

SAP PRESS

Design Patterns in ABAP® Objects

- Use design patterns to write better code, faster
- Implement common creational, structural, and behavioral design patterns
- See design patterns in action with real-world SAP applications

Kerem Koseoglu

Rheinwerk Publishing



SAP PRESS is a joint initiative of SAP and Rheinwerk Publishing. The know-how offered by SAP specialists combined with the expertise of Rheinwerk Publishing offers the reader expert books in the field. SAP PRESS features first-hand information and expert advice, and provides useful skills for professional decision-making.

SAP PRESS offers a variety of books on technical and business-related topics for the SAP user. For further information, please visit our website: http://www.sap-press.com.

Paul Hardy
ABAP to the Future (2nd edition)
2016, 801 pages, hardcover and e-book
www.sap-press.com/4161

Kiran Bandari
Complete ABAP
2016, 1,047 pages, hardcover and e-book
www.sap-press.com/3921

James Wood, Joseph Rupert
Object-Oriented Programming with ABAP Objects (2nd edition)
2016, 470 pages, hardcover and e-book
www.sap-press.com/3597

Gahm, Schneider, Swanepoel, Westenberger
ABAP Development for SAP HANA (2nd edition)
2016, 641 pages, hardcover and e-book
www.sap-press.com/3973

Kerem Koseoglu

Design Patterns in ABAP® Objects

Rheinwerk® Publishing

Bonn • Boston

Dear Reader,

A well-designed piece of code is a beautiful thing—a work of poetry. Like poetry, what constitutes a good piece of code may differ from person to person, though all can readily spot some bad lines. Repetitive, messy, with not quite the right syntax; bad code jumps out at you. Enter design patterns!

While design patterns may not be the solution to all your coding woes, they will help you design cleaner code (and hopefully save you some time along the way!) Like a poet choosing a form the fits the theme—a haiku, sonnet, or sestina—you will choose the pattern that fits your need. Expert author Kerem Koseoglu has provided you with the 27 patterns most useful for ABAP programming, and leads you through each with practical examples and good humor.

As always, your comments and suggestions are the most useful tools to help us make our books the best they can be. Let us know what you thought about *Design Patterns in ABAP Objects*! Please feel free to contact me and share any praise or criticism you may have.

Thank you for purchasing a book from SAP PRESS!

Meagan White
Editor, SAP PRESS

Rheinwerk Publishing
Boston, MA

meaganw@rheinwerk-publishing.com
http://www.sap-press.com

Notes on Usage

This e-book is **protected by copyright**. By purchasing this e-book, you have agreed to accept and adhere to the copyrights. You are entitled to use this e-book for personal purposes. You may print and copy it, too, but also only for personal use. Sharing an electronic or printed copy with others, however, is not permitted, neither as a whole nor in parts. Of course, making them available on the Internet or in a company network is illegal as well.

For detailed and legally binding usage conditions, please refer to the section Legal Notes.

Imprint

This e-book is a publication many contributed to, specifically:

Editor Meagan White
Acquisitions Editor Hareem Shafi
Copyeditor Yvette Chin
Cover Design Graham Geary
Photo Credit Shutterstock.com/125870006/© Gil.K
Production E-Book Graham Geary
Typesetting E-Book SatzPro, Krefeld (Germany)

We hope that you liked this e-book. Please share your feedback with us and read the Service Pages to find out how to contact us.

Library of Congress Cataloging-in-Publication Control Number: 2016044544

ISBN 978-1-4932-1464-8 (print)
ISBN 978-1-4932-1465-5 (e-book)
ISBN 978-1-4932-1466-2 (print and e-book)

© 2017 by Rheinwerk Publishing, Inc., Boston (MA)
1st edition 2017

Dedicated to Bilen Cekic and Isil Demir:
two great programmers of the newer generation,
living on separate continents,
whom I had the honor and pleasure to teach,
and whose presence I miss.

Special thanks to:
Melissa Levi for the author portrait and
Yaman Koseoglu for the initial editing.

Contents

Preface 17

PART I Architectural Design Patterns

1 MVC 27

1.1 Case Study: Read, Process, Display, and Post 28
1.2 Passing Select Options 33
1.3 Distributing Application Logic 34
1.4 Related Patterns 34
1.5 Summary 35

PART II Creational Design Patterns

2 Abstract Factory 39

2.1 Case Study: Log Analysis 40
2.2 Related Patterns 47
2.3 Summary 47

3 Builder 49

3.1 Case Study: Jobs for Text Files 49
3.2 When to Use 66
3.3 Privacy 66
3.4 Summary 67

4 Factory 69

4.1 Case Study: FI Documents for Parties 69
4.2 Advantages 75
4.3 Related Patterns 75
4.4 Summary 76

5 Lazy Initialization 77

5.1 Case Study: Logging Errors 77
5.2 Advantages 81
5.3 Related Patterns 81
5.4 Summary 82

6 Multiton 83

6.1 Case Study: Vendor Balance 84
6.2 When to Use 88
6.3 When to Avoid 88
6.4 State Modification 89
6.5 Summary 89

7 Prototype 91

7.1 Case Study: Item Clone 91
7.2 Changing Class Properties 95
7.3 Clone Operations 96
7.4 Related Patterns 96
7.5 Summary 97

8 Singleton 99

8.1 Case Study: Subcomponents 99
8.2 Advantages and Disadvantages 106
8.3 Related Patterns 106
8.4 Summary 106

PART III Structural Design Patterns

9 Adapter 109

9.1 Case Study: Project Management Tools 109
9.2 Glue Code 118
9.3 Two-Way Adapters 118
9.4 Legacy Classes 122
9.5 Summary 122

