these one by one.

Table

The Table object allows us to define how the data is stored in the database, as well as in business logic.

The only database option for Microsoft Dynamics NAV is Microsoft SQL Server. The SQL Table definitions are automatically created, based on the Table object in Dynamics NAV.

All the business logic is handled by the Dynamics NAV runtime. The SQL Server options, such as foreign keys, triggers, or stored procedures are not used.

Unlike many other applications, Dynamics NAV is not fully **normalized**, given a few exceptions. Tables are bound one-to-one with the user interface. This adds a certain level of simplicity to the design that makes it easy to understand and work with. It also allows us to store historical information in a relatively simple way. For example, the customer address is copied to the posted invoice, as it was the time the invoice was printed.



This way of designing tables is crucial to understand when we deep dive into the design patterns in the later part of this book.

Page

The user interface of Microsoft Dynamics NAV is defined using the **Page** objects. This is done by a combination of binding a Page object to a Table, and adding metadata.



The designer is not **what you see is what you get** (**WYSIWYG**) as we design for multiple display targets using one object definition. This makes defining the UI a highly abstract task.

The Page object inherits all the **properties** and **methods** from the underlying table. Developers have the option of adding additional properties and methods, as well as business logic that is specific to the behavior of the object.

All the logic to **Create**, **Read**, **Update,** and **Delete** (**CRUD**) from the database is automatically generated by the system. This makes it very easy to work with, but also enforces the relationship between a Page and Table.

We can overrule this by implementing the Model-View-ViewModel Pattern, which we will discuss later in *Chapter 3*, *Architectural Patterns*. This pattern allows us to have a UI, which is unbound to the way the data is stored in the SQL Server database.

Report

In Microsoft Dynamics NAV, the **Report** objects are used for both: printing documents, and creating batch routines for data modification.

Reports are typically used to print business documents such as invoices, reminders, and balance due lists.

In addition to printing, they act as containers for batch processes, such as combined invoicing and batch posting. The reasons for using reports instead of codeunits are the possibilities of adding the UI (request page), and the built-in iteration capabilities.

This object type is outside of the scope of this book, although we occasionally mention it when describing patterns.

Codeunit

When writing business logic, Codeunits are the preferred place to do this. They act as containers for the code that allow us to structure our programming using known concepts such as **encapsulation**.

In this book, we will discuss different types of codeunits that are used as part of the design patterns, as well as best practices to structure our code.

We will discuss how, and why codeunits should be used for code in *Chapter 5*, *Coding Best Practices*.

Query

Most business logic in Microsoft Dynamics NAV is bound to the table object, since the table behaves as a class with methods and properties.

However, when using complex iterations over many data items, it can be costly to loop over them one-by-one.

To solve this, we can define **Query** objects that act as predefined SQL Server queries, allowing us to join multiple tables in one read and only read the columns that we need.

XMLPort

To interface with Microsoft Dynamics NAV, we can use **XMLPort** objects. They are outside the scope of this book.

MenuSuite

The **MenuSuite** object helps us to search the application. They are outside the scope of this book.



To learn more about these object types and what you can do with them, please read *Programming Microsoft NAV 2015*, by *David Studebaker*, published by *Packt Publishing*.

Metadata

Table objects behave as classes in Microsoft Dynamics NAV, and are the base of our application. They are bound to tables in the SQL Server, and need to contain all the fields that will be used in the Page objects.

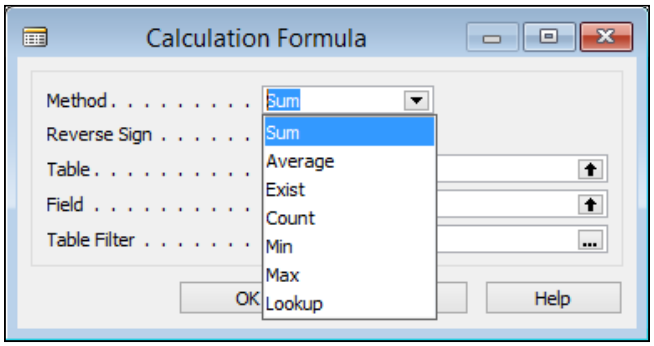
This does not mean that we cannot add references to other tables in a table object. We can reference to other tables using **Flowfields** and **Flowfilters** objects.

Flowfields

When we define a field in a Table, it gets added to the SQL Server table definition automatically, unless we change the **FieldClass** property. Flowfields and Flowfilters objects are only a part of the Dynamics NAV Metadata. They do not exist as a part of the SQL Server table definition.

If we change the property, we can join fields from other tables that we can display on all the Page objects, since they are treated as normal fields. We can also use them in C/AL code for filtering and sorting. The latter is a new option in Dynamics NAV 2015 where we can, for example, do `GLAccount.SETCURRENTKEY(Balance)`. This used to be impossible.

The following screenshot shows the property in the Object Designer:



Flowfilters

When using **Flowfields**, you can use any field in the table as a `where` clause. However, sometimes you might need values that are set by users at runtime. This is what **Flowfilters** are used for. They are defined as fields in the table, but are not added
to the SQL Table definition, or available as filter or sorting criteria.

Base application

Most development projects in Microsoft Dynamics NAV don't start from scratch as empty projects. They often start from the base application that Microsoft ships. This is an ERP application that allows their users to streamline business processes.

The application originated in the 1980s just as the programming language did, and has grown organically since its first introduction.

To be an excellent Microsoft Dynamics NAV developer, understanding business processes is maybe even more crucial than understanding the language, objects, and design patterns.

Functional processes

Microsoft Dynamics NAV allows you to administer many of the functional processes. Although they all have different purposes, such as general ledger, inventory, sales, purchasing, jobs, and manufacturing, they all use the same elements that make the application recognizable for the end user.

If a user is able to post something to the general ledger, it is very likely that they can easily understand posting to the inventory. Posting sales and purchase documents have the same similarities.

This is possible by using the same structures across the application. This also makes it easier for developers to learn new parts of the application.

Many of these structures have been in the application for a long time, since they are very closely related to ERP programming. Others have been added more recently, since they are more related to intuitive UI in correlation with app-like programming.



To learn more about these functional areas and what you can do with them, please read the *Microsoft NAV 2013 Application Design* book by *Marije Brummel*, published by *Packt Publishing*.

Summary

In this chapter, we have learned what design patterns are, and how they started and were introduced in computer science. We learned how design patterns can be applied to Microsoft Dynamics NAV. We also learned about anti-patterns.

In the next chapter, we will build a small application in Microsoft Dynamics NAV that consists of different design patterns brought together.

2
Architectural Patterns

Architectural Patterns are the larger building blocks of your application. They define how the data is stored in the database.

Microsoft Dynamics NAV uses the SQL Server as a database platform. Within the SQL Server, there are a large number of ways to store data. This chapter will discuss a number of ways that are best practice, which will be recognized by the experienced developers.

The members of the Architectural Patterns are the Design Patterns that typically contain field elements and pieces of business logic. Other parts of the business logic are provided by the internal API's generic containers with reusable functions.

All the patterns are described with an abstract, technical description, and with a few examples of the functional usages in the Microsoft Dynamics NAV application.

In this chapter, we will cover:

- Singleton
- Supplemental & Subsidiary
- Compound
- Rules
- Master Data
- Journal Template-Batch-Line
- Entries
- Document
- Argument

The Singleton Pattern

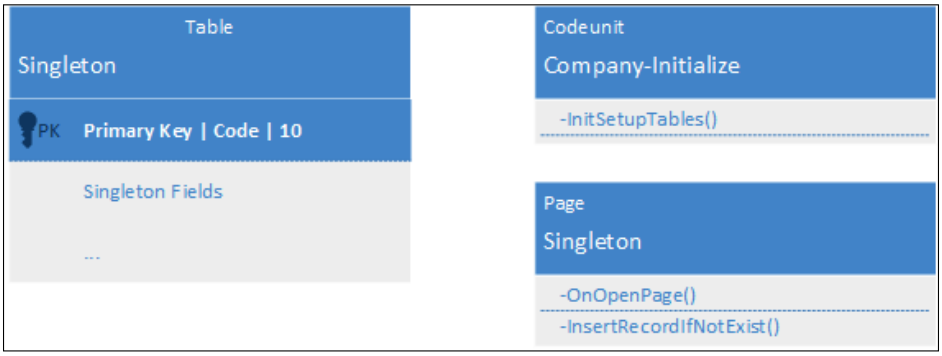
Within the EARP application, we need to store information that is shared across the system for all the users. These values are unique in the application, and when changed they are valid immediately for everyone inside a company. They can be held in memory, and are only acquired from the database when changed. For this requirement, we can implement the **Singleton Pattern**.

Technical description

The Singleton pattern contains a table object with a specific primary key structure that is enforced by the system, allowing the table object to contain only one record. Users cannot add or remove records. When a new company is created, a record in each **Singleton table** is created automatically, although this is optional.

To enforce the existence of a record when it is required, each Page object that is based on the table should contain business logic to create the record, in case it does not exist yet. The reasons for the record to be missing can be that the table was a part of an update, acquired after the company was created.

The following diagram explains how to implement the Singleton Pattern:



Since the Page objects, by default, allow users to create and delete records in tables, this should be blocked by setting the **InsertAllowed** and **DeleteAllowed** properties to `No` on the page.

The table object always has a field named `Primary Key` as a single Primary Key. This field has `ID 1`, and `Code 10`. The value of the key field is blank.

Company-Initialize

One of the oldest Codeunit objects in the Microsoft Dynamics NAV Application is **Company-Initialize**. This **Codeunit** has `ID 2`.

Each time a new company is created **Codeunit** is executed automatically, and it ensures the creation of many Singleton tables. When creating a new Singleton table, you can consider adding this to the **Codeunit**, although this is not mandatory.



When making changes directly in **Codeunit 2**, you might create possible merge conflicts. Consider using the **Hooks pattern** to avoid this conflict. This pattern will be discussed in *Chapter 5*, *Coding Best Practices*.

Implementations

The Singleton Pattern is used for Single Record Setup Tables and Role Center Activity Pages.

Any table that forms a functional perspective, which only requires one record, is eligible for this Pattern, such as `Company Information`.

Examples

In the standard Microsoft Dynamics NAV application, the examples of the usage are as follows:

Table ID	Description
79	The Company Information
98	The General Ledger Setup
242	The Source Code Setup
311	Sales & Receivables Setup
312	Purchases & Payables Setup
1313	The Mini Activities Cue
9050	The Warehouse Basic Cue

Enhancements

A possible enhancement to this pattern would be in providing the default setup rules when generating records in the database. This should be functioning as a member of the table. The function can be called from the Codeunit Company-Initialize, or from the Page UI.

The Supplemental and Subsidiary Pattern

For Microsoft Dynamics NAV to run properly, we also require tables that contain more records to allow multiple values to be used in various scenarios. These values are almost as static as the Singleton tables, although the new records can be created during the lifecycle of the application.

The pattern for these tables is **Supplemental** & **Subsidiary**.

Technical description

The Supplemental & Subsidiary table has a single field Primary Key that is typically called Code; it has the Data type code, and has a length of 10 characters. Values in the Primary Key are manually determined by the application administrators. Normal users of the system have read-only access to these tables.

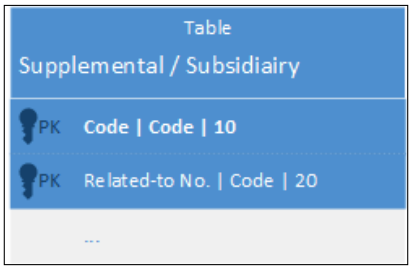
When the attributes in the table relate to another entity, the reference to this entity can be added as a second primary key.



The referenced entity is typically of the **Master Data** pattern. We will discuss Master Data later in this chapter.

The other fields in the table contain the specific setup values that are required from a functional perspective, but can also contain references to design patterns, such as the address integration.

The following diagram explains how to implement the Supplemental & Subsidiary Pattern:



The Page objects, connected to these tables are typically the editable List pages. Card pages are only required when the complexity exceeds the number of columns that fit on a screen with a normal resolution.

Implementations

The Supplemental & Subsidiary pattern is used for Setup tables that have more than one value in the application.

Examples

The pattern is used in hundreds of tables, making it the most implemented Architectural Pattern. In the standard Microsoft Dynamics NAV application, the examples of its usage are as follows:

Table ID	Description
3	The Payment Terms
4	Currency
14	Location
91	The User Setup
92	Customer Posting Group
5404	The Item Unit of Measure

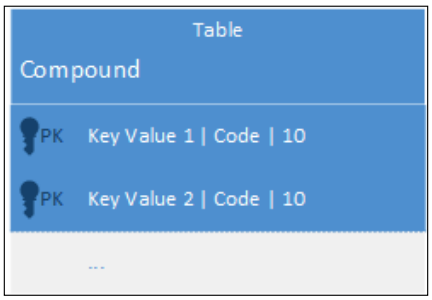
The Compound Pattern

A Compound pattern helps us to define attributes that depend on a combination of two entities. This allows us to create additional flexibility when designing the application setup, creating values that connect through a many-to-many relationship.

Technical description

When applying this pattern, the table has two fields of type code and a length of 10 characters. Both the values should allow blank values.

The following diagram explains how to implement the Compound pattern:



To use the compound table in the application, both the fields are members of the table that uses the attributes. Typically, the compound key values are inherited from the fields in Master Data.

Implementation

The Compound table is used to define attributes for a combination of two values in the system.

Examples

The pattern is used in Microsoft Dynamics NAV for the General and VAT/TAX Posting Setup, which is as follows:

Table ID	Description
252	The General Posting Setup
325	The VAT Posting Setup
5813	The Inventory Posting Setup

The Rules Pattern

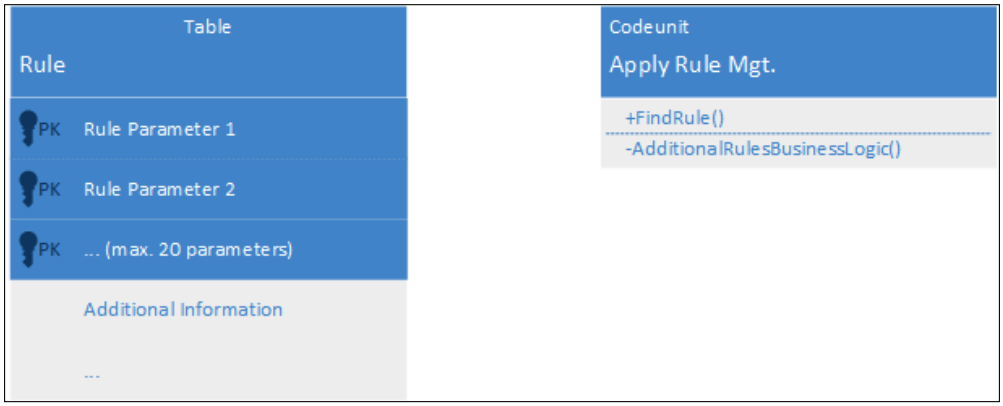
The Microsoft Dynamics NAV EARP application is designed to work for small, medium-sized, and larger businesses. The **Rules Pattern** enables us to store information in the database with a sliding level of complexity.

Technical description

These tables have a composite Primary Key that has up to 20 fields, limited by the SQL Server or 900 bytes. The value being used in the application is based on rules. These rules are programmed in Codeunits.

The non-Primary Key fields typically determine the values that are being used after applying the rule, although these fields can also be a part of the Primary Key. For example, the selected printer, after filtering on the criteria, is an additional field in the table.

The following diagram explains how to implement the Rules pattern:



Implementations

The Rules Pattern is used for the setup tables, and for either pricing or discount information. It is also used to implement a **Select Distinct** statement; for example, during the posting of **Documents**.



To learn more about the Select Distinct statement in T-SQL, go to `http://www.w3schools.com/sql/sql_distinct.asp`.

Examples

In the standard Microsoft Dynamics NAV application, the examples of the usage are as follows:

Table ID	Description
49	Invoice Post. Buffer
78	The Printer Selection
7002	Sales Price
7004	Sales Line Discount
7012	The Purchase Price

The Master Data Pattern

The EARP systems are designed to deal with people and things. These people can be customers, vendors, or employees. The things can be the G/L Accounts, Items, Fixed Assets, or Loaners.

We will refer to people as the **Person Master Data**, and to Things as the **Product Master Data**.

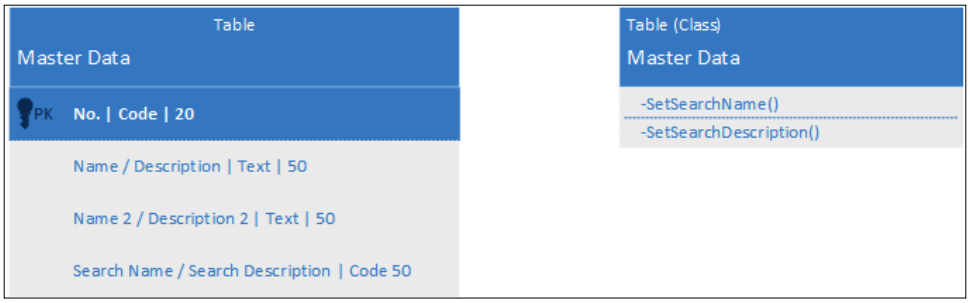
Technical description

The Master Data tables have a single Primary Key of the Code length as 20. This code field is typically and automatically determined using the **Number Series Pattern**, although this is optional. To have a more humanly readable way to find records in the table, each Master Data has two text-fields of length **50**, and a code-field of length **50**.



The Number Series Pattern will be discussed in *Chapter 3, Design Patterns*.

The following diagram explains how to implement the Master Data Pattern:



For the People Master Data, these text fields are called **Name** and **Name 2**. The code fields is named **Search Name**. The Product Master Data has text fields called **Description** and **Description 2**. Here, the code field is called **Search Description**.

The Search Description field is determined based on the first text field using the following algorithm:

```
IF ("Search Description" = UPPERCASE(xRec.Description)) OR
  ("Search Description" = '')
THEN
  "Search Description" := Description;
```



Downloading the example code

You can download the example code files from your account at `http://www.packtpub.com` for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Master Data of the type Person typically has the **Address Integration Pattern** implemented. Other related Design Patterns are **Entity State** and **Dimensions**.

Implementations

Microsoft Dynamics NAV has about a dozen tables that contain Master Data. About half of these are Persons, and the other half are Things.

Some of the Person Master Data tables are linked using the Contact Table as Master Record.



The link between the Contact Master Data and other Person Master Data is described in *Chapter 2, Squash Court*, of the book *Microsoft Dynamics NAV 2013 Application Design* that was published by *Packt Publishing*.

The G/L Account table is an example of Master Data without the implementation of the No. Series Design Pattern.

Examples

These tables, in Microsoft Dynamics NAV, use the Master Data Pattern:

Table ID	Description
15	The G/L Account
18	Customer
23	Vendor
27	Item
156	Resource
270	The Bank Account
5200	Employee
5600	The Fixed Asset
5628	Insurance
5913	Loaner
99000754	The Work Center
99000758	The Machine Center

The Journal Template-Batch-Line Pattern

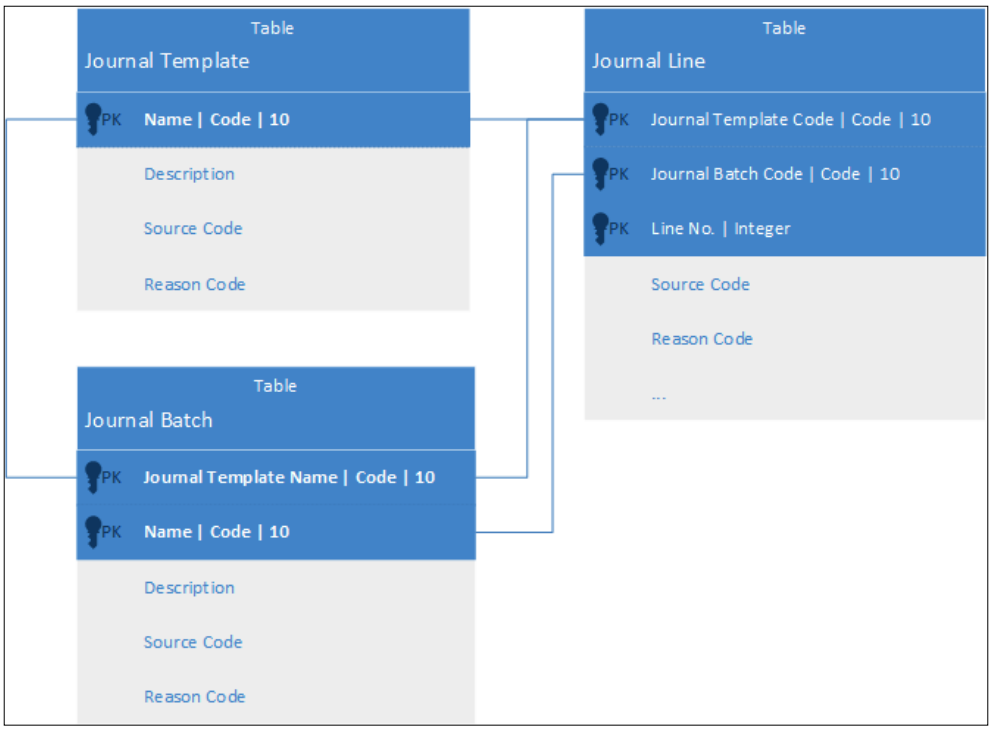
In the Financial systems, Journals are used for accounting. Microsoft Dynamics NAV has an Architectural Pattern called **Journal Template-Batch-Line** that ensures data integrity, and acts as a datacontract for each transaction.

Technical description

Journals are the datacontracts, employed to create entries that are used for the accounting purposes. They can be used either manually via the user interface, or automated via processes, such as documents or batch programs.

Each Journal has a Journal Line table that has a Journal Template Code, and a Journal Batch Code. The Template can be used to influence the functional user interface, and the reports are being used in the process. The Batch allows different users to use the Journal at the same time without interfering with each other, or temporarily keep data in a Journal until it is ready to be posted.

The following diagram explains how to implement the Journal Template-Batch-Line Pattern:



The Journal Line table contains information that needs to be carried over to the entry table that is created by it, as well as all the references to other data that are required for implementing the Journal-Template-Batch-Line Pattern. The creation of the entry tables is done using the Posting Pattern, which we will discuss later in this chapter. A Journal also has a predefined number of management Codeunits that help in the behavior of the template, the batch selection, and the selection of the posting process.



The Entry Table pattern is discussed later in this chapter.

The previous diagram shows the basic components of a Journal.

Let's step through the tables one-by-one.

The Journal Template table

The Primary Key of a Journal Template table is called Name. It is of the type Code, and has a length of 10 characters. Other fields in a Journal template table are Description, the Reason and Source Code, and one or more references to Posting processes or testing reports.

The Optional fields in a Journal Template table are fields that act as predefined values for the journals, such as No., Series, and the Bal. Account No. The Journal Template table has two list page objects. One is editable to be used for Application Administrators. Another one is non-editable, and helps the users to select the template that they want to use.

The Journal Batch table

The Primary Key of the Journal Batch table contains a reference to the Journal Template table, and a field called Name of type Code with the length of 10 characters. Together, these fields are the Primary Key, and are numbered 1 and 2.

Other fields in this table are similar to the Journal Template table. These fields overrule the template. For example, if the template has Reason Code "SALES" and the batch has Reason Code "INVENTORY", then the batch gets priority. So, the reason code "INVENTORY" is used. If there is a functional requirement, the Journal batch can also contain a Boolean called Recurring, indicating that the Journal Lines act as a template, and will be periodically repeated.

Journal batch has an an editable list page that allows users to select a batch, and create their own batches.

The Journal Line table

The Journal line table is the actual posting data-contract that contains the data to be processed. The Primary Key has three fields:

Field ID	Name	Type	Description
1	The Journal Template Name	Code 10	It references to the Journal Template table
2	The Line No.	Integer	This is a unique value determined by AutoSplitKey
51	The Journal Batch Name	Code 10	This references to the Journal Batch table

Other than these fields, the Journal Line table contains all the fields that are required to create the entry records.

Recurring Journals

Journals that are recurring contain two other fields that are used for the date calculation, which are the following:

- Recurring Method |Option |, F Fixed, V Variable, B Balance, RF Reversing Fixed, RV Reversing Variable, RB Reversing Balance
- Recurring Frequency, DateFormula



An explanation of the use of recurring journals and its methods is available at `http://www.archerpoint.com/blog/Posts/microsoft-dynamics-nav-2009-r2-creating-recurring-general-journals`.

Pages

Journals have at least one journal page, but some journals in Microsoft Dynamics NAV have multiple pages, showing different columns for different usages. For example, the General Journal has a Bank Journal, a Payment Journal, and a Cash Receipt Journal, all based on the same tables.

Codeunits

Each journal has a set of Codeunits that are required as a minimum:

Name	Description
JnlManagement	This is the Helper object that contains the functions to select templates, manage working with batches, and other functions specific to a journal.
Jnl.-B.Post+Print	This Codeunit can be called with a Journal Batch table as a parameter. All the lines in the journal that are related to the batch are posted and printed.
Jnl.-B.Post	This Codeunit can be called with a Journal Batch table as a parameter. All the lines in the journal related to the batch are posted.
Jnl.-Post+Print	This Codeunit can be called from a Journal Line table. All the lines in the selection are posted and printed.
Jnl.-Post	This Codeunit can be called from a Journal Line. All the lines in the selection are posted.
Jnl.-Post Batch	This Codeunit can be called from a Journal Line table. All the lines within the same Journal Batch are posted.

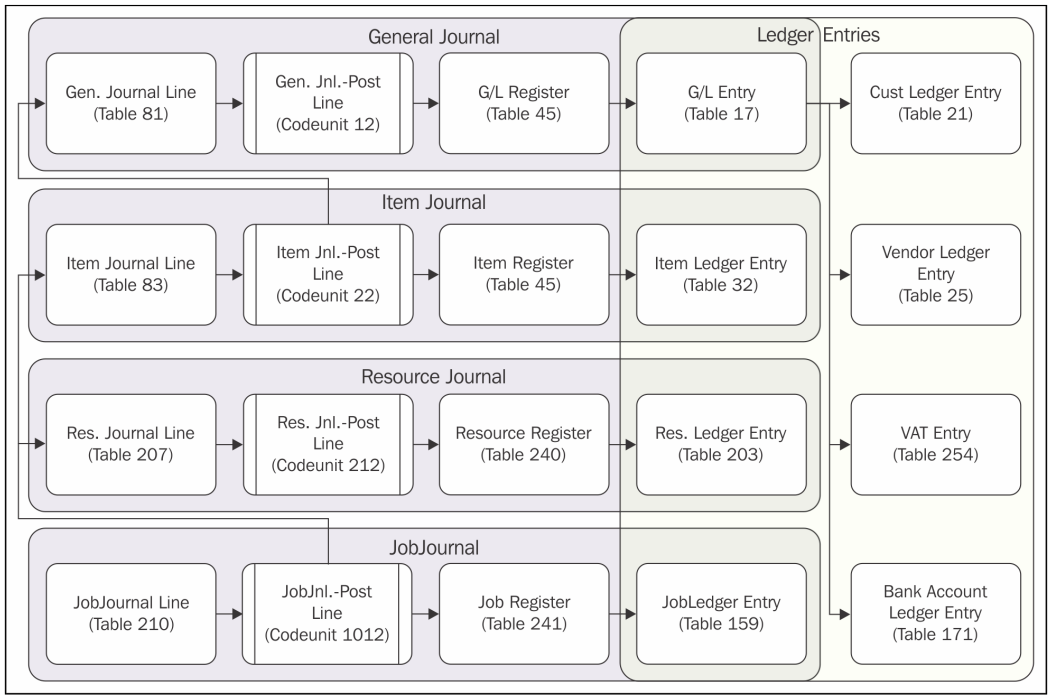
Name	Description
Jnl.-Check Line	This Codeunit is called automatically before posting the lines, and it contains the business logic to test the journal lines for valid values.
Jnl.-Post Line	This is the actual routine that moves the data from the Journal to one or more Entry tables. This also creates a Registry table.

Implementations

Journals are used in Microsoft Dynamics NAV for, among others, the General Ledger, Inventory, Intrastat, Resources, Jobs, Cost Accounting, Fixed Assets, Insurance, and Warehouse. Examples of pages, based on the same journal tables, are the Payment Journal, the Phys. Inventory Journal, and the Production Journal.

Hierarchy

The Job Journal, the Resource Journal, the Journal Item, and the General Journal in Microsoft Dynamics NAV have a hierarchy. The following diagram illustrates this:



When a Job Journal Line is posted, the Item consumption, or the Resource Consumption can be posted using the required journal, which then might trigger the General Journal again, if this has implications for the financial backend. This posting can be executed asynchronously for performance reasons.



The hierarchy is described in *Chapter 1*, *Introduction to Microsoft Dynamics NAV*, in *Microsoft Dynamics NAV 2013 Application Design*, published by *Packt Publishing*.

Journals as contracts

When journals are used in hierarchy, or in any way nested in posting routines, there is a standard pattern for populating the values. This pattern is described in the section *Posting* in the later part of this chapter.

Examples

The following journals exist in Microsoft Dynamics NAV:

Table IDs	Description
232, 80, 81	The General Journal
233, 82, 83	The Item Journal (Inventory)
236, 206, 207	The Resource Journal
237, 209, 210	The Job Journal
1100, 1101, 1102	The Cost Journal (Cost Accounting)
5619, 5620, 5621	The Fixed Asset Journal
5622, 5623, 5624	The Fixed Asset Reclassify Journal
5633, 5634, 5635	The Insurance Journal
7309, 7310, 7311	The Warehouse Journal
261, 262, 263	The Intrastat Jnl. Line

The Entry Pattern

Most EARP systems are data cemeteries. A lot of data comes in, nothing much ever comes out. To store data in Microsoft Dynamics NAV, the **Entry Pattern** is most popular with over 75 implementations in the standard product.

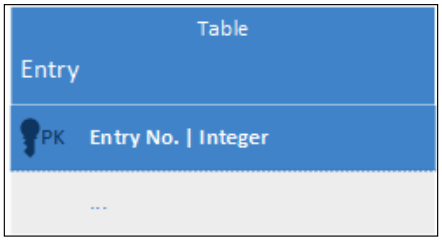
Technical description

Although there are many variations, all entry tables have the same technical structure that can be expanded upon the functional requirements.

The Primary Key for all the entry tables is a field called Entry No. of the type Integer. The value of the field is determined using either the **AutoIncrement** property, or a code algorithm as the following:

```
IF NextEntryNo = 0 THEN BEGIN
  ExLedgEntry.LOCKTABLE;
  IF ExLedgEntry.FINDLAST THEN
    NextEntryNo := ExLedgEntry."Entry No.";
    NextEntryNo := NextEntryNo + 1;
  END;

WITH ExLedgEntry DO
  "Entry No." := NextEntryNo;
```

The following diagram explains how to implement the Entry Pattern:



Entry tables contain reference fields to other tables, often Master Data or Documents, for which they determine things such as balances.

The Entry Table types

In Microsoft Dynamics NAV, we differentiate between the historical and temporary entries.

Historical entries

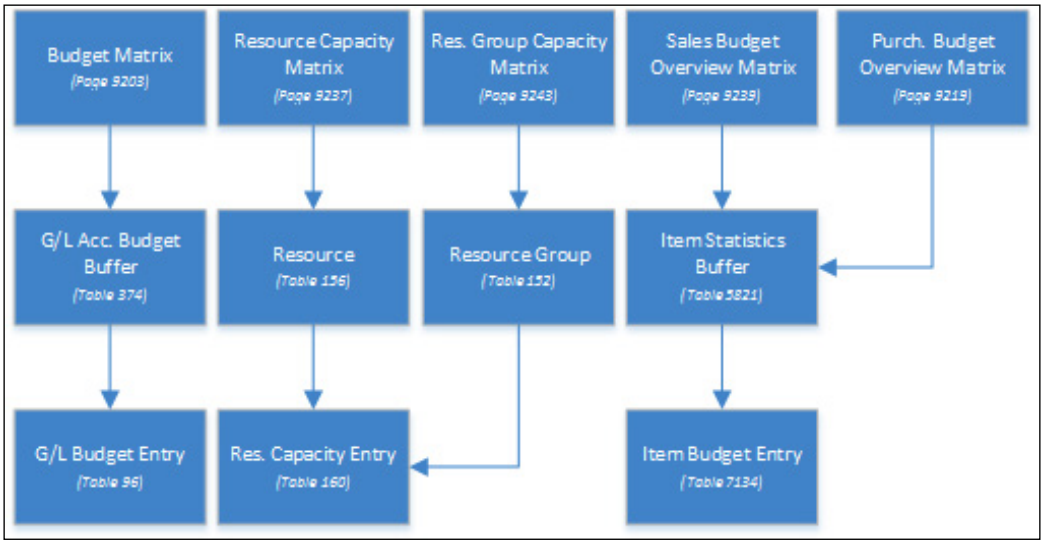
Historical entries are created using Journals, and are tightly connected to registers. If they can be deleted, such as the reservation entries, they are managed using a clean-up algorithm that ensures balances. We can find Historical Entries using the Navigate API, as they are typically related to Master Data and Documents.

Temporary entries

There are a number of ways to get temporary entries in the system. One of them is using the flow field metadata, as discussed in *Chapter 1*, *Introduction to Patterns and Software Architecture*. Flow fields are designed to calculate totals, based on the table fields and flow filters. However, this process can also be reversed.

When we validate a flow field a delta entry is created, based on the filter values to ensure that the totals match the new total value.

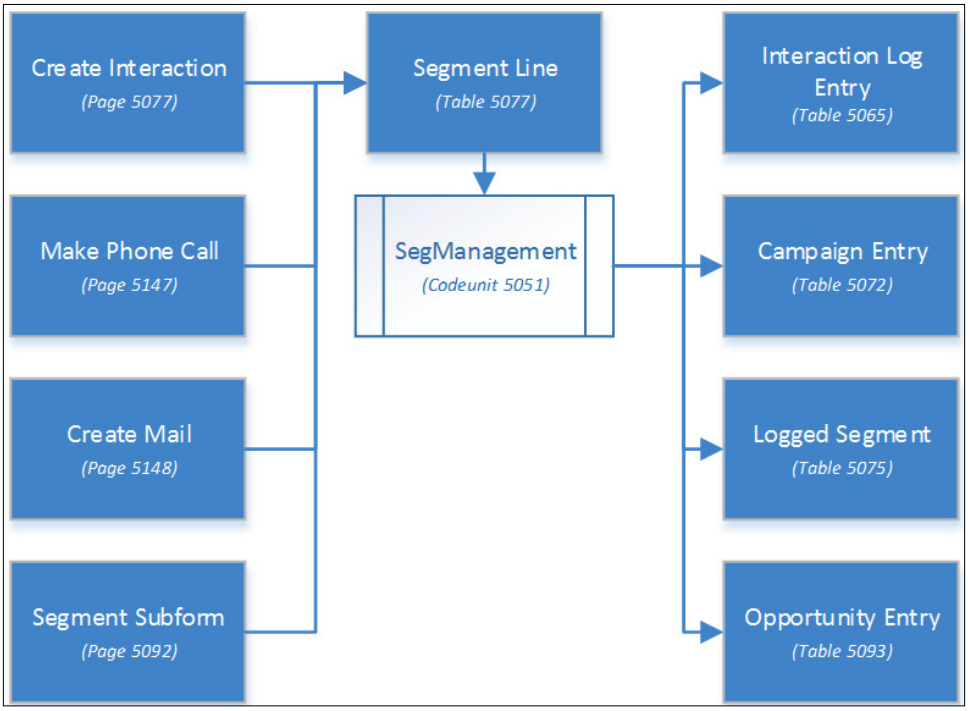
This is used in the Budgeting pages that we can find with the following diagram:



Entries that can be deleted

When the business case requires the possibility of deleting entries, we can make the auditability less strict. We still need code to create the entries, but if they are not used for accounting purposes, users have the option of deleting them again, or changing the contents.

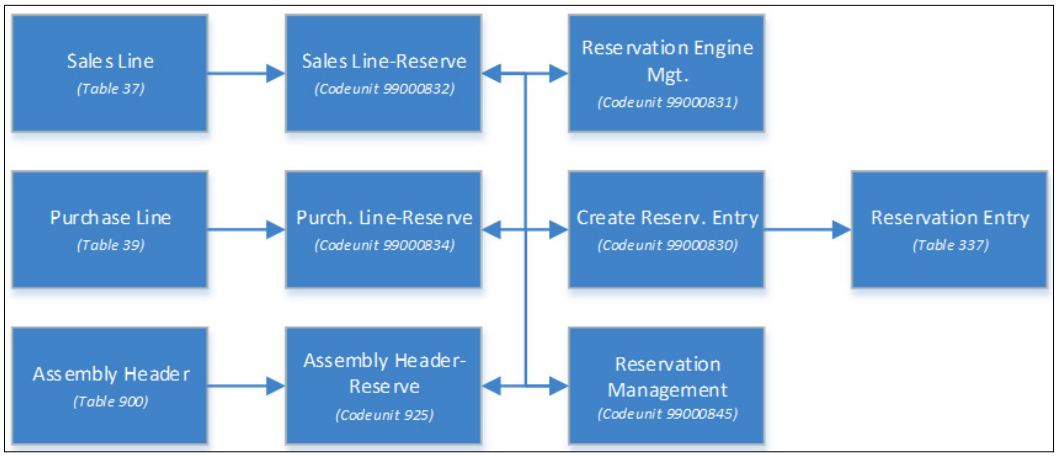
An example of this can be found in Relationship Management, in the following diagram:



Temporary tracking buckets

In some cases, we need tables to track data during the existence of a Document or Process in the system. We can use Entry Pattern for this in combination with a set of Codeunits that ensure data integrity, and ensure having contracts to the application areas where the data is required.

An example of implementing entries can be found in the application area Reservation Entries. Each application area, such as Sales or Purchase has a Codeunit entry for the Reservation Engine. The Reservation Engine consists of three Codeunits. The following diagram explains the structure:



Implementations

There are over 75 implementations of Entry Pattern, varying from the accounting areas like General Ledger and Inventory, to the Flow Field implementations in Budgeting.

Other examples are Job Queue Entries, and Change Log Entries.

Examples

This is a subset of tables using Entry Pattern in Microsoft Dynamics NAV:

Table ID	Description
17	The G/L Entry
96	The G/L Budget Entry
337	The Reservation Entry
405	The Change Log Entry



Not all the tables in Microsoft Dynamics NAV that have the text "Entry" use Entry Pattern. The Document Entry table is an example that does not follow the pattern.

Document Pattern

The EARP systems are traditionally designed to help companies administer the paper trail, required to run their businesses, often from a legal perspective. To be able to register the paper documents in an electronic form, Document Pattern can be used.

Technical description

The Document pattern allows us to enter a header and line information. The header information is often related to Person Master Data (with whom I am doing business), whereas the lines are of the type Product (the kind of business that I am doing).

Each Document Pattern has two tables: a header and line. These names are important, the Sales *Header* and the Sales *Line* tables follow this convention, just as others.

The Header table

The header table has a primary key that is called No., and has a type Code with a length of 20 characters. Optionally, the primary key can also contain a reference to a normalized document type. Normalized in this case means that the share tables and business logic, for physically different documents, have a functional resemblance, such as Sales Quotes and Sales Invoices.

The value of the primary key is most often determined using Number Series Design Pattern.

The header tables, most often, contain a reference to the required Master Data or Supplemental & Subsidiary Tables. This includes using the Address Integration Pattern.

From a workflow perspective, the Entity State Design Pattern is popular in combination with the Document pattern. This tells the system the status of a document. We can have multiple instances of the Entity State Design Pattern in one document.



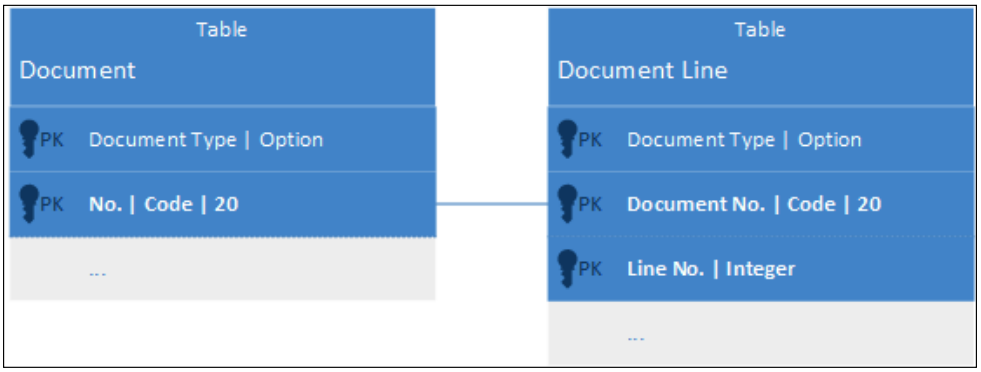
The Address Integration and the Entity State patterns are mentioned in *Chapter 3, Design Patterns*.

The header table has at least two page objects: a non-editable list page, and an editable document page.

The Line table

The primary key of a line table contains a reference to the primary key of the header table with an additional field called Line No. of the type Integer. This field should be the last field in the Primary Key, because the Line No. field is populated using the **AutoSplitKey** property on the subpage. The Line table can contain references to one or more Product Master Data tables. This table has a single page object of the type called **ListPart** that is used as a SubPage for the Document Page.

The following diagram explains how to implement Document Pattern. Please note that the Document Type is optional:



Posting and archiving

Documents follow a flow through the company, and sometimes even leave the company in the form of a shipping bill or sales invoice. Sometimes, a single document creates several other documents, and triggers the creation of entries in the accounting or inventory part of the system.

To enable this, the Create History pattern is used. We will discuss this later, in *Chapter 3, Design Patterns*.

Implementations

There are over fifty implementations of the Document pattern in Microsoft Dynamics NAV. The most well-known implementations are the Sales and Purchase documents.

Examples

This is a subset of the tables that use Document Pattern:

Table ID	Description
36,37	Sales Header, Sales Line
110,111	Sales Shipment Header, Sales Shipment Line
295, 296	Reminder Header, Reminder Line
5076, 5077	Segment Header, Segment Line

The Argument table

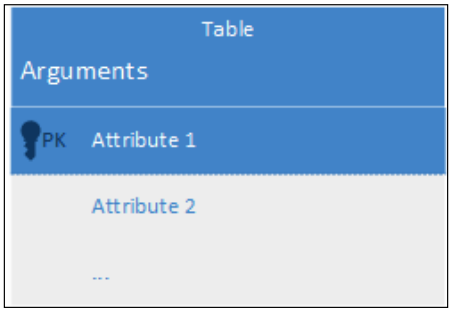
The Argument table has an ambiguous place in the categorization of patterns. The table is used to combine fields when we call a function. This can be compared to a class in object-oriented programming. Since Microsoft Dynamics NAV does not allow us to create a class, we use the table object instead. This will automatically create a table in the SQL Server, but this table will never contain any data. However, some tables are used for both arguments and normal data.

In computer programming, the general rule of thumb is to avoid functions that have too many arguments in the signature. When a function has more than three or four arguments, we should combine them into an argument table.

Technical description

The definition of the table should contain all the fields used in the arguments. The primary key of the table is not important, since the table will never contain actual data. However, the metadata in Microsoft Dynamics NAV requires at least one field as the primary key. This can be any field in the argument table, preferably field 1.

The following diagram explains how to implement Argument Pattern:



This Argument table is bound to the implementation of the function and should not be reused.

Implementations

Microsoft Dynamics NAV uses the concept of dedicated tables as argument tables for e-mail handling. Also, the Journal Line tables are used as argument tables to create ledger entries during posting.



We will use the Argument table in *Chapter 5*, *Coding Best Practices*, to help us organize code and implement the overloading in Microsoft Dynamics NAV.

Summary

In this chapter, we have learned what Architectural Patterns are, and how to implement them in Microsoft Dynamics NAV. We have covered nine patterns that cover more than 95 percent of the tables that are defined with the standard application.

Using Architectural Patterns helps us to define tables when building our applications that others will recognize. This creates a common syntax within the community, and reduces the learning curve when bringing in new developers and architects.

In the next chapter, we will implement Design Patterns that will add elements to Architectural Patterns that define the usability of the application.

3
Design Patterns

In the previous chapter, we learned how to create Architectural Patterns that are the large building blocks. Now, we will learn how to enhance these by adding a flavor to them using Design Patterns.