10 Bridge 123

10.1 Case Study: Messaging Framework 123
10.2 Advantages 131
10.3 Summary 131

11 Composite 133

11.1 Recursive Programming: A Refresher 133
11.2 Case Study: HR Positions 137
11.3 Advantages 144
11.4 Disadvantages 144
11.5 When to Use 145
11.6 Related Patterns 145
11.7 Summary 146

12 Data Access Object 147

12.1 Case Study: Potential Customers 147
12.2 Redundant Code Prevention 152
12.3 Related Patterns 153
12.4 Summary 153

13 Decorator 155

13.1 Case Study: User Exit 155
13.2 Advantages and Challenges 161
13.3 Related Patterns 162
13.4 Summary 162

14 Façade 165

14.1 Case Study: Bonus Calculation 166
14.2 When and Where to Use 168
14.3 Related Patterns 170
14.4 Summary 170

15 Flyweight 171

15.1 Case Study: Negative Stock Forecast 171
15.2 Disadvantages 181

15.3 When to Use 181
15.4 Related Patterns 182
15.5 Summary 182

16 Property Container 185

16.1 Case Study: Enhancing an SAP Program 185
16.2 Static vs. Instance Containers 194
16.3 Sharing Variables 195
16.4 Variable Uniqueness 196
16.5 Related Patterns 196
16.6 Summary 198

17 Proxy 199

17.1 Case Study: Sensitive Salary Information 199
17.2 When to Use 206
17.3 Related Patterns 207
17.4 Summary 207

18 Chain of Responsibility 209

18.1 Case Study: Purchase Order Approver Determination 209
18.2 Risks 217
18.3 Related Patterns 217
18.4 Summary 218

PART IV Behavioral Design Patterns

19 Command 221

19.1 Case Study: SD Documents 221
19.2 When to Use or Avoid 229
19.3 Related Patterns 229
19.4 Summary 230

20 Mediator 231

20.1 Case Study: Stock Movement Simulation 232
20.2 When to Use 241

20.3 Disadvantages 241
20.4 Summary 241

21 Memento 243

21.1 Case Study: Budget Planning 243
21.2 Risks 252
21.3 Redo 253
21.4 Summary 254

22 Observer 255

22.1 Case Study: Target Sales Values 256
22.2 Advantages 261
22.3 Disadvantages 261
22.4 Multiple Subjects 262
22.5 Related Patterns 263
22.6 Summary 264

23 Servant 265

23.1 Case Study: Address Builder 266
23.2 Extensions 274
23.3 Related Patterns 275
23.4 Summary 275

24 State 277

24.1 Case Study: Authorization-Based Class Behavior 277
24.2 Advantages 283
24.3 Related Patterns 284
24.4 Summary 284

25 Strategy 285

25.1 Case Study: Sending Material Master Data 286
25.2 Advantages 291
25.3 Passing Data to the Strategy Object 291
25.4 Optional Strategies 293
25.5 Intermediate Abstract Classes 297

25.6 Related Patterns 302
25.7 Summary 302

26 Template Method 305

26.1 Case Study: Average Transaction Volume 306
26.2 When to Use 318
26.3 Advantages and Risks 319
26.4 Degree of Abstraction 320
26.5 The "Hollywood Principle" 320
26.6 Summary 321

27 Visitor 323

27.1 Case Study: Incoming Invoice Processing 323
27.2 When to Use 333
27.3 Related Patterns 334
27.4 Summary 335

Appendices 337

A Object-Oriented Programming 339
A.1 Object-Oriented ABAP Development Environment 339
A.2 Class 341
A.3 Superclass 343
A.4 Abstract Class 345
A.5 Interface 351
A.6 UML 355
A.7 Summary 359
B Subclass Determination 361
B.1 Hardcoding 361
B.2 Convention over Configuration 361
B.3 SAP Class Tables 363
B.4 Custom Table 368
C Principles 371
C.1 Object-Oriented Principles 371
C.1.1 Abstraction 371
C.1.2 Composition 371
C.1.3 Inheritance 372
C.1.4 Encapsulation 372
C.1.5 Polymorphism 372
C.1.6 Decoupling 373

C.2 Design Principles 373
C.2.1 Single Responsibility 373
C.2.2 Open–Closed 374
C.2.3 Liskov Substitution 374
C.2.4 Interface Segregation 374
C.2.5 Dependency Inversion 375
C.3 Anti-Patterns 375
C.3.1 Blob 376
C.3.2 Copy–Paste Programming 376
C.3.3 Functional Decomposition 376
C.3.4 Golden Hammer 377
C.3.5 Grand Old Duke of York 378
C.3.6 Input Kludge 378
C.3.7 Jumble 378
C.3.8 Lava Flow 379
C.3.9 Object Orgy 379
C.3.10 Poltergeist 380
C.3.11 Reinvent the Wheel 380
C.3.12 Spaghetti Code 380
C.3.13 Swiss Army Knife 381
C.3.14 Vendor Lock-In 381
D The Author 383

Index 385

Service Pages I
Legal Notes II

Preface

When an architect starts planning a new building, he/she doesn't reinvent the wheel. Instead, time-tested, proven principles and designs are passed on from former generations and reused.

The same approach applies to software architects. Object-oriented programming (OOP) provides many concepts you can take advantage of, such as interfaces, abstract classes, concrete classes, properties, methods, encapsulation, inheritance, polymorphism, abstraction, etc. For those unfamiliar with the basics of object-oriented programming, Appendix A will provide you with a primer. Once you are familiar with these concepts, the next step is to use them correctly.

When you are expected to design new software, some of the first questions you'll want to consider are about the structure of classes. How many classes do you need to create? Do you need interfaces and/or abstract classes? What should be static? Should you prefer inheritance or composition? How will you determine the names of subclasses? Should you use casting, or should you create a distinct variable for each object? Will your design be flexible enough for possible future changes/expansions? Will it scale? The questions are endless, and often, there isn't a single correct answer.

Design patterns answer such questions by providing simple, flexible, and scalable solutions to common software requirements.

The first comprehensively documented resource on the subject is the book *Design Patterns Elements of Reusable Object-Oriented Software*, written by the "Gang of Four": Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Although published in 1994, many modern applications of today are still based on the concepts explained in that book, which is still considered the bible of design patterns. Although new patterns emerge every now and then, only some of them linger long enough to become a new standard, while others fade away over time. The patterns in *Design Patterns Elements of Reusable Object-Oriented Software*, however, seem to be here to stay.

Jazz Standards

Design patterns are the "jazz standards" of software development. You wouldn't force yourself to play exactly what's written in the book, but the patterns give you the main logic and base vocabulary to become a world-class musician.

If you want to advance from a developer role towards an architectural role, it is crucial to know the standard patterns, their advantages/disadvantages, and when to use/avoid them. When you read a certain software specification document, you should be able to pinpoint the patterns (emphasis on the plural) that correspond to the requirements. After you design the system in Unified Modeling Language (UML), who does the ABAP coding doesn't matter at all—as long as the developers have the required technical skills.

Becoming an architect will not happen overnight. However, the journey matters at least as much as the destination. On each step on the road, your technical skills will improve, your applications will have better designs, and you will feel more empowered than before.

Become an Architect!

Knowledge of design patterns is one of the key elements of becoming a software architect and is one of the major differences between a developer and an architect.

Be careful though—if you imagine design patterns like a golden hammer, you might start to see everything as a nail. In some cases, a pattern may have a precise correspondence with your current requirement. In that case, there is nothing wrong with going forward and using the design pattern as it is. In other cases, a combination of design patterns will be what's needed—in that case, there is nothing wrong with using a combination of patterns. Just be careful not to overcomplicate things.

However, there are also cases where no pattern or combination can provide a feasible model—you may have performance/security concerns, or the structure of the application can simply be unique. In such cases, we generally advise not bending your application to the patterns—instead, bend the patterns to your application. You can take some known patterns as a basis and morph them into an ad-hoc UML design. If you have enough experience with design patterns that they've become second nature, you can even come up with an entirely new design pattern—that's how Facebook's flux design pattern was born.

Requirement Is King

Don't bend your requirements to match design patterns. Instead, bend the patterns to match your requirements.

Design Pattern Categories

The traditional hierarchy of design patterns contains three widely accepted categories (creational, structural, and behavioral), which we will use here with the minor additional fourth category: architectural. In short, each has the following characteristics:

- Architectural design patterns are all about the overall framework of the application. If you are designing an application from scratch and looking for a skeletal structure to build the components on, you are in the territory of this category. MVC (model–view–controller) is the only architectural design pattern covered in this book, in Chapter 1.
- Creational design patterns are all about creation of object instances. Simply executing a `CREATE OBJECT` or `NEW` command can't cover everything, and the patterns in this category deal exactly with that limitation. Abstract factory, builder, factory, lazy initialization, multiton, prototype, and singleton are the creational design patterns covered in this book (Chapter 2 through Chapter 8).
- Structural design patterns are all about the relationships between objects. If you are looking for a simple, effective, and flexible way to make your objects interact and work together without making them completely interdependent, you are likely to find a pattern here. Adapter, bridge, composite, data access object, decorator, façade, flyweight, property container, and proxy are the structural design patterns covered in this book (Chapter 9 through Chapter 17).
- Behavioral design patterns are all about the communication between objects. If you need to share information between your objects without making them completely interdependent, you are likely to find a pattern here. Chain of responsibility, command, mediator, memento, observer, servant, state, strategy, template method, and visitor are the behavioral design patterns covered in this book (Chapter 18 through Chapter 27).

If you are already familiar with design patterns, you may notice that we have included some nontraditional patterns and excluded a few traditional ones. The reason is that this book is focused design patterns in ABAP, not language-independent

design pattern theory. Some traditional patterns, such as iterator, have a correspondence in ABAP that is directly built within the language itself. In that case, using a design pattern is probably redundant. On the other hand, some nontraditional patterns, such as data access object and lazy initialization, are extremely useful in many cases. Therefore, mixing them with traditional patterns should cause no harm.

How to Learn

In this book, design patterns are classified into categories and sorted in alphabetical order. However, the book is written so that you don't have to read and learn the patterns in that order. The learning process of design patterns depends on your approach. In other words, you may use this book in multiple ways. In the following sections, we will look at four of the most common ways you might choose to use this book.