Design Patterns are groups of fields that add the functional behavior to an application. The elements that we use for these Design Patterns are typically fields and/or some C/AL code snippets. This is a different use of the term Design Patterns than what is generally used in other books.

The Entity State Pattern

In Microsoft Dynamics NAV, the UI elements in the page objects are bound one to one with tables. The tables in the product represent functional elements, such as **Customers**, **Sales Invoices**, or **Production Orders**. The fields that we need in the page have to exist in the table object.

An example of an **Entity State** field is adding a Boolean field called **Blocked**, indicating that the record in the table should not be used in a transaction. This field tells us something about the state of the record.

To add information about the records in these tables, we can add fields with predefined options. These field elements are referred to as Entity state. Because we predefine the state in metadata, we can write code against these fields by creating business rules and processes.

Technical description

To implement this Design Pattern, we need to add a field to the table that describes the state. The field is either of the **Boolean** or **Option** type. The combination of the field name and the option values define metadata in context with the business logic, which we will create while writing the code.



Using other field types, such as Text or Decimal would multiply the amount of states, making it impossible to code against.

When using the **Boolean** datatype, the record in the table the name of the field is the only element that we can use. The record either has the state (the value is `yes`) or it doesn't (the value is `no`).



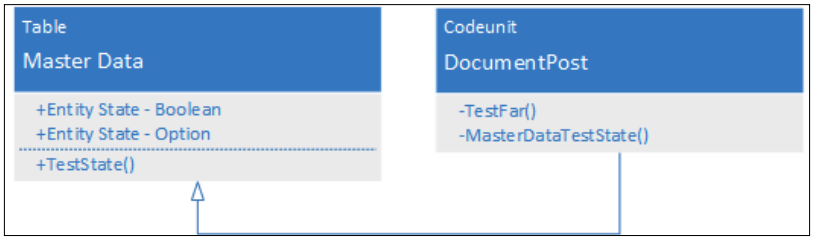
The video at `https://www.youtube.com/watch?v=O2R-fTSup1o&list=PLhZ3P-LY7CqmVszuvtJLujFyHpsVN0U_w&index=16` explains the implementation of the Entity State Pattern.

When using the option field, the state can have multiple values. The combination of the field name, and the option values allow a more complex array of states. For example, the Status field in a Sales document has the values Open, Released, Pending Approval, and Pending Prepayment. The combination of the field name and the option captions determine the functional behavior.

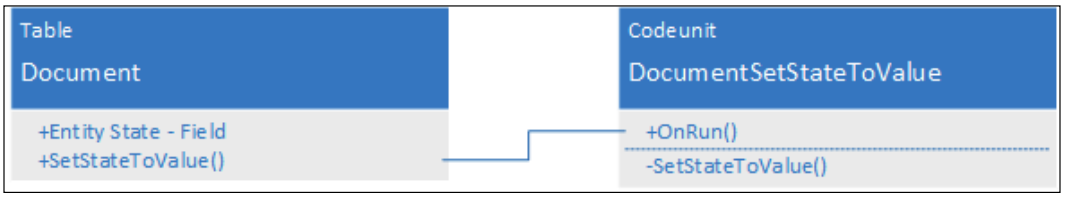
Setting the State

The simplest way to set the State is by allowing the user to select a value manually. To do this, the field should be editable and available via the user interface. The `InitValue` property can be used to change the default value.

The following image illustrates adding an Entity State field to the **Master Data** table, and implementing a function to test the state, which is called in **Codeunit**. The **State** word should be replaced with a better functional description.



The State can also be determined by a function that is called via business logic in **Codeunit**. This business logic can also be executed directly as a function, or it can be nested into other transactions.



Setting the state may trigger additional transactions to be started.

Testing the State

The typical scenario to use the Entity State Pattern is in preventing transactions from happening in the systems. In order to do this, typically, the **TESTFIELD** function is implemented. This command allows developers to test a value in a field. The system will generate an error with a predefined syntax. This should be programmed into the transactions where it is applicable.



The Error Message Processing Cookbook recipe helps in testing multiple fields and states in one transaction. You can read more about this in the Microsoft Design Patterns Wiki at `https://community.dynamics.com/nav/w/designpatterns/236.error-message-processing`.

Alternatively, the Entity State field can also be a part of the Table Relation to the table that the field is a member of. This prevents the record from being used. If this is implemented, then the TESTFIELD function should always be implemented as a safety precaution. An example is testing the Blocked field when selecting an **Item** in a **Sales Line**.

Implementations

The Entity State pattern can be implemented in all the tables, but in the Microsoft Dynamics NAV application, they are typically implemented in Master Data or Documents.

Examples

These objects contain examples of the following Pattern:

Table ID	Field ID	Type	Description
18	39	Integer	Blocked
36	120	Option	Status

The Select Pattern

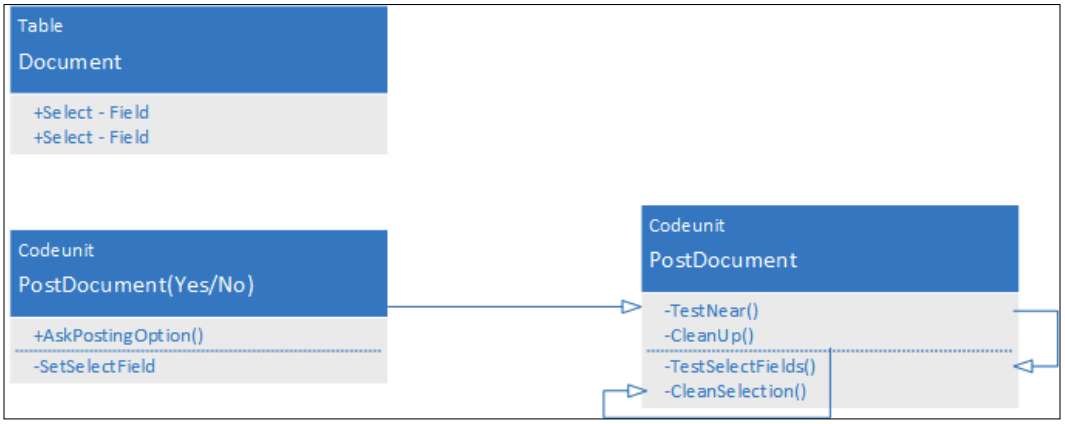
When a transaction is started with a record as a parameter, the transaction sometimes need extra information to decide how the transaction needs to behave. When using the `OnRun()` trigger of a Codeunit to start the transaction, we can only use one table as a parameter to the function. The **Select** pattern helps us in working around this limitation in an elegant way, allowing us to write clean and readable code.



More information about the pattern can be found in the *How Do I* video at `https://www.youtube.com/watch?v=SxywT2XSpcI&list=PLhZ3P-LY7CqmVszuvtJLujFyHpsVN0U_w&index=7`.

Technical description

To be able to give the extra information to the function, we need to add extra fields to the table that acts as **placeholders** or **properties**. These fields are not available from the user interface, and should be non-editable.



They need to be set prior to calling the transaction, but they don't have to be written to the database. They are kept in the memory during the transaction. At the beginning of the transactions, the **values** need to be verified. When the transaction ends, the values need to be reset to the **default values**.

Implementations

The Select Pattern can be implemented in tables that are used in transactions to create history; or in any scenario where the `OnRun` trigger of a Codeunit is used, and extra parameters need to be passed to the code in the Codeunit.



Alternatively, the Codeunit can have Global variables that can be set via functions prior to calling the Codeunit. Microsoft uses a mix of both solutions. The Select Pattern is an elegant way, since it requires less coding than using functions. An example of a function can be found in Codeunit 80, Sales-Post. This has a function called `SetPostingDate` that is called from reports 296, Batch Post Sales Orders and 297, Batch Post Sales Invoices.

Examples

These objects contain examples of the following Pattern:

Table ID	Field ID	Type	Description
36	57	Boolean	Ship
36	58	Boolean	Invoice

The Number Series Pattern

When working with the ERP systems, issue numbers are issued to identify records in tables, which is a key requirement. Microsoft Dynamics NAV has a similar numbering engine built into the system as a part of the application. The engine is written in C/AL, and is accessible to be changed by developers.



More information can be found in the *How Do I* video, available at `https://www.youtube.com/watch?v=1lG9rY_dmM4&list=PLhZ3P-LY7CqmVszuvtJLujFyHpsVN0U_w&index=8`.

Although it is possible to change the behavior of the engine, best practice is not to do so. The engine can be used from any part of the system by implementing a few steps.

Technical description

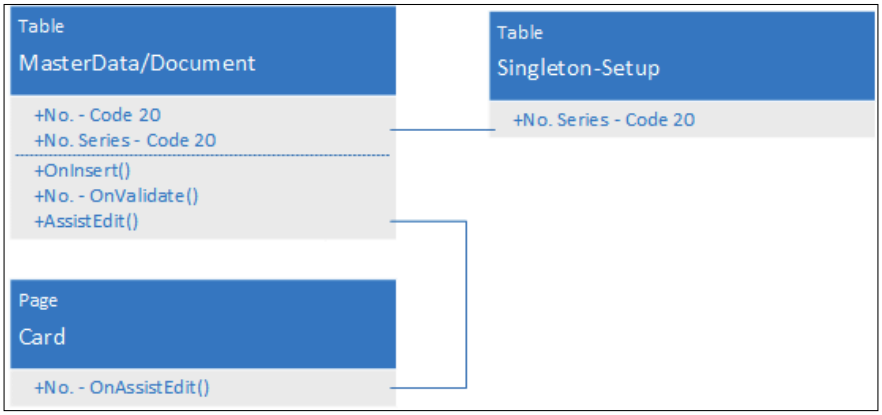
The **NoSeriesManagement** Codeunit has a number of functions, such as **TestManual**, **InitSeries**, and **LookupSeries**. We will focus on how to implement them as part of the primary key. Another example is to implement Number Series to automatically populate the Document No. field in the **Journal Lines**. This is outside the scope of this book.

To implement Number Series so as to uniquely assign values to the primary key, the following steps need to be implemented.

The Table reference field

As a reference to the Number Series that was used, a field is required as part of the table. This field is called **No. Series**, and has a table relation with the **No. Series** table. This field is not a part of the user interface, and should be non-editable.

When a table needs to have more than one **No. Series**, a functional prefix should be added to explain what it is used for.



Setup reference

In order for the system to know which Number Series to use for which implementation, a reference should be implemented in a Setup table. Each implementation of the Number Series should have a minimum of one Setup Reference.



The setup is most often done in Singleton tables, as described in *Chapter 2, Architectural Patterns*.

C/AL functions and variables

In order for a Number Series implementation to behave as expected, we need to implement three C/AL Code snippets, and create a reference from the UI. The code is implemented in the **Table** using the Number Series. The following points are the features of Number Series:

- The first implementation is in the `OnInsert()` trigger of the table:

```
IF "No." = '' THEN BEGIN
  ExampleSetup.GET;
  ExampleSetup.TESTFIELD("Example Person Nos.");
  NoSeriesMgt.InitSeries(ExampleSetup."Example Person Nos.",
    xRec."No. Series",0D,"No.","No. Series");
END;
```

  In this example, the `ExampleSetup` is a reference to the Setup table. `NoSeriesMgt` represents Codeunit 396 NoSeriesManagement, and No. Series is a reference field to the Number Series that is used.

- Number Series allows us to block users from manually assigning values. Therefore, we need to test this when assigning values to the **No.** field. This code should be in the `OnValidate` of the **No.** field:

```
IF "No." <> xRec."No." THEN BEGIN
  ExampleSetup.GET;
  NoSeriesMgt.TestManual(ExampleSetup."Example Person
Nos.");
  "No. Series" := '';
END;
```

- Number Series also allows to link **multiple series**. From the UI, users can select any one of them. This is done using a function called `AssistEdit`:

```
AssistEdit() : Boolean
ExampleSetup.GET;
ExampleSetup.TESTFIELD("Example Product Nos.");
IF NoSeriesMgt.SelectSeries(ExampleSetup."Example Product
Nos.",xRec."No. Series","No. Series") THEN BEGIN
  NoSeriesMgt.SetSeries("No.");
  EXIT(TRUE);
END;
```

- This function is called from the `OnAssistEdit` trigger of the **No.** field in the page object:

```
No. - OnAssistEdit()
IF AssistEdit THEN
  CurrPage.UPDATE;
```



An explanation of the functional use of the Number Series in Microsoft Dynamics NAV is available at `http://dynamicsnavfinancials.com/2013/12/11/tip-33-number-series-nav-50-tips-series/`.

Implementations

Number Series is implemented in almost all the Master Data and Document tables in Microsoft Dynamics NAV using this pattern. An alternative implementation can be found using the Document No. assignment from the Journals.



The G/L Account master data does not use Number Series. All the countries have their own Chart of Accounts. Number Series don't add functional value here.

When issuing numbers for **history records**, such as Posted Sales Invoices or Issues Reminders, the Number Series are typically assigned in the original document.

Examples

These objects contain examples of the following Pattern:

Table ID	Description
18	Customer
36	Sales Header

Address Integration Pattern

Just as Number Series, implementing addresses is typical when working with ERP applications. Microsoft Dynamics NAV follows a number of steps when using the addresses.

Addresses in Microsoft Dynamics NAV are not fully **normalized**. Each implementation creates a new set of predefined fields in a table, and *copies* the values. This has some advantages such as simplicity and filtering. The downside is that addresses use a lot of space in the **record sets**.



This video, available at `https://www.youtube.com/watch?v=60Wrx9N-gfY&list=PLhZ3P-LY7CqmVszuvtJLujFyHpsVN0U_w&index=19`, explains the Address Integration Pattern.

Technical description

Implementing addresses in Microsoft Dynamics NAV requires a set of fields to be implemented, values to be inherited, and the definition of a **formatting** function in the FormatAddress Codeunit.

Fields

When using Addresses in Microsoft Dynamics NAV, the following fields are typically implemented:

Field Name	Type	Length	Comment
Name	Text	50	
Name 2	Text	50	
Address	Text	50	
Address 2	Text	50	
City	Text	30	Table Relation with table 225 Post Code
Contact	Text	50	
Country/ Region Code	Code	10	Table Relation with table 9 Country/Region
Post Code	Code	20	Table Relation with table 225 Post Code
County	Text	30	Caption is State in North America

When more than one address is used in one table, the field names should be more specific; for example, **Bill-to Name** and **Ship-to Name**.

The fields called City and Post Code are related to the Post Code table. This relation is validated by the following C/AL code:

```
City - OnValidate()
PostCode.ValidateCity(City,"Post Code",County,"Country/Region
Code",(CurrFieldNo <> 0) AND GUIALLOWED);
Post Code - OnValidate()
PostCode.ValidatePostCode(City,"Post Code",County,"Country/Region
Code",(CurrFieldNo <> 0) AND GUIALLOWED);
```

Copying the address

Addresses are related to the Master Data of the type Person that we discussed in *Chapter 2*, *Architectural Patterns*.

In Microsoft Dynamics NAV, the address information is synchronized between the **Contact** table and their related **Customer**, **Vendor**, and **Bank Account** records.



The synchronization is described in *Chapter 2*, *Application Design*, of the Microsoft Dynamics NAV 2013 book.



An explanation of how the Customer and Vendors are linked to a Contact is available at `http://www.olofsimren.com/connect-a-customer-with-a-vendor/`.

When addresses are used in Documents, the values are copied from Master Data or the source document. When the address changes at the source, the inherited values are not updated.

The following code snippet, taken from the Sales Header table, shows how the data is copied from one record to another:

```
Bill-to Customer No. - OnValidate ()\
…
GetCust("Bill-to Customer No.");
…
"Bill-to Customer Template Code" := '';
"Bill-to Name" := Cust.Name;
"Bill-to Name 2" := Cust."Name 2";
"Bill-to Address" := Cust.Address;
"Bill-to Address 2" := Cust."Address 2";
"Bill-to City" := Cust.City;
```

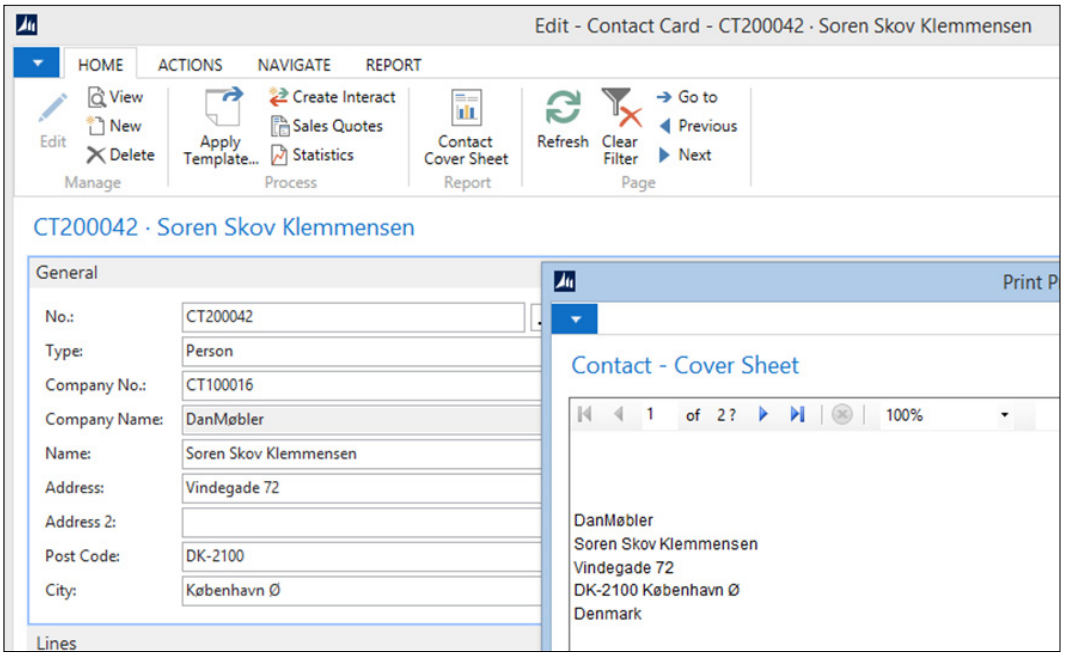
```
"Bill-to Post Code" := Cust."Post Code";
"Bill-to County" := Cust.County;
"Bill-to Country/Region Code" := Cust."Country/Region Code";
```

When creating history from documents, the values are typically forwarded using the **TRANSFERFIELDS** function. We will discuss that later in this chapter.

Printing addresses

The primary reason for using addresses is to print them on documents and envelopes. When we do this, we need to make sure that the empty values don't use unnecessary space. Also, the formatting of addresses, especially the Post Code, is sometimes different per country. The Codeunit 365 **FormatAddress** takes care of all the formatting, and converts the field values into an Array of text values.

In the following screenshot, we can see that **Address 2** does not leave a blank line when printing the address on a **Contact Cover Sheet**:



Each implementation of the **Address** fields has a unique function in this Codeunit. Out of the box, this Codeunit can format addresses from over 65 sources.

The signature of each function is always the Text Array and the source table:

```
Company(VAR AddrArray : ARRAY [8] OF Text[50];VAR CompanyInfo :
Record "Company Information")
WITH CompanyInfo DO
  FormatAddr(
    AddrArray,Name,"Name 2",'',Address,"Address 2",
    City,"Post Code",County,'');
```

When required, this Codeunit should be declared as well as the Text Array. The Text Array can then be used in the **dataset** of **reports**.

Implementations

Addresses are implemented where required, often in Master Data and Documents. The Company Information Singleton table has Address Information, as does the Location Supplemental table.

Some tables, such as the Sales & Purchase documents have up to three addresses.

Examples

These objects contain examples of the following Pattern:

Table	Description
14	Location
18	Customer
36	Sales Header
79	Company Information

Multi-Level Setup

When designing software for both small-sized and medium-sized businesses, there is often a balancing act in the designing setup options. Where some companies only require a feature to be turned on or off, other businesses require them to be more flexible.



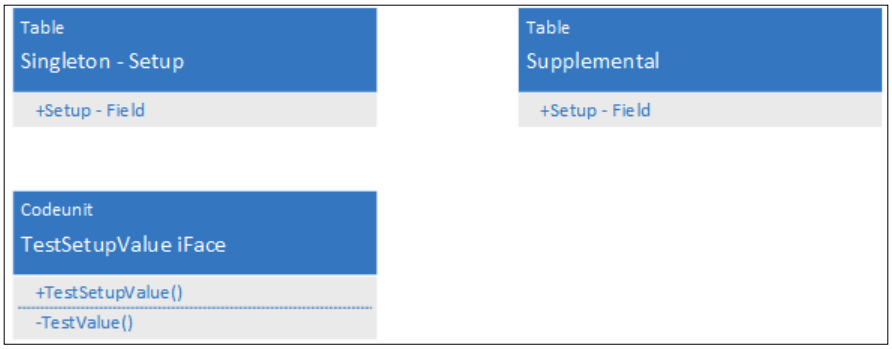
The Multi-Level Setup pattern is explained at `https://www.youtube.com/watch?v=kMtzKFjU22U&list=PLhZ3P-LY7CqmVszuvtJLujFyHpsVN0U_w&index=30`.

In order to allow the best of both worlds, we can simply add the setup options to both the tables, and write a small interface to read the values based on importance.

Technical description

The setup fields need to be implemented on at least two levels, for example, a **Singleton** and **Supplemental** table. The setup options should be identical.

To read the setup values, we need to provide a function that first checks the most detailed level, the **Supplemental** table, and then the Singleton table.



The following code snippet is a part of Codeunit 11 **Gen. Jnl.-Check Line**, and checks the values of **Allow Posting From** and **Allow Posting To** in reference to a **PostingDate** variable:

```
DateNotAllowed(PostingDate : Date) : Boolean
IF (AllowPostingFrom = 0D) AND (AllowPostingTo = 0D) THEN BEGIN
  IF USERID <> '' THEN
    IF UserSetup.GET(USERID) THEN BEGIN
      AllowPostingFrom := UserSetup."Allow Posting From";
      AllowPostingTo := UserSetup."Allow Posting To";
    END;
  IF (AllowPostingFrom = 0D) AND (AllowPostingTo = 0D) THEN BEGIN
    GLSetup.GET;
    AllowPostingFrom := GLSetup."Allow Posting From";
    AllowPostingTo := GLSetup."Allow Posting To";
  END;
  IF AllowPostingTo = 0D THEN
    AllowPostingTo := 31129999D;
END;
EXIT((PostingDate < AllowPostingFrom) OR
  (PostingDate > AllowPostingTo));
```

Implementations

This feature can be implemented across the application where it is required.

Examples

These objects contain examples of the following Pattern:

Field Name	Type	Table	Description
Allow Posting From	Date	98	The General Ledger Setup
Allow Posting To	Date	98	The General Ledger Setup
Allow Posting From	Date	91	The User Setup
Allow Posting To	Date	91	The User Setup

Document History

In Microsoft Dynamics NAV, Documents have a limited life cycle and once obsolete, they need to be removed from the table. The Document History pattern helps us to create this. Examples of historical documents are Sales Order Archive, Sales Shipments, Invoices and Credit Memos. Issues Reminders, and Service Invoices. They act as data containers, and are often used to print documents. Once the documents are printed, they can be deleted.



An explanation of how to read and understand posting routines in Microsoft Dynamics NAV is available at `https://www.youtube.com/watch?v=Wetd1Fa0rmw&list=PLhZ3P-LY7CqmVszuvtJLujFyHpsVN0U_w&index=4`.

Technical description

When creating **History** for a Document, we move the data from one set of Document tables to another. This is done using a number of Codeunit objects.

The naming conventions

From a functional perspective, this process has many names, such as Posting, Archiving, Issuing, and Registering. This term is used in the User Interface, and should functionally apply to the task.

The Document tables, where we move the data to, have no business logic and exist only for historical reasons. In the standard Microsoft Dynamics NAV Application, they can be removed after printing.



The History Document Tables should therefore never be used for data analysis and reporting purposes.

Codeunit Structure

The Codeunit object used for the process has a logical flow called **Test Near**, **Test Far**, **Do It**, and **Clean up**.

Test Near, Test Far

Before the process starts, we need to make sure that everything is in place. It does not make sense to create records in the database, and then later roll back the **transaction** when a **mandatory** field is not populated. We should test the data that is closest first, which are the fields in the **Document Header** and **Document Line** table.

If the data in these tables is good, we can read singleton and rules tables to see if the external setup values are okay to proceed.

Do it

When the traffic lights are green, we can move the data to the next set of tables, and process other nested transactions.

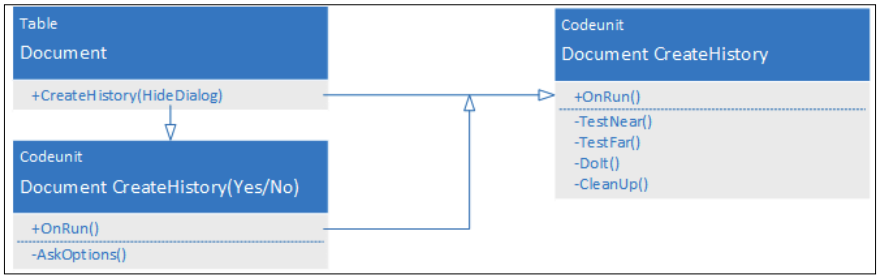
The nested transactions can be creating entries via **Journals** or other **Documents**.

Clean up

At the end of our process, we can clean up the data. This may involve deleting the original records from the database.

An Example

The following diagram illustrates how a table can call a function to create a history document. The **CreateHistory** function can be called Post, the **Codeunit** can be called Sales-Post, such as Codeunit 80.

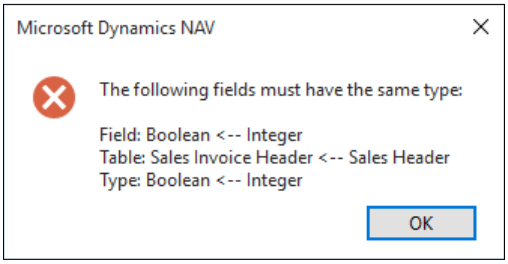


TRANSFERFIELDS

When we move Documents from one set of tables to another, we use the C/AL built-in command called **TRANSFERFIELDS**. This automatically copies all the field values from one table to another, based on the field numbers.

The MSDN article, at `https://msdn.microsoft.com/en-us/library/dd301438.aspx`, explains the use of the TRANSFERFIELDS command.

This makes field numbering very important. If a field has a different functional meaning, we get garbage; and if fields with the same number have different datatypes, we might get runtime errors. The following screenshot illustrates the error:



If it is not possible to use the same field numbers, the history tables need to be populated field-by-field.



An example of the TRANSFERFIELDS command when creating documents can be found in Codeunit 80, Sales-Post. The 36, 110, 112, and 114 tables have the same structure. Even the 37,111,113, and 115 tables have the same structure.

If fields do not exist in the tables, we move data to, the data is not written to the history tables and is therefore lost.

Implementations

The most well-known examples are **Sales-Post** and **Purchase-Post**. The helper objects are **Sales-Post (Yes/No)** and **Purch.-Post + Print**. Another implementation is **Reminder-Issue.**

Examples

These objects contain examples of the following Pattern:

Table ID	Description
900	Assembly-Post
5760	Whse.-Post Receipt
5980	Service-Post

Copy Document Pattern

Entering a Document with all the options and multiple lines can be a lot of work, and sometimes similar Documents already exist in the system. This is often a reverse action of the Document History Pattern, but we can also copy the outstanding documents.



An explanation on the use of Copy Document Pattern is available at `https://www.youtube.com/watch?v=aTiwroXwW_0&list=PLhZ3P-LY7CqmVszuvtJLujFyHpsVN0U_w&index=17`.

An example of this is creating a Credit Memo from a Posted Sales Invoice where all the information is copied, including the Customer, Items, and the correct posting dates.

Technical description

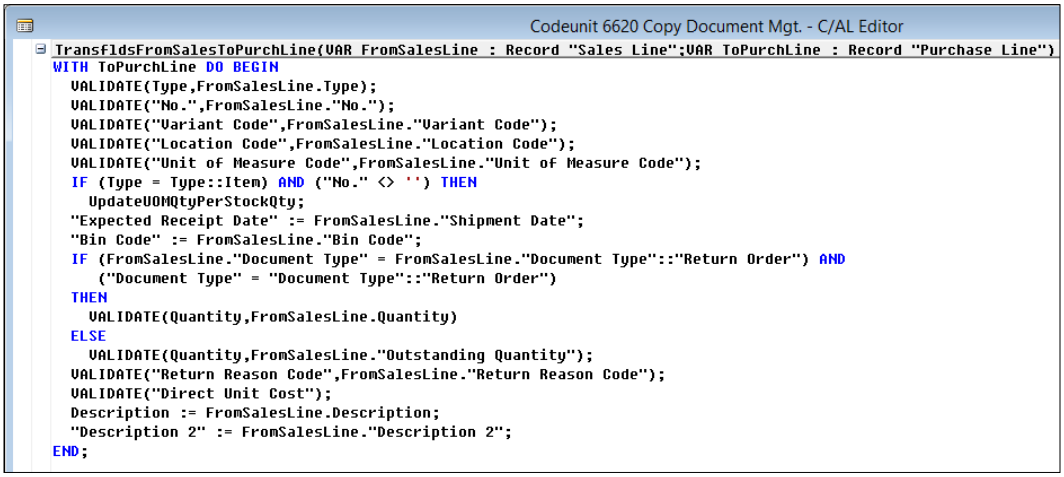
The **Copy Document Pattern** objects are a Processing Only Report that calls into an internal function called the Codeunit. The Report object needs to understand the valid options to copy the data around. We can, for example, copy a Posted Sales Shipment to a Sales Order, but we cannot copy a Sales Order to a Posted Sales Invoice.

The direction where we copy the data always needs to be an unprocessed document.

Validation logic

Unlike the posting pattern, we cannot only use **TRANSFERFIELDS**. We need to apply business logic that checks if the values are still valid. If we copy a document that has been in the archive for a year, the Product Master Data might be discontinued. Each Document has a unique set of validation protocols that follow functional data entry patterns. This protocol should be used in the Copy Document functions.

This following screenshot illustrates the validation logic to create a Sales Line:



This means that if the validation logic in the Document changes because we add fields or change the data entry order we need to change the Copy Document too.

Implementations

Copy Document Pattern is implemented in the major functional areas of Microsoft Dynamics NAV: Sales, Purchasing, Assembly Management, and Service Management. The production orders are allowed to be copied, but they don't follow this pattern.

Examples

The following objects are used by the Pattern:

Object	Description
Report 292	Copy Sales Document
Report 492	Copy Purchase Document
Report 901	Copy Assembly Document
Report 5979	Copy Service Document
Codeunit 6620	Copy Document

Job Queue Entry Pattern

In Microsoft Dynamics NAV, processes can be automated using the Job Queue. The Job Queue asynchronously processes requests as a background service using an NAV Application Service.



An explanation of the implementation of Job Queue Entry Pattern is available at `https://www.youtube.com/watch?v=UA2Pq7fv-Dk&list=PLhZ3P-LY7CqmVszuvtJLujFyHpsVN0U_w&index=15`.

Most processes can be wrapped in a Job Queue process by implementing Job Queue Entry Pattern.



Job Queue can also execute a report. This pattern describes the usage of Codeunits in Job Queue.

Technical description

Job Queue Pattern requires us to wrap the process into a Codeunit that takes a Job Queue Entry as a parameter. This Codeunit then follows a number of steps.

Preprocessing

During this step, we can test if the Job Queue Entry table contains the information that we need.

Getting records

The Job Queue Entry contains a **Record ID** that allows us to read a record from a specific table. This record will be the one to be processed.

The Main function

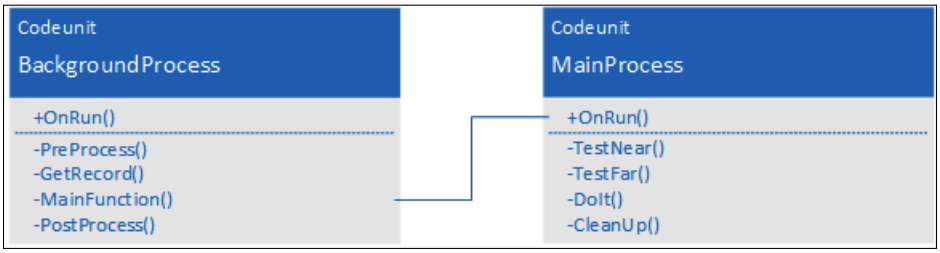
In this step, we execute the automated process. Each process in Dynamics NAV should be designed to be wrapped in a Job Queue Entry, and should therefore avoid any runtime decisions; or if any, they should be omitted.



Runtime decisions are the implementations of STRMENU, CONFIRM, or the running MODAL page objects.

Try to avoid the code cloning of the main function to allow running via the Job Queue Entry.

The following illustrates the structure of the Job Queue Entry Codeunits:



Post-processing

Specific tasks related to the background process can be handled in this step, such as handling the Job Queue Status.

Error handling

The source table for the automated process should have a new field Job Queue Status. This field is of the type Option, and has the options: (blank), **Scheduled for Posting**, and **Error & Posting**.

The Helper Codeunit, as discussed in the pattern Document History, should set this field to the **Scheduled for Posting** value when the background processing is activated.

Implementations

This is implemented for Sales and Purchase Posting in Microsoft Dynamics NAV.

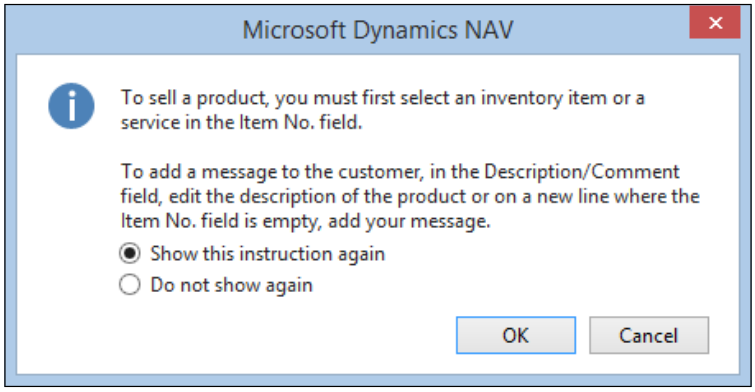
Examples

These objects contain examples of the following Pattern:

Object	Description
Codeunit 88	Sales Post via Job Queue
Codeunit 98	Purchase Post via Job Queue

Dismissible dialogs and save preferences

In order to design user-friendly systems that can be used intuitively, we need to store the user's preferences, and allow them to specify them at runtime.



Each user can make their own settings, based on the level of guidance they feel comfortable with.

This pattern is relatively new to the Microsoft Dynamics NAV product, and was first implemented in the 2013 R2 version.



An explanation of the implementation of embedding instructions in Microsoft Dynamics NAV is available at `https://www.youtube.com/watch?v=loobQ1TVO3o&list=PLhZ3P-LY7CqmVszuvtJLujFyHpsVN0U_w&index=14`.

Technical description

The preferences are stored in a table with a combination of User ID and a code that represents the setting. The settings are handled in functions that are the members of the function.

To test if a user has not disabled something, simply use the following code:

```
IsGettingStartedVisible() : Boolean
EXIT(NOT MiniUserRemovedInstructions.GET(USERID,GetGettingStartedCode
));
```

If a user wants to disable a feature, the `DisableInstruction` function can be used:

```
DisableInstruction(InstrCode : Code[20])
IF NOT MiniUserRemovedInstructions.GET(USERID,InstrCode) THEN
BEGIN
  MiniUserRemovedInstructions.INIT;
  MiniUserRemovedInstructions."User ID" := USERID;
  MiniUserRemovedInstructions."Instruction Code" := InstrCode;
  MiniUserRemovedInstructions.INSERT;
END;
```

Dismissible dialog

To allow a user to dismiss a message, a STRMENU should be implemented as the following:

```
ShowCreateLineInstruction(ItemNo : Code[20])
IF (ItemNo = '') AND GUIALLOWED AND IsEnabled THEN
  IF STRMENU(OptionsStringQst,1,CreateNewLineInstructionTxt) = 2
THEN
    DisableMessageForCurrentUser;
```

The `OptionsStringQst` is a Text Constant with the `Show this instruction again` and `Do not show again` options.

Implementations

This pattern is implemented in some areas of the Small Business Application.



I've explain what the Small Business Application is about at the `http://nav-skills.com/2013/09/26/dynamics-c5-2014-welcome-miniapp/ blog entry`.

Examples

Object	Description
Table 1306	Mini User Removed Instructions
Page 9022	Small Business Role Center
Codeunit 1320	Mini Lines Instruction Mgt.
Codeunit 1321	Mini Getting Started Mgt.

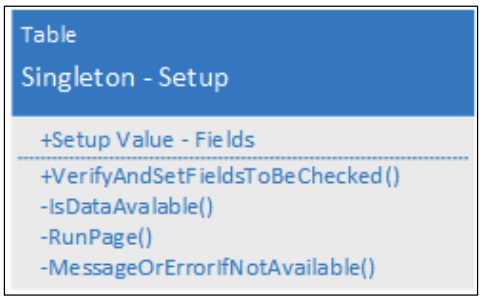
Setup Update Pattern

Implementing the correct setup for a complex ERP system like Microsoft Dynamics NAV can be a lot of work and is error prone.

Setup Update Pattern can be implemented to guide end users to set up the system if they try to use a feature that is not set up correctly.

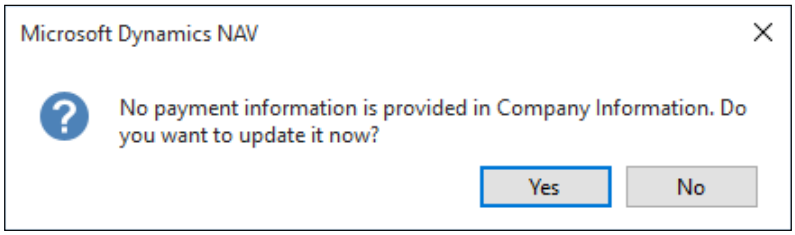
Technical description

Instead of programming TESTFIELD in a setup field that would result in an error message to the user, we need to implement a function that guides users through the process.



The function is a member of the setup table, and should explain its purpose. As an example, we will look at `CompanyInfo.VerifyAndSetPaymentInfo`.

This function is called from the Sales Invoice report, and it shows the following message when executed:



The following is the code:

```
VerifyAndSetPaymentInfo()
GET;
IF IsPaymentInfoAvailble THEN
  EXIT;
IF CONFIRM(NoPaymentInfoQst,TRUE,TABLECAPTION) THEN BEGIN
  CompanyInformationPage.SETRECORD(Rec);
  CompanyInformationPage.EDITABLE(TRUE);
  IF CompanyInformationPage.RUNMODAL = ACTION::OK THEN
    CompanyInformationPage.GETRECORD(Rec);
END;
IF NOT IsPaymentInfoAvailble THEN
  MESSAGE(NoPaymentInfoMsg,TABLECAPTION);
```

This function has a logical structure that tests whether the setup is correct, and exits if so.

The user gets a question asking if they would like to complete the required setup for this feature. If yes, the system will show the UI element that allows the setup.

If the user closes the UI element without completing the system, a message or error can be displayed, based on the setup being **optional** or **mandatory**.

After this process, the system will complete the transaction using the new setup values.

Restrictions

This pattern should only be implemented when the user who is getting the message is also allowed to change the setup.

Implementations

This pattern is implemented when printing the Sales Invoice in Microsoft Dynamics NAV.

Example

Object	Description
Report 206	Sales Invoice
Table 79	Company Information

Model-View ViewModel Pattern

As explained in *Chapter 1, Introduction to Patterns and Software Architecture*, the Page objects are bound to the Table objects. The way the data is displayed to the user is the way we store it in the SQL Server database.

This is not always what we want. In some cases, we want to display the information in a different, way than how we store it in the database.

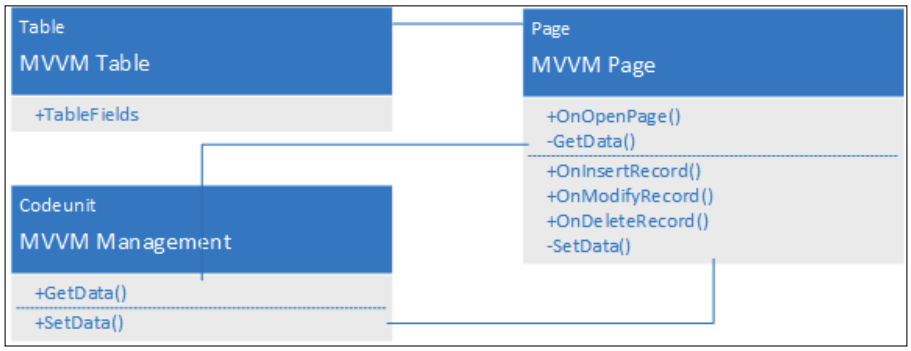
For this reason, each Page has the property **SourceTableTemporary**, and Reports have the Temporary property since Microsoft Dynamics NAV version 2015.

Technical description

The basics for this Pattern is the Temporary Dataset, which is a table that contains the field layout that we want to display to the user. This needs to have a matching Primary key, which is often Rules Pattern or Entry Pattern.

When this is set and we start up the Page or Report, we won't see any data. We need to load it into memory first.

This can be achieved by calling a function in either the `OnOpenPage` trigger, an Action, or a combination of these. This function should read the data and populate the source table of the **Page** or Report.



Once the Page or Report is closed, the data is deleted from memory.

Implementations

The most well-known implementation of the MVVM pattern in Microsoft Dynamics NAV is the Navigate page object. This has a function called `FindRecords`.

Examples

These objects contain examples of the following Pattern:

Object ID	Description
Page 344	Navigate

Summary

In this chapter, we have learned 11 commonly implemented Design Patterns for Microsoft Dynamics NAV using different elements. In the next chapter, we will combine Architectural Patterns and Design Patterns into an example application.

4
Building an Example Application Using Patterns

In this chapter, we will combine the Patterns that we have learned in the previous chapters into an application. Since the focus of this book is studying Patterns, we will implement them in an abstract way to avoid any functional flavor that might pollute this.

In the last chapter of this book, we will add the functional information that gives meaning to the abstract Patterns. The name of our application will be `Example` as the name of the chapter implies.

The design elements

We want to use as many Patterns in the application as possible to demonstrate their interaction. Although the Patterns are abstract, they have typical functional implementations that make them recognizable in an ERP environment, where Microsoft Dynamics NAV is typically used.

We will create an application with Documents using Master Data that creates the Entries using a Journal.

The Pattern dependencies

Within Microsoft Dynamics NAV, the Patterns have internal dependencies, and they work together in a hierarchy from a functional perspective.

We will construct the example application in four steps, starting with the foundation elements, and then building on top of this until we have four layers. Although each step will build upon the previous step, we will create elements in multiple layers in some of the steps.

Step one – Foundation

Before we can start with our application, we need some foundation elements such as Setup and a Role Center.

These are the basics of almost any functional application in Microsoft Dynamics NAV.

Example setup

When adding a new module to Microsoft Dynamics NAV, or doing a customization for a specific customer, it is considered best practice to create your own Setup. Doing this avoids software merge issues during upgrades or when applying rollup updates. Creating a Setup of a specific Module also allows organizations to separate responsibilities between different departments.

Which Patterns to use

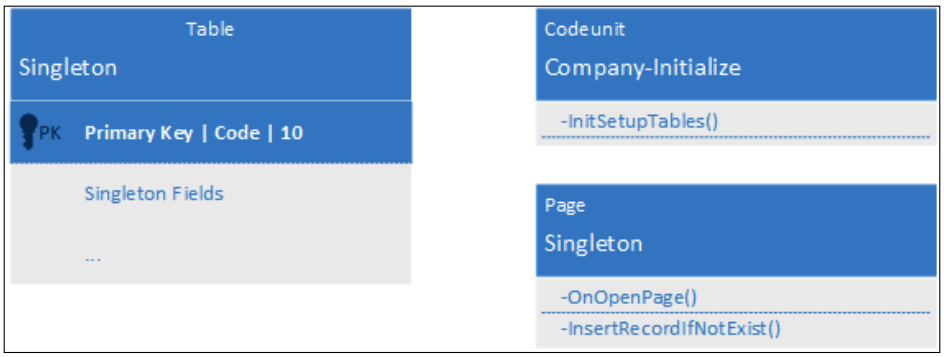
The Pattern that we use for the Setup is the Singleton Pattern. This Pattern allows us to create a single set of variables that is kept in memory throughout the session of the application that is shared with all the users; exactly what we need.