Pattern-Based Learning

If you have no experience of design patterns and want to learn them all, it makes sense to start from basic patterns and move to more advanced patterns. In this slow learning process, you would study each design pattern thoroughly and apply it to your next development project the following day. Some patterns are built on top of others, while some patterns make good pairs. These dependencies are more important than the pattern categories; therefore, we recommend you learn and apply them in the following order, which will provide you with a decent curriculum:

- MVC (the most fundamental pattern of them all): Chapter 1
- Factory (prerequisite for all creational design patterns): Chapter 4
- Builder: Chapter 3
- Singleton (references factory; prerequisite for multiton): Chapter 8
- Multiton (references singleton): Chapter 6
- Flyweight (references factory and multiton): Chapter 15
- Lazy initialization (references singleton and multiton): Chapter 5
- Prototype: Chapter 7

- Abstract factory (references factory, builder, singleton): Chapter 2
- Template method: Chapter 26
- Strategy (references template method and flyweight): Chapter 25
- Data access object (references strategy): Chapter 12
- State (references template method, strategy, singleton, flyweight): Chapter 24
- Adapter: Chapter 9
- Proxy (references adapter, lazy initialization, state): Chapter 17
- Façade (references singleton, proxy, adapter): Chapter 14
- Composite (references flyweight): Chapter 11
- Property container (good background for decorator, builder, bridge, chain of responsibility, mediator, observer, strategy): Chapter 16
- Mediator (references singleton): Chapter 20
- Decorator (references template method, property container, mediator): Chapter 13
- Observer (references template method, property container, mediator, singleton): Chapter 22
- Chain of responsibility (references singleton, composite, strategy, property container): Chapter 18
- Visitor (good background for servant): Chapter 27
- Servant (references template method, visitor): Chapter 23
- Memento (good background for command): Chapter 21
- Command (references memento, template method): Chapter 19
- Bridge: Chapter 10

Category-Based Learning

If you have experience in design patterns already and want to improve your knowledge in a certain category, you can pick a category and study the patterns in an order that makes sense. Following the dependencies mentioned in the previous section, we recommend the following order for each category.

- Architectural design patterns:
  - MVC (the one and only!): Chapter 1

- Creational design patterns:
  - Factory (prerequisite for all creational design patterns): Chapter 4
  - Builder: Chapter 3
  - Singleton (references factory; prerequisite for multiton): Chapter 8
  - Multiton (references singleton): Chapter 6
  - Lazy initialization (references singleton and multiton): Chapter 5
  - Prototype: Chapter 7
  - Abstract factory (references factory, builder, singleton): Chapter 2
- Structural design patterns:
  - Adapter: Chapter 9
  - Proxy (references adapter, lazy initialization, state): Chapter 17
  - Façade (references singleton, proxy, adapter): Chapter 14
  - Flyweight (references factory and multiton): Chapter 15
  - Composite (references flyweight): Chapter 11
  - Property container (good background for decorator, builder, bridge, chain of responsibility, mediator, observer, strategy): Chapter 16
  - Decorator (references template method, property container, mediator): Chapter 13
  - Bridge: Chapter 10
- Behavioral design patterns:
  - Template method: Chapter 26
  - Strategy (references template method and flyweight): Chapter 25
  - Data access object (DAO; references strategy): Chapter 12
  - Chain of responsibility (references singleton, composite, strategy, property container): Chapter 18
  - State (references template method, strategy, singleton, flyweight): Chapter 24
  - Mediator (references singleton): Chapter 20
  - Observer (references template method, property container, mediator, singleton): Chapter 22
  - Memento (good background for command): Chapter 21

- Command (references memento, template method): Chapter 19
- Visitor (good background servant): Chapter 27
- Servant (references template method, visitor): Chapter 23

Quick and Dirty Learning

If you lack time, you might want to learn the most significant patterns first. It is true that some patterns are used frequently in ABAP while others are needed only occasionally. If you feel like going quick and dirty, here are the patterns that are generally used more often than others:

- MVC, used all the time: Chapter 1
- Factory and builder to create new objects in a centralized point: Chapter 4
- Singleton and multiton to prevent creation of multiple instances of the same object: Chapter 8 and Chapter 6, respectively
- Lazy initialization for overall performance improvement: Chapter 5
- Template method to prevent algorithm duplication among similar classes: Chapter 26
- Strategy to make algorithms interchangeable: Chapter 25
- Façade to make the life of other developers easier: Chapter 14
- Decorator to modify an object or dataset by multiple classes: Chapter 13
- Observer to make objects communicate without making them interdependent: Chapter 22
- Visitor to extend the functionality of classes without modifying legacy code: Chapter 27

This list doesn't mean that other patterns are less important. In fact, every pattern has its place. A pattern that may seem insignificant to you today may be a life-saver tomorrow when you encounter the corresponding requirement. This list is just a suggestion to get you started.

Ad-Hoc Learning

You may be in a situation where you have experience with design patterns already and want to use this book as reference material. You may want to quickly remember the UML structure of a pattern, use the book as a reference for your

thesis, or simply learn the few remaining patterns you don't have too much experience with. To help with ad-hoc learning, all patterns are grouped under their corresponding category and sorted alphabetically. Pinpointing a pattern should be no trouble.

Sharpening Your Skills

Once you have a few design patterns under your belt, consider taking a look at Appendix C, Section C.3 regarding anti-patterns (the bad practices of the OOP world). You can also check out the anti-patterns in the index to see them in practice.

PART I

Architectural Design Patterns

In this chapter, we'll get to know one of the most fundamental design patterns of the entire software industry: MVC. Countless commercial applications are built with MVC, and there is no reason why your ABAP application shouldn't take advantage of this time-proven pattern. If you are building a GUI-based application, MVC can isolate the application logic.

1 MVC

The *model–view–controller* design pattern, or MVC design pattern, focuses on isolating the application logic from the GUI-related code. Generally, MVC is one of the most fundamental design patterns in the entire range. MVC is so common that some development tools even have MVC project types. If you were to learn a single design pattern and use it for the rest of your life, it would be MVC. During training sessions, MVC also happens to be one of the first design patterns demonstrated.

Traditional MVC suggests an approach where you divide your application into three pieces:

- **Model ("M")**
  Class(es) that contain your application logic. A model class shouldn't contain any code related to GUI operations. In the ABAP world, the model class would correspond to your backend classes in Transaction SE24.
- **View ("V")**
  Class(es) that contain your GUI-related stuff. Textboxes, combo boxes, forms, etc. are all in this category. In the ABAP world, the view may correspond to your SAP List Viewer (ALV) grid, Web Dynpro ABAP components, etc. In a typical project, we reuse elements provided by SAP and don't code views from the scratch.
- **Controller ("C")**
  The client application that binds the model and view together. In the ABAP world, the controller would correspond to your executable application in Transaction SE38.

Let's move forward and see MVC in action, beginning with a case study before moving on to other patterns. Whatever additional pattern you might use, MVC will probably be the most fundamental pattern of any GUI application.

1.1 Case Study: Read, Process, Display, and Post

Our case study will be one of the most common ABAP requirements ever—a typical CRUD-like application (create, read, update, delete). Our program needs to get data from somewhere, do some calculations, display the result with ALV, and post something to the database when the user clicks a button. We all have likely written such an application a thousand times.

To make it more concrete, we're going to use an example where we need to do the following:

1. Read: Read customer orders from tables `VBAK` and `VBAP`.
2. Process: Eliminate orders of blocked customers.
3. Display: Show results using ALV.
4. Post: Create delivery documents for selected items.

Using classic ABAP, we could develop a program in Transaction SE38, which would roughly look like the code in Listing 1.1.

```
REPORT zrep.

" Some data definitions
" Some selection-screen parameters

START-OF-SELECTION.
  PERFORM read_orders.
  PERFORM eliminate_blocked.
  PERFORM display_alv.
END-OF-SELECTION.

FORM read_orders.
  " Some code to read VBAK, VBAP, etc
ENDFORM.

FORM eliminate_blocked.
  " Some code to read KN* tables and remove entries from the ITAB
ENDFORM.
```

```
FORM display_alv.
  " Some code to call REUSE_ALV_GRID_DISPLAY
ENDFORM.

FORM user_command USING rucomm rs_selfield.
  CHECK rucomm EQ c_ucomm_save.
  PERFORM create_dlv.
ENDFORM.

FORM create_dlv.
  " Some code to loop through the ITAB & create deliveries
ENDFORM.

" Some further forms
```

Listing 1.1 Simple Application in Transaction SE38

So far, so good. We have created a typical program structure encountered frequently among thousands of SAP clients. However, this typical structure has a major flaw: Code for database operations is mixed with code for managing the GUI. The database logic is locked inside Transaction SE38 and is not reusable.

To see how this can cause problems, imagine that a new requirement has emerged that says that we need to include an RFC function (remote function call). The RFC is supposed to get order numbers from an external system, eliminate blocked clients, and create deliveries for whatever remains. Basically, we want the same application logic in the form of an RFC function.

To take this example a step further, an additional requirement could be to write a new GUI using Web Dynpro ABAP targeting web users without SAP GUI.

What can be done in classical ABAP is very limited and blunt, as you can see with the following options:

- You could simply copy and paste the code from Transaction SE38 to Transaction SE37. However, this is not the best idea. Why? If, for instance, you need to modify the blocked customer elimination logic or add a new field to the BAPI, you would need to modify multiple spots.
- You could take advantage of include files so forms are available everywhere. Not an elegant solution, because the data shared between forms would need to be defined in Transaction SE37/SE38 multiple times. If you need to add a new field to the internal table, you need to modify multiple spots.

- You could create a huge function group and turn forms into functions. This option is better than the others, but you wouldn't be taking advantage of object-oriented features, such as polymorphism, inheritance, and encapsulation. See Appendix C for more information on these advantages.

Instead, we can turn our gaze to the design patterns and see what MVC can do for us. As we saw at the start of chapter, MVC tells us to split the application into three components: the model, the view, and the controller.

In our example, however, we have multiple controllers. Aside from the ALV application, the Web Dynpro ABAP application is also a controller. The RFC function can roughly be seen as a controller as well (although there is no GUI). To begin to work through this, let's see what the Unified Modeling Language (UML) diagram of our MVC design would look like. We will start with the simple ALV application first, which is outlined in Figure 1.1.

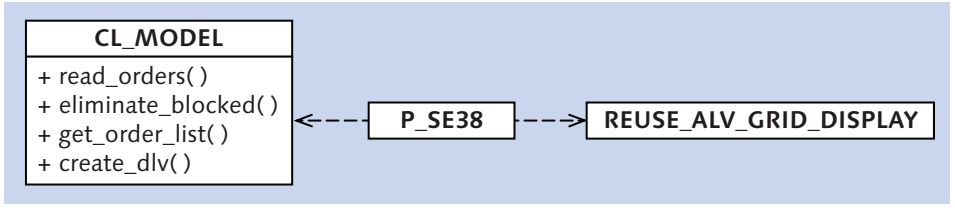


Figure 1.1 MVC Overview

As you see, we have moved the entire application logic into `CL_MODEL`. Before jumping to the advantages of MVC over the classic procedural approach, let's see what the code would look like (Listing 1.2).

```
CLASS zcl_model DEFINITION
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.

    METHODS read_orders
      IMPORTING
        !it_vbeln_rng TYPE ztt_vbeln_rng OPTIONAL.

    METHODS eliminate_blocked.

    METHODS get_order_list
      RETURNING
        VALUE(rt_list) TYPE ztt_order.
```

```
    METHODS create_dlv
      RETURNING
        VALUE(rt_bapiret2) TYPE bapiret2_tab.

  PRIVATE SECTION.
    DATA gt_order TYPE ztt_order.
    " Some data definitions
    " Some private methods
  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_model IMPLEMENTATION.

  METHOD read_orders.
    " Some code to read VBAK, VBAP, etc and fill GT_ORDER
  ENDMETHOD.

  METHOD eliminate_blocked.
    " Some code to read KN* tables & remove entries from GT_ORDER
  ENDMETHOD.

  METHOD get_order_list.
    rt_list[] = gt_order[].
  ENDMETHOD.

  METHOD create_dlv.
    " Some code to loop through GT_ORDER & create deliveries
  ENDMETHOD.

  " Some further methods

ENDCLASS.
```