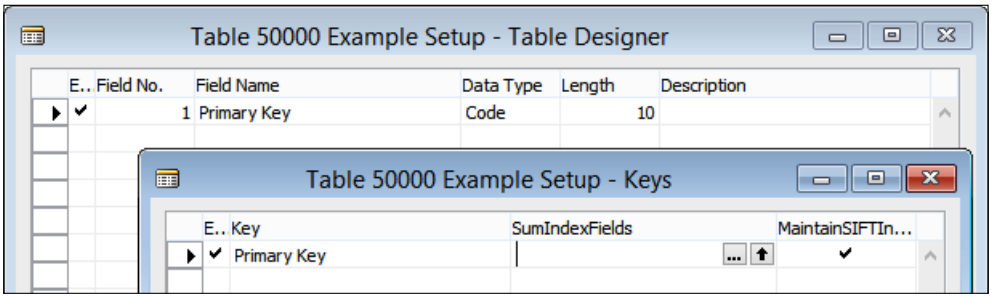
NAV 2013 introduced a global cache that is not only shared by users, but is also synchronized across the NAV servers. There are essentially three cache levels to keep you from having to perform these costly disk reads.

The implementation

The Singleton Pattern is an Architectural Pattern, which means we need to create at least one table and a page. The Pattern tells us how to structure these:

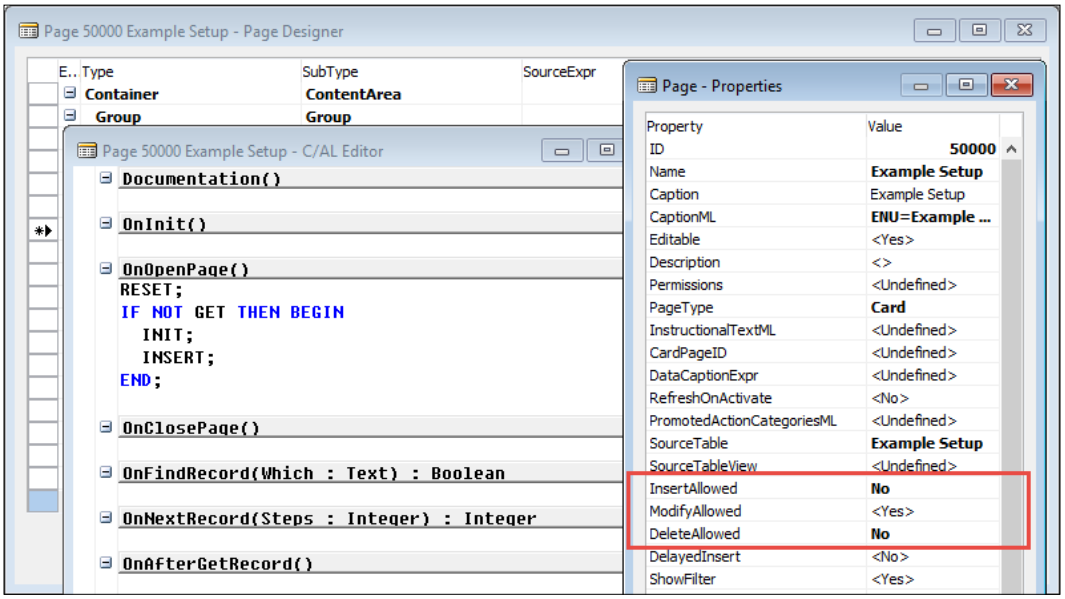


Let's create a new table, and define the primary key as in the diagram. We have to save the table, and name it as `Example Setup` with an ID of `50000`, as shown:



We also need to define a Page where the users can edit the setup. We have to prevent users from accidentally deleting the setup or creating additional records in the table.

The Page contains a small C/AL code snippet that creates a record in the setup table, just in case the record does not exist:



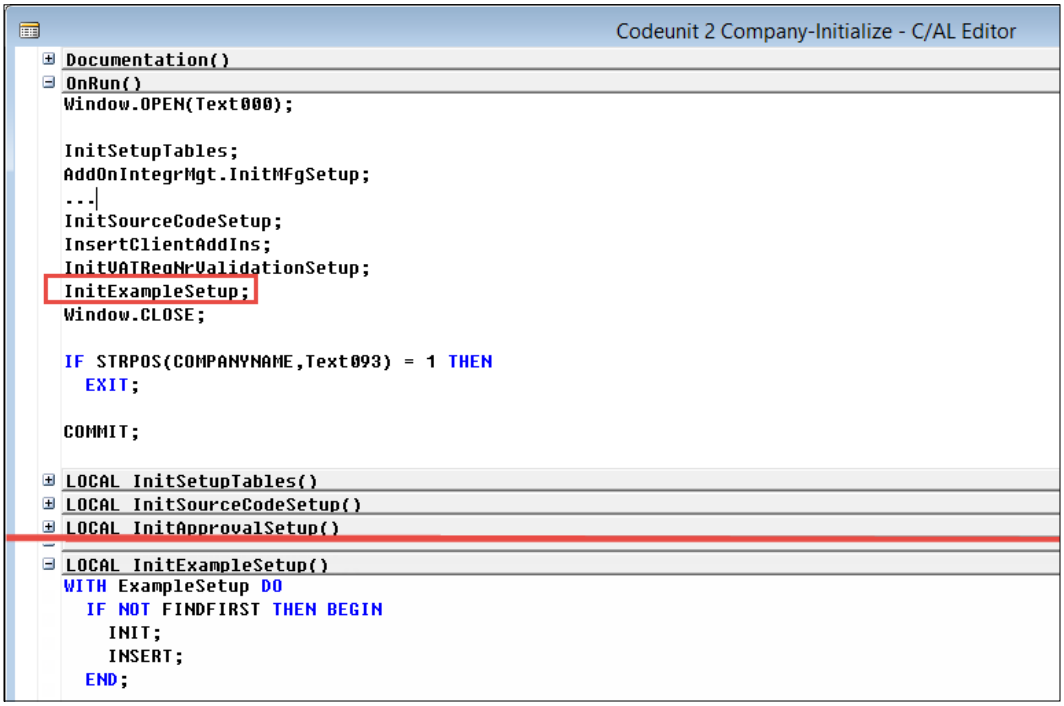
We also change the **InsertAllowed** and **DeleteAllowed** properties to `No`.

Although we have already made sure that a record always exists in the `OnOpenPage` trigger, it is also recommended to create the record in the Company-Initialize Codeunit. This Codeunit gets called each time we create a new company, or when upgrading to a new version.

In fact, the only reason the record is created in the Page is the scenario of adding a module to a Dynamics NAV system after the initial implementation, without upgrading to a new version of the base application.



When shipping a module as a product, it is recommended to add an Upgrade Codeunit to the package that creates the required settings in the database. More information can be found on MSDN, available at `https://msdn.microsoft.com/en-us/library/dn757164(v=nav.80).aspx`.



To make sure our code is executed during the Company-Initialize process, we need to make a raw source code modification to the standard Dynamics NAV code. There are a range of best practices to do this while ensuring upgradability and maintainability. We will discuss these best practices in the next chapter, *Chapter 5, Coding Best Practices*.

This is it. We have successfully implemented the Singleton Pattern to create a Setup table for our Example application.

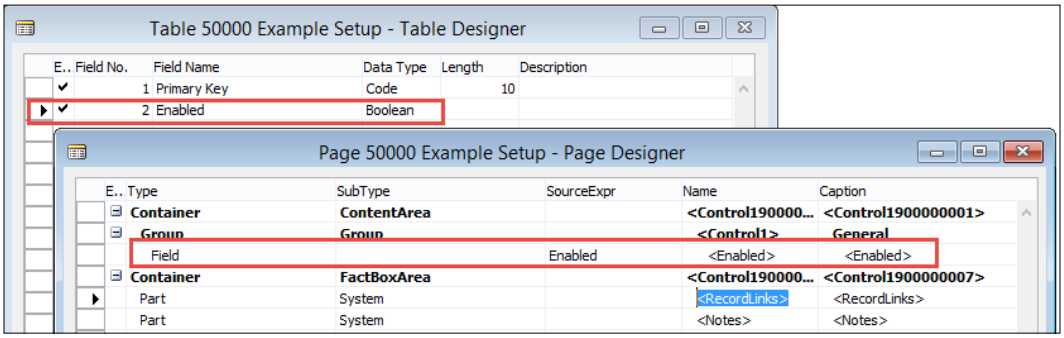
Let's implement another Pattern to make this a little bit more interesting, and see how the Patterns can work together.

The Multi-Level Setup Pattern

From an architectural perspective, the Singleton Pattern works together quite well with the Multi-Level Setup Pattern. We will implement this in our Example application by adding a Boolean that enables or disables the use of our little add-on.

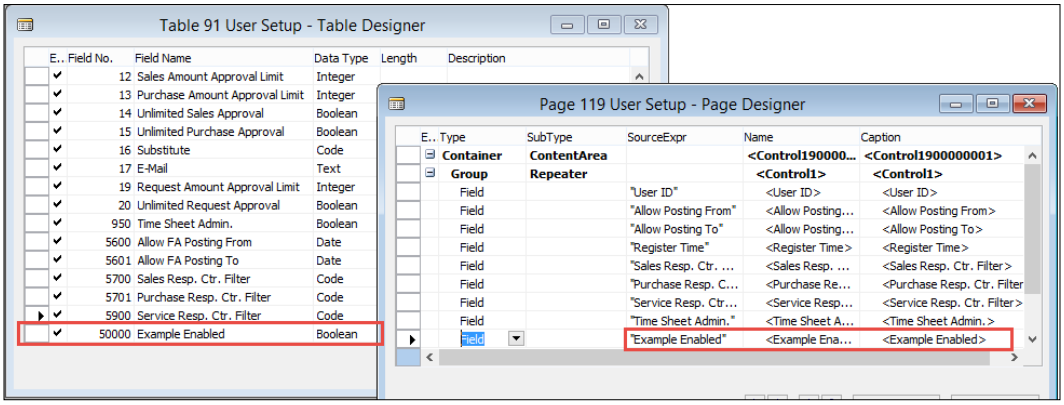
The idea is to be able to enable the application for individual users or on a database level.

Let's start by implementing a new field on the Setup table that allows us to enable the application. We will also add this field to the page.



Microsoft Dynamics NAV already has a table (User Setup) where users can enter preferences. Instead of creating our own, we will add a field to this table. This is what users of the product expect when working with the system. Each module in NAV has a limited set of options in this table. If we add a large range of setup options we may create our own table. In case of a new table, we are probably building a large vertical application, rather than a small module.

The following image shows the added field to the **User Setup** table and Page:



The **User Setup** table is implemented using the Supplementary and Subsidiary Tables Pattern, and has an editable list page that enables users to enter the setup options.

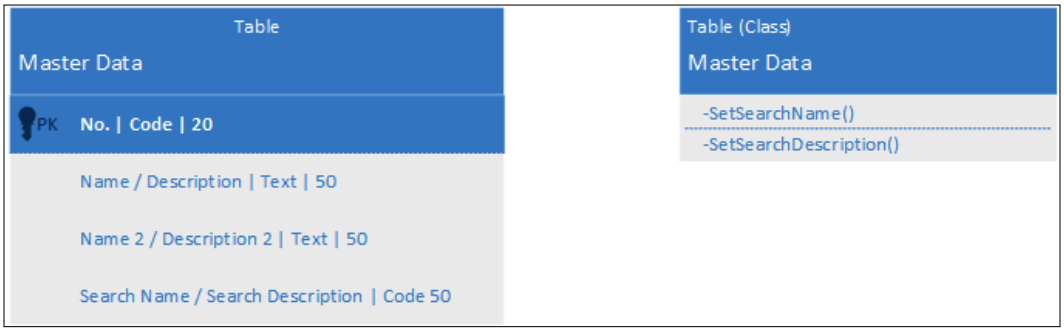
Now, you have completed the foundation elements of our application. Let's move on to the second step.

Step two – Master Data

When you decide to use Microsoft Dynamics NAV as a platform to build new applications, it is very likely that your application has a close resemblance to the ERP software. This is what Dynamics NAV was originally designed to do best, and the Meta Data elements are also optimized to work as such.

Within the ERP applications, we typically have Master Data that resembles people or things. In our example application, we will call them Example Person and Example Product.

The Master Data pattern schema is shown in the following diagram:

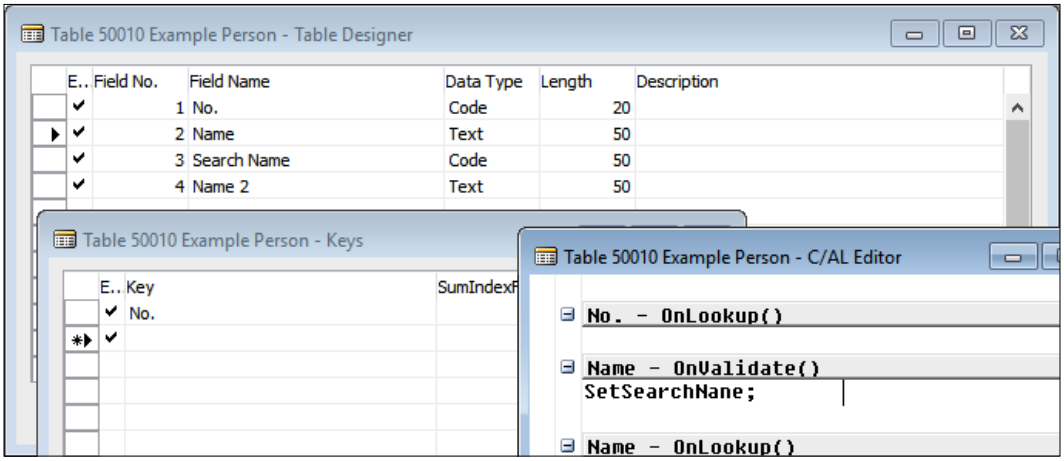


Example Person

For our Example Person entity, we need to create a table that contains the elements of our Pattern. We will call our table called Example Person, and save it with an ID of `50010`.



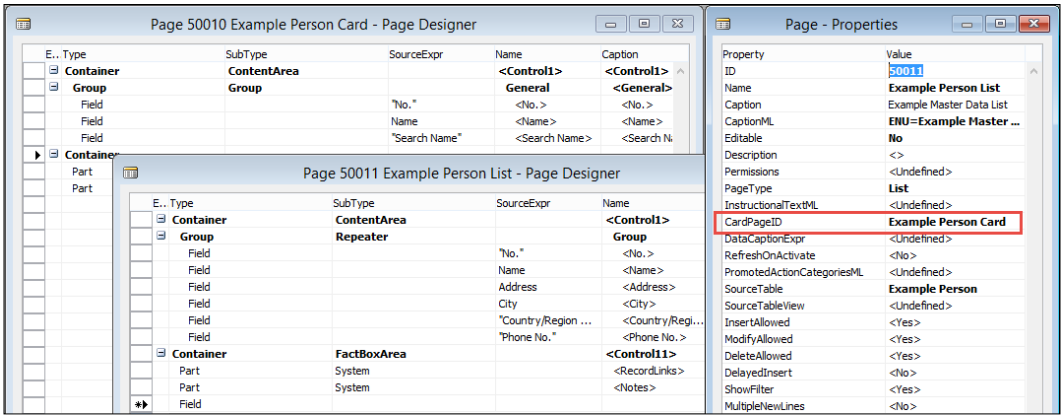
Leaving a small gap in the field and table IDs in our module gives us the option of later adding other tables in between, if this makes sense from a perspective of grouping similar objects together.





The field numbering in Master Data follows a specific Pattern, which is explained later in this chapter.

The Master Data Pattern also tells us to add the number of Pages. We need two: a non-editable list page, and a card page that allows entering new records, maintaining values, and removing records.

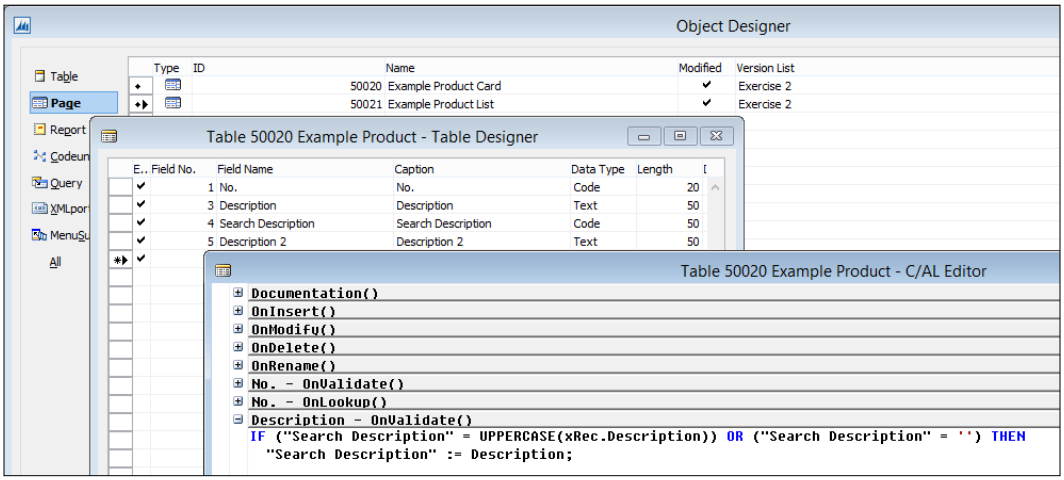


We will use the `50010` and `50011` IDs for the pages. The pages are linked using the **CardPage** ID, allowing the user to open the card by double-clicking on a record from the list.

When creating Page objects, there are some naming conventions to be followed. We will discuss them in the next chapter, *Chapter 5*, *Coding Best Practices*.

The Example Product

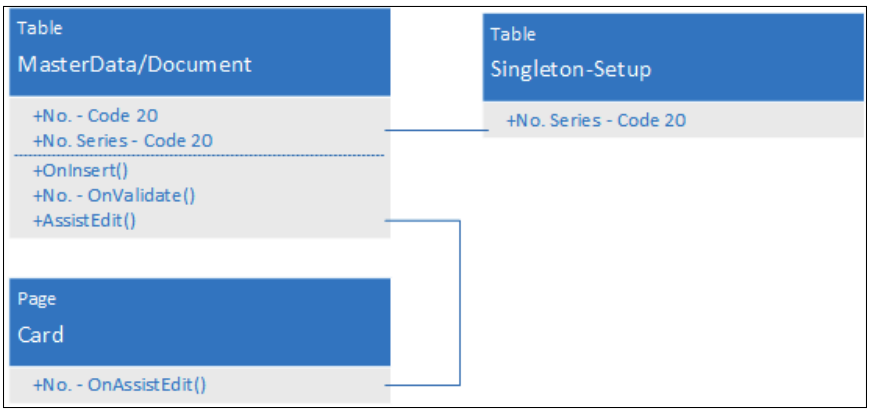
This exercise can be repeated for the Example Product Master Data. By simply adjusting the elements from people to things we will change the flavor of the Pattern. We will use the ID `50020` for the table and `50020` and `50021` for the pages, as shown in the following screenshot:



Using the same Pattern illustrates the power of using Patterns to create recognizable software.

Number Series

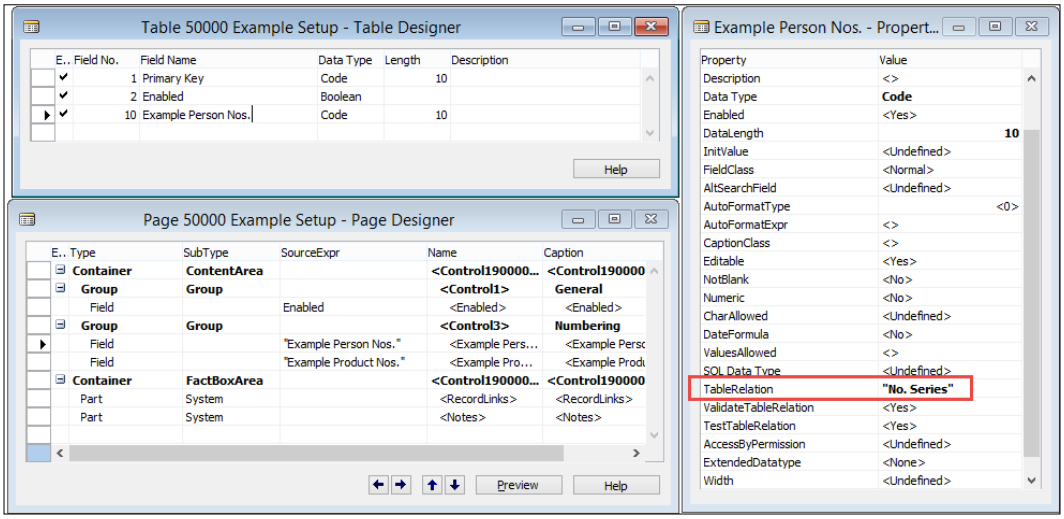
When using the Master Data Architectural Pattern, there are quite a number of Design Patterns that we can use to add functional behavior to our software. One of them is Number Series. Let's see how this can be done by looking at the diagram:



We will implement the Pattern on the Example Person table by adding the elements one by one. Although we can perform some of the steps in different orders, there is a logical order that enables the objects to be compiled each time.

Adding Setup fields

The first step is to add a field that contains a reference to which Number Series we want the application to use. This is typically done in a setup table. We will use the Example Setup table that we've created in step one of this chapter.



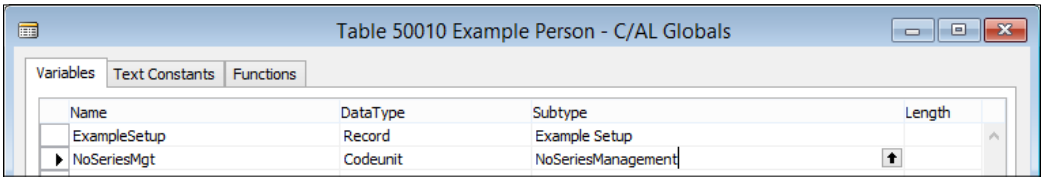
The field is of the type **Code**, has a length of **10**, and has a table relation with the No. Series table `308`. This is typical for a lot of fields in Dynamics NAV that relate to the Subsidiary or Supplementary tables. On the Setup page, we can add a group for all the numbering fields, since there is a fair chance that we will need more than one.



References to Number Series are always called `Nos.`, which is short for **Numbers**. When Number Series is used, we always use `No.`, which is short for **Number**.

Adding variables

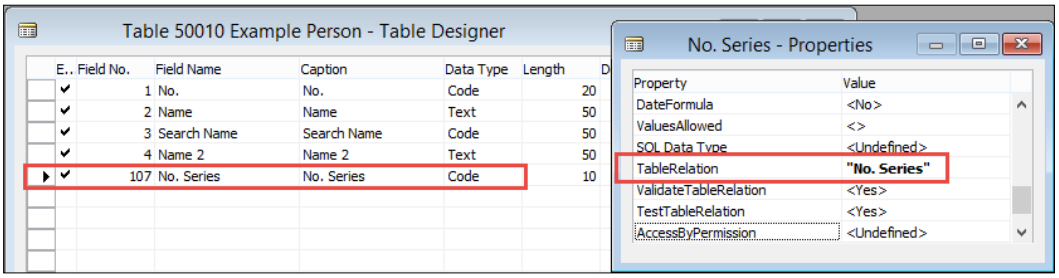
If we choose to implement the Pattern exactly as done in many Microsoft implementations, we need to add two global variables to our **Example Person** table. In the next chapter, *Chapter 5*, *Coding Best Practices*, we will discuss how to avoid using global variables. However, for the purpose of creating recognizable software, we will use global variables.



We need references to our Example Setup table and to the **NoSeriesManagement** Codeunit.

Adding the No. Series field

When reading the UML diagram, we can see that we need to add a new field to our Master Data table. This field is called `No. Series`, and it also has a relationship with the No. Series table (308). From a functional perspective, this field will remember which No. Series was used when creating a new record.



This field is **Field No. 107**. The numbering is critical here, since the Master Data of subtype people uses the TRANSFERFIELDS command when synchronizing with Contacts. We will discuss this command in the next chapter.

If we need more than one `No. Series field` in one table, we have to specify the functional use of the field, for example, Posting No. Series.

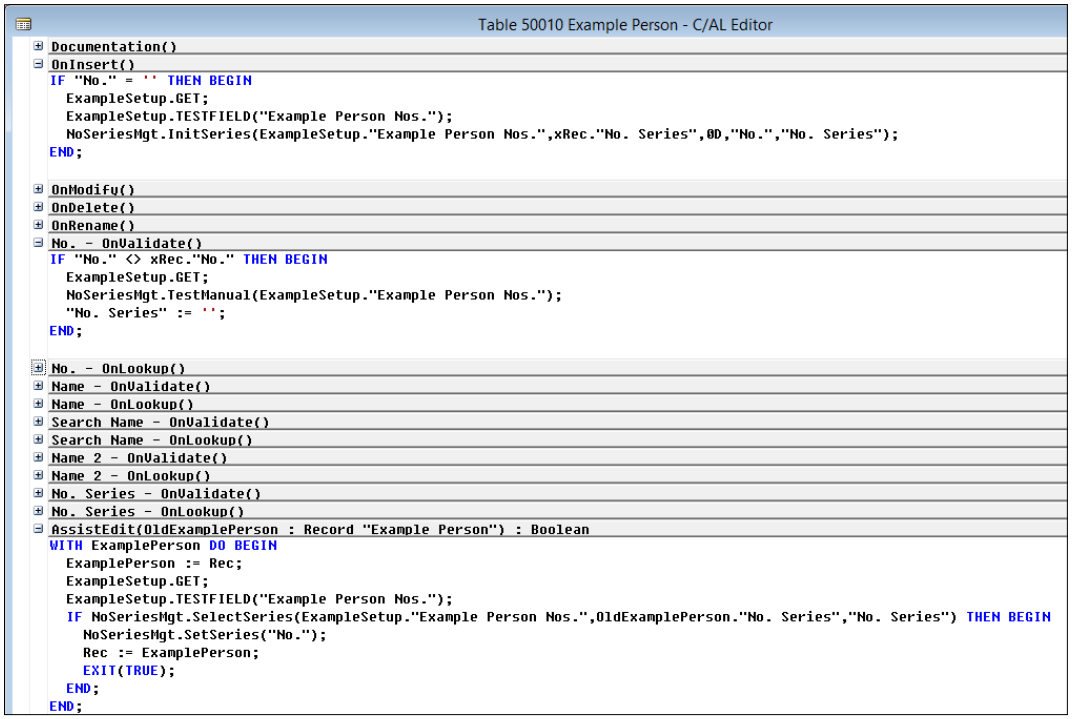


Before continuing with the steps, first close the table and reopen it. This enables the compiler to know about the new field.

Code snippets

When implementing the No. Series Pattern, we need a total of four C/AL code snippets. Three of them are in the table, the fourth is on a Page trigger.

Let's start with the table. This screenshot shows all the snippets:



The first `C/AL` snippet is in the `OnInsert` trigger, and calls out to the `NoSeriesManagement` Codeunit that generates the `No.` value.

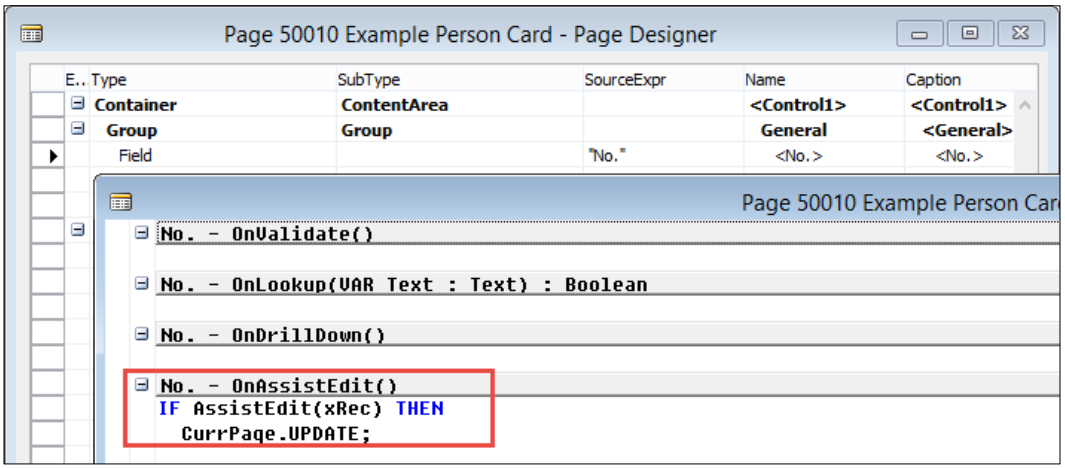
The second `C/AL` snippet is called from the `OnValidate` of the `No.` field. It verifies whether the user is allowed to manually assign numbers using a method from the `NoSeriesManagement` Codeunit.

The last table element is a new function with the name `AssistEdit`. This function is called from the `OnAssistEdit` trigger of the `No.` field of the Card Page. This code allows users to select from multiple No. Series Lines if they exist.



There are two implementations of the `OnAssistEdit` trigger that behave the same from a functional perspective. An alternative implementation can be found in the Item table (27).

The following screen capture shows calling the new `AssistEdit` function from the `No.` field on the Page. This is the fourth and last C/AL code snippet, and it finalizes the implementation of No. Series on our Master Data.

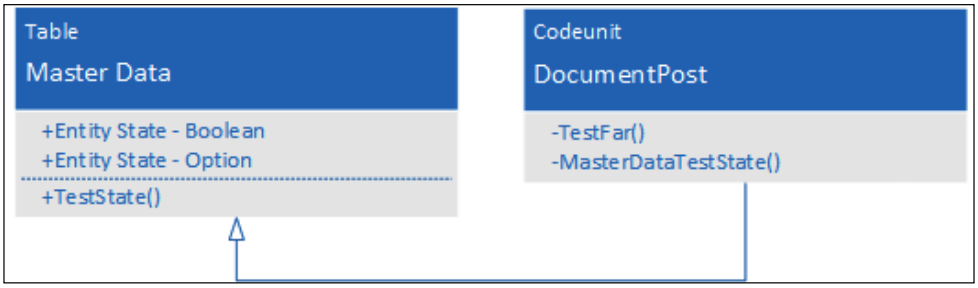


Entity State

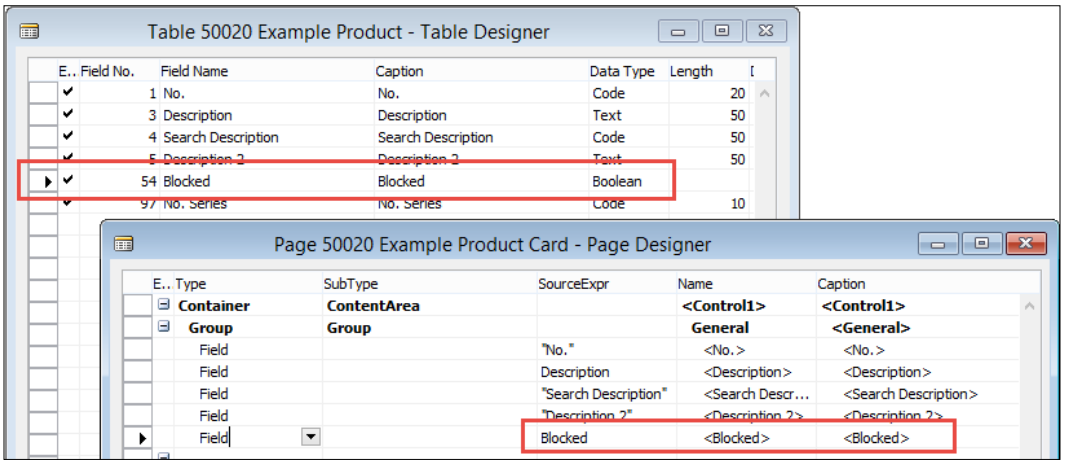
The Entity State Pattern allows us to understand something about our records. This can be a wide range of information, and a well-known example is adding the blocked state.

Let's implement this on our Example Product table. We will use the Boolean implementation. This can be used when the state is either true or false.

The following screenshot illustrates how the state fields can be added and how to add functions to test the state:



To implement this Pattern we will have to add the field, and add a function to test the state. Our field is called **Blocked**, as shown in the following screenshot. It explains the functional implementation of the state.

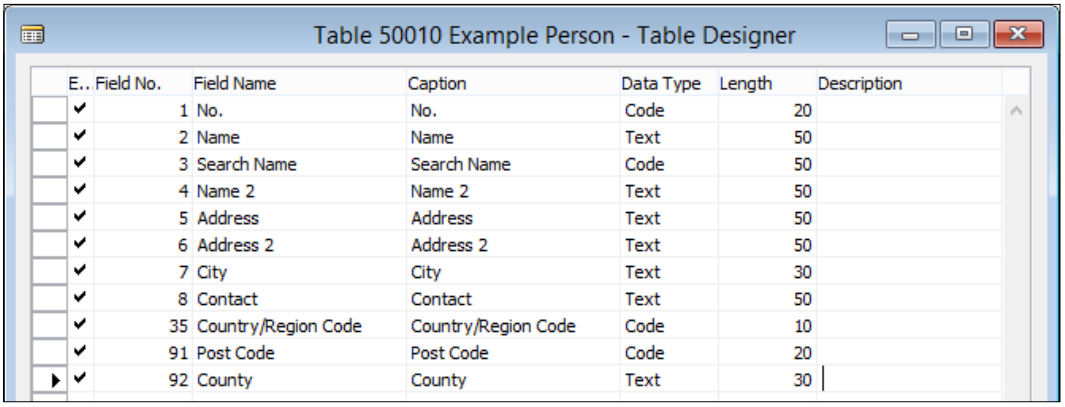


Since our state field only allows two values, the function is relatively simple.

```
TestNotBlocked()
TESTFIELD(Blocked, FALSE);
```

Implementing the Address Integration Pattern

We can implement the Address Integration pattern on our **Example Person** table. This involves adding the fields that allow us to enter an address as well as adding an option to the Format Address `Codeunit 365` to print the address:



The fields `Name` and `Name 2` already exist in the **Example Person** table, we need to add the other fields as described in *Chapter 3, Design Patterns*.



By copying and pasting the fields from another table we ensure the correct field names & numbers, table relations and validation code.

The fields `City` and `Post Code` also have cross reference checking using the `Post Code table 225`. In order for this to be enabled, we need to declare the Post Code table as global variable. The following screenshot shows the validation logic in the `City` and `Post Code` field:

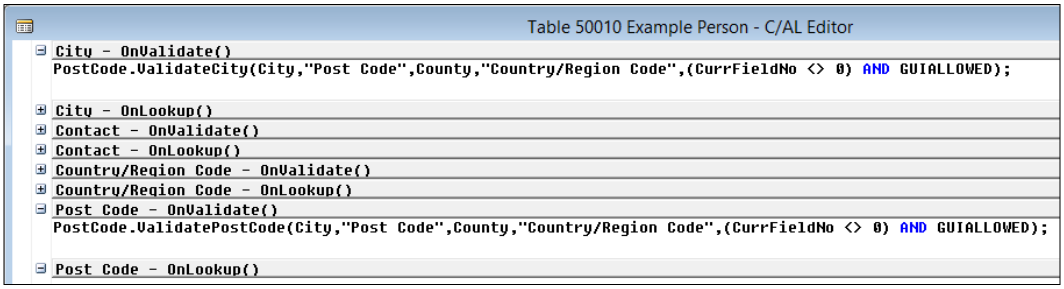
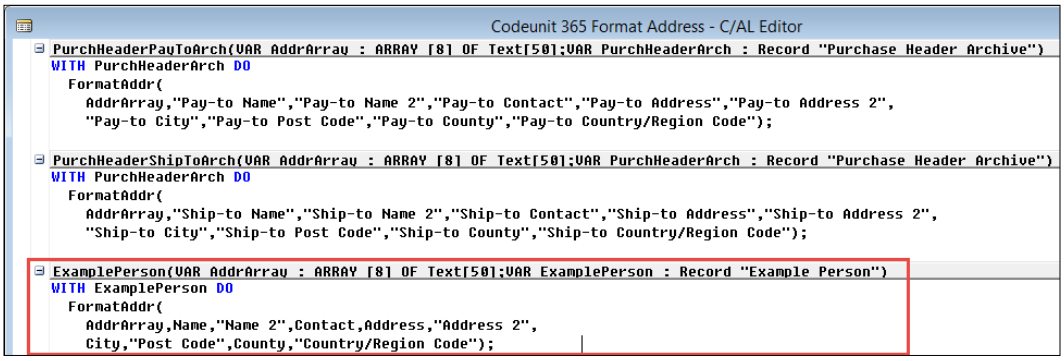


Table 50010 Example Person - C/AL Editor

```
City - OnValidate()
PostCode.ValidateCity(City,"Post Code",County,"Country/Region Code",(CurrFieldNo <> 0) AND GUIALLOWED);

City - OnLookup()
Contact - OnValidate()
Contact - OnLookup()
Country/Region Code - OnValidate()
Country/Region Code - OnLookup()
Post Code - OnValidate()
PostCode.ValidatePostCode(City,"Post Code",County,"Country/Region Code",(CurrFieldNo <> 0) AND GUIALLOWED);

Post Code - OnLookup()
```

The address fields should also be added to the card page, and if required, also on the list page.

To enable printing of the address we need to add a function to the Format Address `Codeunit 365`. Each table that needs address fields to be printed has a similar function in this `Codeunit`. The following screenshot illustrates our function as compared to two existing functions:



Codeunit 365 Format Address - C/AL Editor

```
PurchHeaderPayToArch(VAR AddrArray : ARRAY [8] OF Text[50];VAR PurchHeaderArch : Record "Purchase Header Archive")
WITH PurchHeaderArch DO
  FormatAddr(
    AddrArray,"Pay-to Name","Pay-to Name 2","Pay-to Contact","Pay-to Address","Pay-to Address 2",
    "Pay-to City","Pay-to Post Code","Pay-to County","Pay-to Country/Region Code");

PurchHeaderShipToArch(VAR AddrArray : ARRAY [8] OF Text[50];VAR PurchHeaderArch : Record "Purchase Header Archive")
WITH PurchHeaderArch DO
  FormatAddr(
    AddrArray,"Ship-to Name","Ship-to Name 2","Ship-to Contact","Ship-to Address","Ship-to Address 2",
    "Ship-to City","Ship-to Post Code","Ship-to County","Ship-to Country/Region Code");

ExamplePerson(VAR AddrArray : ARRAY [8] OF Text[50];VAR ExamplePerson : Record "Example Person")
WITH ExamplePerson DO
  FormatAddr(
    AddrArray,Name,"Name 2",Contact,Address,"Address 2",
    City,"Post Code",County,"Country/Region Code");
```

This function calls out to a generic function (`FormatAddr`) that compresses the address fields into a text array with a length of `8`.

From the perspective of upgradability and maintainability, this is not considered good architecture. In the next chapter, we will discuss the alternatives.

Step three – Journals & Entries

Now that we have Master Data, we can start recording transactions that are linked to the master data. We will implement this using an Example Journal and Example Entries. The Entries will be linked to both the Example Person and the Example Product table.

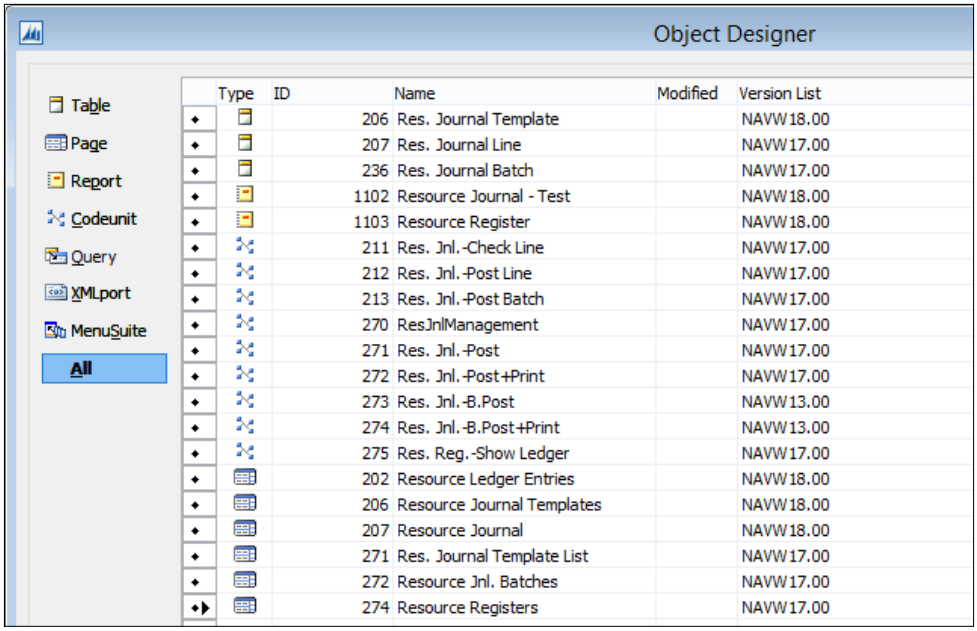
Creating your own Journal and Entry

When creating a Journal we need a set of Tables, Pages, Codeunits and Reports. There are quite a few Journals in Microsoft Dynamics NAV with different levels of complexity, but they all have the same basic structure.

Creating a brand new Journal is not easily documented step-by-step, and would cost a considerable page count in this book. For most architects, creating a new journal is not a frequent task. Even in the standard Dynamics NAV application, new journals are added only in every few big releases.

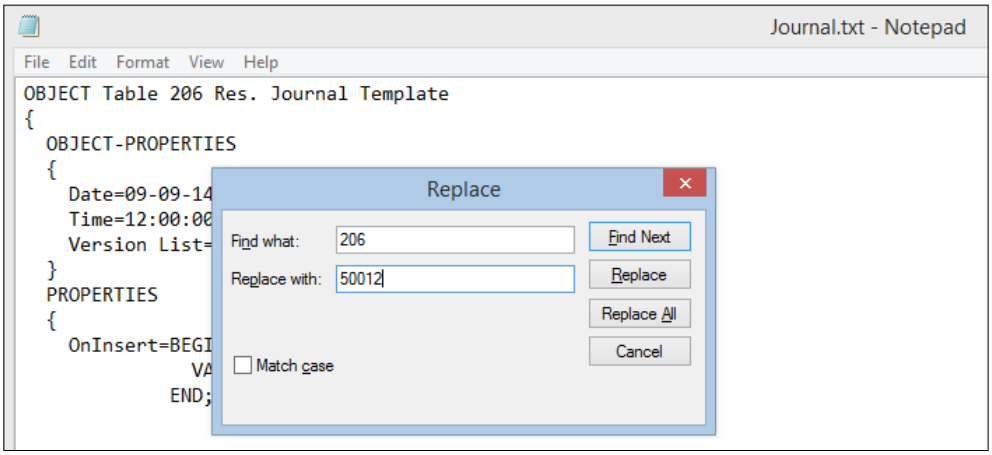
When creating a new Journal in Dynamics NAV there is a trick that you can use, but it requires basic knowledge of the syntax that is used for external object text files.