Listing 1.2 Model Class

As you see, we have ripped the runtime logic out of the report and placed it into `CL_MODEL`. As a result, the report will be simplified dramatically, as demonstrated in Listing 1.3.

```
REPORT zrep.

DATA:
  go_model TYPE REF TO zcl_model,
  gt_order TYPE ztt_order.

" Some data definitions
" Some selection-screen parameters

START-OF-SELECTION.
```

```
  go_model = NEW #( ).
  go_model->read_orders( ).
  go_model->eliminate_blocked( ).
  gt_order = go_model->get_order_list( ).
  PERFORM display_alv.
END-OF-SELECTION.

FORM display_alv.
  " Some code to call REUSE_ALV_GRID_DISPLAY
ENDFORM.

FORM user_command USING rucomm rs_selfield.
  CHECK rucomm EQ c_ucomm_save.
  go_model->create_dlv( ).
ENDFORM.
```

Listing 1.3 Controller Application

A neat and simple approach, all that ZREP ever needs to do is to be a "messenger" (controller) between CL_MODEL (model) and ALV (view).

Now, if we want to extend the functionality and bring the other requirements of Web Dynpro ABAP and an RFC into the game, the true value of MVC will be much more evident. Let's take a look at the UML diagram in Figure 1.2.

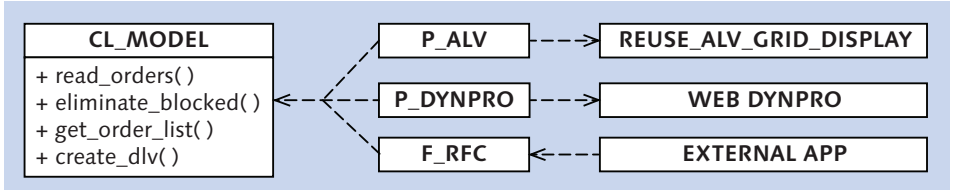


Figure 1.2 Extended MVC Functionality

As you see, the Web Dynpro ABAP application would be as simple as the ALV application. Instead of copying and pasting code between two applications, or dealing with dreaded include files/function modules, we simply and elegantly reuse the runtime logic contained in CL_MODEL. Although it has no GUI, the same applies to F_RFC as well.

If, in the future, we make an improvement in CL_MODEL, it will automatically improve all three applications. For instance, if we need to check an additional table to detect blocked customers, the only place we need to touch is CL_MODEL~ELIMINATE_BLOCKED. The change will automatically reflect in all three applications.

MVC as "the" Standard

MVC is arguably the industry standard for GUI-related programming today. In order to maintain the MVC approach, even your simplest GUI-related application should have a model class in Transaction SE24 (which knows nothing about the GUI) and a program in Transaction SE38 (which knows nothing about the business logic).

The code of a classic ABAP program with more than 5,000 lines usually looks confusing. Despite the best efforts of good programmers to split forms into distinct include files, the code related to GUI operations is often mixed in with the code related to the application logic. New programmers (or the original programmer after five years) may find the program hard to understand.

The same program in MVC has a different story though. The program in Transaction SE38 contains GUI-related code only. The class in Transaction SE24 contains the business logic only. Nothing is mixed, which is cleaner and easier to understand. If another programmer wants to modify a GUI-related functionality, he/she will work in Transaction SE38 only and won't need to worry about messing up the business logic. Another programmer who needs to modify the business logic will work in Transaction SE24 only and will have an easier time understanding the logic.

1.2 Passing Select Options

If you need to transfer select options from your program to the class, a good way to do this is to pass them as parameters to the constructor of the model class. Inside the class, you would store them in global variables. Instead of passing select options as distinct variables, you could define a nested type containing all the parameters and pass one single value to the constructor. This approach makes the method signature look nice and clean. See Listing 1.4 for an example.

```
CLASS zcl_model DEFINITION
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.

    TYPES:
      BEGIN OF t_param,
        s_bukrs TYPE RANGE OF bkpf-bukrs,
        s_belnr TYPE RANGE OF bkpf-belnr,
        s_gjahr TYPE RANGE OF bkpf-gjahr,
      END OF t_param.

    METHODS read_docs
      IMPORTING
        !is_param TYPE t_param.
```

```
  PRIVATE SECTION.
  PROTECTED SECTION.
ENDCLASS.
```

Listing 1.4 Passing Select Options at Once

In this sample code, we have consolidated all select options into `T_PARAM` instead of passing them one by one. Imagine having thirty select options, and all the clutter it would create on `READ_DOCS`. With our approach, the signature is kept clean—no matter how many select options are passed.

1.3 Distributing Application Logic

A simple model class may only contain a handful of methods. However, if the application logic is comprehensive, don't feel the pressure to contain everything within one single model class. Doing so may lead you to the blob anti-pattern (more information on the blob anti-pattern can be found in Appendix C) where a central object contains the lion's share of responsibilities.

You should distribute distinct components of your application logic into loosely coupled classes and contain them inside a central model class (composition).

Distribution of Responsibilities

A bloated central class to take care of everything under the sun is not advisable. Instead, consider splitting the responsibilities into distinct classes. Each class should be responsible of one task alone.

You can even take advantage of other design patterns and make use of them within the model class! For instance, you can announce significant events using the observer design pattern (Chapter 22) and make other classes take action as needed. That way, your actions will be extendible: Just add a new observer class and that's it—you wouldn't even touch the central model class. Many other patterns can be used by the model class as well.

1.4 Related Patterns

You may have heard that Facebook has invented a new design pattern, which it favors over MVC: Flux. The flux design pattern has a simple and manageable

structure for cases where you have a complex GUI with independent dynamic elements, such as a Facebook page. If this sounds interesting to you, there is no reason why you shouldn't get online and check some examples of flux.

Further Resources

If you want to go deeper and understand how flux works, Facebook has a great overview on GitHub: *https://facebook.github.io/flux/docs/overview.html*. This page also contains guides, references, and other resources and can be considered the official technical homepage of flux.

Don't worry, though, MVC has been tested and proven against time, and countless significant commercial applications are using MVC successfully. Therefore, there's no reason not to use MVC in your ABAP applications if you choose so. It is unlikely for a typical ABAP application to get so complicated that flux becomes a necessity (though you might prefer to do so). As always, use what is appropriate for the requirement in question.

1.5 Summary

MVC is the most fundamental design pattern and should generally be the base pattern for any GUI application. It works by separating the model class, the controller application, and the view completely so that each can operate independently. This separation ensures that each ABAP element is reusable for different purposes.

In SAP, we generally don't program views from the scratch. Instead, we use views provided by SAP. Some examples are ALV grid, table control, Web Dynpro ABAP breadcrumbs, and SAP Fiori views.

MVC doesn't force you to pack everything into the model class; you can (and should) distribute the application logic to multiple classes if the logic is too advanced. Remember that each class should be responsible of one task alone.

Although newer architectural patterns, such as flux, emerge over time, MVC can still be considered the industry standard at this time.

PART II

Creational Design Patterns

The factory design pattern, discussed in detail in Chapter 4, centralizes the steps to create an instance of a class. However, there are cases where those steps vary. A typical situation is a multiplatform application where the steps to be taken depend on the underlying OS. For such cases, the factory can be made abstract, so that a distinct factory implementation can be coded for each supported OS.

2 Abstract Factory

One of the biggest reasons to use design patterns is to increase the level of abstraction, which, in return, gives us more flexibility and reusability.

When creating an object, the most concrete way is to have a concrete constructor and to create an object via the command `CREATE OBJECT` (or `NEW`, if you are using ABAP 7.4). Adding one level of abstraction to this command might lead us to the factory (Chapter 4) or builder (Chapter 3) design patterns. In essence, after adding a level of abstraction, we would have a factory method that runs a complex code and returns a new object instance. Instead of using the command `CREATE OBJECT GO_OBJ`, we get a new instance via a method (e.g. `GO_OBJ = CL_OBJ=>GET_INSTANCE( )`). We would highly recommended understanding the factory and builder patterns before using this chapter.

In some cases, however, we need a second degree of abstraction. The typical case would be the situation where the code is going to be executed on multiple operating systems. Under the hood, an object targeting Windows might need to behave much differently than an object targeting Unix, despite the fact that they would share the same interface. For such cases, the abstract factory design pattern is an invaluable tool.

Note

Having multiple operating systems is a rare situation for many typical SAP clients. Companies usually pick one server operating system and build everything on top of it, so ABAP programmers rarely need to worry about the underlying server OS, if ever. However, to demonstrate the design pattern in question, this situation presents the perfect case study.

2.1 Case Study: Log Analysis

Imagine a huge company running SAP. The company is so large that it runs fifteen distinct, live SAP systems with various purposes. Some of those systems are Windows installations, and some of them are Unix. Our goal will be to develop an ABAP program that will build a log file (about whatever you might imagine), download the log file to the application server, and execute a program/script that resides on the server.

Basically, we need two interfaces: a writer (to download the file) and an executer (to execute the program), which are outlined in Figure 2.1.

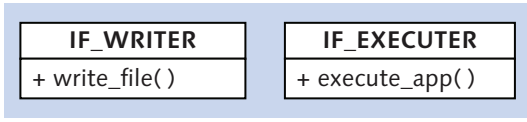


Figure 2.1 Interfaces Needed

Listing 2.1 demonstrates what the writer interface might look like.

```
INTERFACE zif_writer
  PUBLIC .

  METHODS write_file
    IMPORTING
      !iv_path TYPE clike.

ENDINTERFACE.
```

Listing 2.1 Writer Interface

Listing 2.2 provides a simple portrait of the executer interface.

```
INTERFACE zif_executer
  PUBLIC .

  METHODS execute_app.

ENDINTERFACE.
```

Listing 2.2 Executer Interface

So far, so good. Now, the architecture of our sample application seems to be more advanced than before. Depending on the OS, we need to use an `IF_WRITER` implementation and an `IF_EXECUTER` implementation to write the log file to the disk

and execute the subsequent steps. The problem is that the download and execution processes differ between operating systems. If the server runs on Windows, we need the following to happen:

- `write_file`
  Validate folder, write file
- `execute_app`
  Start *run.exe*

If the server runs on Unix, we need the following to happen:

- `write_file`
  Write file, run CHMOD (the command line UNIX command to change file permissions) to arrange permissions
- `execute_app`
  Start run.sh in administrator mode

Therefore, we need a pair of distinct, concrete classes for each OS, which is demonstrated in Figure 2.2.