In Microsoft Dynamics NAV the Resource Journal & Entry structure is very isolated and close to the Pattern definition that we use. We can mark the following objects in the object designer as illustrated in the next screenshot, and export them as a text file:

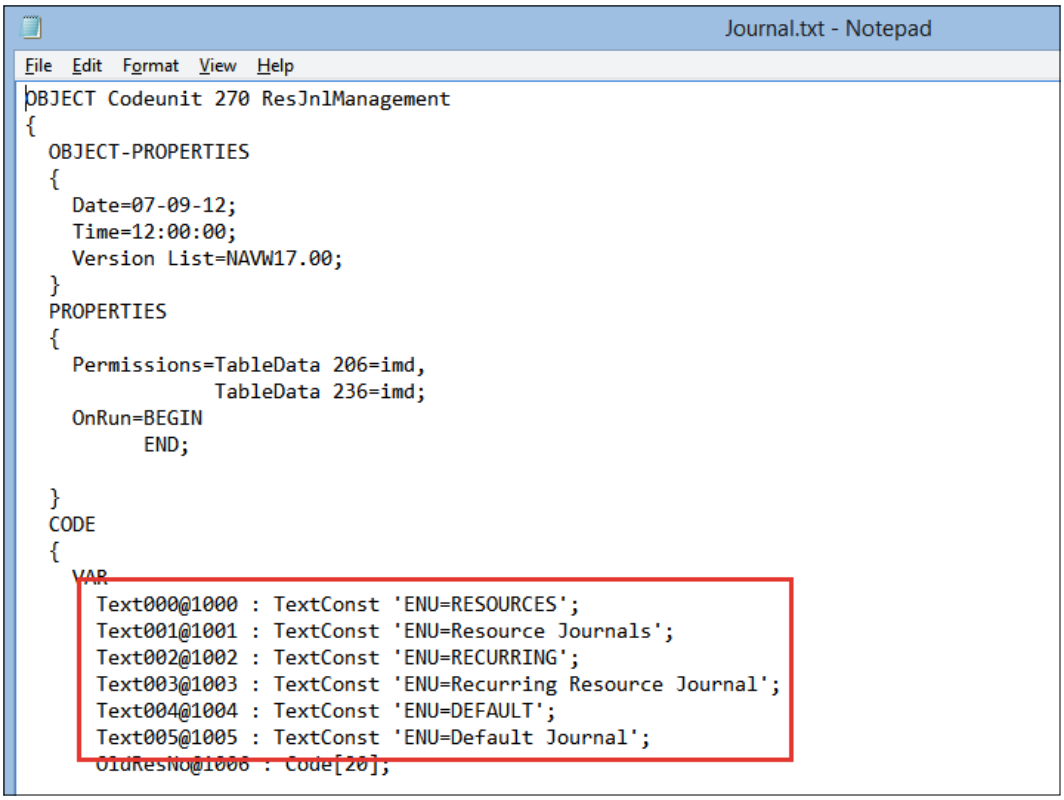


Using any text editor, we can then use the Find and Replace to create our own journal and entry using the following steps:

1. Create a new object number structure for your add-on and replace the numbers. Remember to do this in an intelligent way, as the number you are replacing might be used for other purposes than object numbering:



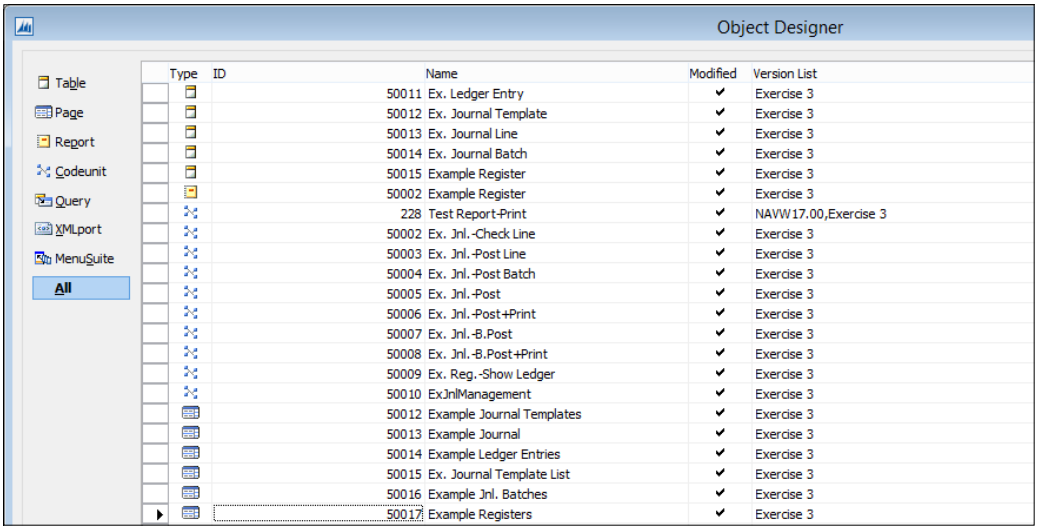
2. Find and Replace the object names with your names. The Resource Journal uses the `Resource` and `Res` names. In our Journal we will use `Example` and `Ex`.
3. Change Table Relations to your tables.
4. Replace the Text Constants in the `ResJnlManagement` Codeunit:



After importing your text file you can fine-tune your journal. In our example journal we have removed the recurring option to simplify the process.

The Resource Journal also uses the Test Report-Print `Codeunit 228`. This Codeunit needs to be modified to support the Example Journal.

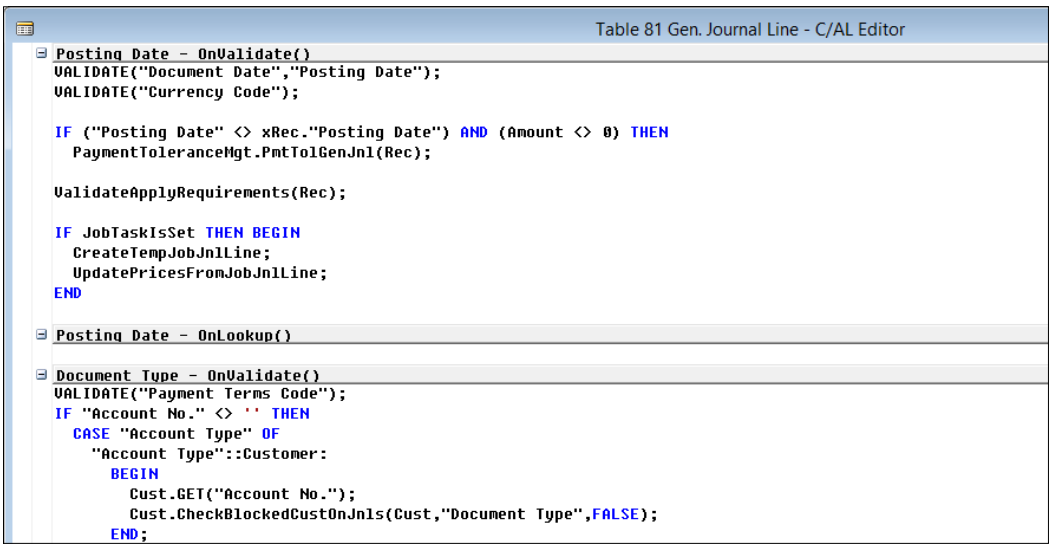
5. After importing the text file and cleaning it up, we should have the following objects:



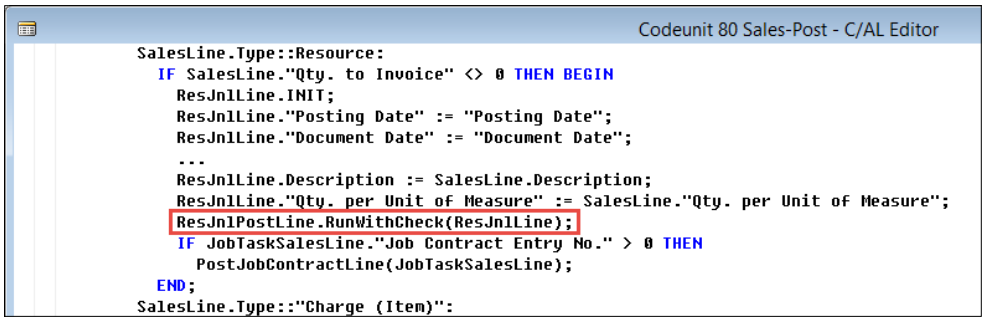
Validation in Journals

Although many Journal Line tables in Dynamics NAV contain `C/AL` code in the `OnValidate` triggers of the table, this code is only intended to be used when the table is used from the user interface as a journal.

An example of `OnValidate` code in the General Journal Line is displayed in the following screenshot:



Journals are also used during the posting of Documents and other processes in NAV. During these processes the Journal Line table acts as an Argument table as illustrated in the following screenshot of `Codeunit` Sales-Post (80):



During these processes, the data is already validated during the entering of the Document and tested in the Test Near and Test Far parts of the posting routine.

Step four – Documents

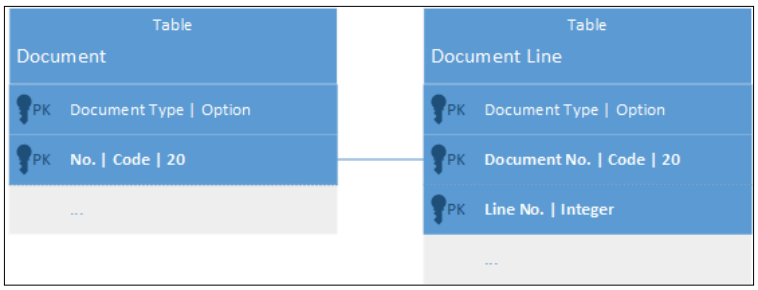
When designing applications in Microsoft Dynamics NAV, the Documents allow us to implement a functional workflow that resembles the traditional paper flow in a company. From a pure accounting and auditing perspective, we can run the application without Documents and have the same entries as a result in the database.

Documents act as containers of functionality for end users to be able to work with the application in a human way.

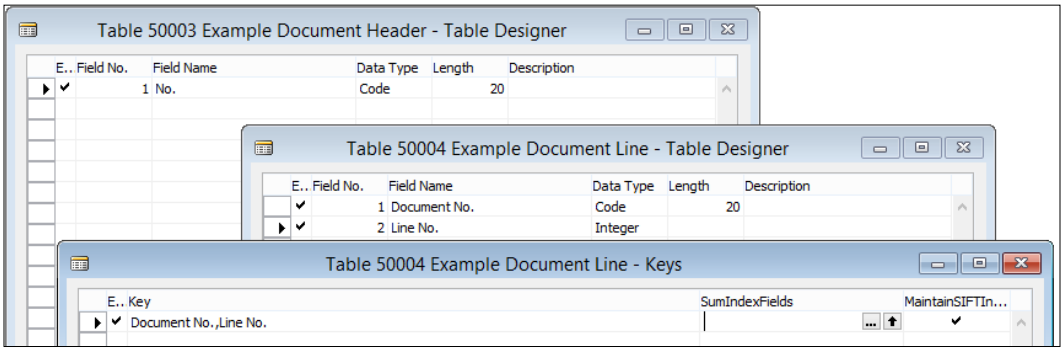
Let's add a Document in our example application and see how we can connect this to our journal during a posting process.

Implementing a Document

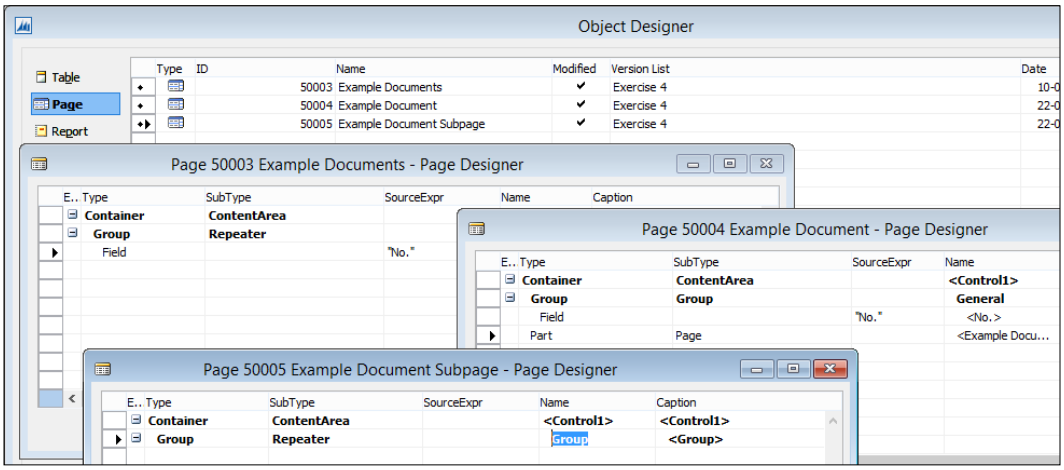
To implement a Document we must follow the following Diagram. Remember that the Document Type is optional.



For our **Example Document** the table structures look as displayed in the following screenshot:



We also need a set of page objects. A list and a card page, as created for the Master Data, and a page for the lines as subpage for the card:



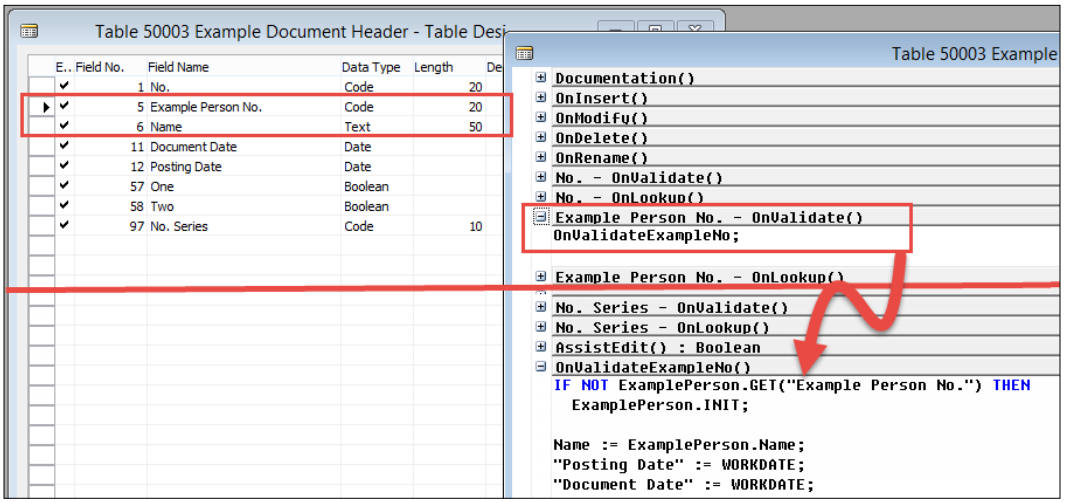


Please note that although Microsoft internally still uses the term **Subform**, we will call ours Subpage. Subform was a term used in the Classic Client in Dynamics NAV 2009 and earlier versions.

We now have the basic structure that we need to create a document, but in order to be able to implement the Example Journal as part of a posting process we also need to add references to our Master Data. We will reference our Example Person as part of the Document, and the Example Product as part of the Document Line. This way we can create an Example Journal Line for each Document Line when posting a Document.

To implement these Master Data we need to implement fields with a reference to the primary key as well as inherit details that are required for our process.

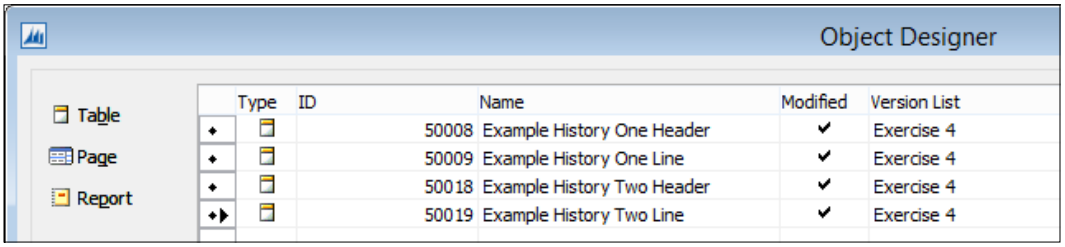
The following image illustrates the **Document** table with the Master Data fields and Validation Code, as well as some additional fields:



Document history

When Documents have a limited life expectancy, we can implement a posting routine to remove the document and create the next document. An example of this process is posting a Sales Order into a Posted Sales Invoice. During these routines, we can create additional historical information, such as entries using the journal as a contract.

In our abstract example model we will create two of these history tables. We will call them `Document History One` and `Document History Two`. They are abstract names. Examples in NAV would be the Posted Purchase Invoice and Posted Purchase Receipt.

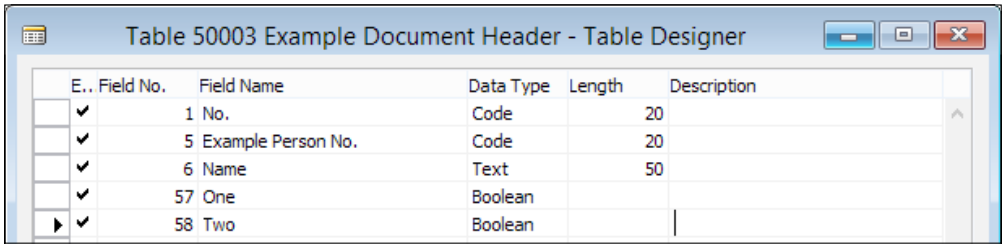


Selecting a Pattern

For our posting routine to be able to understand if it has to create a record in **History One**, **History Two** or both, we will implement the Select Pattern.

This Pattern allows us to parse extra parameters to a `Codeunit` using the `OnRun` trigger by adding fields to the table that act as placeholders.

The fields never appear on the user interface:

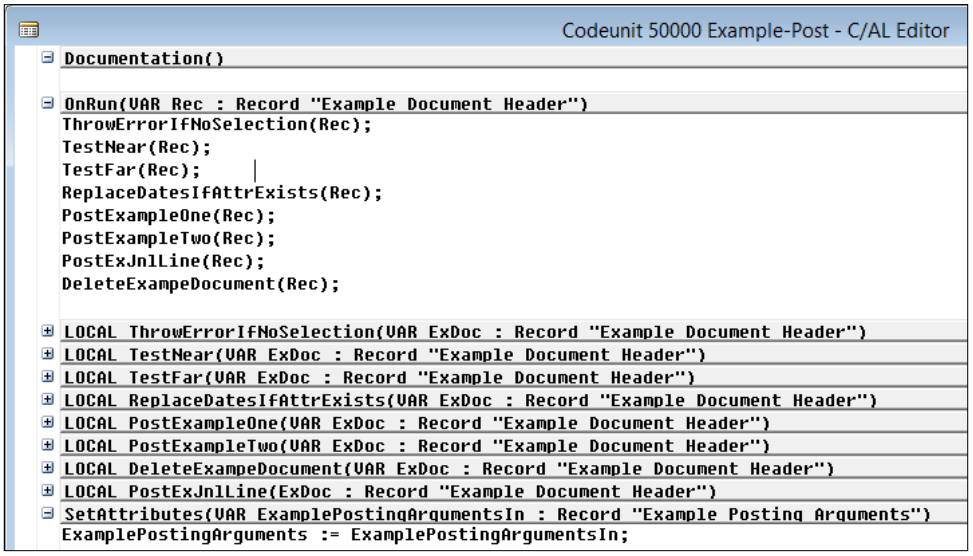


Other Patterns

There are other Patterns that implement nicely with the Documents such as No. Series, Address Integration and Entity State. They are implemented in the Example Objects that can be downloaded as part of this book.

Posting

Let's have a look at the posting routine that we can create to move data from the Documents to the Document History while creating Example Entries:



The **Codeunit Example-Post 50000** is engineered with a concept called Natural Language Programming that we will discuss in the next chapter, *Chapter 5*, *Coding Best Practices*. The `OnRun` trigger does not contain any business logic code, it only explains in readable English what steps will be processed. Let's look at some of the steps.

The ThrowErrorIfNoSelection function

When the Codeunit is called, we need to know which history document to create. This is done using the Select fields. If none are selected, we need to throw an error:

```
WITH ExDoc DO
  IF NOT (One OR Two) THEN
    ERROR(
      NoSelectionError,
      FIELDCAPTION(One), FIELDCAPTION(Two));
```

The TestNear function

Before starting I/O to the SQL Server, we need to do some basic testing to see whether the Documents contain mandatory data. This is preferably done in the `TestNear` function:

```
WITH ExDoc DO BEGIN
  TESTFIELD("Example Person No.");
  TESTFIELD("Posting Date");
END;
```

The TestFar function

When the data in the Documents are valid, we need to check if the relevant setup is correct. This is done in the `TestFar` function:

```
IF GenJnlCheckLine.DateNotAllowed(ExDoc."Posting Date") THEN
  ExDoc.FIELDERROR("Posting Date", DateNotAllowed);
```

The PostExampleOne/PostExampleTwo function

This is where we create the History Documents using `TRANSFERFIELDS`; the following is the code:

```
WITH ExampleHistoryOne DO BEGIN
  TRANSFERFIELDS(ExDoc);
  INSERT;
END;
```

```
WITH ExampleDocumentLine DO BEGIN
  SETRANGE("Document No.", ExDoc."No.");
  IF FINDSET THEN REPEAT
    ExampleHistoryOneLine.TRANSFERFIELDS(ExampleDocumentLine);
    ExampleHistoryOneLine.INSERT;
  UNTIL NEXT = 0;
END;
```

The PostExJnlLine function

This function creates the Entry based on the Master Data in the Document. This is done using the Journal line as a contract:

```
WITH ExDoc DO BEGIN
  ExampleDocumentLine.SETRANGE("Document No.", "No.");
  IF ExampleDocumentLine.FINDSET THEN
    REPEAT
      ExJnlLine.INIT;
      ExJnlLine."Posting Date" := "Posting Date";
      ExJnlLine."Document Date" := "Document Date";
      ExJnlLine."Example Person No." := "Example Person No.";
      ExJnlLine."Example Product No." :=
        ExampleDocumentLine."Example Product No.";
      ExJnlLine.Description := ExampleDocumentLine.Description;
      ExJnlLine.Quantity := 1;
      ExJnlPostLine.RunWithCheck(ExJnlLine);
    UNTIL ExampleDocumentLine.NEXT = 0;
END;
```

The DeleteExampeDocument function

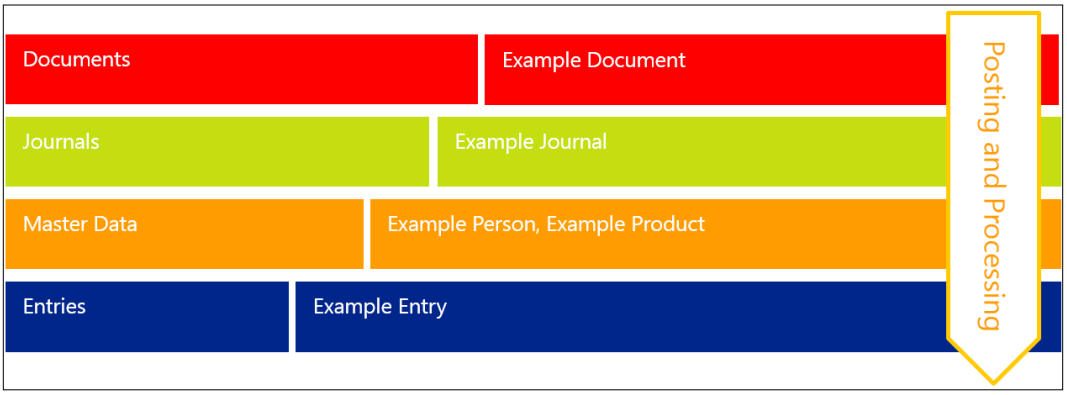
At the end of our routine we can delete the document by implementing the following:

```
WITH ExDoc DO
  DELETE;
```

The layered model

We have built an application in four steps by implementing Architectural Patterns and adding Design Pattern elements.

This allows us to look at our model from a top-down perspective, as shown in the following figure:



The entries are the heart of our application that act for analysis and reporting. They are related to Master Data, the people we do business with and the products we use. Journals allow us to implement rules and manage transactions. Documents are the workflow elements that allow people to use the application in a way that makes sense.

In Microsoft Dynamics NAV, this model has one extra layer, the processes, that we've out scoped in this chapter for reasons of simplicity.



For the processes that have multiple documents related and/or more static, Dynamics NAV has processes. From a Pattern perspective, they use the same elements as Master Data.

Summary

In this chapter we have successfully combined the Patterns we have learned in the previous chapter into an abstract application. We also combined Architectural Patterns and Design Patterns to create functional behavior, and learned how the Architectural Patterns have a hierarchy in a layered model, allowing users to work with the system in an understandable way.

These elements can be used over and over again, and will always result in a software that is recognizable as Dynamics NAV and maintainable by people who understand these Patterns. This makes Patterns a very powerful set of knowledge when designing applications in Microsoft Dynamics NAV.

In the next chapter, we will look at coding best practices that will help you design software that others can maintain and is easily upgradable to newer versions.

5
Coding Best Practices

In the previous chapters, we learned about Patterns that help us design objects and object elements. Now, we will start adding code to make them work together.

There are Design Patterns about structuring code, but most of the things that we will discuss in this chapter are considered best practices for writing code. Some of them are fundamental basics of object-oriented programming (OOP), such as Encapsulation, although Encapsulation is older than OOP and applies to Microsoft Dynamics NAV as well.

Good coding practices will improve the maintainability of your application, and make it easier to work with a team of developers on the same application, or share the code outside of your team. The latter is very common in the Microsoft Dynamics NAV ecosystem, and starts with Microsoft sharing their code with NAV partners. Partners then again share the code with other partners and also customers. Microsoft Dynamics NAV is a popular ERP system with customers who require a flexible system that can be easily customized.

Coding guidelines

Best practices for writing code in any language start with a set of programming guidelines. This is important for any development team, but crucial for projects that share code with different teams, such as Dynamics NAV.

The programming guidelines for C/AL have hardly changed in 30 years. For Navision Software A/S and later, Microsoft has always had very clear coding guidelines. Throughout the lifecycle of the application, they have been very consistent.

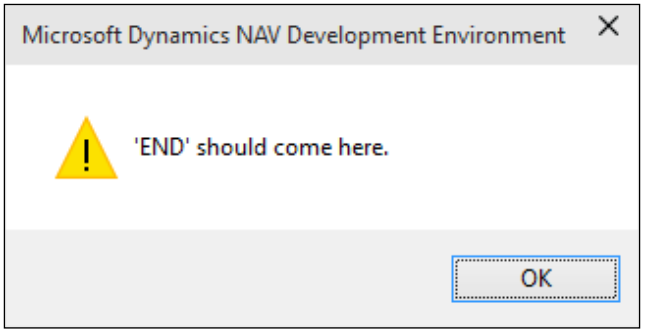
In January 2015, Microsoft released an updated set of guidelines, based on NAV 2015. We will discuss the most important ones.



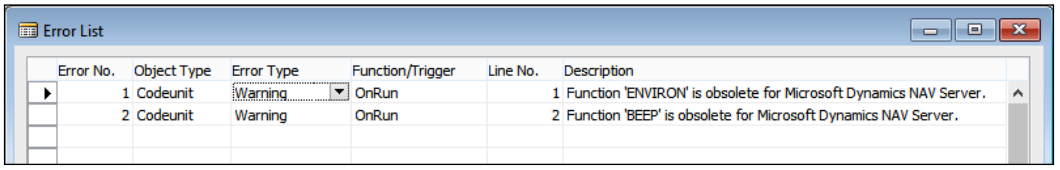
You can find the guidelines at `https://community.dynamics.com/nav/w/designpatterns/156.cal-coding-guidelines`, and `https://msdn.microsoft.com/en-us/library/ee414237(v=nav.80).aspx`.

Code design

The C/AL compiler checks our code for programming errors such as missing object references, and `BEGIN`/`END` or `REPEAT`/`UNTIL` statements that are wrong.



The Microsoft Dynamics NAV Development Environment has a limited warning mechanism that only checks the C/AL functions that used to be part of the **Classic Client**, but no longer worked in the three-tier stack. The examples are `BEEP` and `ENVIRON`:



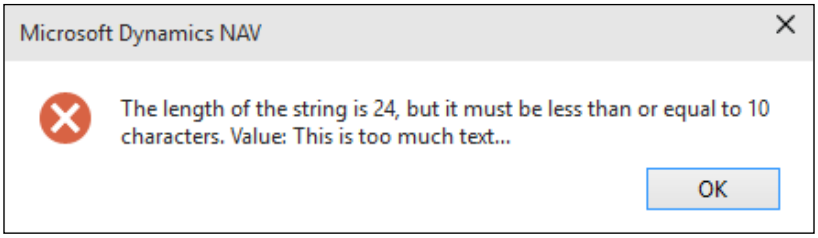
The code design rules are implemented to add an extra layer of rules that are not enforced by the compiler or generate warnings but when applied, they improve the readability of the code.

Unused Variables	Every variable that is declared global, local, or as a parameter should be used. Unused variables should always be removed by the developer.
By Reference	When assigning a variable as parameter, we have the option to declare the variable as By Reference. When this is applied, changes to the variables also apply to the original, since they share the same memory space. In Microsoft Dynamics NAV, the column is called `Var`. By Reference should only be enabled when there is a clear intention of changing the variables.
Match Placeholders	Although C/AL does not support the function with a variable arguments list, there is a limited set of commands that allow the use of multiple parameters. The examples of these functions are `MESSAGE`, `ERROR`, and `STRSUBSTNO`. When applied, these functions should be called with placeholders in a text string called `%1`, `%2`, and more. These placeholders should always match but are not checked at compile time or runtime.

Set Variables	Variables should be used in business logic. If variables are set to a specific value but don't influence the decision tree, they should be removed. For example:
	`Count := 0;`
	
	`Vendor.SETFILTER("No.",FilterStr);`
	`IF Vendor.FINDSET THEN`
	`  REPEAT`
	`    "User ID" := USERID;`
	`    "Vendor No." := Vendor."No.";`
	`      IF INSERT THEN`
	`        Count += 1;`
	`  UNTIL Vendor.NEXT = 0;`
	This is an example of a potential boat anchor anti-pattern. We will discuss more about anti-patterns in *Chapter 6, Anti-patterns and Handling Legacy Code*.

Variable capacity

Within C/AL there is no compile time to check for a possible overflow scenario. An overflow can occur when a large string is moved in a variable or field with limited storage capacity. The following screenshot shows an error when the length of the string usage is exceeded:



Since Dynamics NAV version 2013 R2, it is possible to have Text variables with unlimited length, which are mapped to the `DotNet` string datatype. However, when writing to the database, the maximum length of a string is 250 characters.

The Cyclomatic Complexity

When programming conditions such as `IF/ELSE` or `CASE/OF`, and `FOR/DO`, we add complexity to our code.

The **Cyclomatic Complexity** is a methodology to calculate the complexity of your C/AL functions. There are multiple officially-documented algorithms, and for Dynamics NAV, we can use **Cyclomatic Complexity Three**, or **CC3**.

This means that each `IF/ELSE/CASE` and `FOR` condition gets one point. After adding up all the points, a function gets a score. The higher the score, the more complex the function is and therefore, high scores should be avoided.



You can find more information on Wikipedia at `https://en.wikipedia.org/wiki/Cyclomatic_complexity`.

Internally, Microsoft programmers working on the product use a threshold of 25 points. I would advise you to use a maximum value of 10.

The function size

Each function that we write in Dynamics NAV, whether it is a built in trigger or manually added function, should not exceed the size of a normal modern computer screen.

When using Microsoft Dynamics NAV 2015 Development Environment on a HD screen, the maximum size of a function should not exceed 40 lines.

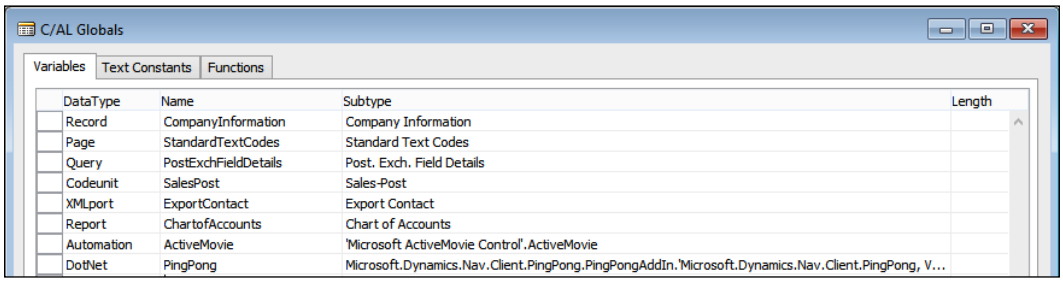


Internally, Microsoft programmers use a threshold of 100 lines. Since a lot of legacy functions in Dynamics NAV have much more than this, another rule internally used at Microsoft is when having to add one line of code to a function that has over 100 lines, two other lines must be moved out, and refactored into a function.

This is considered to be a good methodology for refactoring while avoiding disturbing the upgrade as much as possible.

Using complex data types

To avoid complexity in a single function, there is a maximum threshold of complex data types in C/AL, such as Record, Page, Query, and Codeunit, as shown:

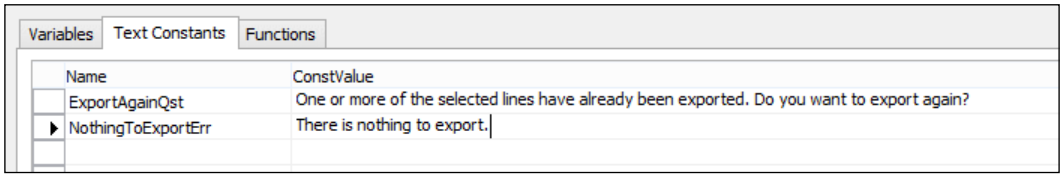




I advise a maximum threshold of 10. Internally, Microsoft developers use 30.

Localizability

Since Navision version 2.01, 2.60, and 3.00, the application can be translated independently of the code. This can be done using the Text Constants. They can be either declared globally or locally. The following screenshot show the **Text Constant** tab:



These Text Constants should always be used, unless the text constant is used internally only by the application, as illustrated in the following screenshot:

```
Documentation()
OnRun()
ShowCreateLineInstruction(ItemNo : Code[20])
DisableMessageForCurrentUser()
SalesCheckAllLinesHaveQuantityAssigned(SalesHeader : Record "Sales Header")
PurchaseCheckAllLinesHaveQuantityAssigned(PurchaseHeader : Record "Purchase Header")
LOCAL IsEnabled() : Boolean
EXIT(NOT MiniUserRemovedInstructions.GET(USERID,GetInstructionCode));

LOCAL GetInstructionCode() : Code[20]
EXIT('LINESINSTRUCTION');
```



Internally, Microsoft always declares the Text Constant as a global variable.

The naming of the Text Constant should be self-explanatory, indicating if it is used as Message, Error, Question, or something else. Errors and Message constants should end with a period, and a Question should end with a question mark always.



The naming conventions for Text Constants have changed since version 2013. Before this, the constants were numbered.

Date formulas

When hardcoded **Date Formulas** are used, they should be surrounded by the <> characters. When this is applied, Date Formulas are interpreted in the ENU language, which is the default language in which Dynamics NAV is shipped.

```
Codeunit 88887 CALCDATE - C/AL Editor
Documentation()

OnRun()
NewDate := CALCDATE('<+1M>', WORKDATE);
```

Code readability

When working with multiple developers on a single codebase and sharing software as open code with external partners, using a generic set of rules regarding readability is important from a maintenance perspective.

This includes a long list of small things such as using spaces, comments, white-lines, BEGIN/END, TRUE/FALSE, indentation, and parenthesis:

Spaces	This keeps statements together when they belong together; add spaces only if readability becomes an issue:
	`IF NOT(Condition) THEN`
	`-NegativeValue`
	`Result := A / B;`
	`Complex := Calculate((A + B) + POWER(C));`
	`Type = Type::Item:`
	`IF FIND('<>=') THEN`

Comments	Add this only if required; comments should start directly after the code with a single space: `ERROR(ThisIsWrongErr); // Show Error`
White lines	Blank lines should only be used when they improve the readability of the function. Remember that adding a blank line is a good indication to break your function into multiple other functions. They should never be implemented as the first line of a function, or within a conditional block.
BEGIN/END	Only use a BEGIN and END statement if this is required for the code to function in the desired way. If not required, never implement it.

TRUE/FALSE	Boolean operators require no use of TRUE when positive. When negative, the operators are required; use NOT:
	`IF ThisShouldReturnTrue THEN`
	`IF NOT SomethingHappened THEN`

IF/THEN	The THEN keyword should always be on the same line as the IF keyword, except in a multi-line IF statement. In this case, the THEN keyword should be on a new line:
	`IF Setup.GET THEN`
	`IF DoSomethingWithALongDescription AND`
	`AlsoSomethingElseWithAnEvenLongerDescription`
	`THEN`

Indentation	Indentation is always two spaces or one tab, following each condition:
	`IF ThisIsTrue AND`
	`  ThatIsTrue`
	`THEN`
	The only exception is the multi-line `IF` statement; in which case, the second statement is indented three spaces:
	`IF ThisIsTrue THEN`
	`  DoSomethingWithIt;`
	`CASE TRUE OF`
	`  A=B+C:`
	`  GoToSchool;`
	`  K=L/M:`
	`  FlyAway;`
	`  X=Y-Z`
	`  EndTheStory;`
	`END;`

Parenthesis	Parentheses should only be used if they cannot be avoided. Functions in C/AL that don't have parameters do not have, and should not be called with empty parentheses `()`.

When applied consistently, these rules will enable the readability of your application, making it easier to share code with people both within and outside your organization.

Variable naming

Although Microsoft Dynamics NAV Development Environment does not have a built-in data dictionary that is enforced by the compiler, it has very strong naming conventions that have been stable throughout the lifecycle of the product:

Complex datatypes	The name of the complex datatype should be the variable name without removing the special characters and spaces; for example: Sales-Post: `SalesPost` G/L Account: `GLAccount` IC G/L Account Import/Export - `ICGLAccountImportExport`

Fundamental datatypes	Simple datatypes should be self-explaining without adding the datatype to the variable name: • Busy: `boolean` • Initials: `text` • NoOfApples: `integer` • Size: `integer` • Amount: `decimal` • Students[1]: `array`
Function Names	A function name should contain a verb and noun. This indicates the intention of business logic: • `CalculateVat` • `FormatAddress` • `PrintInvoice` • `IssueReminder`



In Microsoft Dynamics NAV 2015, the complex variable names are automatically generated, based on the name of the object and the rules.

Self-explaining

When it is required to have multiple instances of the same complex datatype, they should be explained rather than numbered:

```
MoveReservationFromSalesToPurchase(ReservationEntrySales)
WITH ReservationEntrySales DO
  TRANSFERFIELDS(ReservationEntryPurchase)
```

Suffix versus prefix

When working with the C/AL symbol menu or **Intellisense**, using suffixes is easier than prefixes.

Approved abbreviations

Frequently used variable names in Microsoft Dynamics NAV have approved abbreviations. This can enhance the readability of the code, because we avoid very long variable names.



This list contains the abbreviations used by Microsoft Developers, which is available at `https://community.dynamics.com/nav/w/designpatterns/162.suggested-abbreviations`.

Class-Method-Property

The Microsoft Dynamics NAV Development Environment allows us to write code in almost any place within the application. Each object has dozens of triggers that allow writing C/AL code.

If we just write bits and pieces of code everywhere in the objects, the maintainability of the application would be hard for anyone, including the original developer.

To write code in such a way that other developers can find their way requires a set of rules. If we apply these rules, it will be easier for others to see what we did, because we will reduce the number of places where we write business logic.

The Table as a class

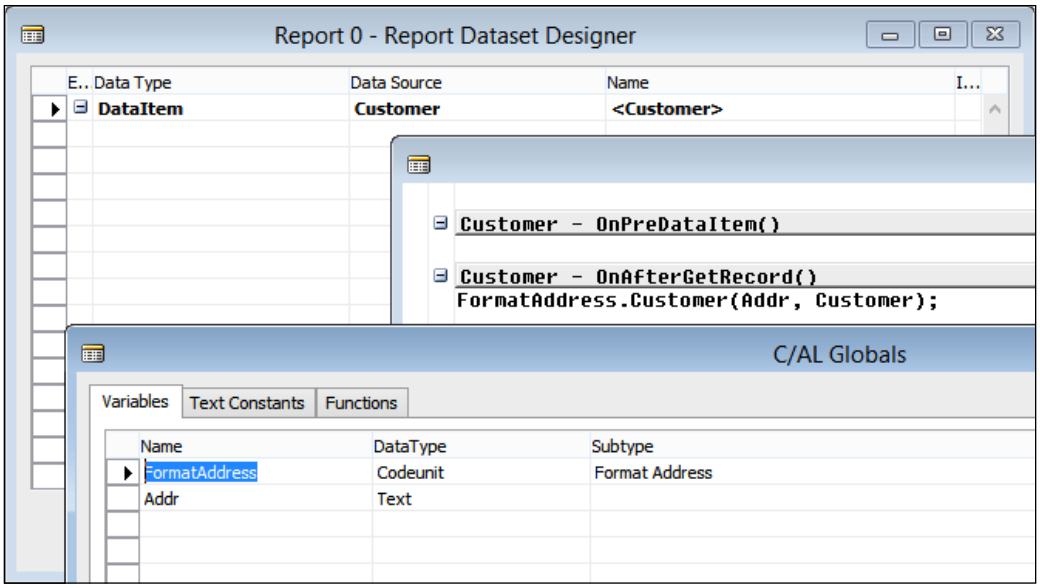
Before we start coding, we need to find the owner of a process. If we explicitly connect the process to the owner, it will be easier for everyone who works with the object to understand what it can do.

Let's try to clarify this with an example. One of the processes that has a lot of owners in Microsoft Dynamics NAV is printing an address. We can print addresses for customers, vendors, documents, and more. In *Chapter 2*, *Design Patterns*, we have learned the Pattern to use and print addresses.

If we want to print a customer address on a report, we have to perform the following steps:

1. First, create a report that uses the Customer as a `dataitem`.
2. Then, declare a global text variable called `Addr` with a length of `50` characters, and an array length of `8` characters.
3. Then, we declare Codeunit 365 as a variable.
4. Finally, we call `FormatAddress.Customer(Addr, Customer).`

The following screenshot shows the **Report Designer** window:



This will return a populated address array that we can use on a report.

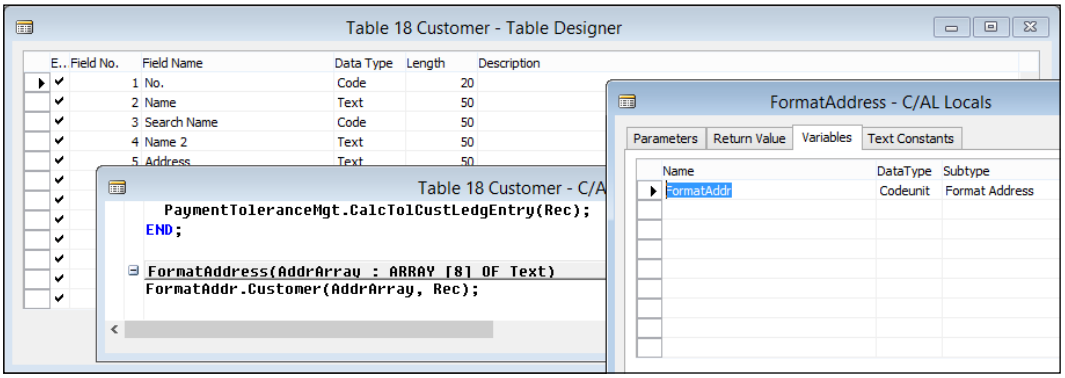
But how do we know that the Codeunit `FormatAddress` exists, and has a function that can be called with a Customer record?

The answer is experience, training, and reverse engineering.

A Codeunit as a method

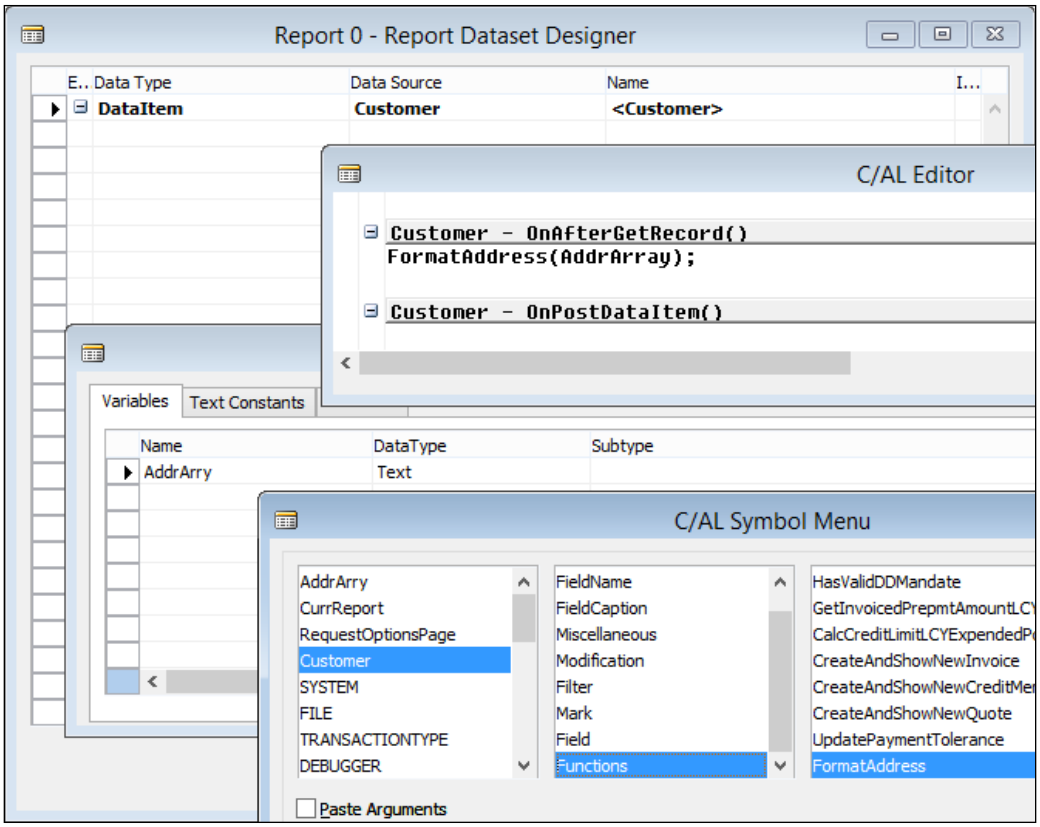
Wouldn't it be nice if we could look at the Customer table, and see what we can do with it?

To do this, we need to implement a new function on the Customer table. Let's call it `FormatAddress`. We'll give it the `Addr` text variable as a parameter. Codeunit 365 can be a local variable in the function, as in the next screenshot:



When we do this, we can use the `FormatAddress` function each time we have access to a customer record without declaring a Codeunit, or even knowing that it exists. If we apply logical naming, and group similar functions together, other developers will find it easier to know about these functions. The suggested naming should always contain a verb and noun.

Now that this function exists, we can simply refer to it from our report without declaring the Codeunit. We only need the Text array.

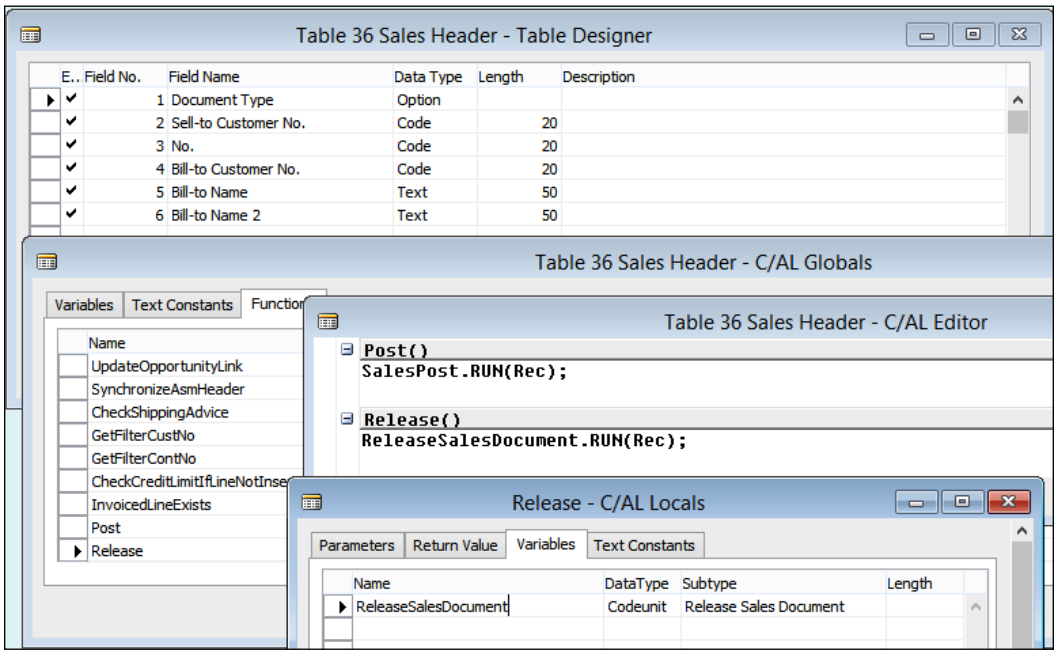


This way of working with functions as a member of the process owner is normal in object-oriented programming, where we will use classes. Using this methodology, we treat an object in Dynamics NAV as a class. We preferably add business logic to a table, and create a Codeunit if the code becomes complex, which typically is when we need more than 10 lines of code.



This video illustrates the use of this concept including some examples:
`https://www.youtube.com/watch?v=d_sGFP40evw`

Another example where implementing this methodology makes sense is working with Documents. Most documents have multiple processes such as release, reopen, post, copy, and more. Most of these processes are in Codeunits in the system. By creating these processes as a function in the table, it would be easy for developers to call them from the user interface.



Function libraries and interfaces

Not every function in Microsoft Dynamics NAV has a clear owner. In some cases, we have a function that can be accessed using simple variables that perform some calculation or a small piece of business logic.

These functions can be put in function libraries or interfaces. By doing this, it is easy for other developers to reuse them.

Examples of function libraries can be string manipulations or date functions. Microsoft Dynamics NAV does not have a lot of built-in functions for this, and many developers have created their own collection of functions through the years.

An Interface is a reusable component that can be accessed from anywhere in the application. A very good example of what could and should be an interface is the function called `DateNotAllowed` in Codeunit 11, Gen. Jnl.-Check Line:

```
IF (AllowPostingFrom = 0D) AND (AllowPostingTo = 0D) THEN BEGIN
  IF USERID <> '' THEN
    IF UserSetup.GET(USERID) THEN BEGIN
      AllowPostingFrom := UserSetup."Allow Posting From";
      AllowPostingTo := UserSetup."Allow Posting To";
    END;
  IF (AllowPostingFrom = 0D) AND (AllowPostingTo = 0D) THEN BEGIN
    GLSetup.GET;
    AllowPostingFrom := GLSetup."Allow Posting From";
    AllowPostingTo := GLSetup."Allow Posting To";
  END;
  IF AllowPostingTo = 0D THEN
    AllowPostingTo := 31129999D;
END;
```

This function uses the Multi-Level setup Pattern to check the User Setup and General Ledger Setup for allowed posting dates. This function is reused in Codeunit 80 and Sales-Post, and is cloned in other functions such as Codeunit 211, Res. Jnl.-Check Line and Codeunit 21, and Item Jnl.-Check Line.

Instead of calling a function from another Codeunit or cloning functions, we could create a new Codeunit `Posting Date Allowed Iface`. This Codeunit only has one global function that does what we expect; or it may even have multiple global functions if we have a requirement for having small variances.

The naming of Codeunits is very important to increase the chance of the functions to be reused. When not reused the result will be code cloning, an anti-pattern, which we will discuss in *Chapter 6, Anti-patterns and Handling Legacy Code*.

Some development environments such as eclipse have plugins that search for code clones. Microsoft Dynamics NAV does not have a plugin as this.

Natural Language Programming

The first thing that many people think of when discussing Natural Language Programming is the very scientific concept of talking to a computer in a natural language and having the code generated automatically. These concepts are important for search engines and artificial intelligence (AI).

However, Natural Language Programming can also help us in writing code that is easier to read, understand, maintain, and even debug. Although it sounds very modern, a lot of traditional programming languages, such as Cobol and SQL use Natural Language to make the code easier to read. Take the following example:

```
select [No_], [Name] from [Contact]
where [Name] like '%Mark%'
```

The C/AL programming language that we use in Microsoft Dynamics NAV is based on Pascal, and is traditionally not considered readable as a natural language. If we want to apply readability to C/AL, we have to implement a concept that we will refer to as **multilevel code**.



This video explains Natural Language Programming for C/AL. You can watch it at `https://www.youtube.com/watch?v=18b710d6nMg`.

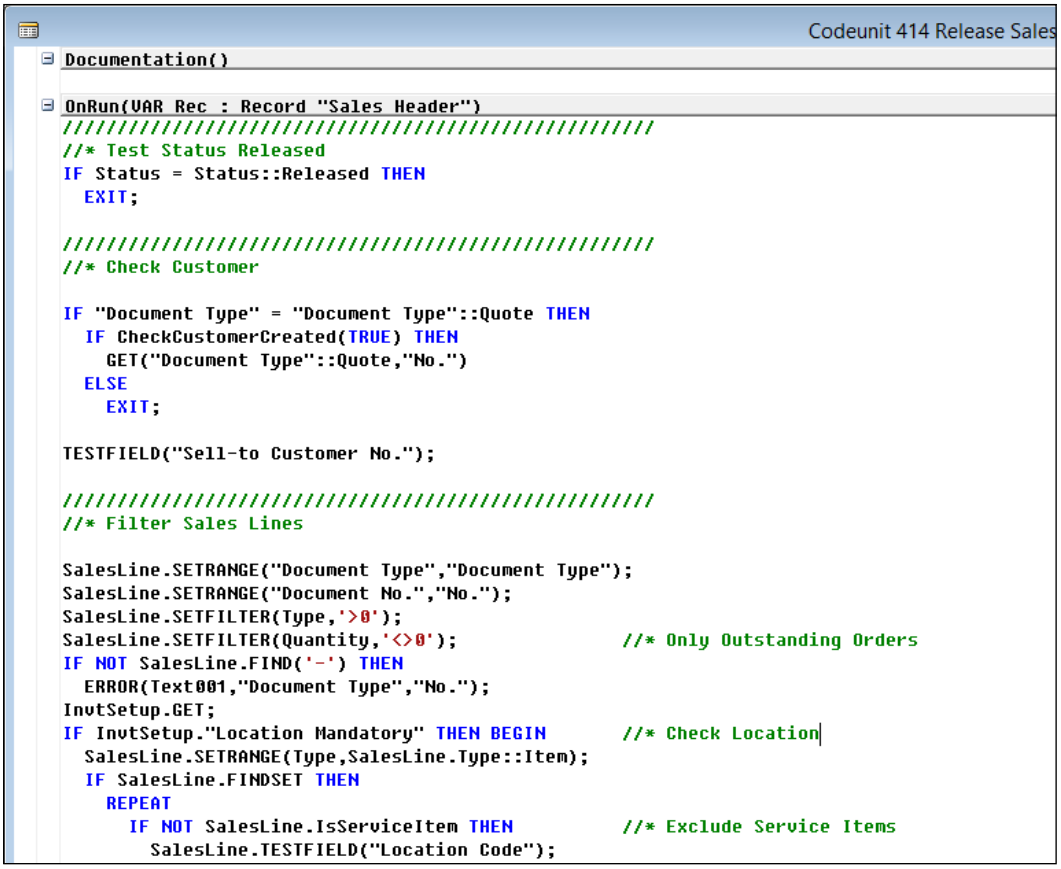
With multilevel code, we will split the technical code, which we will refer to as "nerdy code" from readable functions.

Let's look at an example, releasing a Sales Order, Codeunit 414:

Codeunit 414 Release Sales Document - C/AL Editor

```
Documentation()

OnRun(VAR Rec : Record "Sales Header")
IF Status = Status::Released THEN
  EXIT;

IF "Document Type" = "Document Type"::Quote THEN
  IF CheckCustomerCreated(TRUE) THEN
    GET("Document Type"::Quote,"No.")
  ELSE
    EXIT;

TESTFIELD("Sell-to Customer No.");

SalesLine.SETRANGE("Document Type","Document Type");
SalesLine.SETRANGE("Document No.","No.");
SalesLine.SETFILTER(Type,'>0');
SalesLine.SETFILTER(Quantity,'<>0');
IF NOT SalesLine.FIND('-') THEN
  ERROR(Text001,"Document Type","No.");
InvtSetup.GET;
IF InvtSetup."Location Mandatory" THEN BEGIN
  SalesLine.SETRANGE(Type,SalesLine.Type::Item);
  IF SalesLine.FINDSET THEN
    REPEAT
      IF NOT SalesLine.IsServiceItem THEN
        SalesLine.TESTFIELD("Location Code");
    UNTIL SalesLine.NEXT = 0;
```

This Codeunit does a lot of things that are hard to understand by just reading the code, and intuitively most developers will start adding comments to explain what they are doing and in which step; both for themselves to understand, and for others after them to be able to maintain the code.



However, adding these comments does not make the code easier to read. It adds lines to the function, spreading logic across multiple pages. In this example, the commenting code is an anti-pattern.

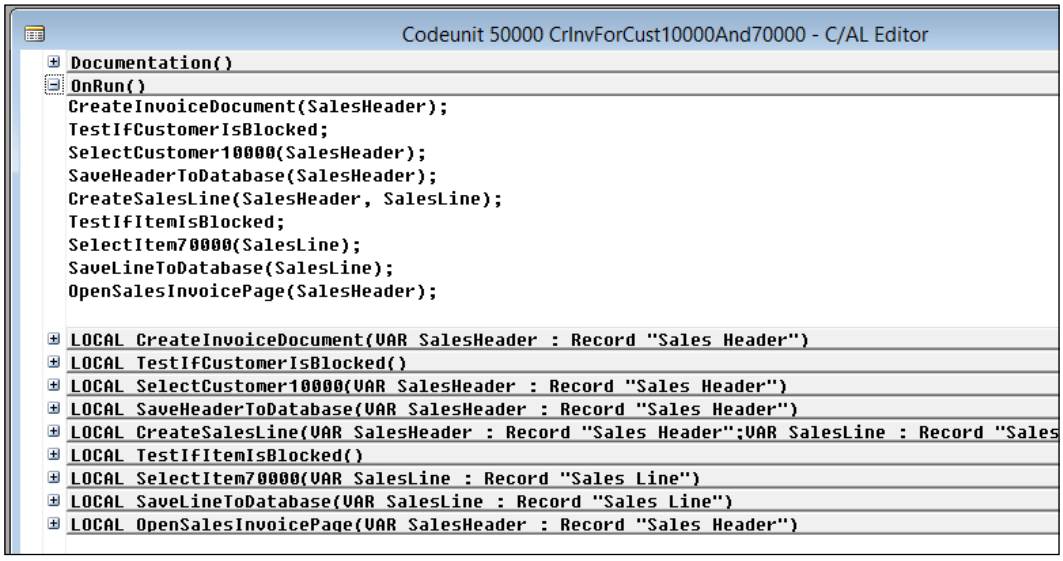
Let's see how we can implement a complex function using Natural Language Programming, and how this makes the code more readable and easier to maintain.

We will implement a relatively straightforward programming task. Create a script to make an invoice for customer number 10000 (The Cannon Group PLC), and item number 70000 (Side Panel).

Before jumping into the code, we can write down the required steps for this function, which are as follows:

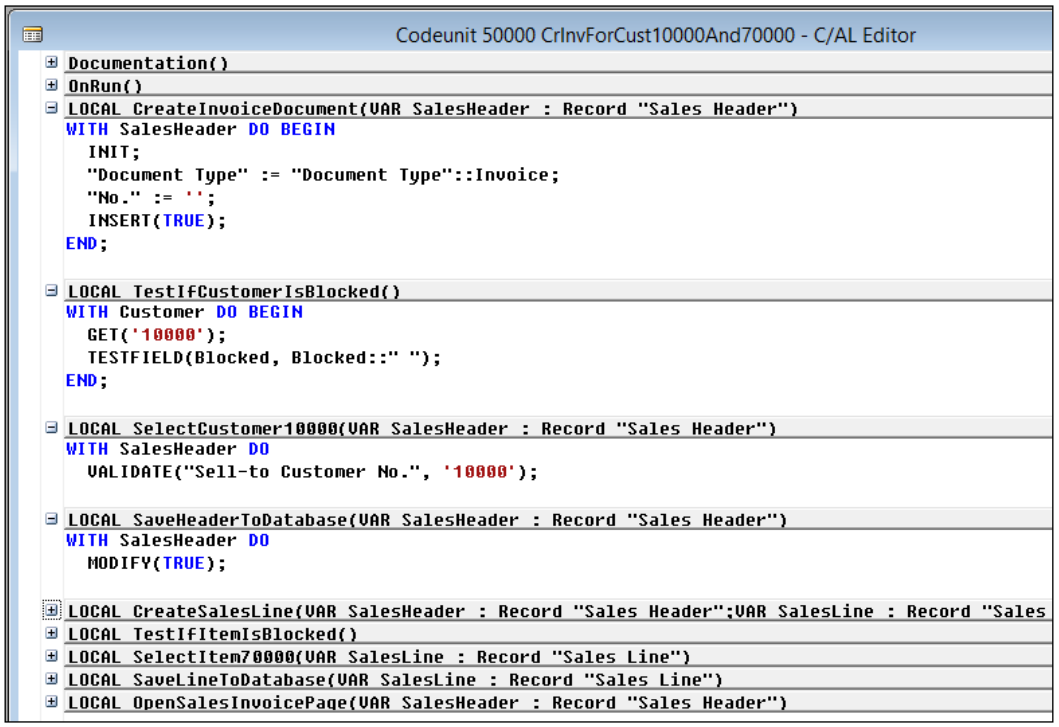
- First, create a New Document.
- Then, test if the Customer is not blocked.
- Read Customer 10000.
- Next, save the Document to the Database.
- Create a Sales Line.
- Test if the Item is not blocked.
- Then, select the Item 70000.
- Save Line to Database.
- Finally, open the Invoice Page.

To implement this, instead of directly writing code we will create a function for each step, such as the following screenshot:



This makes very clear what we want to achieve, and it is easy to read even for non-technical people.

The functions contain the "nerdy code", which is harder to read:



This code can be as simple as MODIFY if we look at the `SaveHeaderToDatabase` function, which means to save the header to the database, which is more readable than the programming term.

Pitfalls

There are a couple of things to realize when starting to implement this methodology.

When we want to change what a function does, we have to check if the name still makes sense. The name of the function explains what we want to do.

If our function requires us to loop through a dataset and perform a function on each of the rows in the dataset, we cannot put all the readable functions in one place if some of the functions have to be inside the `REPEAT UNTIL` loop, which is inside the function.

Since Microsoft Dynamics NAV 2013 a function name, it can contain up to 132 characters, allowing us to write functions as follows:

```
ForEachSalesLineCreateShipmentLineAndItemLedgerEntry.
```

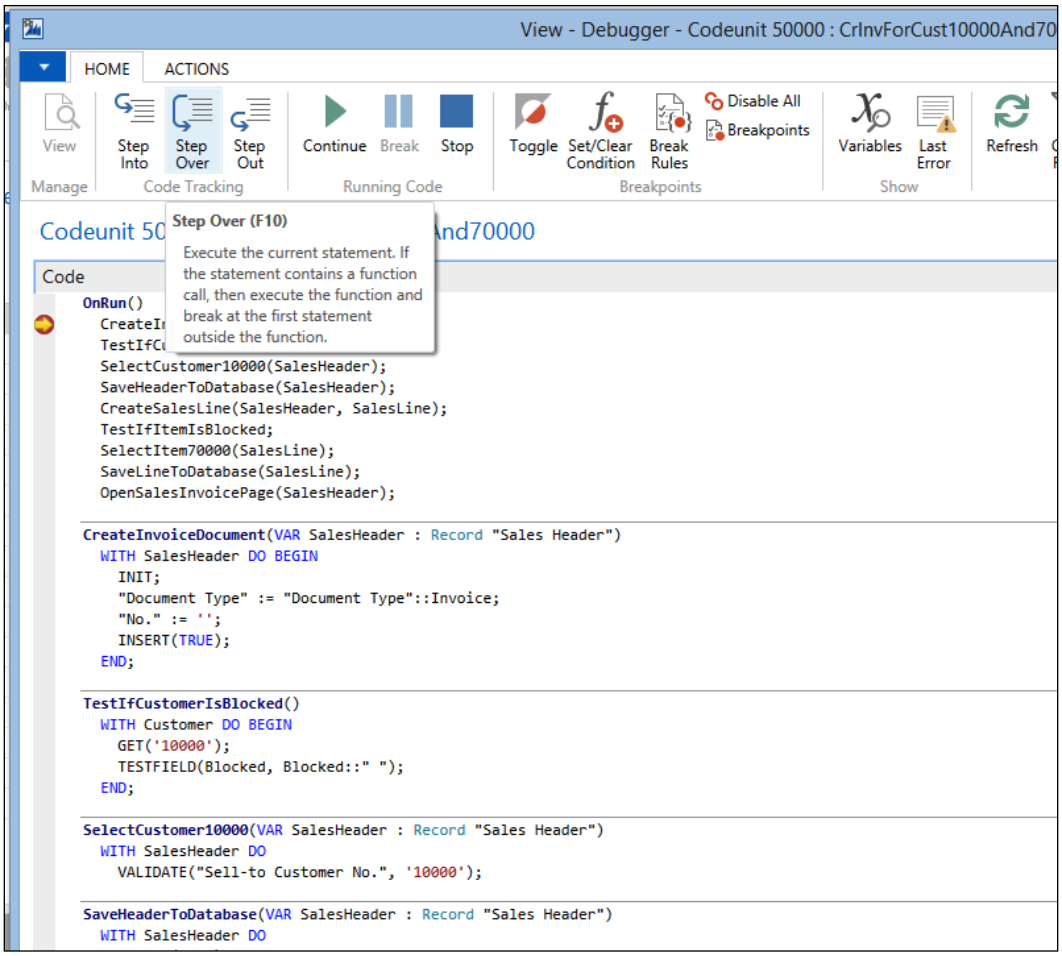
This way, the reader understands that the function contains more than one step, and uses the `Go To` Definition to further investigate. Of course, this function will also contain logical function names that make the code as readable as possible.

Bonus – debugging

Implementing Natural Language Programming forces developers to think before they act and write code that does a better job of explaining itself.

This becomes very useful when debugging. Debugging in Microsoft Dynamics NAV is very often done by Functional Consultants or Power Users.

When they see a code that is structured in a more readable way, it will be easier for them to understand the intention of the developer.



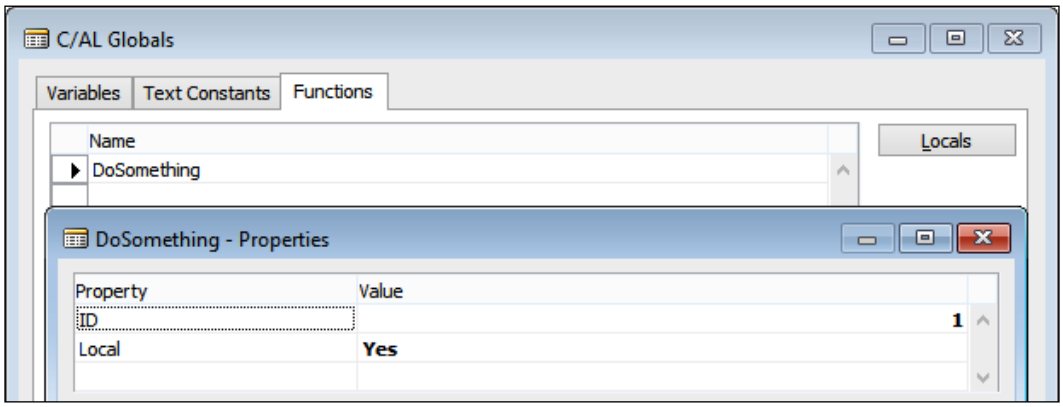
Encapsulation

Although encapsulation is considered one of the four cornerstones of OOP it is not limited to this, and can be implemented in Dynamics NAV as well. When implementing encapsulation, we hide the components from being wrongly used.

In other words, we indicate to other developers which functions can, or cannot be used outside the class, or as in Microsoft Dynamics NAV, the object.

This is achieved by implementing the local functions. Microsoft Dynamics NAV allows functions to be global or local. Global functions are allowed to be called from outside the object; local functions can only be used inside an object. In C#, these would be public and private functions.

When we want a function to be local, traditionally in C/AL we have to explicitly set the property. However, this has changed in Microsoft Dynamics NAV 2015. In this version, all the functions are explicitly set to local, and as developers, we have to change the property to No if we want the function to be available outside the scope of the object.



Before this change was implemented, the consideration of making a function global or local was often ignored by developers. When accidentally calling to a function that was designed only to be used internally inside an object, unexpected things might occur, especially when the original developer changed the behavior of the function, assuming that it was never used outside the scope of his object.

Encapsulation can be related to the class-method-property model, interfaces, and Natural Language Programming.

If we treat a table like a class and make functions a method of the class, these functions should be global. Therefore, the default value of the Local property of a function has a negative impact on this model. If we implement this methodology, we have to change the property to `No` always, so the function is global and callable as a method of the table.



Whenever a function in a table could be local, it indicates that we should switch to a Codeunit, implementing Natural Language Programming.

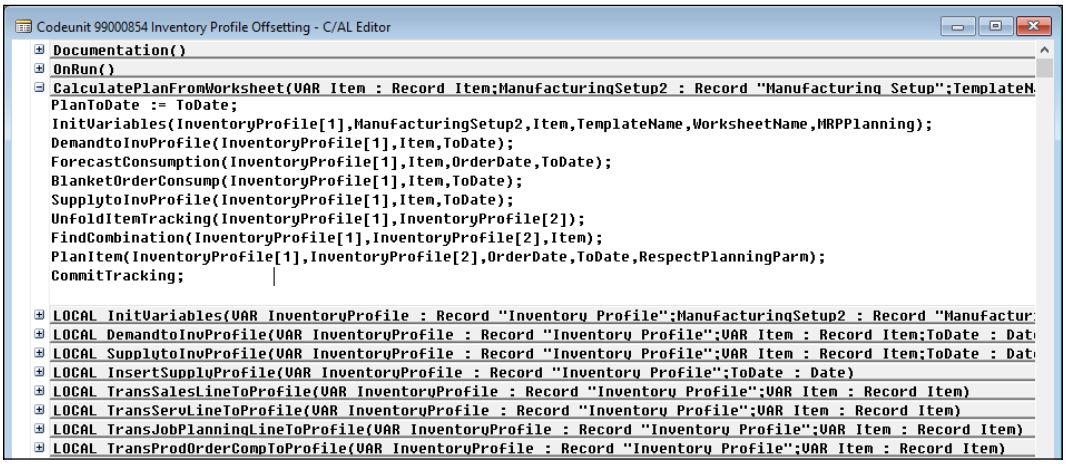
Functions in a table should almost always be callable as a member of the table, and be related to the table as an owner of the process.



This video explains how to implement encapsulation in C/AL. You can watch it at `https://www.youtube.com/watch?v=xsi2BjUY4IU`.

Interfaces and Function Libraries can have more than one global function. Function Libraries are often collections of the related features. These Codeunits can have local functions that perform logic, which is shared across the functions. Interfaces typically have one global function, which represents "the interface". However, there could be more global functions that allow small variations of the interface, requiring other parameters, comparable to overloading in other programming languages.

Encapsulation really becomes powerful when designing functions using Natural Language Programming. With this combination, we can separate the internal code from the owner. An example of an encapsulated function in Microsoft Dynamics NAV is Codeunit `99000854, Inventory Profile Offsetting`, as displayed in the next screenshot:



This Codeunit handles a complex algorithm for Inventory Calculation, and uses a variety of Patterns and methodologies, which makes it interesting material to study.

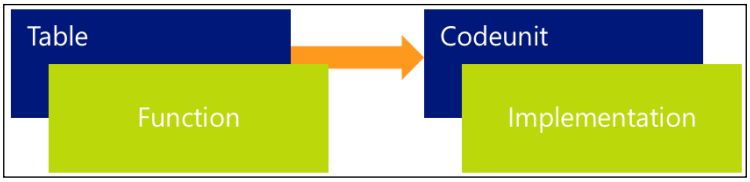
Override Functions

When Encapsulation is implemented, many programming languages allow developers to override local functions and replace their code. This is not possible in C/AL. Once an encapsulated module is called, there are no exit points for customizations. When we expect that our software is likely to be extended or even replaced by some other code, we can implement a Façade or Hook.

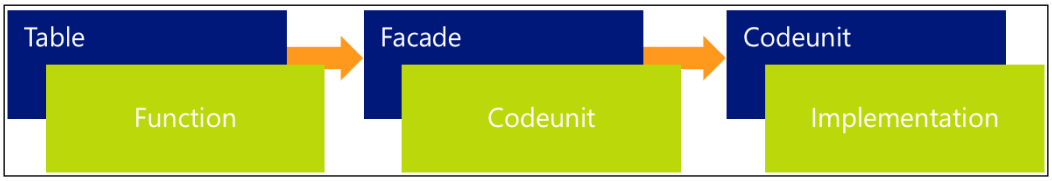
Façade

When working with objects and methods in ERP, we traditionally implement them directly. In other words, we implement tight coupling. The class calls the method, and the method only accepts calls from the object. We cannot run a Codeunit with a record that it does not relate to. This ensures a contract, the table, between the class and the method, allows the compiler to check integrity and prevents the software from being wrongly used.

In C/AL Code, this would look like: `SalesPost.RUN(SalesHeader)`, which ensures a tight contract between the Codeunit (`SalesPost`) and the table (`SalesHeader`).



In OOP, we can also implement loose coupling. When this is implemented, we implement an intermediate class in between the object and the method. This allows flexibility, but it increases the risk of the wrong use of the software. At design time, the method does not know its caller, and the caller may not know the method.



The **Façade** Pattern allows us to design software with great flexibility, since a change in the method does not require a change in the source code. We can simply create a new implementation and call this instead.

This requires great discipline from the software architect and developers. When used wrongly, anti-patterns such as code cloning or boat anchors are just waiting to happen. We will discuss these in the next *Chapter 6*, *Anti-patterns and Handling Legacy Code*.

To avoid being wrongly used, in OOP, we can implement abstract classes. When doing this, we can force the method to have a specific signature.

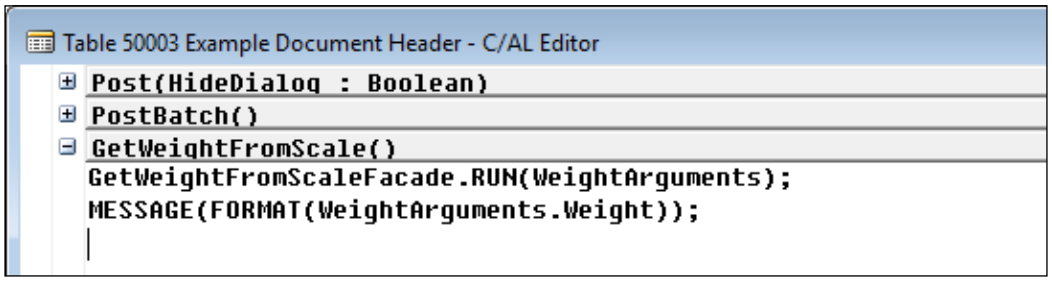
C/AL does not allow this flexibility of calling a Codeunit with multiple records. We can only ensure the type of record variable that we send to a function. When working with Variants, which we will discuss later in this chapter, we don't even have this assurance.

When the record itself is not suitable as a contract for the function, we may consider implementing the arguments Pattern as a contract for the Façade.

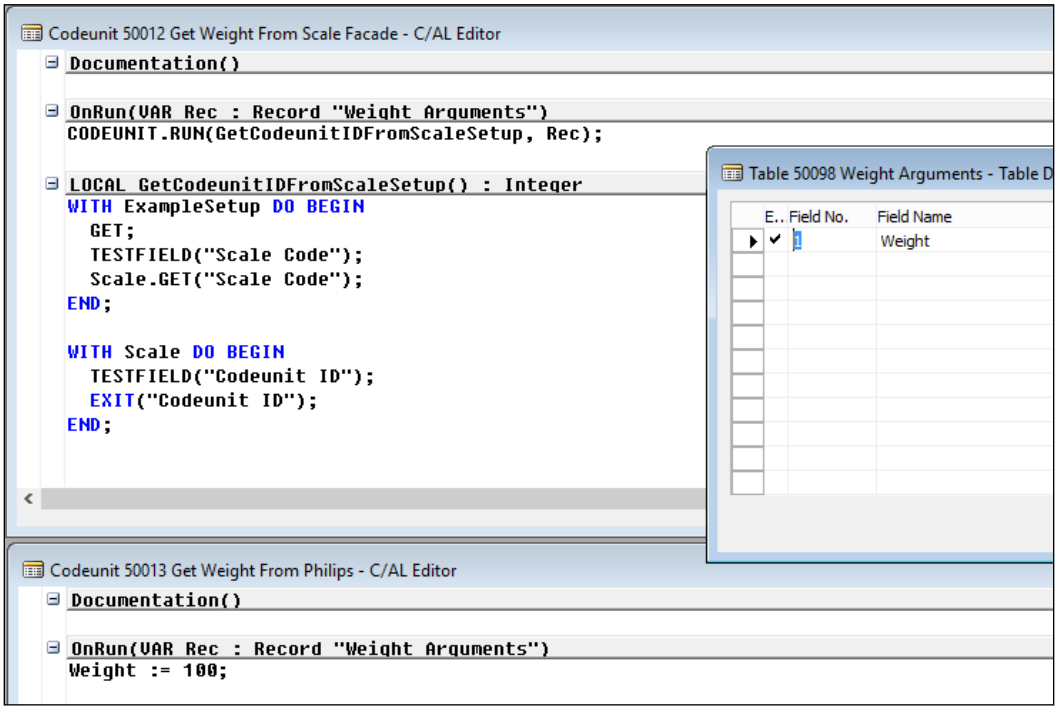
Let's have a look at the Façade Pattern with an example.

As discussed earlier, we should only implement Façade when flexibility is required. This can be the case in the scenario of getting the weight from a weighing scale. There are many manufacturers of scales that have different interfaces, but from a business perspective we only require a single value—the weight.

To implement this, we need to add a member to a table called `GetWeightFromScale`, which returns a decimal value for the weight. This function calls in a Façade Codeunit that is able to determine the type of scale, based on setup. If we need this weight in other application areas, we have to consider creating an interface.



For the sake of simplicity, we will print the weight to a message box:



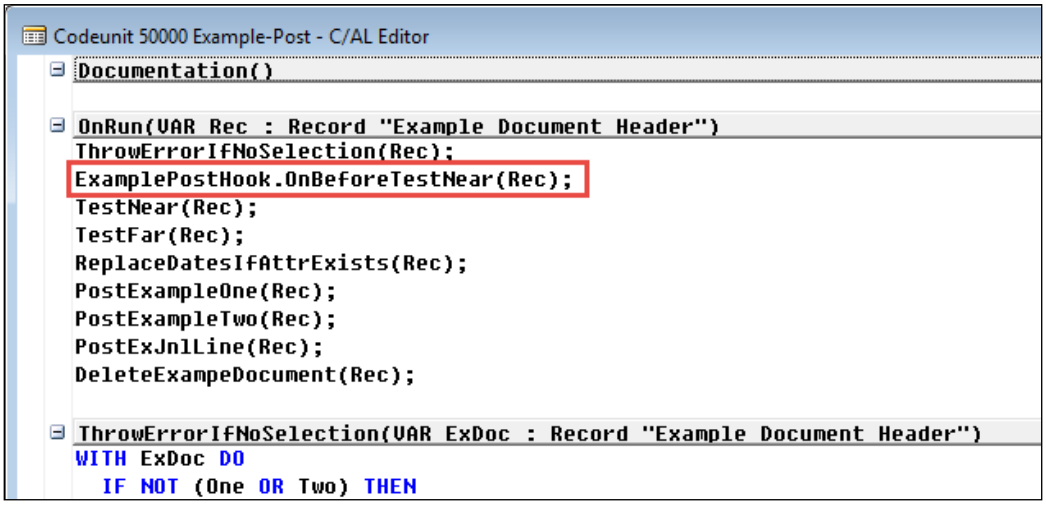
The Codeunit reads a Scale from the Setup table. Based on this setup, a specific Codeunit is called. The use of an argument table ensures the contract and that the Codeunit is not called with wrong parameters by accident.

Hooks

While the Façade has to be explicitly implemented to allow different code paths at runtime, there might also be scenarios where we are required to make changes to the code that we don't own. In these scenarios, we can implement a hook, which is nothing more than a function call that executes business logic outside of the code, but inside the transaction. The alternative to implementing hooks is raw source code modification, including creating additional variables.

The advantages of implementing hooks can be found in the area of upgradability. It does not increase maintainability, mainly because the name of the hook does not indicate what the hook does, but where it is called from. Hence we can see in the original source that there is a hook, but we need to use a `Go To` definition in order to see what business logic is executed.

Let's look at hooks with the help of an example:



Remember, we only implement hooks when we don't want to disturb the original source code. In this example, we have a posting routine. Instead of adding code to this routine, we call out to a hook at a certain point. The hook is named accordingly.

We need to make sure that the code is called with the required parameters, and return the changed values when required.

Disabling the original source code

In some scenarios, we maybe required to disable large portions of original source code. If this code is not encapsulated into a single function, the anti-pattern would be to comment out a large block of code.

Instead of doing this, we need to ensure that the original code still works when our modification is disabled. We can do this by adding an enabled flag to the setup of our module. If the module is disabled, we run the original code path.

This will also allow the automatic merge routines that Microsoft ships as part of the development tools to understand the intention of what we want to do. Another benefit is being able to execute the standard test routines.

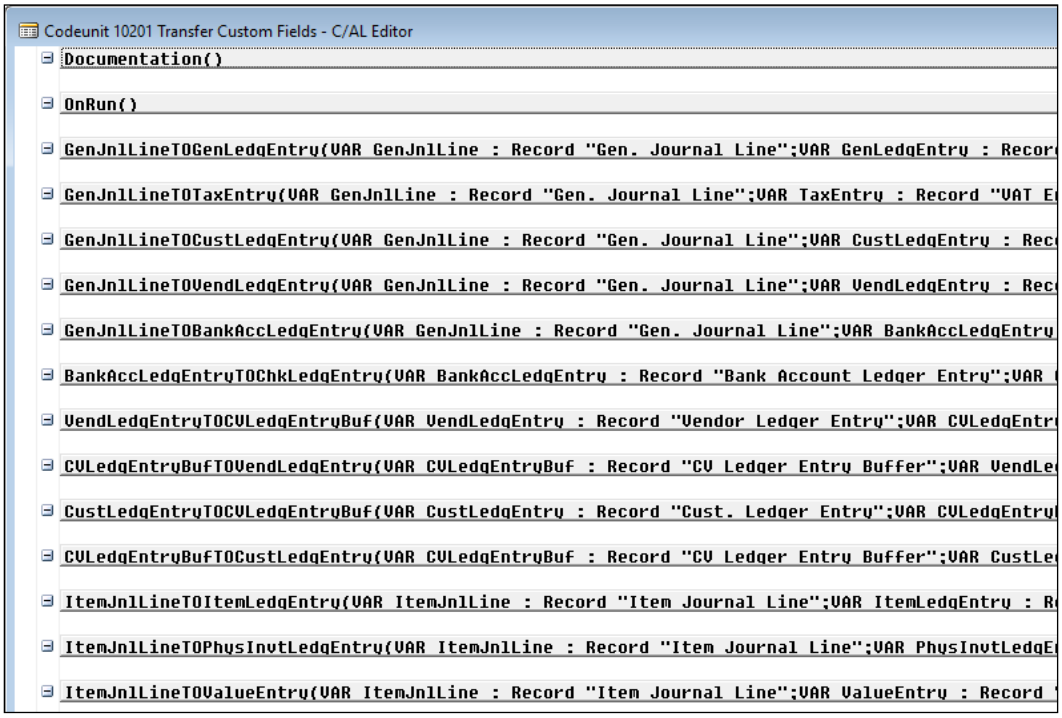
Predefined hooks

Hooks are not only used by developers who want to make modifications to the code, but also to the code that they don't own. When an architect expects others to make changes to certain processes, they can also implement Predefined hooks.



This video explains the implementation of the Predefined hooks. You can view it at `https://www.youtube.com/watch?v=cGaBqwfGCws&list=PLhZ3P-LY7CqmVszuvtJLujFyHpsVN0U_w&index=9`.

An example of a Predefined hook can be found in the localization for North America. This localization has Codeunit `10201` and `Transfer Custom Fields`, and is called from all the large posting routines where Dynamics NAV creates Entry tables.



Variant variables

Although it is not generally recommended, it might be required from time-to-time to be able to call a function with different types of records as parameters. With the strongly typed record variable parameters, this is not possible. However, we can use the Variant datatype as a workaround. We can store every variable type in NAV in a Variant.



All the C/AL code that we write is converted into C# code in the background, and this C# code generates a class for each object. The Variant datatype maps to the C# type of the object. So, if we use this datatype as a parameter of a function, we can call this function with any object in NAV.

In the object, we can change the object back to a specific table using the `RecordRef` datatype if required. This also allows us to test whether the function is called with a table that we expect it to be called with.

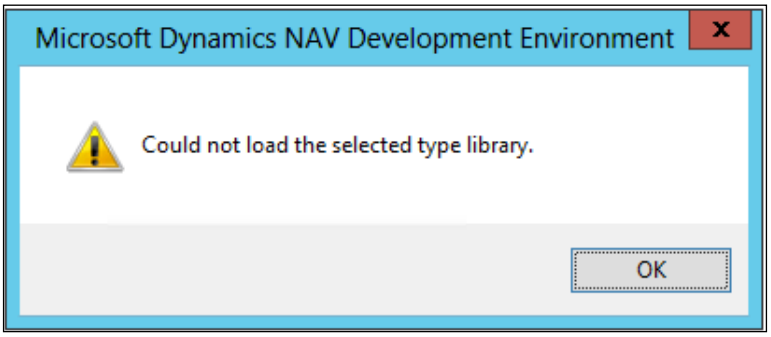
This explains what options we have when using variants as parameters in functions. You can view it at `http://www.waldo.be/2013/04/25/the-power-of-variants/`.

Managing DLL

Before Microsoft introduced dot net, we had automation control of the DLL files. These DLL files needed to be registered on each machine. This is what people refer to as **DLL Hell**.

In the case of Microsoft Dynamics NAV, we refer to DLL Hell in a different way.

The challenge of using DLL in Dynamics NAV, whether it is an automation control DLL or a custom made .NET DLL, is that the compiler checks the references. This screenshot shows the error message in the case of a missing DLL reference:

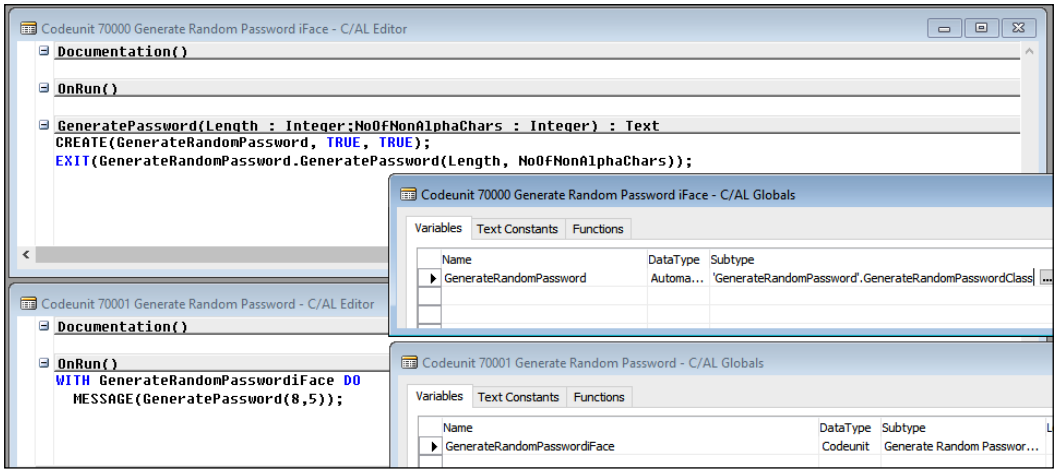


Since it cannot be expected that every developer has every customer's DLLs installed, it is considered best practice to wrap DLL in a one-to-one Codeunit. Each function in DLL gets a function in the wrapper Codeunit.

Instead of calling DLL directly as a variable, we will use the wrapper Codeunit. This enables developers to change the code *without having the DLL installed*.

This can potentially also be interpreted as an adaptor or a Façade.

The following screenshot illustrates creating a wrapper Codeunit and calling that from a new Codeunit. The new Codeunit can always be compiled, even if the DLL does not exist:



Summary

In this chapter, we learned some best practices that will help us structure our code in such a way that others can read, maintain, and upgrade it. The methodologies in this chapter give you a range of options to choose from, and some for when to use what scenarios. When implemented, you will realise that upgrading will become easier, and the automatic merge routines in PowerShell will do a great job for you.

In the next chapter, we will focus on anti-patterns and handling legacy code. This will help you in the transition from traditional C/AL coding, to the methodologies and Patterns discussed in this book.

6
Anti-patterns and Handling Legacy Code

From the perspective of Architectural and Design Patterns and coding guidelines, Microsoft Dynamics NAV is very consistent, but even though the Patterns have always been the secret source, the product is not always nice and clean. Many developers have worked on solutions for many years with or without guidelines about the maintainability and upgradability of the patterns.

The Dynamics NAV application from scratch has been built in quite a neat and clean way, but throughout the years, modules that Microsoft acquired through acquisition or sub-contracting have been added. They have not always been cleaned up, and following the rules that we learned in the earlier chapters of this book.

We call this phenomenon **legacy code**, and since our ecosystem is three decades old, there is a lot of it.

From a business perspective, it is hard to get a budget to spend a few years in a basement, changing your legacy code, which has been working for many years to work with these patterns and practices, and come up with a way to upgrade your existing customer base.

Therefore, we will learn some guidelines on how to transform legacy code into modern code. Some of these guidelines are used by Microsoft, which is experiencing similar challenges. This process is also referred to as **refactoring**.

Anti-patterns

In a world that defines good design and best practices, it might be useful to also define bad design, and discuss why we consider it bad, and in which context. Where Design Patterns are best practices, Anti-patterns are considered to be proven as counter-productive practices.

There is a lot of information to be found on the internet on Anti-patterns, and it is a perfect material to be discussed over a beer during a barbeque. In this book, we will not try to come up with a complete list of anti-patterns for Dynamics NAV, but we will discuss some of them and see how they can be avoided.

Code cloning

One of the most well-known anti-patterns is **code cloning**. Code cloning means copying code or a part of the code in order to reuse it. When learning object-oriented programming (OOP), it is good to try to avoid this at almost any cost, whereas in Microsoft Dynamics NAV, we have sales people telling customers how easy it is to copy something in NAV and customize it. Dynamics NAV itself is full of code cloning. How can this add up to being good?

Well, some of it does not add up, but some of it also needs to be put into perspective.

In order to put it into perspective, we will divide code cloning into four subtypes. This will enable us to justify some instances of code cloning in Dynamics NAV, and allow us to focus on what parts should be addressed first.

Forking

When **forking** is applied, we create a new application by creating a copy of an existing application, and let it grow organically from there.

Templating

If we reuse objects because they behave just as we need, but they are not designed to be either reused or used outside their functional implementation, we call it **templating**.

Customizing

Sometimes when we design a new feature, we don't want to immediately replace the original feature. If we start with a copy of a feature with the intention of ultimately replacing the original, we refer to it as **customizing**.

Exact match

When we copy objects or code rather than making generic functions, we call it **exact match**. This type of code cloning is what most software engineers think of when we talk about code cloning.



This document describes these forms of code cloning outside the context of Microsoft Dynamics NAV. You can find this at `http://plg.uwaterloo.ca/~migod/papers/2006/wcre06-clonePatterns.pdf`.

Examples

Let's look at an example of each category to understand how they are implemented in Microsoft Dynamics NAV.

Transaction mirroring

One of the fundamental design principles of Microsoft Dynamics NAV is transaction mirroring. This is best illustrated by looking at the Sales and Purchase side of the application, but the examples of it can be found anywhere in Dynamics NAV.

When we open the **Sales Line** (**37**) and **Purchase Line** (**39**) tables and look at the fields, the comparison is amazing.

Up to the level of field number, type, and often name, the fields match exactly:

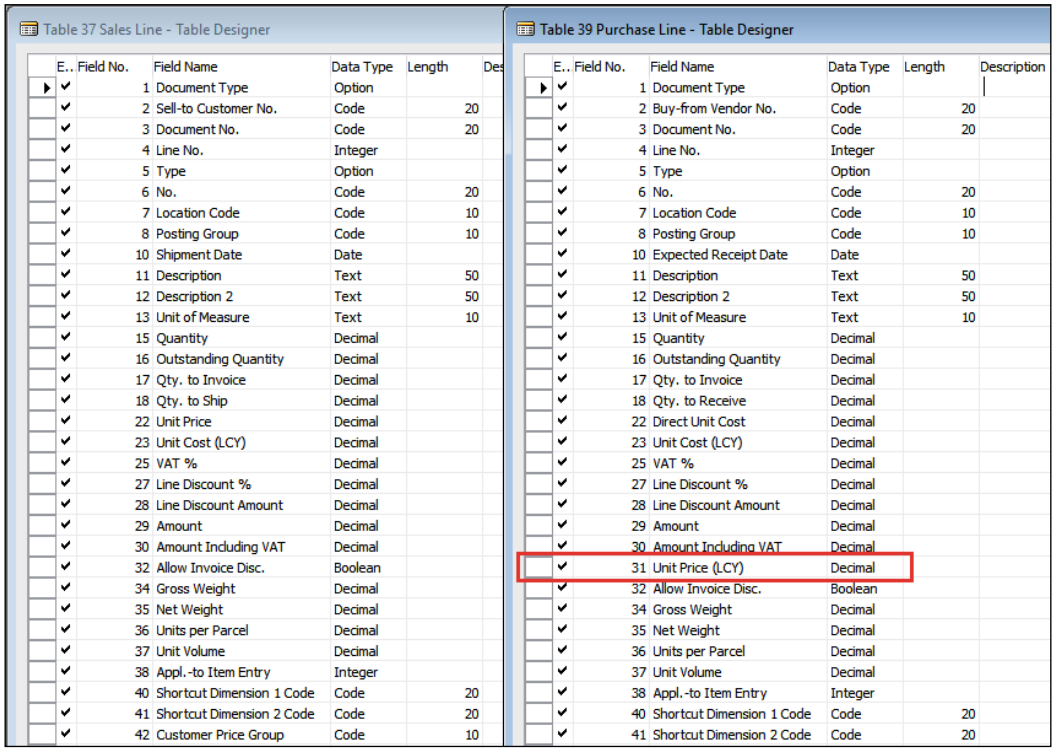


Table 37 Sales Line - Table Designer

E..	Field No.	Field Name	Data Type	Length	Des
✔	1	Document Type	Option		
✔	2	Sell-to Customer No.	Code	20	
✔	3	Document No.	Code	20	
✔	4	Line No.	Integer		
✔	5	Type	Option		
✔	6	No.	Code	20	
✔	7	Location Code	Code	10	
✔	8	Posting Group	Code	10	
✔	10	Shipment Date	Date		
✔	11	Description	Text	50	
✔	12	Description 2	Text	50	
✔	13	Unit of Measure	Text	10	
✔	15	Quantity	Decimal		
✔	16	Outstanding Quantity	Decimal		
✔	17	Qty. to Invoice	Decimal		
✔	18	Qty. to Ship	Decimal		
✔	22	Unit Price	Decimal		
✔	23	Unit Cost (LCY)	Decimal		
✔	25	VAT %	Decimal		
✔	27	Line Discount %	Decimal		
✔	28	Line Discount Amount	Decimal		
✔	29	Amount	Decimal		
✔	30	Amount Including VAT	Decimal		
✔	32	Allow Invoice Disc.	Boolean		
✔	34	Gross Weight	Decimal		
✔	35	Net Weight	Decimal		
✔	36	Units per Parcel	Decimal		
✔	37	Unit Volume	Decimal		
✔	38	Appl.-to Item Entry	Integer		
✔	40	Shortcut Dimension 1 Code	Code	20	
✔	41	Shortcut Dimension 2 Code	Code	20	
✔	42	Customer Price Group	Code	10	

Table 39 Purchase Line - Table Designer

E..	Field No.	Field Name	Data Type	Length	Description
✔	1	Document Type	Option		
✔	2	Buy-from Vendor No.	Code	20	
✔	3	Document No.	Code	20	
✔	4	Line No.	Integer		
✔	5	Type	Option		
✔	6	No.	Code	20	
✔	7	Location Code	Code	10	
✔	8	Posting Group	Code	10	
✔	10	Expected Receipt Date	Date		
✔	11	Description	Text	50	
✔	12	Description 2	Text	50	
✔	13	Unit of Measure	Text	10	
✔	15	Quantity	Decimal		
✔	16	Outstanding Quantity	Decimal		
✔	17	Qty. to Invoice	Decimal		
✔	18	Qty. to Receive	Decimal		
✔	22	Direct Unit Cost	Decimal		
✔	23	Unit Cost (LCY)	Decimal		
✔	25	VAT %	Decimal		
✔	27	Line Discount %	Decimal		
✔	28	Line Discount Amount	Decimal		
✔	29	Amount	Decimal		
✔	30	Amount Including VAT	Decimal		
✔	31	Unit Price (LCY)	Decimal		
✔	32	Allow Invoice Disc.	Boolean		
✔	34	Gross Weight	Decimal		
✔	35	Net Weight	Decimal		
✔	36	Units per Parcel	Decimal		
✔	37	Unit Volume	Decimal		
✔	38	Appl.-to Item Entry	Integer		
✔	40	Shortcut Dimension 1 Code	Code	20	
✔	41	Shortcut Dimension 2 Code	Code	20	

This is an example of forking. One of the advantages is the creation of recognizable software. Much like the abstract classes in OOP, the signatures of many functions are the same.

However, there is a thin line between forking and exact match. Many functions in Microsoft Dynamics NAV can be normalized.

Code normalization and writing clean code

When we discuss the term **normalization**, most engineers think about database modeling, and this is an important part of our job. Another part of normalization is writing clean code; avoiding code duplicates and making sure that the code is reusable and extendable. In Microsoft Dynamics NAV, we have some interesting examples of both: database normalization and code normalization.

The Document type normalization

Traditionally, Dynamics NAV is not referred to as being normalized by software engineers. However, there are some examples where the software is normalized, or may even be considered to be over normalized. An example of normalization is the Dimension structure that was introduced in Dynamics NAV 2013. With this model, customers can save up to an average of about 30 percent of database space compared to the old model.



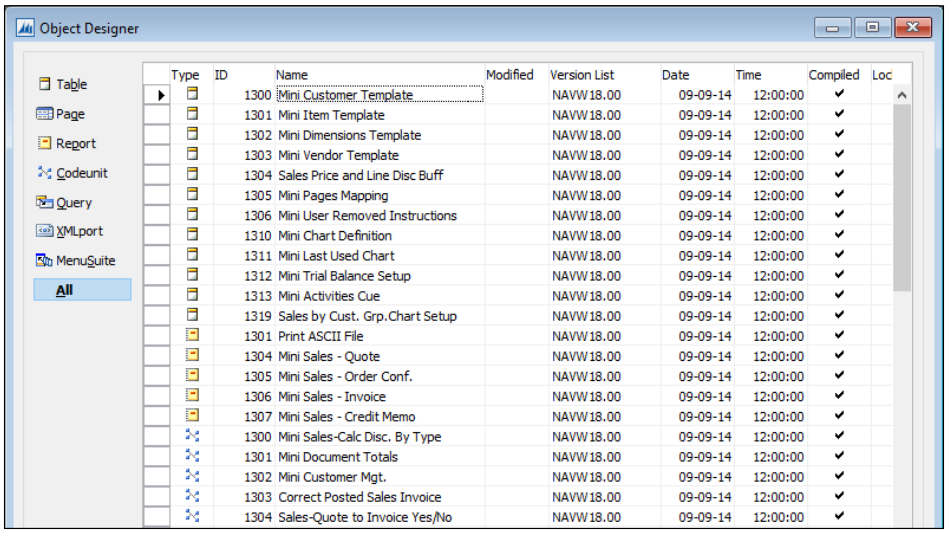
In this blog post, I've explained how the dimensions were refactored in NAV 2013. You can find it at `http://nav-skills.com/2015/08/21/dynamics-nav-2013-dimensions-i/` and `http://nav-skills.com/2015/08/21/dynamics-nav-2013-dimensions-ii/`.

Another example of normalization can be found in the Sales and Purchase documents before they are posted. Here a quote, blanket order, order, invoice, credit memo, and return order all live in the same table objects. This is considered by many as an example of templating. This is emphasized by the large quantity of CASE "Document Type" OF statements and IF "Document Type" = "Document Type"::"" THEN statements we find in the application.

The small business application

Inside the Microsoft Dynamics NAV objects, we can find a specific set of objects that we refer to as the Small Business Application. Internally, at Microsoft, they are referred to as **MiniApp**.

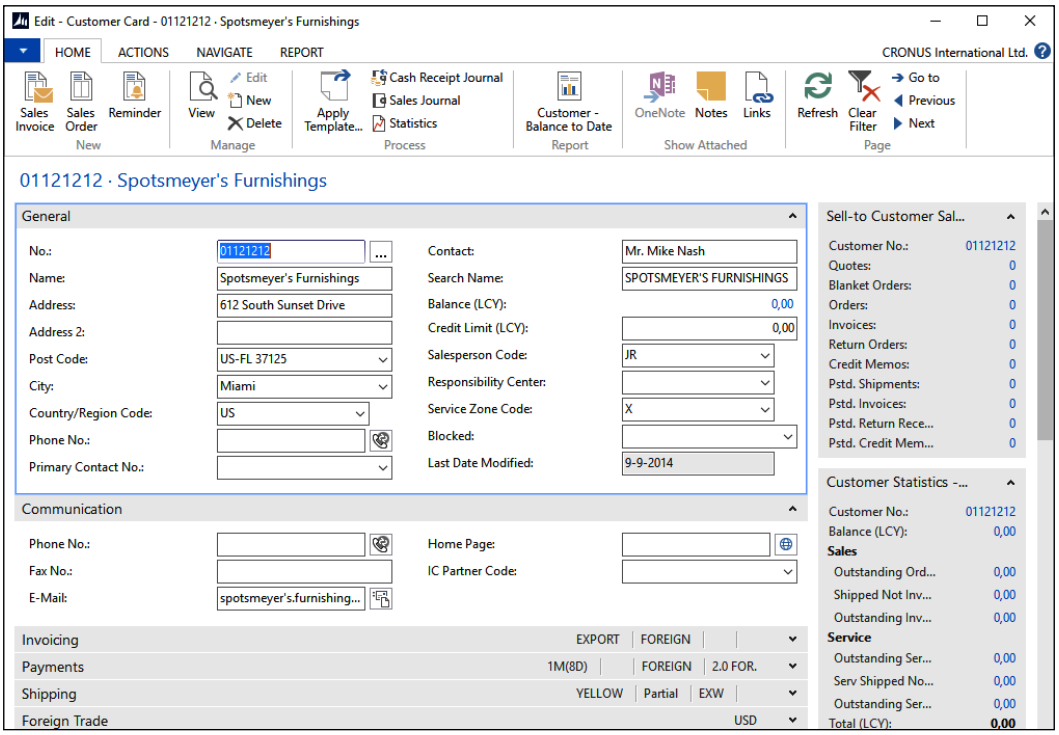
The following screenshot shows objects being used in the small business application. All the objects except the Rolecenter (page 9022) are in the 1300 – 1399 number range.



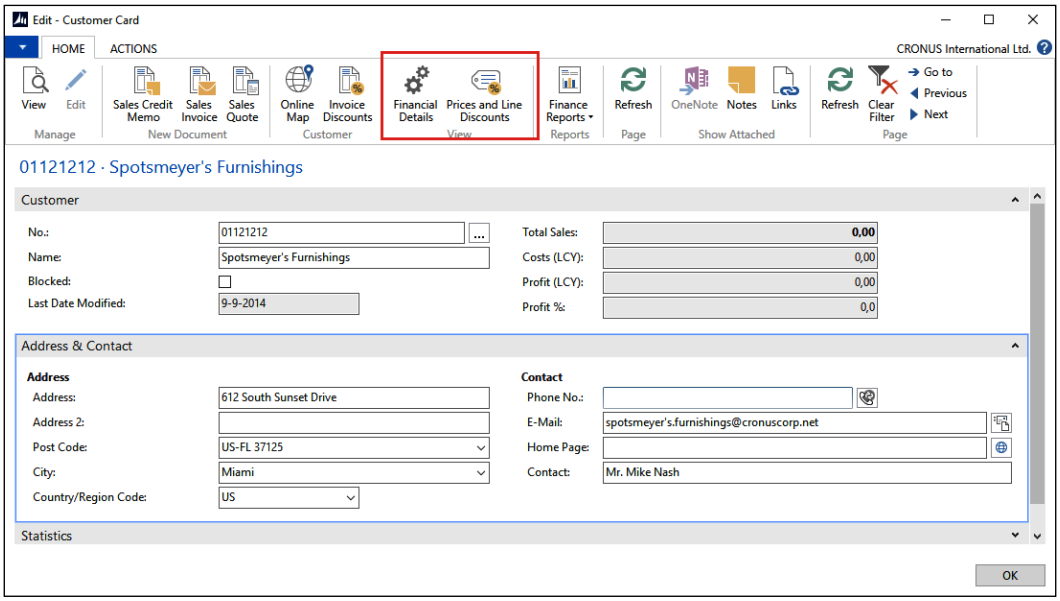
Type	ID	Name	Modified	Version List	Date	Time	Compiled	Loc
Table	1300	Mini Customer Template		NAVW18.00	09-09-14	12:00:00	✔	
Table	1301	Mini Item Template		NAVW18.00	09-09-14	12:00:00	✔	
Table	1302	Mini Dimensions Template		NAVW18.00	09-09-14	12:00:00	✔	
Table	1303	Mini Vendor Template		NAVW18.00	09-09-14	12:00:00	✔	
Table	1304	Sales Price and Line Disc Buff		NAVW18.00	09-09-14	12:00:00	✔	
Table	1305	Mini Pages Mapping		NAVW18.00	09-09-14	12:00:00	✔	
Table	1306	Mini User Removed Instructions		NAVW18.00	09-09-14	12:00:00	✔	
Table	1310	Mini Chart Definition		NAVW18.00	09-09-14	12:00:00	✔	
Table	1311	Mini Last Used Chart		NAVW18.00	09-09-14	12:00:00	✔	
Table	1312	Mini Trial Balance Setup		NAVW18.00	09-09-14	12:00:00	✔	
Table	1313	Mini Activities Cue		NAVW18.00	09-09-14	12:00:00	✔	
Table	1319	Sales by Cust. Grp.Chart Setup		NAVW18.00	09-09-14	12:00:00	✔	
Page	1301	Print ASCII File		NAVW18.00	09-09-14	12:00:00	✔	
Page	1304	Mini Sales - Quote		NAVW18.00	09-09-14	12:00:00	✔	
Page	1305	Mini Sales - Order Conf.		NAVW18.00	09-09-14	12:00:00	✔	
Page	1306	Mini Sales - Invoice		NAVW18.00	09-09-14	12:00:00	✔	
Page	1307	Mini Sales - Credit Memo		NAVW18.00	09-09-14	12:00:00	✔	
Codeunit	1300	Mini Sales-Calc Disc. By Type		NAVW18.00	09-09-14	12:00:00	✔	
Codeunit	1301	Mini Document Totals		NAVW18.00	09-09-14	12:00:00	✔	
Codeunit	1302	Mini Customer Mgt.		NAVW18.00	09-09-14	12:00:00	✔	
Codeunit	1303	Correct Posted Sales Invoice		NAVW18.00	09-09-14	12:00:00	✔	
Codeunit	1304	Sales-Quote to Invoice Yes/No		NAVW18.00	09-09-14	12:00:00	✔	

This application contains a lot of UI clones of the existing objects in NAV that behave slightly different than the original object. This was introduced in NAV2013R2 and NAV2015, some design principles are inherited back to the main objects. This is an example of the `Customize` subtype of code cloning.

The following screenshot illustrates the full Customer Card in the Dynamics NAV application:

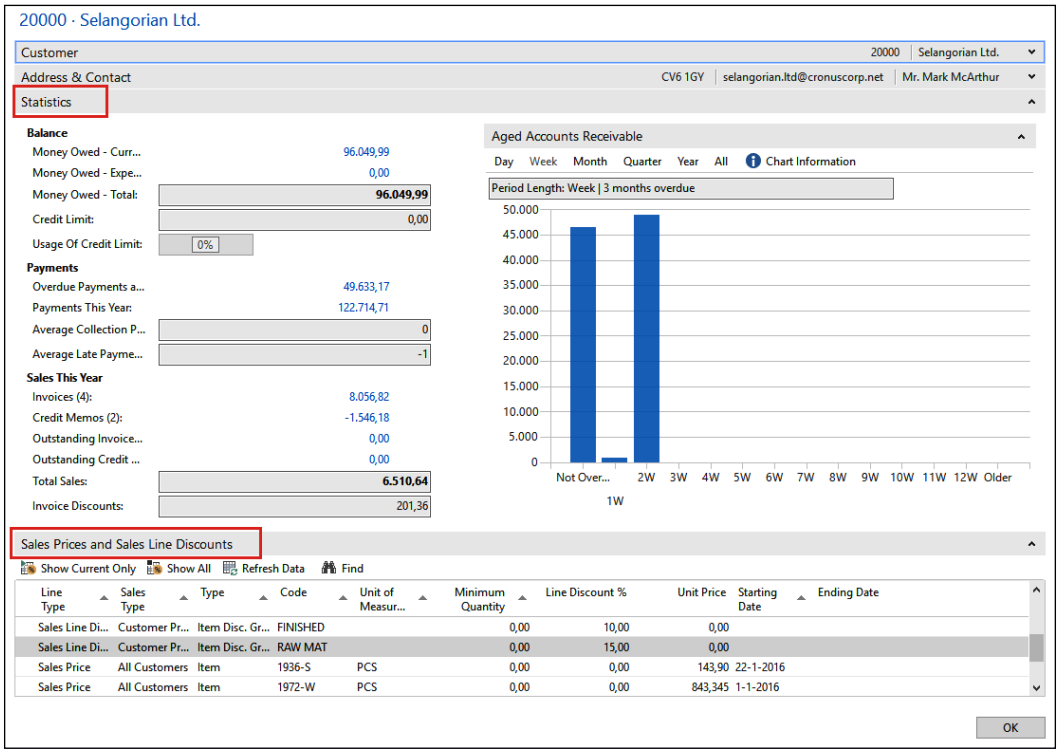


In the small business application, there is less information, and fields are grouped together in an easier way:



Information that is not always required is hidden and can be activated using actions. The examples are **Financial Details** and **Prices and Line Discounts**. The latter is a SubPage, which is displayed directly on the card; it is something that we would normally not do on Master Data.

The next screenshot illustrates the SubPage, and also shows **Statistics** using a graph of a Master Data card:



The VAT and Sales Tax calculation

One of the best examples of exact match code cloning in Microsoft Dynamics NAV is the way in which VAT and Sales Tax is calculated. Each table that has this calculation has an almost identical set of functions that do the calculation:

```
CASE "VAT Calculation Type" OF
  "VAT Calculation Type"::"Normal VAT",
  "VAT Calculation Type"::"Reverse Charge VAT":
    BEGIN
      "VAT Amount" :=
        ROUND(Amount * "VAT %" / (100 + "VAT %"),
          Currency."Amount Rounding Precision",
          Currency.VATRoundingDirection);
```

```
        "VAT Base Amount" :=
          ROUND(Amount - "VAT Amount",
            Currency."Amount Rounding Precision");
      END;
    "VAT Calculation Type"::"Full VAT":
      "VAT Amount" := Amount;
    "VAT Calculation Type"::"Sales Tax":
      IF ("Gen. Posting Type" = "Gen. Posting Type"::Purchase) AND
         "Use Tax"
      THEN BEGIN
        "VAT Amount" := 0;
        "VAT %" := 0;
      END ELSE BEGIN
        "VAT Amount" :=
          Amount -
          SalesTaxCalculate.ReverseCalculateTax(
            "Tax Area Code","Tax Group Code","Tax Liable",
            "Posting Date",Amount,Quantity,"Currency Factor");
        IF Amount - "VAT Amount" <> 0 THEN
          "VAT %" := ROUND(100 * "VAT Amount" /
            (Amount - "VAT Amount"),0.00001)
        ELSE
          "VAT %" := 0;
        "VAT Amount" :=
          ROUND("VAT Amount",Currency."Amount Rounding Precision");
      END;
  END;
```

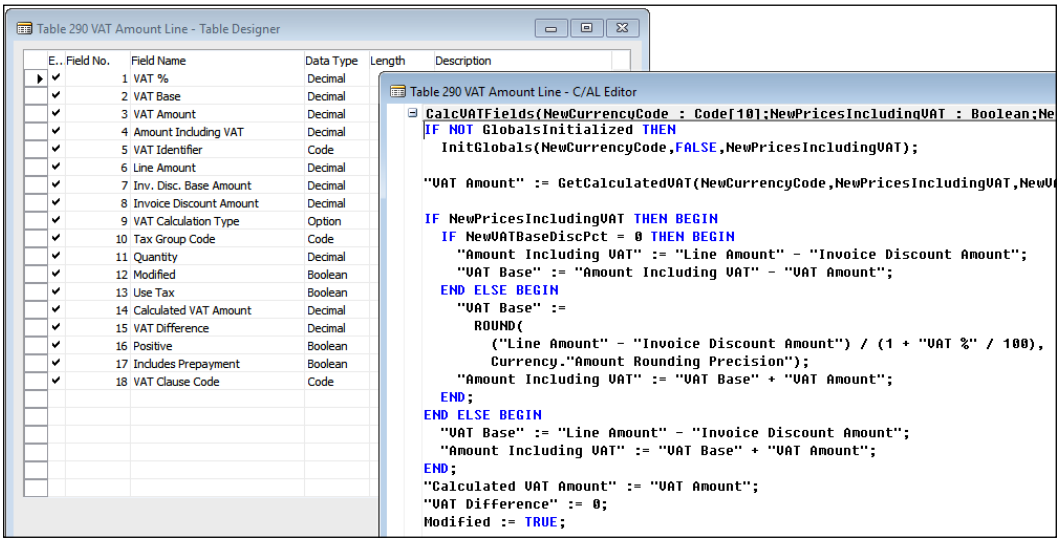
Some objects where this code is implemented are Codeunits 12, 442, 444 and Tables 37, 39, 5902, 751, 81.

Avoiding code cloning

Code clones of the type called exact match should always be avoided. There are Patterns and practices against this. Function libraries and interfaces are the best examples.

The Argument table is a perfect Pattern that can help you against code cloning. One of the challenges of normalizing code is the number of parameters that you end up having in functions. The Argument table is a perfect answer to this.

In case of our VAT and Sales Tax calculation example, we can potentially create a VAT and Sales Tax argument table where the tax is calculated and the results are passed back to the caller. In fact, a VAT Amount Line table already exists in Microsoft Dynamics NAV with a function called `CalculateVATFields`, as illustrated in the next screenshot:

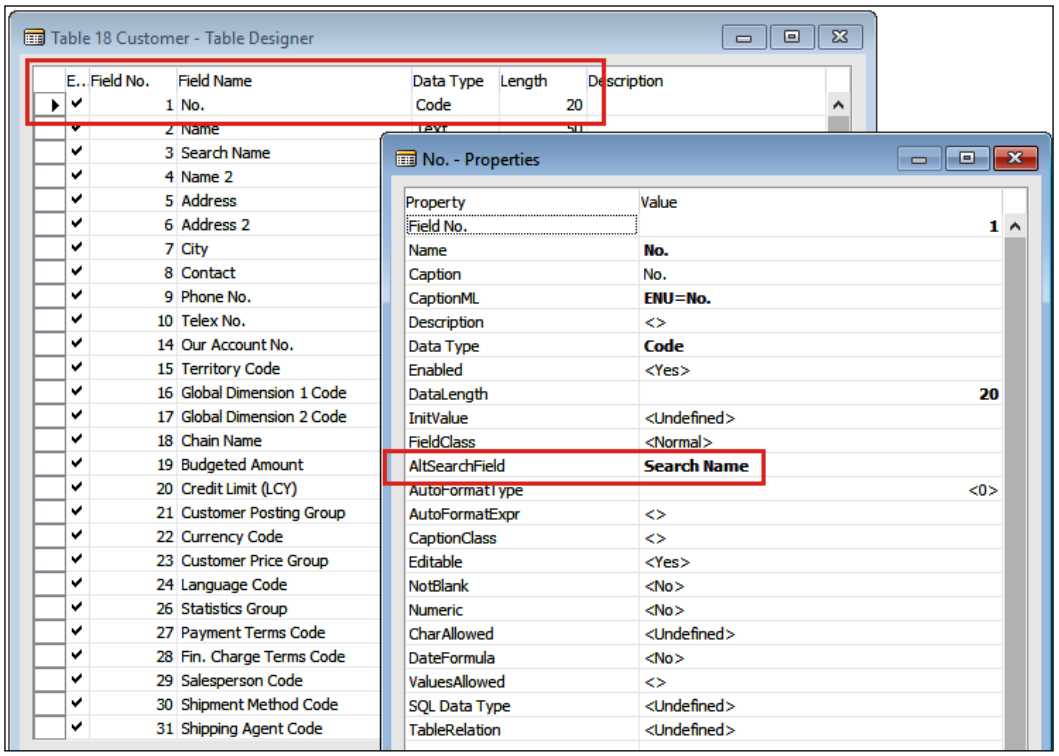


Alternatively, when you require VAT and Sales Tax calculation for your add-on, you might consider using the General Journal Line as an argument table for your calculation. This way you are guaranteed to have a working solution if Microsoft changes the calculation; or you have proper warning that the contract might be changed in a new release of the product.

Boat anchors

Another typical anti-pattern is a **boat anchor**. This Pattern is implemented by leaving code or metadata in your application that is no longer used.

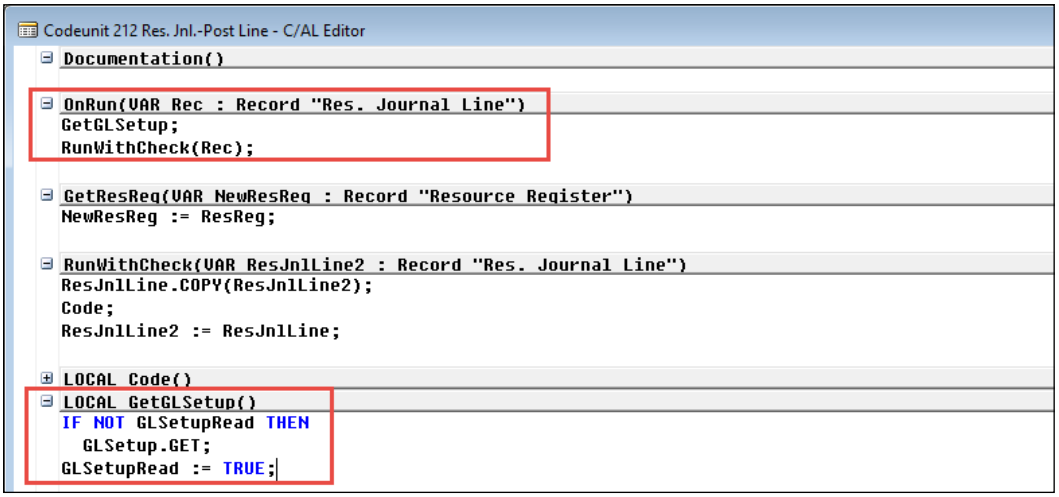
There are at least a dozen examples of boat anchors in the standard Microsoft Dynamics NAV code. One of them is the `AltSearchField` property in the Master Data tables, as shown:



This property was used by the classic client. When a field has a table relation with the primary key of another table, a user can also type the value of `AltSeachField`, and the client will automatically search in both the columns.

Since the introduction of the **Role Tailored Client**, this mechanism has been replaced by the **DropDown Field Group**.

Other examples can be found in the Journal Posting Codeunits, as in this screenshot:



The `OnRun` trigger was used before Navision Software introduced **Dimensions** in version 3.0. After its introduction, the Dimensions had to be passed and the `RunWithCheck` function was implemented. However, when Dimensions was refactored in NAV2013, they did not move back to the `OnRun` Trigger. Therefore, either the code in `OnRun` is never called and should be removed, or `RunWithCheck` should be refactored.

The second boat anchor in this Codeunit is getting the General Ledger Setup in the `GetGLSetup` function. To avoid roundtrips to the SQL Server, it is best practice to remember whether we already read the record inside a loop. This has been obsolete ever since the introduction of the data cache on the Service Tier. This automatically avoids the roundtrips to the server.



Other examples of boat anchors in Microsoft Dynamics NAV can be found at `http://dynamicsuser.net/blogs/vanvugt/archive/tags/cleanup/default.aspx`.

Other anti-patterns and context

Many other anti-patterns can be derived from the design pattern or best practice that solves them. However, a pattern and an anti-pattern should always be applied in context. Sometimes, a comment in the code is unavoidable since you don't own the code; or the addition of a suffix, explaining the datatype makes sense since the name is ambiguous.

A design pattern implemented in the wrong context can become an anti-pattern and cause performance issues or become hard to maintain.

Programming language capabilities

The ability to apply Design Patterns is largely managed by the capabilities that a programming language has. Since Microsoft Dynamics NAV is not object oriented, we have a restricted subset of the traditional OOP patterns. We also have a set of Patterns that is either related to procedural programming, or the metadata definitions as described in *Chapter 1, Introduction to Patterns and Software Architecture*.



All the C/AL code that we write in Microsoft Dynamics NAV is converted to C# code before generating a managed code. The managed code is generated at runtime by the service tier. The C# code is maintained in a system table. This code should never be changed or used.

Whenever Microsoft changes the capabilities of our programming language, the design patterns change. This can essentially lead to the introduction of new patterns, or even anti-patterns and boat anchors. With every release of Dynamics NAV, Microsoft adds features to our programming language, mainly due to the fact that C/AL is translated into C# before it is compiled, making it in essence as object oriented as Microsoft wants it to be, since the underlying language allows the object-oriented programming.

Legacy code

The first versions of Microsoft Dynamics NAV were released in the late 1980s as Navision running on MS-DOS and AS-400. Many of the fundamental architectural concepts were introduced in this version, and many of the objects go as far back as these days.

The language has always been updated and modernized, but Navision and Microsoft always ensured that existing code could be brought forward.

During its life cycle, Dynamics NAV had two moments when the software has had major platform changes. The first change was moving from the DOS platform to Windows in 1995, and the second and last change was the introduction of the three tier stack in 2009.

Both the platform changes allowed developers to bring forward existing applications, ensuring the investment that was made in the intellectual property.

This results in an ecosystem with a large amount of legacy code. Some of the existing Dynamics NAV objects go as far back as 30 years, including the partners and customers' code.

When Navision was originally written by the owners of the company, Design Patterns had a strong presence in the application, but the software has grown organically and has been touched by many other developers since then.

This leads to a potential challenge of having to redesign the legacy code from time to time. Since C/AL is an open code, and others can customize our code, we have to be careful in rewriting it since we might break a change that is being made outside the scope of our project.

We will discuss a few options that will allow us to refactor our application, while maintaining the upgradability for our application.

Refactoring the C/AL Code

When we discuss the methodology for refactoring, we need to focus on two areas. The first focus is on the data model—the way we store data in tables. The other is business logic—the code that is executed.

When we refactor the application, the restriction is that we do not introduce new functional features. We should focus on optimizing our application. This makes it easier to test the application, since the functional result of our transactions should match the old application.

Combining refactoring and implementing new features is very possible, but this introduces a substantial risk to our project that should be avoided if possible.

Refactoring tables

There is only a limited set of scenarios where refactoring tables is required to optimize the maintainability of the code without changing the functional application, since the Dynamics NAV user interface is typically bound to the database structure.

We will focus on the scenarios of renaming and normalization.

Renaming

Even though naming conventions in Dynamics NAV are strong and typically discussed when training new developers, this is considered an area where many mistakes are made. If, after reading this book, you would like to start implementing the naming conventions as discussed in *Chapter 5*, *Coding Best Practices*, we need to start renaming tables and fields.

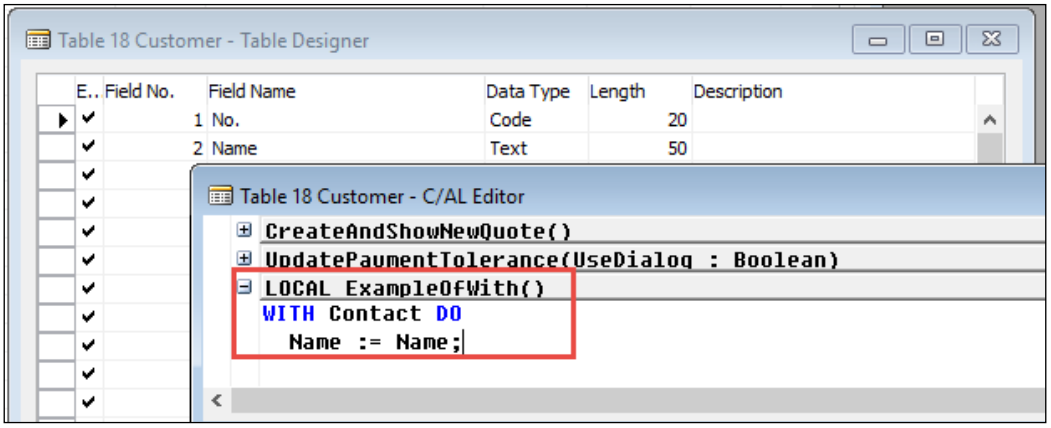
The table and field names can be changed in Microsoft Dynamics NAV without breaking the code with a few exceptions.



When renaming fields and tables, a manual merge with traditional tools will detect a change. The automatic merge with the PowerShell tools, introduced in NAV2013R2, will accept this without a conflict.

When a `WITH` clause is used on a `Table`, `Page`, or `OnRun` trigger of a Codeunit, the scope of the clause is set to the `Rec` variable first, and the `WITH` variable second. When both the tables contain the same fields, this can lead to unexpected results. When renaming fields, there is a chance of breaking the code that used to work if these `WITH` clauses were implemented in the mentioned areas. This should and can be easily avoided by not using the `WITH` scope in these areas.

The following screenshot shows an ambiguous `WITH` statement where both the **Customer** and **Contact** table have a `Name` field:



The `FlowField` definitions are not updated when the referenced fields are renamed in the Table. The `FlowFields` definitions should be updated manually. The compiler checks for these references.

Be careful with reserved words. A field in the table called Action can cause unexpected results, especially when renaming fields during refactoring. The compiler does not check for reserved words, but a list can be found on MSDN.



Information on the MSDN article about reserved words is available at `https://msdn.microsoft.com/en-us/library/ee414230(v=nav.80).aspx`.

Normalization

Although Normalization is not typical for Dynamics NAV, there are some exceptions. When Microsoft shipped Dynamics NAV 2013, the Dimension feature was refactored to be normalized.

Introducing normalization in Dynamics NAV while maintaining the existing functionality requires the implementation of the Model-View-ViewModel Pattern, as introduced in *Chapter 3*, *Design Patterns*. We can find an implementation of this Pattern in the Dimension application area.

When changing the Data Model, we also need to provide an upgrade path for the existing users of the system. There are two ways of doing this upgrade.

Traditionally, we upgrade data by providing a temporary set of tables to which we first move the data, while making sure that the original tables are cleared. This is called step one of the upgrade. When the original tables are empty, we can make changes to the data model and after these are implemented, we can run a script that changes the data structure to the new model. This is called step two of the upgrade.

If the upgrade has time constraints, we can alternatively optimize the upgrade by providing a new set of tables instead of redesigning the existing tables. This makes step one obsolete.

Doing the data conversion directly in SQL can also make the upgrade run much faster.

Code refactoring

I hope that after reading this book, you want to implement concepts such as the Class-Method-Property model, Natural Language Programming, and Encapsulation.

When the existing code does not follow these structures, the code can be refactored by following a few easy steps.

Please note that we must be the original creator of the code in order to refactor. We should never refactor code that we do not own. If we want to refactor the code modifications that we have done in the objects we don't own, we should use the Hooks Pattern.

Finding the process owner

The first step for refactoring is to find the owner of the code. This is typically the record with which you start the transaction. If this is true, the code can be moved to a function of the table. If the code does not have a specific record, we can move it into an interface.

Moving to Codeunit

Depending on the owner, we now have either an interface or a method. Both should be in a Codeunit of their own. When the method is very small, we can consider keeping it in the table. This, typically, is the case when the function has 10 lines of code or fewer and does not have a strong probability of being changed.

Breaking down the code into functions

The next step is to see if we can further break down into functions. These functions should have logical names and when implemented, the main routine should contain readable code in function names, as discussed in *Chapter 5*, *Coding Best Practices*.

Encapsulating functions

The functions in the Codeunit should be encapsulated, meaning the Local property should be set to true. Typically, a method called `Codeunit` only has one global function, or you can use the `OnRun` trigger as the main application.

Normalizing the code

When we cannot encapsulate a function because it is called from multiple places, we should normalize the code, following the same steps. Find the owner, decide if it is an interface or a method, and repeat the steps.

If the function can be encapsulated, there is still a chance that the code can be normalized, or that the code is the result of a clone in the application. This should be evaluated by analyzing the application, and refactoring the clones in interfaces and methods of their own.

We will normalize the code and refactor the functions into members of the table in *Chapter 7*, *Building Solutions using Patterns*.

The boy scout rule

Refactoring can be very complex, time consuming, and can lead to disruption in the release cycles. It should therefore be handled with care and tested thoroughly.

When applications have big pieces of legacy code, the question often rises of where to start and when to implement the methodology, as discussed in this book.

A simple rule is to leave the objects that you touch in better condition than when you started working on them. This rule, however, conflicts with the rule of only refactoring without changing the application.

Refactoring can lead to errors that did not exist before. Making mistakes is what makes us human. Refactoring should therefore never be done as a part of bug fixing.

Summary

In this chapter, we have learned about anti-patterns and how to handle legacy code. We discussed forking, templating, customizing, and exact match—the four types of code cloning.

We talked about boat anchors. We also talked about keeping the application code that is no longer in use, which could potentially harm upgrades and confuse developers. To handle the legacy code, we learned how to refactor tables and code while reducing the risk of introducing errors and maintaining the upgrade paths.

In the next chapter, we will combine the Patterns and methodology that we learned in this book into a functional application.

7
Building Solutions Using Patterns

Let's see if we can use the knowledge acquired in this book and use it to design a new application.

When doing this, we can apply Patterns and coding concepts in a new module that users will recognize to be a Microsoft Dynamics NAV application, and which is easy for other developers to understand and maintain.

The solution that we will make is for a small bed and breakfast (B & B), allowing them to manage their rooms and reservations. This can be integrated into the financial part of Dynamics NAV.

It is not the intention of this chapter to make a full-featured finished product. We will discuss basic design principles and decision-making processes. Therefore, we will simplify the functional process. One of the restrictions in our application is that we rent rooms per night.

This chapter will be covering the following topics:

- Building blocks
- Creating the Table objects
- Applying the Design Pattern
- Defining the methods
- Writing code and linking methods
- Refactoring
- Testing the application

- Upgrading our application
- Building our application
- Managing our source code

Building blocks

We borrowed the term classes from object-oriented programming as a collection of things that belong together. Classes can be Tables or Codeunits in Microsoft Dynamics NAV.

The first step in our process is to define the classes. These will be created as Tables or Codeunits, following the Patterns that we have learned:

Setup	This is a generic set of parameters for the application
Guest	This is a person who stays at our B & B. This can be one person, two persons, or a group (family)
Room	Our B & B has a number of rooms with different attributes that determine the price, together with the season
Season	This is the time of the year
Price	This is the price for one night in a room
Reservation	Rooms can be reserved on a day-by-day basis with a start and end date
Stay	This is a set of one or more consecutive nights at our B & B
Check-In	This is the start of a Stay; checking in for reservation
Check-Out	At the end of a Stay, we would like to send a bill
Clean	Whenever a room is cleaned, we would like to register this
Evaluation	Each Stay can be evaluated by a Customer
Invoice	This generates a Sales Invoice for a Stay

Applying Architectural Patterns

The second step is to decide per class the Architectural Patterns that we can use. We will use the Patterns that we learned in *Chapter 2*, *Architectural Patterns*, but in some special cases we might need to write down some new Patterns, based on the data structures that are not used in the standard application, or the Patterns that are outside the scope of this book.

The application setup

For the application setup, we will use the Singleton Pattern. This allows us to define a single set of values for the entire application that is kept in memory during the lifetime of the system.

Guests

To register our guests, we will use the standard Customer table in Dynamics NAV. This has pros and cons.

The good thing about doing this is the ability to use all the standard analysis options in the application for our customers without reinventing the wheel. Some B & B users might decide to also sell souvenirs or local products, so they can use items and the standard trade part of Dynamics NAV. We can also use campaigns in the Relationship Management module.

The bad part, or the challenge, is upgradability. If we were to add fields to the customer table or modify the standard page elements, we would have to merge these into the application each time we got a new version of the product, which is once a month. We will use the new Delta file to challenge this ,as well as the testability framework.

Room

The Architectural Pattern for a room is a tough decision to make. Most users of our system run a small B & B, so we can consider rooms to be as setup data. Number Series is not a required Pattern. We will therefore decide to implement a Supplemental Table.

Season

Each B & B can setup their own seasons. They are used to determine price, but when not used, the system will have to work too. We will implement a Supplemental Table too.

Price

Rooms can have a default price, or a price per season and guest. Based on this requirement, we will implement the Rules Pattern, which allows us a complex array of setup values.

Reservation

We want to carefully trace the reservations and cancellations per room and per guest. We would like to analyze the data, based on a season. For this feature, we will implement the Journal Template-Batch-Line Pattern, and introduce an Entry table that is managed by the Journal.

Stay

We would like to register each unique stay in our system rather than individual nights. This allows us to easily combine parameters and generate a total price. We will implement this as Master Data, based on the requirement to be able to use Number Series. The Stay does not have requirements for a lines table, nor does it represent a document in our organization.

Check-in

When a guest checks in to the bed and breakfast, we can check a reservation and apply the reservation to the Stay.

Check-out

When a guest leaves, we would like to set up the final bill, and ask to evaluate the stay. This process will be a method on the `Stay` class with encapsulated functions, creating the sales invoice and generating an evaluation document.

Clean

Rooms have to be cleaned each day when a guest stays, and at least once a week when the room is empty.

We will use the Entry Pattern without a journal. `Clean` will be a method on the `Room` class. Each day, we will generate entries using the Job Queue Entry Pattern. The Room will also have a method that indicates if it has been cleaned.

Evaluation

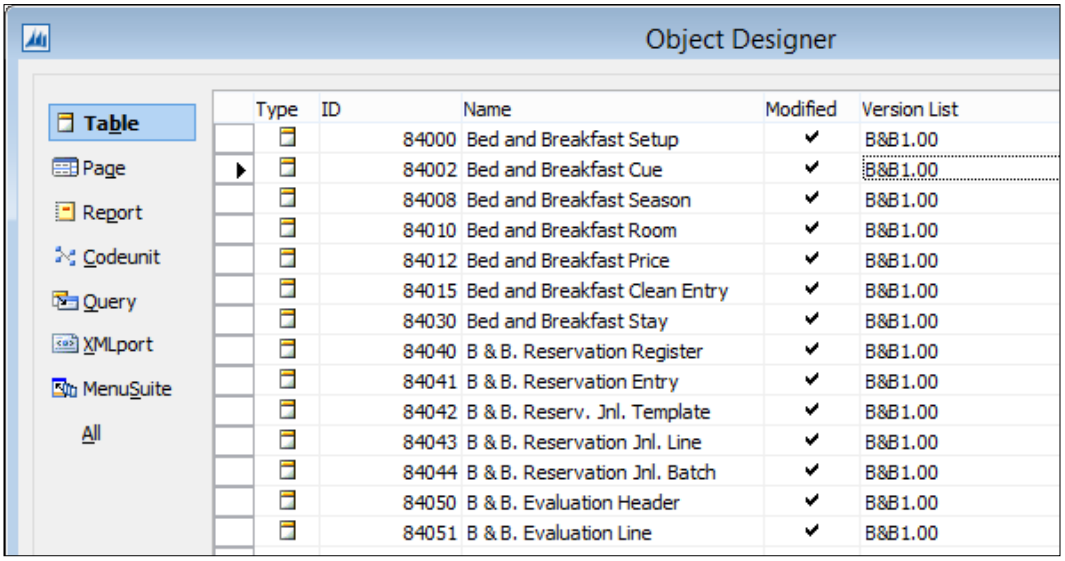
A Stay in our B & B can be evaluated by our guests. The evaluation has different criteria. We will use the Document Pattern.

Invoice

We can create the method as an encapsulated method of the Stay class. In order to link the Sales Invoice to the Stay, we will add the `Stay No.` field to the Sales Header, the Sales Invoice Header, and the Sales Cr.Memo Header tables.

Creating the Table objects

Based on the Architectural Patterns, we can define a set of objects that we can start working with.



Type	ID	Name	Modified	Version List
	84000	Bed and Breakfast Setup	✔	B&B1.00
	84002	Bed and Breakfast Cue	✔	B&B1.00
	84008	Bed and Breakfast Season	✔	B&B1.00
	84010	Bed and Breakfast Room	✔	B&B1.00
	84012	Bed and Breakfast Price	✔	B&B1.00
	84015	Bed and Breakfast Clean Entry	✔	B&B1.00
	84030	Bed and Breakfast Stay	✔	B&B1.00
	84040	B & B. Reservation Register	✔	B&B1.00
	84041	B & B. Reservation Entry	✔	B&B1.00
	84042	B & B. Reserv. Jnl. Template	✔	B&B1.00
	84043	B & B. Reservation Jnl. Line	✔	B&B1.00
	84044	B & B. Reservation Jnl. Batch	✔	B&B1.00
	84050	B & B. Evaluation Header	✔	B&B1.00
	84051	B & B. Evaluation Line	✔	B&B1.00



Object names are limited to 30 characters, which makes it a challenging to name them. The Bed and Breakfast name illustrates this challenge. Only use abbreviations when the length limitation is a problem.

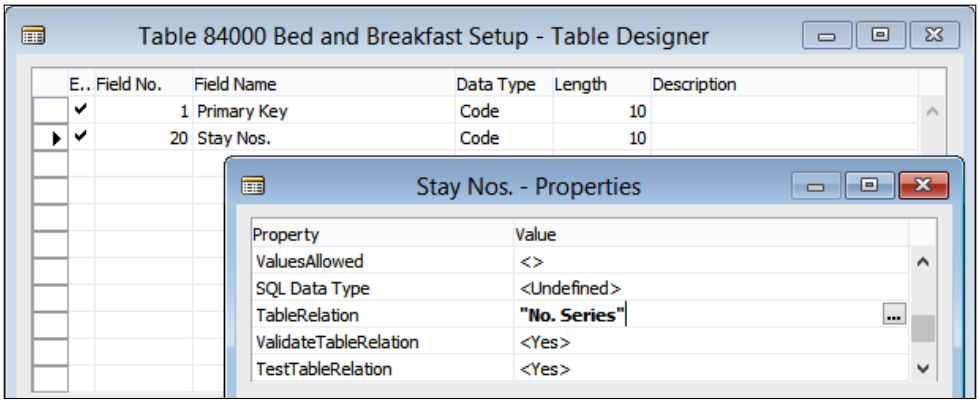
Applying Design Patterns

Most of these tables now only have an **ID**, **Name**, and **Primary Key**. We need to give them functional elements by adding Design Patterns.

The UML schemas for these Patterns can be found in *Chapter 2*, *Architectural Patterns* and *Chapter 3*, *Design Patterns*.

The Bed and Breakfast setup

In this example application, we need the Setup table to store the Number Series for the Stay table. Other than these, we don't have fields to add anything to this table.



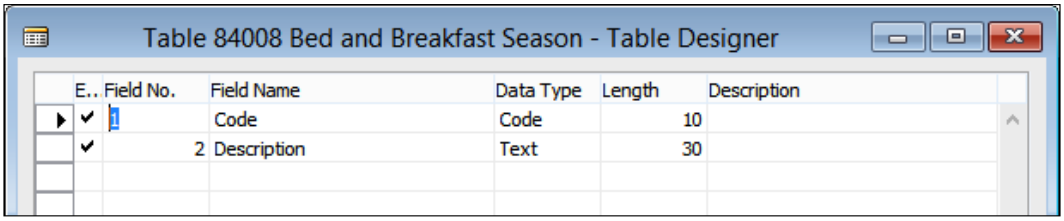
Here's how I usually put table definitions in design documents:

Field No	Field Name	DataType	Comments
1	Primary Key	Code10	This is the Primary Key for this table
20	Stay Nos.	Code10	This provides the Table Relationship to the "No. Series" table

If you need any elaborate descriptions of validation rules, you can add them as bullet items underneath the table.

Bed and Breakfast season

Using the Supplemental Architectural Pattern, we already have a Code and Description field. This is all we need for the Season table.



Bed and Breakfast room

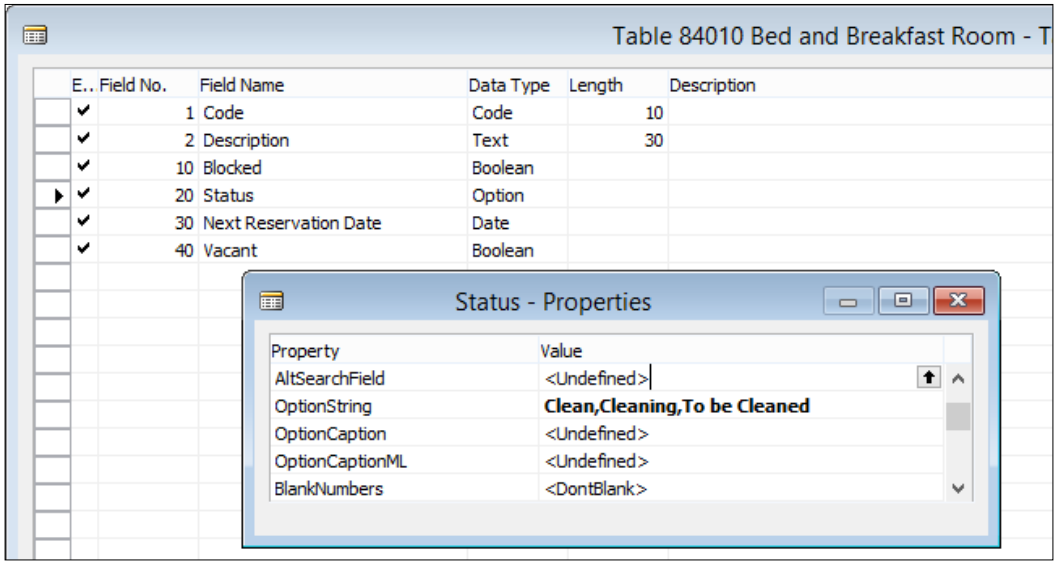
We will add three entity state fields to the room table—Blocked & Clean Status.

Blocked will prevent us from making a reservation for the room or create a new stay. This entity state field is editable.

Clean Status will be an option field with the values—`Clean`, `Cleaning`, `To be Cleaned`. These status values will be managed by methods on the table. The field is non-editable.

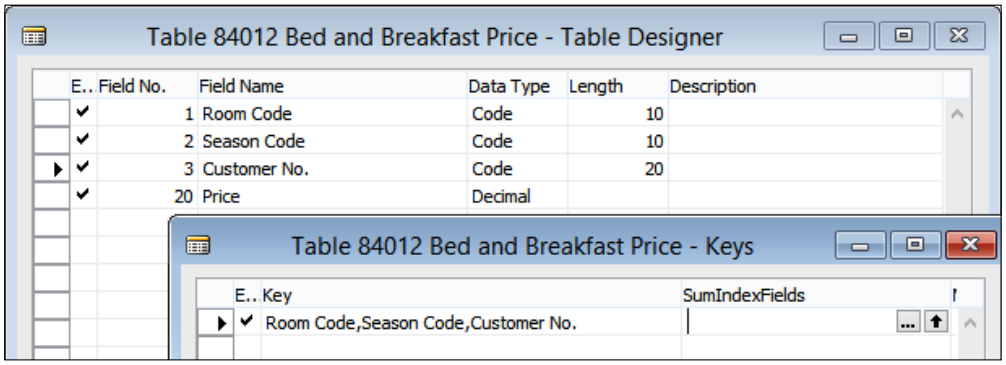
We will also add some flow fields to the table to see whether there is an upcoming reservation, and whether the room is vacant at the moment.

The price calculation will be a method on the table.



Bed and Breakfast price

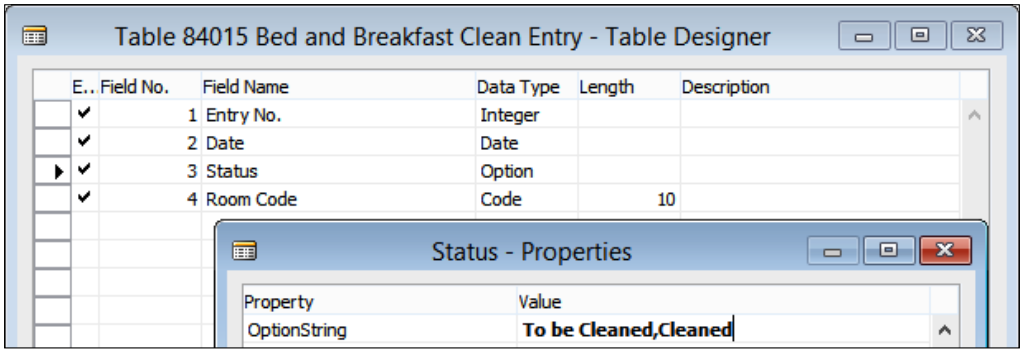
Implementing the Rules Pattern provides us with all the required fields for the price calculation.



Bed and Breakfast clean entry

We will just add some fields to the table to indicate the cleaning date and the cleaning type. The entry table does not contain business logic, and is managed by the Room tables' methods.

The **Entry No.** field is updated using the **AutoIncrement** property.



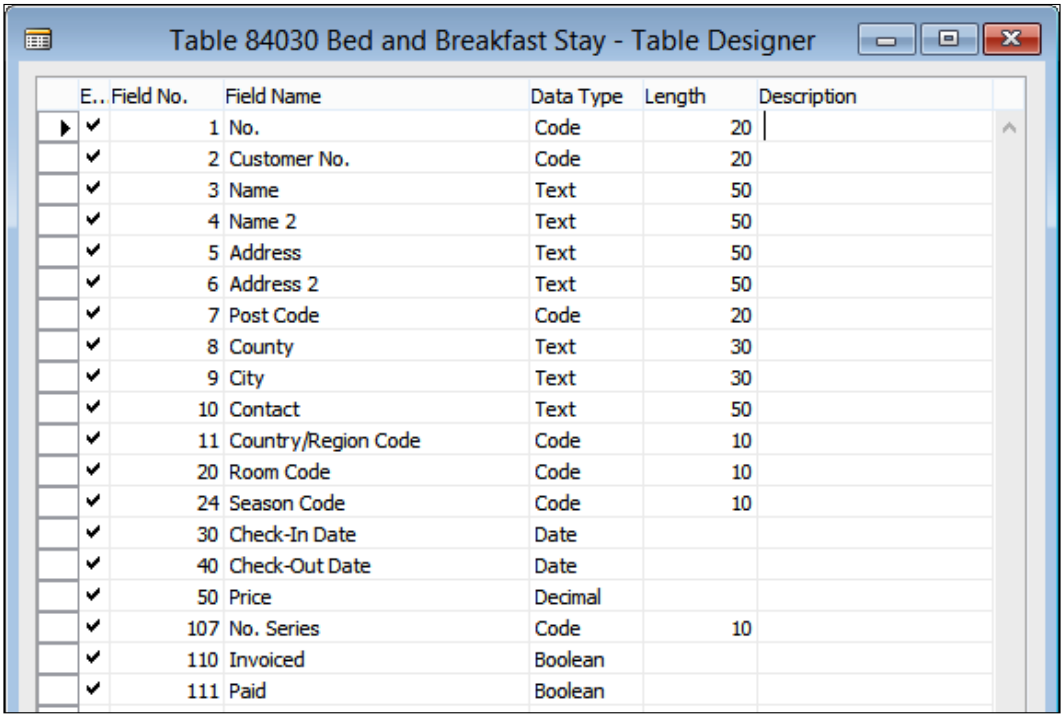
Bed and Breakfast stay

This table will also implement the Number Series, and we will reference the Customer and Room table.

The Stay has a `Check-In` method. When checking in, we can apply the Stay to a reservation. For a Stay at a B & B, a reservation is mandatory. We will implement a method to create a reservation from the Stay.

We will also implement a `Price` field that is calculated using a method on the table that uses the method on the Room.

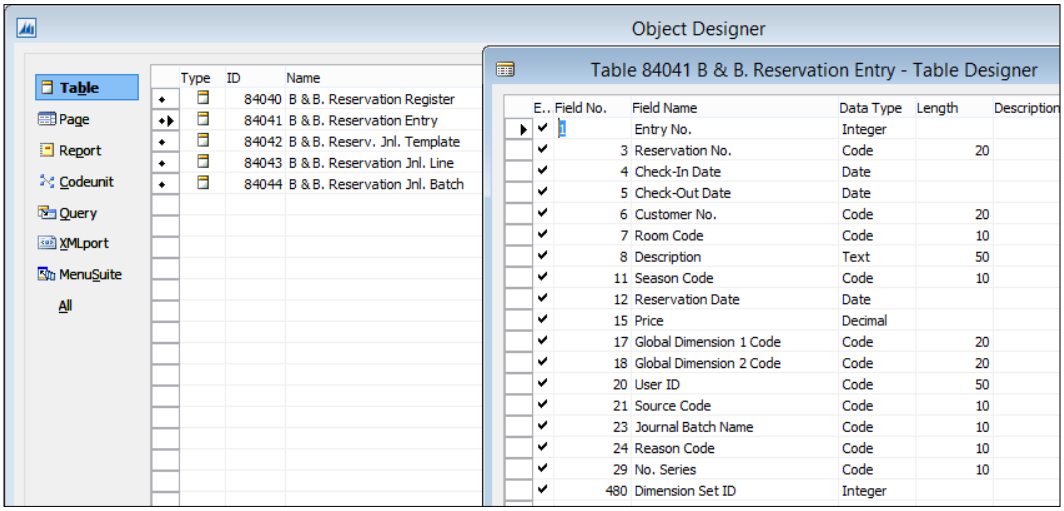
We can end the stay with the Check-Out method, which will create the Sales Invoice and the Evaluation. Both will be encapsulated in their own methods.



E..	Field No.	Field Name	Data Type	Length	Description
✔	1	No.	Code	20	
✔	2	Customer No.	Code	20	
✔	3	Name	Text	50	
✔	4	Name 2	Text	50	
✔	5	Address	Text	50	
✔	6	Address 2	Text	50	
✔	7	Post Code	Code	20	
✔	8	County	Text	30	
✔	9	City	Text	30	
✔	10	Contact	Text	50	
✔	11	Country/Region Code	Code	10	
✔	20	Room Code	Code	10	
✔	24	Season Code	Code	10	
✔	30	Check-In Date	Date		
✔	40	Check-Out Date	Date		
✔	50	Price	Decimal		
✔	107	No. Series	Code	10	
✔	110	Invoiced	Boolean		
✔	111	Paid	Boolean		

Bed and Breakfast journal and entry

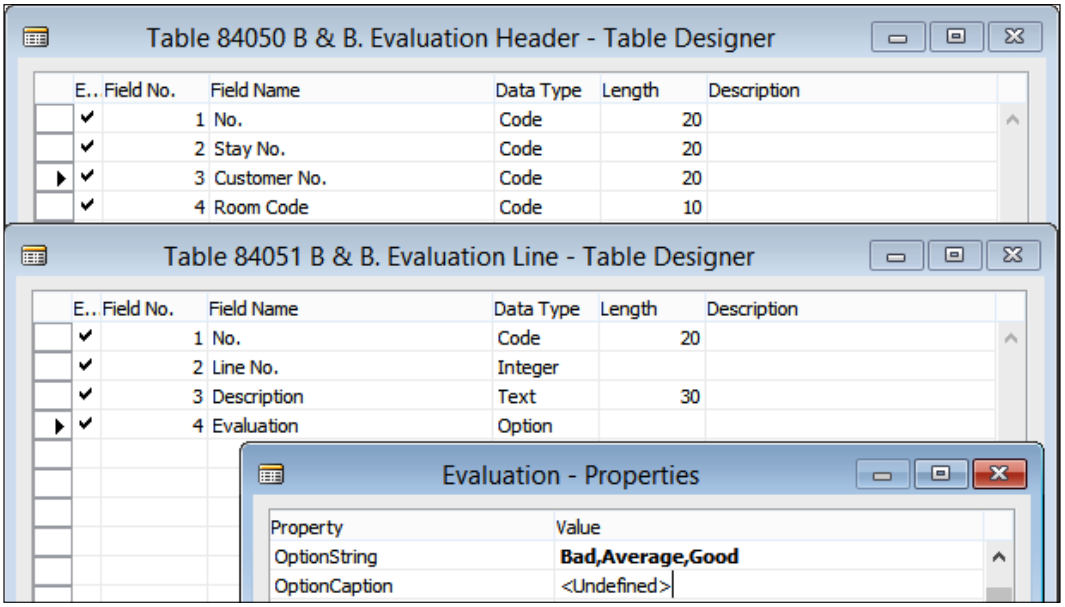
We use the standard Journal Pattern for the Bed and Breakfast reservation process. This means we have three tables for the journal: an entry table and a register table. We won't use the recurring feature or the detailed entries.



The Bed and Breakfast evaluation header and evaluation line

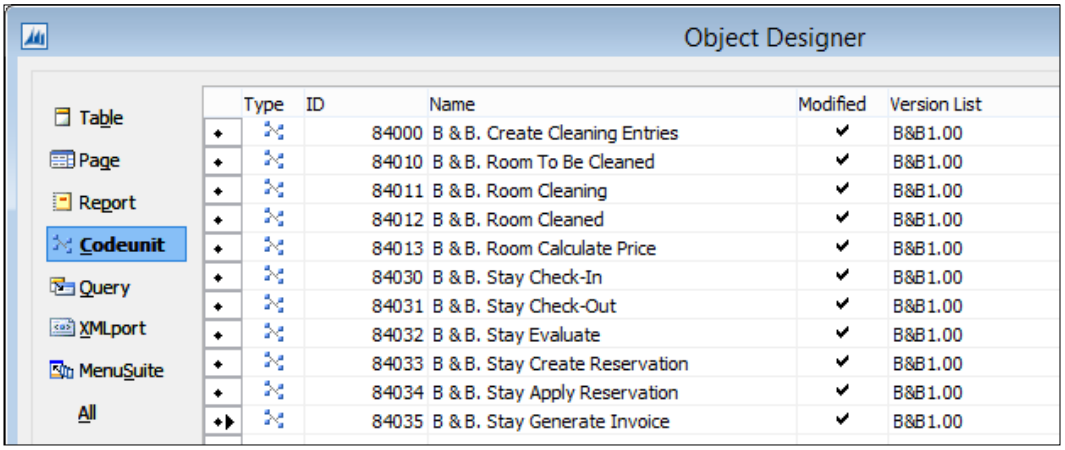
The Evaluation table will reference the Stay table, and inherit the Customer and Room information. We will add an Entity State field, indicating that the evaluation is completed.

The Line table will contain a description and an option field.



Defining the methods

We need a few methods for our B & B application that are linked to the tables. For each method, we will create a Codeunit that will be called as a function on the table.



Writing the Code and Link methods

The next step is to write code and link the methods as a function to the table. Depending on the complexity of the code, we can leave it in one function, or use Natural Language Programming and Encapsulation to create multiple functions that explain our intention.

We will first focus on one function and explain how to structure the code.

Creating the cleaning entries

This function will be called automatically each day by the Job Queue, and has to create a Cleaning Entry for each room that is occupied or has not been cleaned for a week:

```
OnRun()
CreateCleaningEntryForEachRoom;
```

Since our function contains an iteration through a room record set, the Main function is a single description of the iteration.

In the following function, we again describe the business logic while programming the loop. Both the functions that we call have a meaningful name:

```
LOCAL CreateCleaningEntryForEachRoom()
WITH Room DO
  IF FINDSET THEN BEGIN
    REPEAT
    IF OccupiedOrHasNotBeenCleanedForAWeek(Room) THEN
        CreateCleaningEntryForRoom(Room);
  UNTIL NEXT = 0;
  END;
```

By implementing Readability and Reducing **Cyclomatic Complexity,** we can create code that is easy to understand by adding a function called `FindLastCleaningDate` and `LastWeek`:

```
LOCAL OccupiedOrHasNotBeenCleanedForAWeek(Room : Record "Bed and
Breakfast Room") : Boolean
WITH Room DO BEGIN
  IF NOT Vacant THEN
    EXIT(TRUE);
```

```
    IF FindLastCleaningDate(Room) < LastWeek THEN
      EXIT(TRUE)
  END;

  EXIT(FALSE);
```

The following function returns the last record in the Cleaning Entry table for a Room sorted by Date:

```
LOCAL FindLastCleaningDate(Room : Record "Bed and Breakfast Room")
: Date
BEGIN
  WITH CleanEntry DO BEGIN
    SETCURRENTKEY(Date);
    SETRANGE("Room Code", Room.Code);
    IF FINDLAST THEN
    EXIT(Date);
  END;
END;

EXIT(0D);
```

The following function only exists for readability. It returns the last week's date:

```
LOCAL LastWeek() : Date
  EXIT(WORKDATE - 7);
```

And finally, we can insert the cleaning entry:

```
LOCAL CreateCleaningEntryForRoom(Room : Record "Bed and Breakfast
Room")
WITH CleanEntry DO BEGIN
  "Entry No." := 0;
  Date := WORKDATE;
  Status := Status::"To be Cleaned";
  "Room Code" := Room.Code;
  INSERT;
END;
```

Refactoring

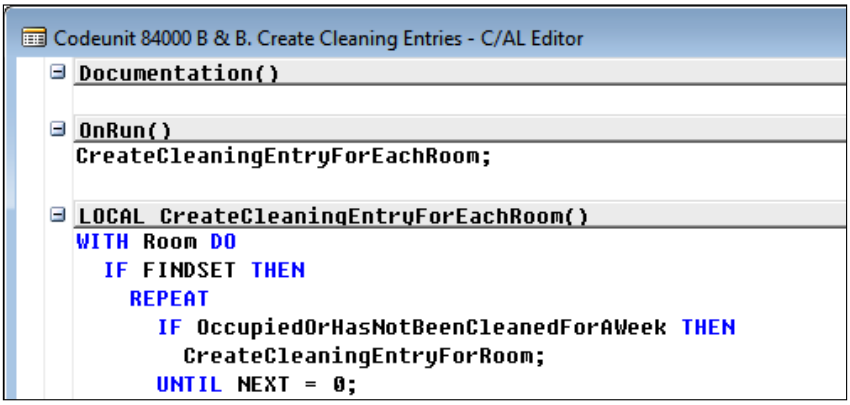
If we read the code that we have just written, we can immediately start cleaning the code, and then start refactoring it.

Let's browse through the functions one by one, see how we can refactor, and see what the result looks like.

The new class members

Quite a number of functions can be the members of a class or the table. We can add the function called `OccupiedOrHasNotBeenCleanedForAWeek`, `CreateCleaningEntryForRoom`, `FindLastCleaningDate`, and `CreateCleaningEntryForRoom` as members of the Room table.

If we do this, our Codeunit will look like the following:

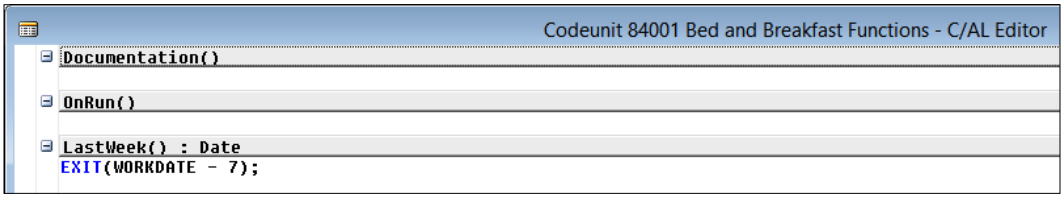


The Codeunit is very clean and refactored.

The new function library

One of the functions cannot be moved as a member of the Room table. The `LastWeek` function is a generic function.

Let's create a new function library and call it the `Bed and Breakfast` function, as shown here:

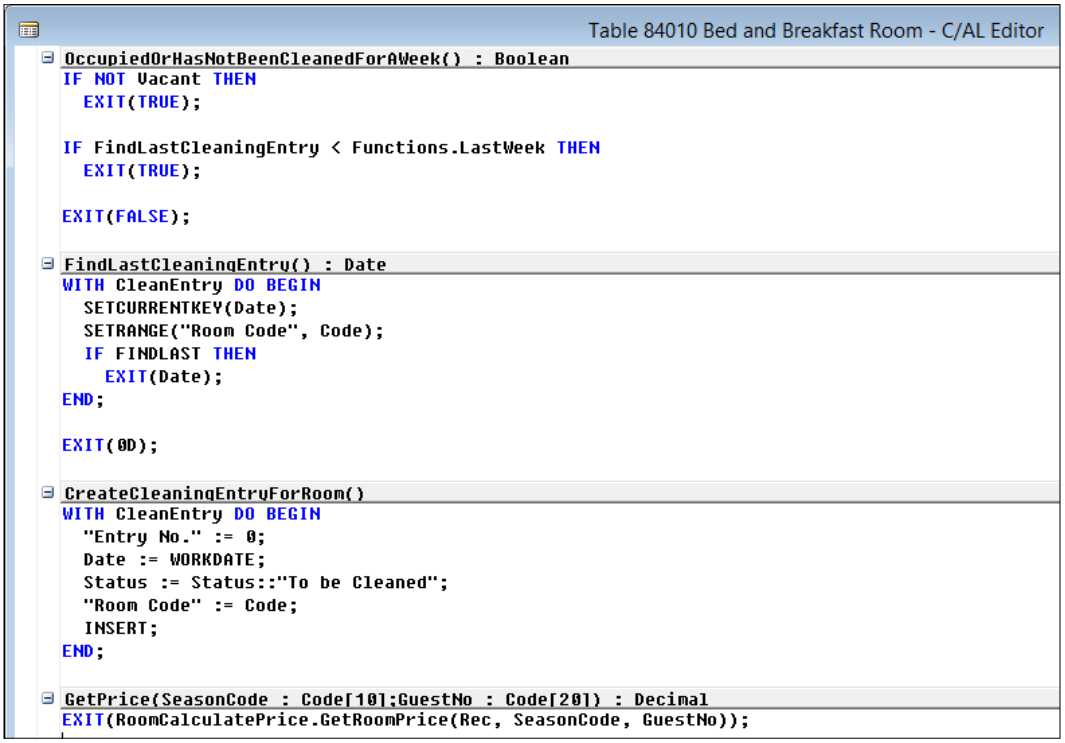


Room members

The Room table now has three members that can be generically reused whenever appropriate.

We can probably make these functions more generic. For example, by adding the Cleaning Entry Status as a parameter to the function.

We can also refactor the functions into their own Codeunit. This will make sense if our development team has different programmers working on application areas that will overlap. If this is not likely, then the extra overhead might not be worth the hassle.



The room price calculation

We need the calculation of a room price in a number of places. At least the Journal and the Stay have this requirement. We can also foresee a future enhancement of a Web Service returning a price, based on an inquiry or a wizard for manual inquiries.

Although the price is based on three elements—Room Code, Season Code, and Guest No., the Room is the only process that is mandatory and can be assigned as the owner of the process. Rather than implementing the price calculation as a member of the price table, we will implement this as a member of the Room. From a functional perspective, the reason is that we know the room and want to know the price, and not the other way around.

The following code is the function on the Room table. It can be called with the optional members—Season Code and Customer No.-and calls into a method Codeunit with these parameters:

```
GetPrice(SeasonCode : Code[10];CustomerNo : Code[20]) : Decimal
EXIT(RoomCalculatePrice.GetRoomPrice(Rec, SeasonCode,
CustomerNo));
```

Each table that requires the calculation of a Room price has its own function that attempts to read the room from the database, and then calls into the `GetPrice` function:

```
GetRoomPrice()
WITH Room DO BEGIN
  IF NOT GET("Room Code") THEN
    EXIT;
  Price := GetPrice("Season Code", "Customer No.");
END;
```

The Codeunit first does the calculation, based on the most detailed price. The following code is the implementation of the Rules table. Alternatively, we can also find the best or lowest price, as it is implemented in Sales pricing in Microsoft Dynamics NAV:

```
GetRoomPrice(Room : Record "Bed and Breakfast Room";SeasonCode :
Code[10];CustomerNo : Code[20]) : Decimal
WITH BedandBreakfastPrice DO BEGIN
  IF GET(Room.Code, SeasonCode, CustomerNo) THEN
    EXIT(Price);

  IF GET(Room.Code, '', CustomerNo) THEN
    EXIT(Price);

  IF GET(Room.Code, SeasonCode) THEN
    EXIT(Price);

  IF GET(Room.Code) THEN
    EXIT(Price);
END;
```



When using compound keys in rules tables, the last values in the `GET` statement can be left blank. This allows flexibility for the key fields without breaking the code.

Testing the application

After we have built all the functions, we can start testing the application. Please note that we have not built any user interface. The basic methods of our application can all be tested without the UI. We will only need to test the UI code if the code was intended to influence the UI.

We need to build a test Codeunit, containing the elements that we would like to test. It should also create the Setup and Master Data that we need.

This can be split into two test code units, but for the sake of simplicity, we have combined them into one.

Codeunit 84098 Bed and Breakfast Test - C/AL Editor

```
Documentation()
OnRun()
[Test] CreateNumberSeries()
InternalCreateNumberSeries('STAY', 'Stays', 'STAY0001');

[Test] CreateSetup()
WITH BedandBreakfastSetup DO BEGIN
  "Stay Nos." := 'STAY';
  INSERT;
END;

[Test] CreateRooms()
[Test] CreateSeasons()
[Test] CreatePrices()
[Test] CreateReservations()
[Test] CreateStays()
[Test] CreateCleaning()
[Test] TestPrice()
[Test] TestCleaning()
[Test] TestVacant()
[Test] TestCheckIn()
[Test] TestCheckOut()
InternalCreateNumberSeries(NoSeriesCode : Code[10];NoSeriesDescription : Text;NoSeriesStartingNo : Code[20])
InternalCreateRoom(RoomCode : Code[10];RoomDescription : Text)
InternalCreateSeaon(SeasonCode : Code[10];SeasonDescription : Text)
InternalCreatePrice(RoomCode : Code[10];SeasonCode : Code[10];GuestNo : Code[20];RoomPrice : Decimal)
InternalCreateReservation(GuestNo : Code[20];RoomCode : Code[10];SeasonCode : Code[10])
InternalCreateStay(GuestNo : Code[20];RoomCode : Code[10];SeasonCode : Code[10])
InternalGetCust(Offset : Integer) : Code[20]
```

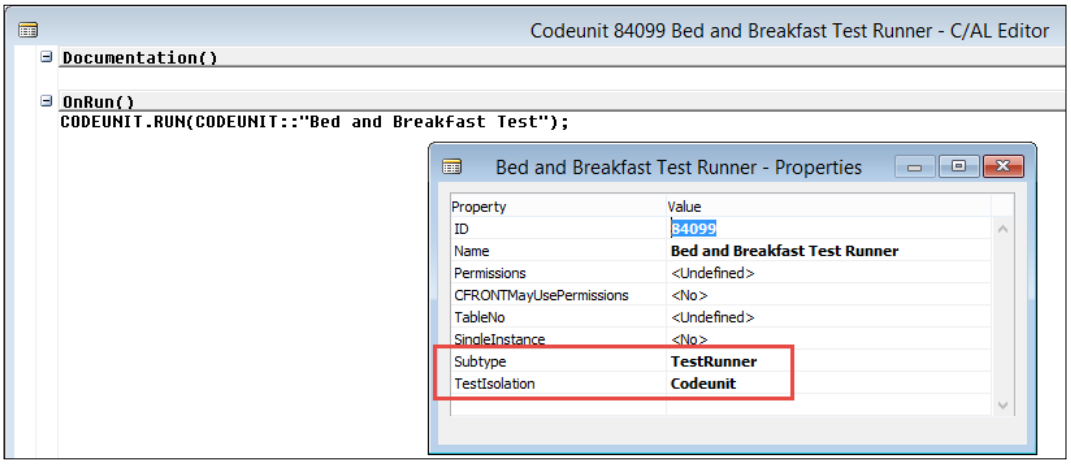
The first set of test functions implements the data, whereas the second set tests the result. We have a set of helper functions to avoid duplicate code.

One of the tests is the price calculation. We can create a specific record in the price table, and then test whether the price is applied:

```
[Test] TestPrice()
Stay.GET('STAY0001');
Stay.TESTFIELD(Price, 10);
```

Test isolation

During development, we might need to run the test multiple times. In order for the test to be complete, we have to start with a fresh situation before each run. This can be achieved by implementing Test Runner, which runs in the `TestIsolation` mode of the Codeunit.



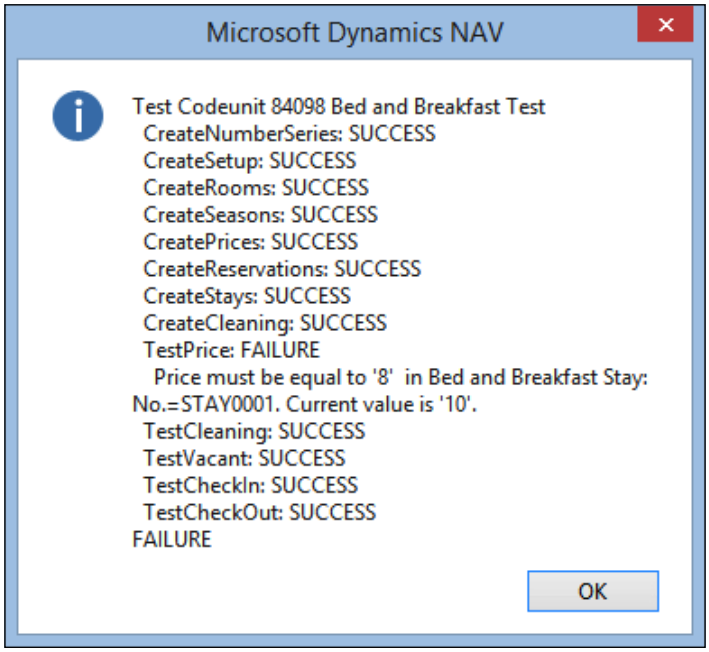
This guarantees that we can run the result, and the data is not actually stored in the database.



The MSDN article at `http://blogs.msdn.com/b/nav/archive/2012/12/03/writing-unit-tests-in-c-al.aspx` explains the writing of test routines, and how the `TestIsolation` function works.

Analyzing the results

If we run the **TestRunner** Codeunit from **Object Designer**, we get a window displaying the result of each test function. If none of the tests fail, the status of the run is a success. Each error is displayed in a window, similar to this:



The `ASSERTERROR` command can be implemented to write code against the expected errors. This allows us to write specific tests against commonly occurring errors, or the typical scenarios that we try to catch.



The MSDN article at `https://msdn.microsoft.com/en-us/library/ee414211.aspx` explains the `ASSERTERROR` function.

Upgrading the application

We would like to package our application to be easily upgraded to other versions of Microsoft Dynamics NAV. This can either be a new version or a cumulative update. We can also package our software to be implemented in a different localization (country), or even downgrade to an older version of the application.

Creating a Delta file

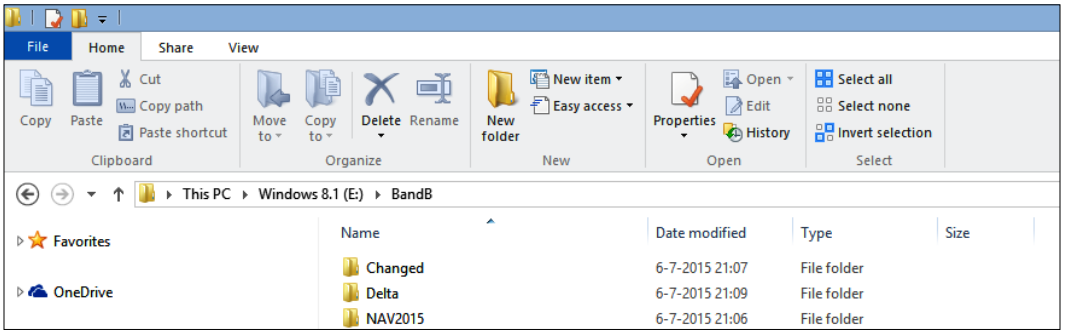
To ship our software safely between versions, we need to create a Delta file. This defines the difference between our application and the standard application that we started off with when we started programming.

The Delta file allows both the entirely new objects and the changes that we have implemented to the standard Dynamics NAV objects. When the Delta file is applied to another version, we will get a warning if the change cannot be applied because of a conflict or missing reference.

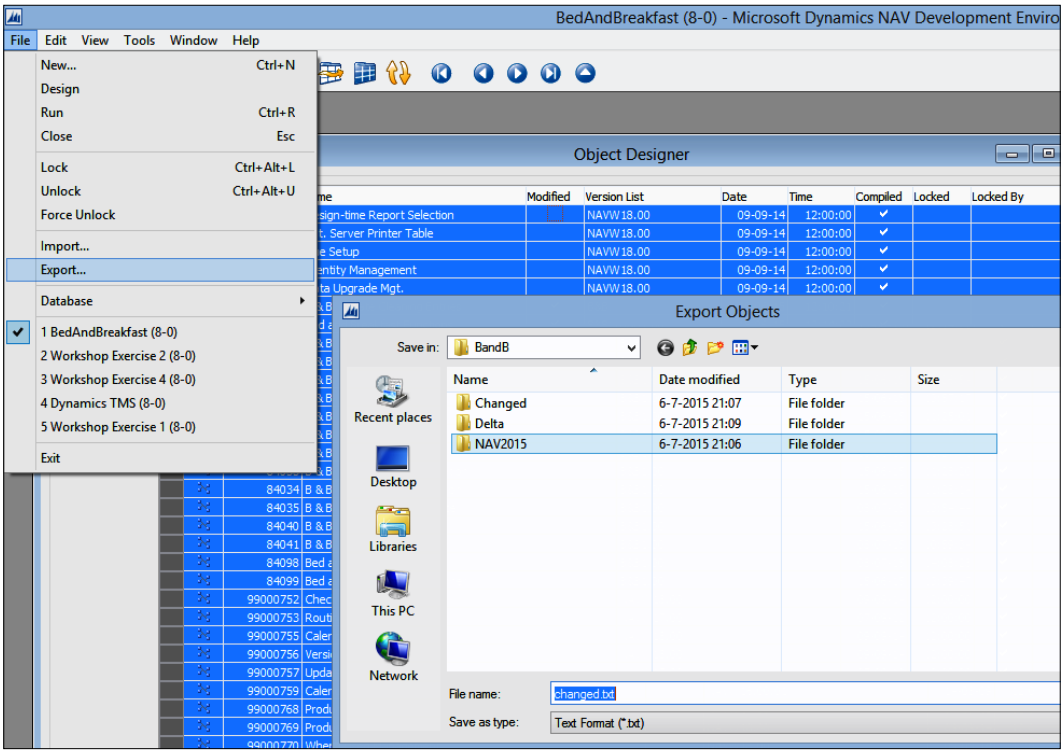
The steps of creating a Delta file

Let's generate the Delta file from our B & B application step by step. To do this, we must use the **PowerShell** commands that are installed when we install Dynamics NAV 2015. These are as follows:

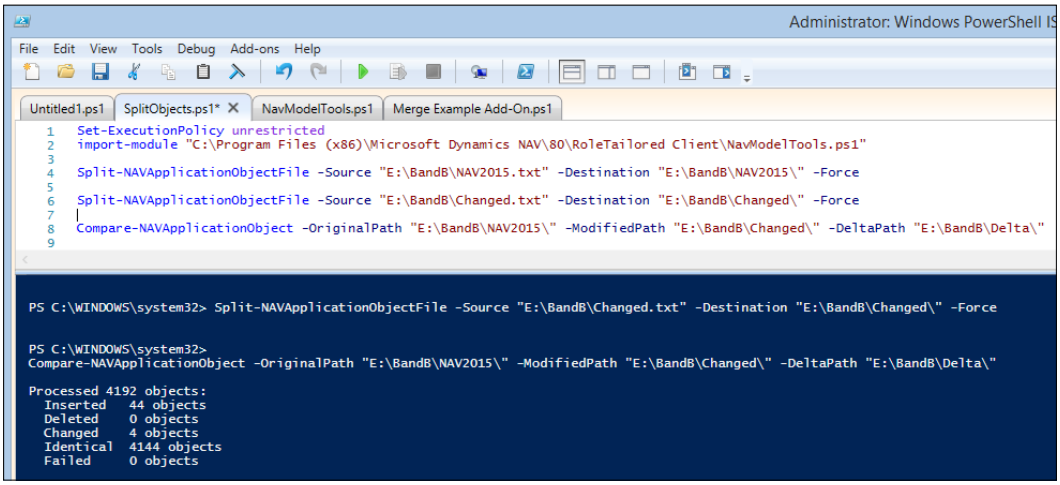
1. **Creating a base version and folder structure**: we need a folder on a drive with three subfolders. We will call the folder `BandB`. Here, we have a `NAV2015` folder that contains the unchanged objects, a `Changed` folder that contains the Bed and Breakfast objects, and a `Delta` folder for the Delta files:



2. **Exporting text files**: We need to export the objects from Dynamics NAV as text files. This is something that can be done manually or via PowerShell. For reasons of simplicity, we will do it manually.



3. **Splitting the objects, and comparing and generating the Delta files**: to compare the objects, we need to split the text file into one file per object, and then run the compare. We'll do this using PowerShell, as shown in the following screenshot:





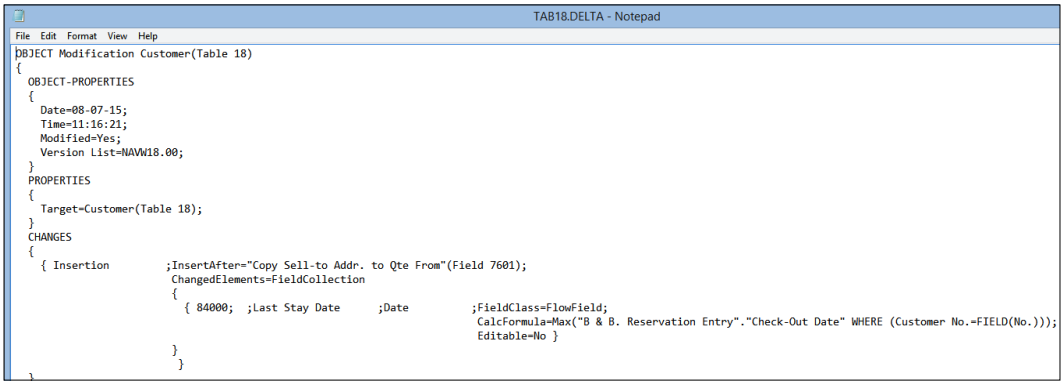
You can find more information at `https://msdn.microsoft.com/en-us/library/dn789581(v=nav.80).aspx` on how to use PowerShell on MSDN.

Inside the Delta file

According to the PowerShell result, we should have 48 Delta files, 44 inserted objects, and 4 changed objects.

The syntax of a Delta file is similar to normal text objects, and an inserted Delta file is identical to a normal text object.

The changed objects are more interesting, as displayed in the following screenshot:



We can see that the file is a modification to the Customer table. The `CHANGES` tag describes an insertion of a flow field with `ID 84000`. This Delta file can be applied to any text file of any version of NAV2009R2 and newer. We can potentially downgrade our solution using the Delta files.



MSDN and the Dynamics NAV team blogs have a lot of articles on using the Delta files and PowerShell. One of them is available at: `http://blogs.msdn.com/b/nav/archive/2014/10/03/merging-application-objects-using-windows-powershell-in-microsoft-dynamics-nav-2015.aspx?PageIndex=1`

Building our application

We can use our Delta file to build our application, based on a new version of the software that also uses PowerShell commands.

PowerShell is, as the name implies, a very powerful tool that not only supports Dynamics NAV, but most Microsoft applications including the SQL Server. This allows us to generate a single script that restores a new SQL Database, applies our Delta file, compiles objects, and runs the test.

Let's go through this process step by step using an example script. In this script, the parameters are hardcoded for readability. It is recommended that you use variables instead.

1. **Prepare**: the first step is to prepare some mandatory variables, such as a reference to the NAV Ide. Next, load the libraries that we need. We need the SQL Server library and both the NAV libraries:

```
$NavIde = "C:\Program Files (x86)\Microsoft Dynamics
NAV\80\RoleTailored Client\finsql.exe"

Set-ExecutionPolicy unrestricted

import-module "C:\Program Files (x86)\Microsoft Dynamics
NAV\80\RoleTailored Client\NavModelTools.ps1"
import-module "C:\Program Files\Microsoft Dynamics
NAV\80\Service\NavAdminTool.ps1"

Push-Location
import-module sqlps
```

2. **Restoring database**: the intention is to start with a fresh copy of the Dynamics NAV database. We do so by stopping the service tier, dropping the existing database, and restoring a new database. After the restore, we connect the service tier user to the database and restart the service tier:

```
Set-NAVServerInstance "bedandbreakfast" –Stop

invoke-sqlcmd -ServerInstance ".\NAVDEMO" -Query "ALTER
DATABASE [BedAndBreakfast (8-0)] SET SINGLE_USER WITH
ROLLBACK IMMEDIATE"
invoke-sqlcmd -ServerInstance ".\NAVDEMO" -Query "DROP
DATABASE [BedAndBreakfast (8-0)]"
```

```
invoke-sqlcmd -ServerInstance ".\NAVDEMO" -Query "RESTORE
DATABASE [BedAndBreakfast (8-0)] FROM DISK = N'E:\Demo
Database NAV (8-0).bak' WITH FILE = 1, MOVE N'Demo Database
NAV (8-0)_Data' TO N'C:\SQLFiles\Demo Database NAV (8-
0)_Data.mdf', MOVE N'Demo Database NAV (8-0)_Log' TO
N'C:\SQLFiles\Demo Database NAV (8-0)_Log.ldf', NOUNLOAD,
STATS = 5"

Invoke-Sqlcmd -ServerInstance ".\NAVDEMO" -Database
"BedAndBreakfast (8-0)" -Query "CREATE USER [NT
AUTHORITY\NETWORK
SERVICE] FOR LOGIN [NT AUTHORITY\NETWORK SERVICE]"

Invoke-Sqlcmd -ServerInstance ".\NAVDEMO" -Database
"BedAndBreakfast (8-0)" -Query "exec sp_addrolemember
'db_owner',
'NT AUTHORITY\NETWORK SERVICE'"

Set-NAVServerInstance "bedandbreakfast" –Start
```

3. **Exporting the text files**: since we cannot import the Delta files directly into the NAV database, we have to generate a set of text files to apply the Delta files. This is done using the PowerShell commandlets:

```
Export-NAVApplicationObject -DatabaseName "BedAndBreakfast
(8-0)" -Path "E:\BandB\Target.txt" -DatabaseServer
".\NAVDEMO"

Split-NAVApplicationObjectFile -Source
"E:\BandB\Target.txt" -Destination "E:\BandB\Target\" -
Force
```

4. **Applying the Delta file**: the next step is the actual merge between the files, and the creation of the result file that we can import into NAV. We remove all objects that are untouched to reduce the amount of objects that are to be compiled:

```
$MergeResult = Update-NAVApplicationObject -TargetPath
"E:\BandB\Target\"
  -DeltaPath "E:\BandB\Delta\" `
  -ResultPath "E:\BandB\Result\" -Force

$MergeResult | Where-Object UpdateResult -eq Unchanged |
foreach { Remove-Item -Path $_.PSPath } -Verbose

Join-NAVApplicationObjectFile -Source "E:\BandB\Result\" -
Destination "E:\BandB\Result.txt"
```

5. **Importing and Compiling**: assuming the file has been generated without conflicts as we designed, we can import the result in the database and compile the objects:

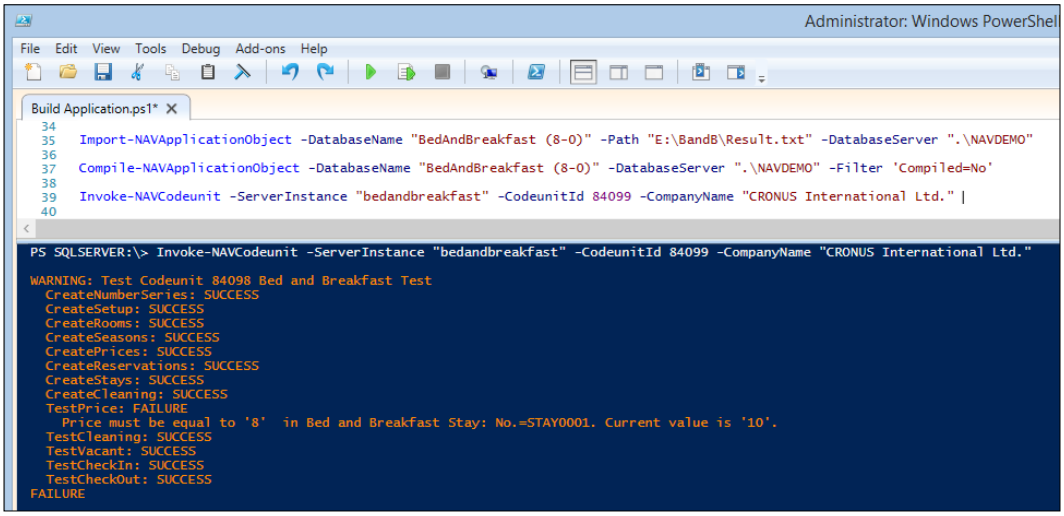
```
Import-NAVApplicationObject -DatabaseName "BedAndBreakfast
(8-0)" -Path "E:\BandB\Result.txt" -DatabaseServer
".\NAVDEMO"

Compile-NAVApplicationObject -DatabaseName "BedAndBreakfast
(8-0)" -DatabaseServer ".\NAVDEMO" -Filter 'Compiled=No'
```

6. **Testing the result**: the last and final build step is to run the Test Codeunit via PowerShell. We do so by invoking the Codeunit:

```
Invoke-NAVCodeunit -ServerInstance "bedandbreakfast" -
CodeunitId 84099 -CompanyName "CRONUS International Ltd."
```

Building the result

The result of the build should look something like the next screenshot. In our case, the build failed because one of the tests failed, as shown in the following screenshot:



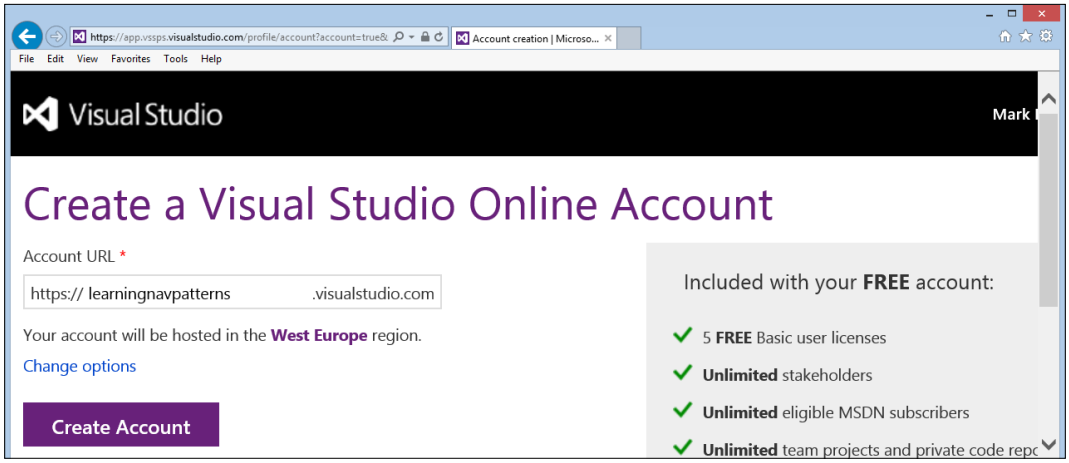
Managing the source code

The sources of our application are now stored as text files on our file system, but this is not a very easy way to collaborate. Keeping the versions of our files is a manual process that requires a lot of discipline.

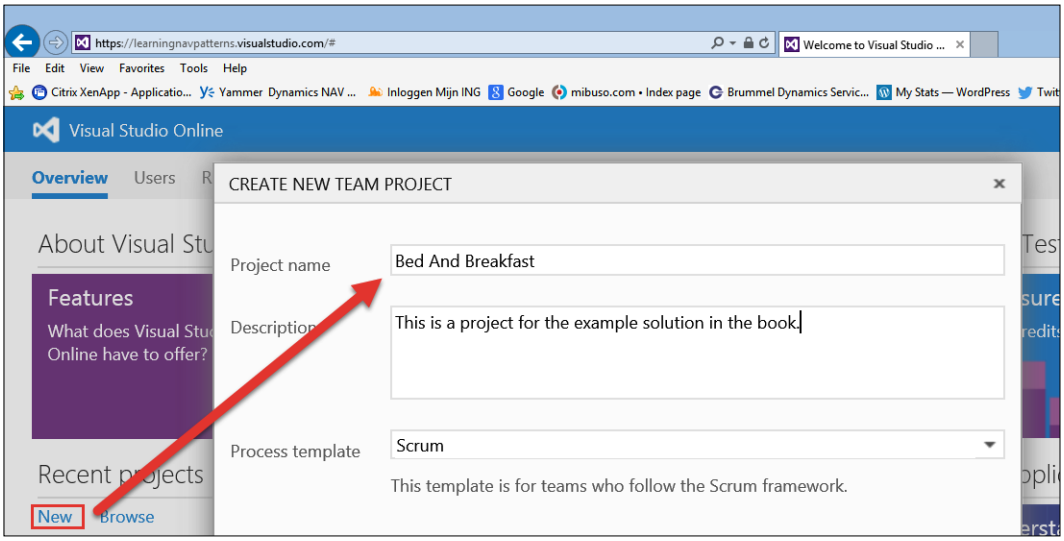
Instead of the manual process, we can use a source code management system that helps us in this process. Since 2014, Microsoft has had a free cloud service for this—**Visual Studio Online**. This is essentially a cloud version of Team Foundation Server. Let's see how we can use this.

Creating the workspace

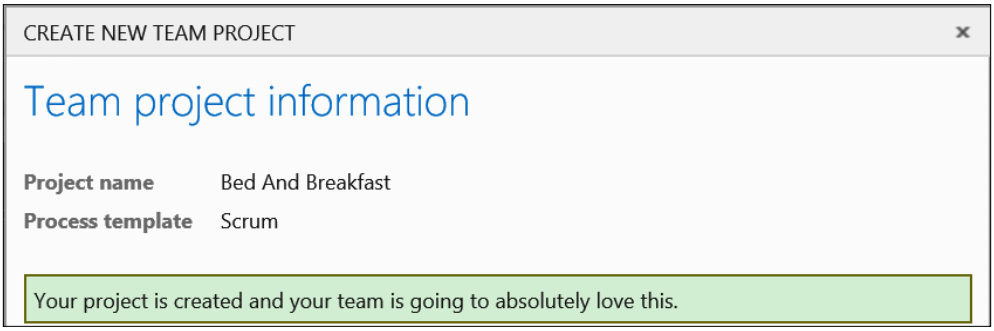
If we go to `http://www.visualstudio.com`, we can set up a new workspace. We will use `http://www.learningnavpatterns.visualstudio.com`. Each workspace gets five free users for the source code control and unlimited users for creating and maintaining tasks. On top of the five free users, you can add users with an MSDN subscription for free too, which is shown as follows:



After the account is created, we can set up a new project; for example; a project for our Bed and Breakfast solution.



After generating the project, it is ready to be used.



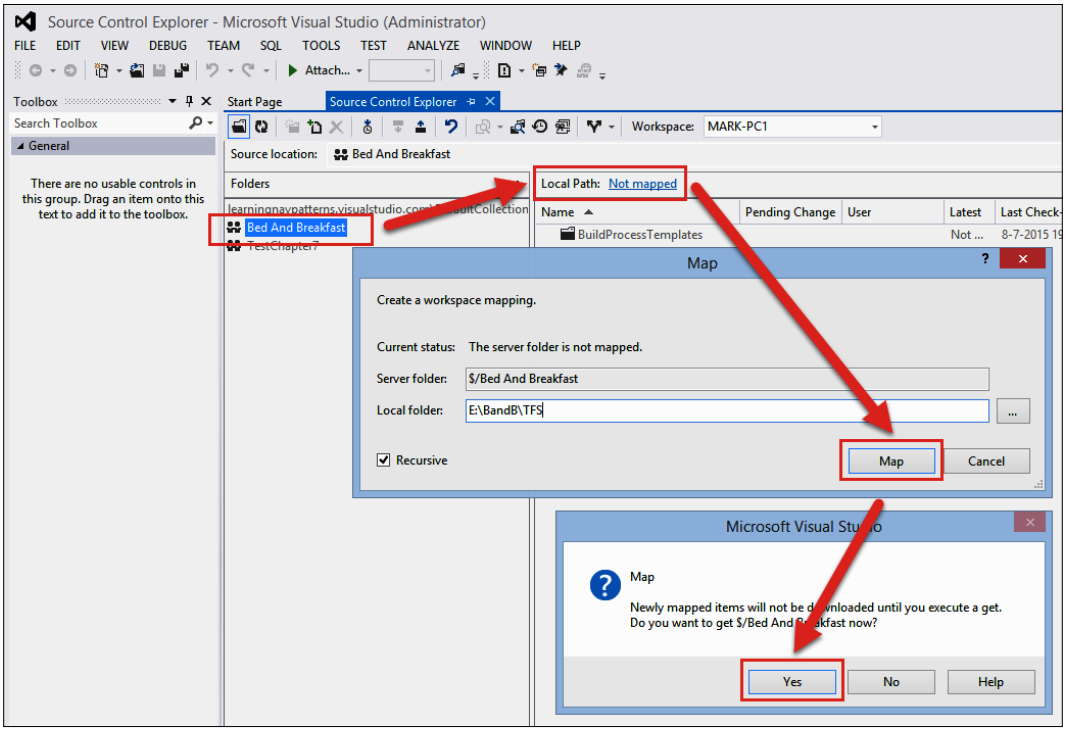
Setting up Visual Studio

We will use Visual Studio 2015 to connect to the cloud service and upload the files. This can be done in almost any version of Visual Studio including the community edition, which is available for free.



To create the connection, you can watch the videos on this YouTube channel at `https://www.youtube.com/playlist?list=PLhZ3P-LY7Cqlh_z1rLOAAAAk9Y2clUhHl`.

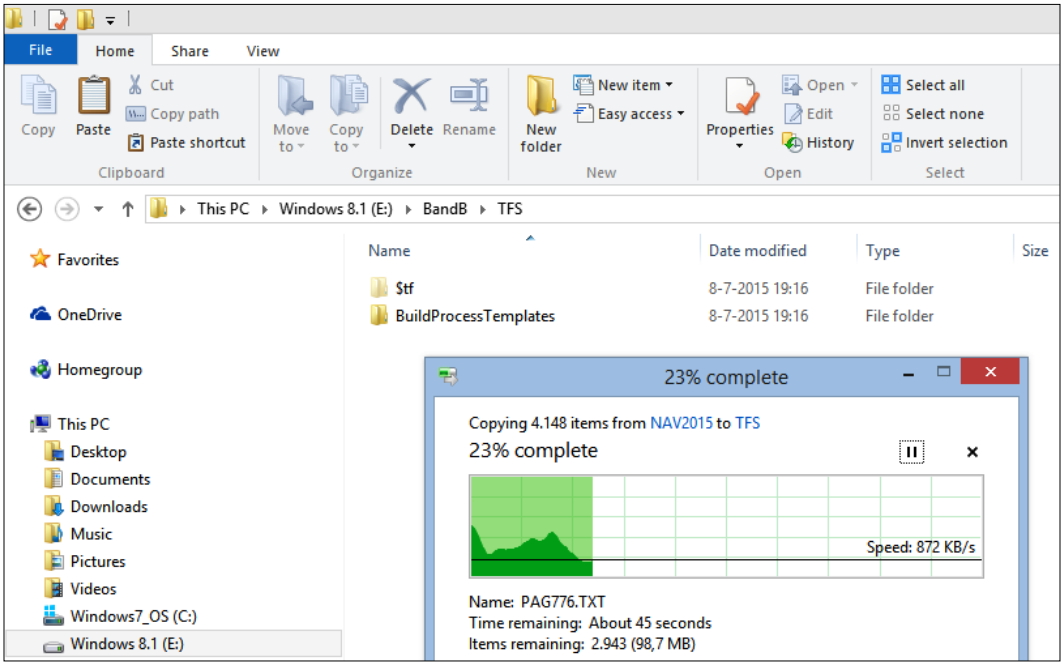
When the connection is created, we can map the files to our local folders.



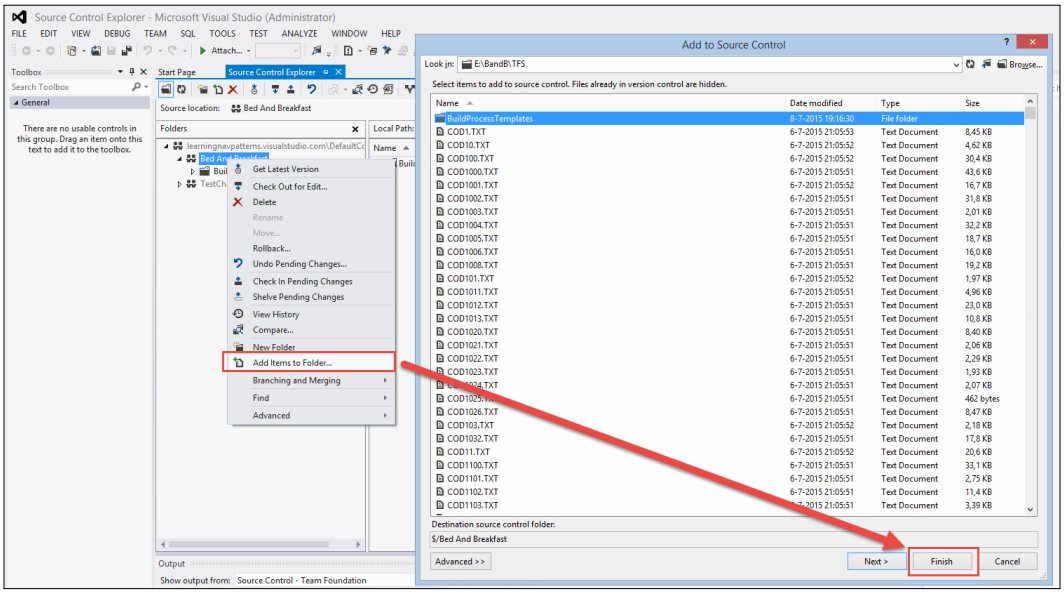
Uploading the files

Now we have to upload the files. There are many ways to do this, and we will use the simplest way. We will first upload the initial version that we started programming with, NAV2015, and then upload the 1.0 version of the software.

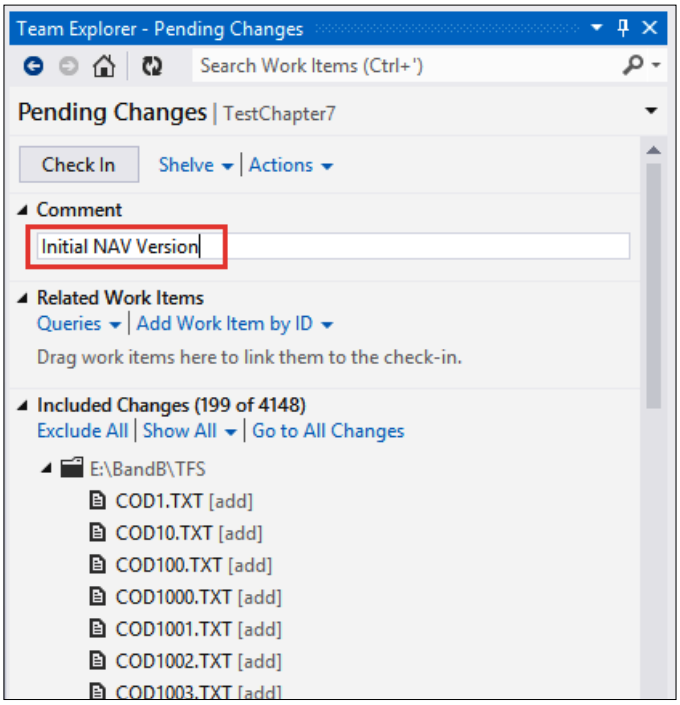
To do this, we need to copy the individual text files from the folder that we had created earlier with PowerShell to the TFS folder.



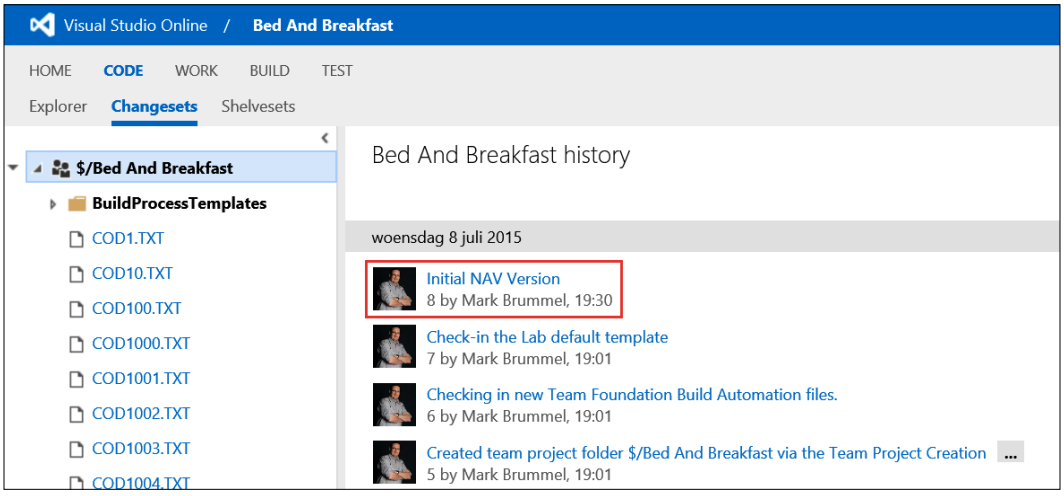
The next step is to add these in Visual Studio. This is not something that happens automatically.



After this is done, the changes are pending to be committed. We can do this in the **Team Explorer** window. We need to make sure that the description makes sense.

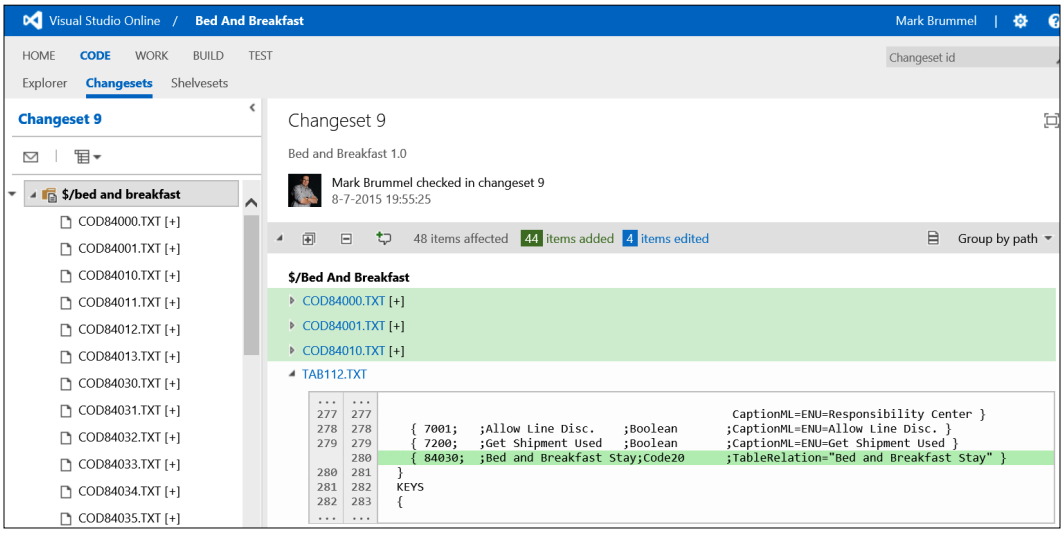


Now, the changes are committed, and we can see the **Changesets** online too, as shown here:



To upload the Bed and Breakfast files, we need to repeat the steps and check in again. Let's fast forward to this.

In the following screenshot, we can see that we've imported **changeset 9**, **Bed and Breakfast 1.0**. It shows the changes per object, compared to other versions:



The simplicity of setting up a versioning system in Visual Studio Online, in combination with the PowerShell command, allows us to make it very powerful for an average Dynamics NAV implementation.

Summary

In this chapter, you have learned how to break up software development into tasks that allow us to implement the methodology we learned in this book. You learned how to map functional elements to classes and methods, and apply the Architectural and Design Patterns.

After defining the base building blocks, we started coding and continued to normalize until the desired level of the code normalization was required.

We wrote automated test scripts that allowed us to test our code with a single action with the data isolation.

This application could then be easily upgraded or downgraded to other versions using the Delta file and PowerShell.

PowerShell also allowed us to build a new database with our software, and let us apply the Delta file and run the automated test. We saw that Visual Studio Online can be used to store our source code in a safe place.

The methodology used in this book is also taught in classroom training. Throughout the year, a team of teachers travels and organizes classes across the globe. Have a look at `http://nav-skills.com/agenda-master-class-nav-architecture/` for the current schedules.

Index

A

Address Integration Pattern
about 48
address, copying 50
address, printing 51
examples 52
implementations 52
reference link 49
technical description 49
anti-patterns
about 5, 124
and context 134
boat anchor 132-134
code cloning 5, 124
examples 125
language capabilities, programming 135
transaction mirroring 125, 126
application, building 163, 164
application, testing
about 157
results, analyzing 159
test isolation 158
application, upgrading
about 159
Delta file, creating 160
Delta file, overview 162
Step-By-Step 160, 161
Architectural Patterns, applying
about 142
application setup 143
check-in 144
check-out 144
clean 144
evaluation 144
guests 143
invoice 145
price 143
reservation 144
room 143
stay 144
Argument table
about 38
implementing 39
technical description 38
ASSERTERROR command
about 159
reference link 159
AutoIncrement property 32
AutoSplitKey property 37

B

base application
about 15
functional processes 15
blocks
building 11, 142
Codeunits 13
MenuSuite object 14
Page objects 12
Query objects 13
Report objects 13
Table object 12
XMLPort objects 13
boat anchor
about 132-134
reference link 134
boy scout rule 140

C

C/AL 3, 45
C/AL Code
refactoring 136
challenges, for upgrading Microsoft Dynamics NAV
Code Upgrade 6
Data Upgrade 6
charting 10
classes 3
Classic Client 94
Class-Method-Property
about 102
Codeunit, as method 104, 105
Table, as class 102, 103
clean code 126
clone 5
code
structuring 4
Code and Link methods, writing
about 152
cleaning entries, creating 152, 153
code cloning
about 124
avoiding 131, 132
customizing 125
exact match 125
forking 124
reference link 125
small business application 127-129
templating 124
VAT and Sales Tax calculation 130, 131
code normalization 126
code refactoring
about 139
breaking down, into functions 139
code, normalizing 140
functions, encapsulating 139
moving, to Codeunit 139
process owner, finding 139
Codeunits, Journal Template-Batch-Line Pattern
Jnl.-B.Post 29
Jnl.-B.Post+Print 29
Jnl.-Check Line 30
JnlManagement 29
Jnl.-Post 29
Jnl.-Post Batch 29
Jnl.-Post Line 30
Jnl.-Post+Print 29
coding guidelines
about 93
design 94, 95
readability 98-100
reference link 94
coding standards 5
Compound Pattern
about 21
examples 22
implementing 22
technical description 21
Copy Document Pattern
about 57
examples 59
implementations 58
reference link 57
technical description 58
Create, Read, Update, and Delete (CRUD) 12
Customer connection, with Vendors
reference link 50
customizing 125
Cyclomatic Complexity
about 96, 152
reference link 96
Cyclomatic Complexity Three (CC3) 96

D

default values 45
delta files 8
design, coding guidelines
about 94, 95
complex data types, using 97
Cyclomatic Complexity 96
date formulas 98
function size 96
localizability 97, 98
variable capacity 96
design elements, example application 67

design pattern
about 4
and repeatability 8
and upgrades 8
lifecycle 9
Design Patterns, applying
about 145
Bed and Breakfast clean entry 148
Bed and Breakfast evaluation header 150
Bed and Breakfast evaluation line 150
Bed and Breakfast journal and entry 150
Bed and Breakfast price 148
Bed and Breakfast room 147
Bed and Breakfast season 146
Bed and Breakfast setup 146
Bed and Breakfast stay 149
Dimensions Pattern 25
Dismissible dialogs & save preferences
about 61
examples 63
implementations 62
reference link 61
technical description 62
DLL
managing 121, 122
DLL Hell 121
Document Header table 55
Document History
about 54
examples 57
implementations 57
technical description 54
Document Line table 55
Document Pattern
about 36
examples 38
implementing 38
technical description 36
documents
about 85
history 87
implementing 85, 86
other patterns 88
posting routine, creating 88, 89
Select Pattern, implementing 88
Document type normalization 127
DropDown Field Group 133

E

encapsulation
about 13, 113, 114
override 116
reference link, for video 115
Entity State field
Blocked 41
Entity State Pattern 25
about 41
examples 44
implementations 43
reference link, for implementation 42
State, setting 42, 43
State, testing 43
technical description 42
Entry Pattern
about 31
examples 35, 36
implementing 35
technical description 32
Temporary Tracking Buckets 34
Entry Table types
about 32
entries, deleting 34
historical entries 33
temporary entries 33
exact match 125
example application
building 67
design elements 67
pattern dependencies 67
example application, building
documents, adding 85
foundation elements, using 68
Journals & Entries, using 81
layered model 90
Master Data pattern, using 72

F

Façade pattern
about 116
example 117, 118
FieldClass property 14
Flowfilters 15
forking 124

formatting function 49
foundation elements
about 68
example setup 68
implementation 68-70
Multi-Level Setup pattern 71
which pattern 68
functional oriented patterns 4
function libraries 106, 107

H

Header table 36
historical entries 33
history records 48
hooks
about 118, 119
original source code, disabling 119
pre-defined hooks 120
Hooks Pattern 19

I

implementations, Journal Template-Batch-Line Pattern
about 30
Hierarchy 30
Journals as Contracts 31
Intellisense 101
interfaces 106, 107

J

Job Queue Entry Pattern
about 59
examples 61
implementations 61
reference link 59
technical description 59
Journal Batch table 28
Journal Line table 28
Journals
about 55
recurring 29
reference link 29
Journals & Entries
about 81
creating 81-83
validations 84, 85
Journal Template-Batch-Line Pattern
about 26
implementing 27
technical description 26, 27
Journal Template table 28

L

layered model 91
legacy code 5, 123, 135, 136
lifecycle, design pattern
about 9
new design patterns 10
software, designing in optimal ways 10
technology changes 9
Line table 37
ListPart 37

M

Master Data Pattern
about 20, 24, 72
Address Integration Pattern, implementing 79-81
Entity State Pattern 78, 79
Example Person entity 72, 73
Example Product 74
examples 26
implementing 25
Number Series 74
technical description 24, 25
MenuSuite object 14
metadata
about 11, 14
Flowfields 14
Flowfilters 15
methods 12
defining 151
Microsoft Design Patterns Wiki
URL 43
Microsoft Dynamics NAV
about 1
software architecture 11
Microsoft Dynamics NAV Design Patterns
about 3
functional oriented patterns 4
non-object-oriented patterns 3

object-oriented patterns 3
MiniApp 127
Model-View ViewModel Pattern
about 65
examples 66
implementations 66
technical description 65
multilevel code 108
Multi-Level Setup
about 52
examples 54
implementations 54
technical description 53
multiple series 47
Multi-Tenant environment 7

N

Natural Language Programming
about 107-110
benefits 112
drawbacks 112
reference link, for video 108
non-object-oriented patterns 3
normalization 126
normalized 12, 49
NoSeriesManagement Codeunit, functions
InitSeries 46
LookupSeries 46
TestManual 46
Number Series, Master Data pattern
about 74, 75
code snippets 77
No. Series field, adding 76
Setup fields, adding 75
variables, adding 76
Number Series Pattern 24
about 45
examples 48
implementations 48
reference link 48
technical description 46

O

object-oriented patterns
about 3
Adapter 3
Bridge 3
Builder 3
Façade 3
Model-View-ViewModel (MVVM) 3
Prototype 3
reference link 3
Singleton 3
object-oriented programming (OOP) 2

P

Page objects 12
pattern dependencies, example application 67
patterns
about 2
software design patterns 2
patterns elements 4
patterns hierarchy 4
Person Master Data 24
placeholders 44
posting routine
DeleteExampeDocument function 90
PostExampleOne/PostExampleTwo function 89
PostExJnlLine function 90
TestFar function 89
TestNear function 89
ThrowErrorIfNoSelection function 89
pre-defined hooks
about 120
reference link, for video 120
Procedural software design patterns 2, 3
Production Orders field 41
Product Master Data 24

Q

Query objects 13

R

readability, coding guidelines
about 98, 99
approved abbreviations 102
self-explaining 101
suffix, versus prefix 101
variable naming 100, 101

Record ID 60
record sets 49
refactoring
about 123, 153
new class members 154
new function library 154
room members 155
room price calculation 155, 156
repeatability
about 8
and design pattern 8
Report objects 13
reserved words, MSDN article
reference link 138
return on investment (ROI) 1
reusable 3
Role Tailored Client 133
Rules Pattern
about 22
examples 23
implementing 23
technical description 22

S

Select Distinct statement
about 23
reference link 23
Select Pattern
about 44
examples 45
implementations 45
reference link 44
technical description 44, 45
Setup Update Pattern
about 63
examples 65
implementations 65
technical description 63, 64
Singleton Pattern
about 18
Company-Initialize 19
enhancements 19
examples 19
implementing 18, 19
technical description 18
Singleton table 18, 53
Small Business Application
reference link 62
software architecture, Microsoft Dynamics NAV
about 11
metadata 11
software design patterns
about 2
Procedural software design patterns 2, 3
reference link 2
source code, managing
about 166
files, uploading 168-171
Visual Studio, setting up 167
workspace, creating 166, 167
structure 4
Subform 86
Supplemental & Subsidiary Pattern
about 20
examples 21
implementing 21
technical description 20
Supplemental table 53

T

Table objects
creating 145
tables, refactoring
about 136
normalization 138
renaming 137, 138
technical description, Address Integration Pattern
about 49
fields 49
technical description, Copy Document Pattern
validation logic 58
technical description, Dismissible dialogs & save preferences
about 62
Dismissible dialog 62
technical description, Document History
about 54
Clean Up 55

Codeunit Structure 55
Do It 55
example 56
naming conventions 54, 55
Test Far 55
Test Near 55
TRANSFERFIELDS 56
technical description, Document Pattern
Header table 36, 37
Line table 37
Posting and Archiving 37
technical description, Job Queue Entry Pattern
about 59
error handling 60
main function 60
post-processing 60
preprocessing 59
records, obtaining 60
technical description, Journal Template-Batch-Line Pattern
about 26, 27
Codeunits 29
Journal Batch 28
Journal Line 28
Journal Template 28
technical description, Number Series Pattern
about 46
C/AL functions, and variables 47
setup reference 46
table reference field 46
technical description, Setup Update Pattern
about 63
Restrictions 64
templating 124
temporary entries 33
TESTFIELD function 43
TestIsolation function
reference link 158
total cost of ownership (TCO) 1
transaction 55
transaction mirroring 125, 126
TRANSFERFIELDS function 51
about 56-58
reference link 56

U

upgradability
about 6
upgrade frequency 7
Upgrade Codeunit
URL 70
Upgrade Toolkit 6

V

values 45
variable naming 5
variant variables
about 121
reference link 121
VAT and Sales Tax calculation 130, 131
Visual Studio
setting up 167

W

what you see is what you get (WYSIWYG) 12

X

XMLPort objects 13



Thank you for buying

Learning Dynamics NAV Patterns

About Packt Publishing

Packt, pronounced 'packed', published its first book, *Mastering phpMyAdmin for Effective MySQL Management*, in April 2004, and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern yet unique publishing company that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at `www.packtpub.com`.

About Packt Enterprise

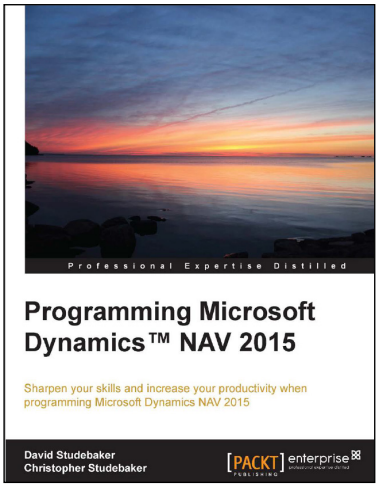
In 2010, Packt launched two new brands, Packt Enterprise and Packt Open Source, in order to continue its focus on specialization. This book is part of the Packt Enterprise brand, home to books published on enterprise software – software created by major vendors, including (but not limited to) IBM, Microsoft, and Oracle, often for use in other corporations. Its titles will offer information relevant to a range of users of this software, including administrators, developers, architects, and end users.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, then please contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

[PACKT] enterprise
professional expertise distilled
PUBLISHING

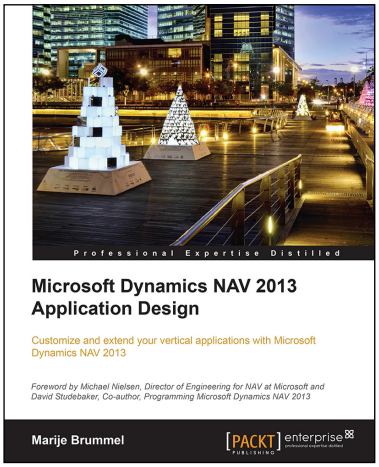


Programming Microsoft Dynamics™ NAV 2015

ISBN: 978-1-78439-420-2 Paperback: 644 pages

Sharpen your skills and increase your productivity when programming Microsoft Dynamics NAV 2015

1. The most comprehensive Dynamics NAV reference guide for programmers, developers, managers, and consultants.
2. Covers all the elements needed to learn how to do development in NAV 2015 C/SIDE and C/AL.
3. A comprehensive guide with detailed explanations and useful examples, presented in a logical, step-by-step manner.



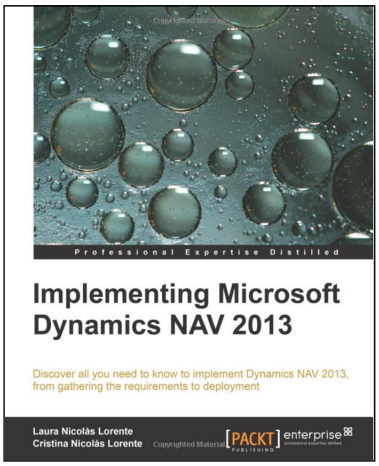
Microsoft Dynamics NAV 2013 Application Design

ISBN: 978-1-78217-036-5 Paperback: 504 pages

Customize and extend your vertical applications with Microsoft Dynamics NAV 2013

1. Set up your application for a number of vertical industries and scenarios.
2. Get acquainted with Dynamics NAV's data model and transaction schema with the help of highly efficient design patterns.
3. Consists of two completely designed and explained vertical solutions, including application objects.

[PACKT] enterprise
professional expertise distilled
PUBLISHING

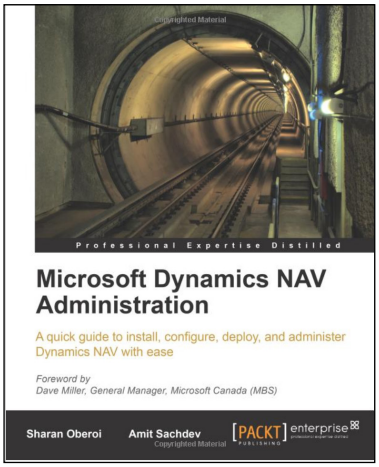


Implementing Microsoft Dynamics NAV 2013

ISBN: 978-1-84968-602-0 Paperback: 554 pages

Discover all you need to know to implement Dynamics NAV 2013, from gathering the requirements to deployment

1. Successfully handle your first Dynamics NAV 2013 implementation.
2. Explore the new features that will help you provide more value to your customers.
3. Full of illustrations and diagrams with clear step-by-step instructions and real-world tips extracted from years of experience.



Microsoft Dynamics NAV Administration

ISBN: 978-1-84719-876-1 Paperback: 190 pages

A quick guide to install, configure, deploy, and administer Dynamics NAV with ease

1. Install, configure, deploy and administer Dynamics NAV with ease.
2. Install Dynamics NAV Classic Client (Dynamics NAV C/SIDE), Dynamics NAV Role Tailored Client (RTC), and Dynamics NAV Classic Database Server on your computer to manage enterprise data.
3. Connect Dynamics NAV clients to the Database Server in the earlier versions and also the latest Dynamics NAV 2009 version.

Please check **www.PacktPub.com** for information on our titles