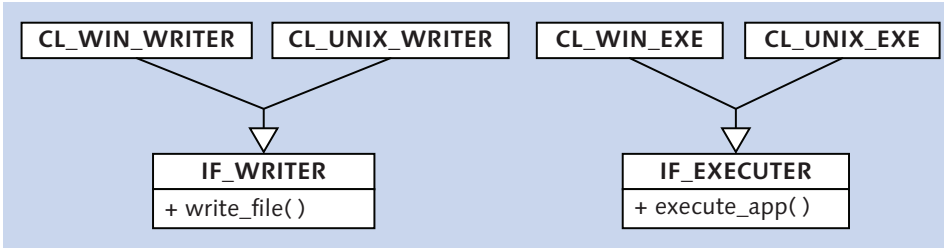


Figure 2.2 Classes Needed

Let's take a look at the classes, starting with the Windows writer demonstrated in Listing 2.3.

```
CLASS zcl_win_writer DEFINITION
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.
    INTERFACES zif_writer.
    METHODS windows_specific_stuff.
  PRIVATE SECTION.
    " Some helpful private definitions & methods
  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_win_writer IMPLEMENTATION.
```

```
  METHOD windows_specific_stuff.
    " Here is some stuff regarding Windows OS
  ENDMETHOD.

  METHOD zif_writer~write_file.
    " Some code to validate folder
    " Some code to open dataset, write file, close dataset
  ENDMETHOD.

  " Some further methods

ENDCLASS.
```

Listing 2.3 Windows Writer Class

Note that we have an additional method called `WINDOWS_SPECIFIC_STUFF`. This method is left to your imagination, with the assumption that preparing the instance might require some additional (and possibly conditional) method calls.

Listing 2.4 contains the writer class for Unix.

```
CLASS zcl_unix_writer DEFINITION
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.
    INTERFACES zif_writer.
    METHODS unix_specific_stuff.
    METHODS unix_specific_further_stuff.
  PRIVATE SECTION.
    " Some helpful private definitions & methods
  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_unix_writer IMPLEMENTATION.
  METHOD unix_specific_stuff.
    " Here is some stuff regarding Unix
  ENDMETHOD.

  METHOD unix_specific_further_stuff.
    " Here is some further stuff regarding Unix
  ENDMETHOD.

  METHOD zif_writer~write_file.
    " Some code to open dataset, write file, close dataset
    " Some code to do CHMOD stuff
  ENDMETHOD.

  " Some further methods

ENDCLASS.
```

Listing 2.4 Unix Writer Class

This class has not one but two additional methods for initialization purposes.

Moving forward, let's take a look at our executer implementations. Up first, let's look at the executer for Windows in Listing 2.5.

```
CLASS zcl_win_exe DEFINITION
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.
    INTERFACES zif_executer.
  PRIVATE SECTION.
    " Some helpful private definitions & methods
  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_win_exe IMPLEMENTATION.

  METHOD zif_executer~execute_app.
    " Some code to execute run.exe
  ENDMETHOD.

  " Some further methods

ENDCLASS.
```

Listing 2.5 Windows Executer Class

This class is not particularly complicated, but the Unix implementation in Listing 2.6 has a bit more to it.

```
CLASS zcl_unix_exe DEFINITION
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.
    INTERFACES zif_executer.
    METHODS enter_admin_mode.
  PRIVATE SECTION.
    " Some helpful private definitions & methods
  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_unix_exe IMPLEMENTATION.

  METHOD zif_executer~execute_app.
    " Some code to execute run.sh
  ENDMETHOD.

  " Some further methods

ENDCLASS.
```

Listing 2.6 Unix Executer Class

At this point, we have written a handful of concrete classes. Listing 2.3 and Listing 2.5 belong to the Windows domain, while Listing 2.4 and Listing 2.6 belong to Unix. We must now build the factory classes that will produce objects targeting the appropriate OS, as shown in Figure 2.3.

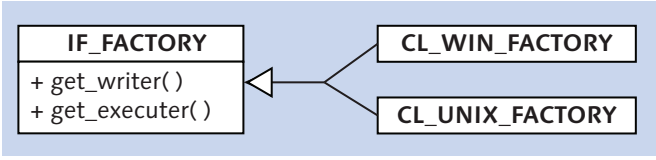


Figure 2.3 Factory Classes per Operating System

As seen in the Unified Modeling Language (UML) diagram in Figure 2.3, we are abstracting the factory in the form of an interface. For each OS, we will have a distinct factory class. Easier coded than explained, let's jump into the example. Listing 2.7 demonstrates what the interface could look like.

```
INTERFACE zif_factory
  PUBLIC .

  METHODS get_writer RETURNING VALUE(ro_wri) TYPE REF TO zif_writer.
  METHODS get_executer RETURNING VALUE(ro_exe) TYPE REF TO zif_executer.

ENDINTERFACE.
```

Listing 2.7 Factory Interface

The Windows factory class is demonstrated in Listing 2.8.

```
CLASS zcl_win_factory DEFINITION
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.
    INTERFACES zif_factory.
  PRIVATE SECTION.
  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_win_factory IMPLEMENTATION.

  METHOD zif_factory~get_writer.
    DATA(lo_win_writer) = NEW zcl_win_writer( ).
    lo_win_writer->windows_specific_stuff( ).
    ro_wri ?= lo_win_writer.
  ENDMETHOD.
```

```
  METHOD zif_factory~get_executer.
    DATA(lo_win_executer) = NEW zcl_win_exe( ).
    ro_exe ?= lo_win_executer.
  ENDMETHOD.

ENDCLASS.
```

Listing 2.8 Windows Factory Class

Taking a closer look at each method, we see that, in `GET_WRITER`, we start by creating a new concrete `CL_WIN_WRITER` object—a writer targeting Windows. Then, we include some extra initialization code by calling `WINDOWS_SPECIFIC_STUFF`. Finally, we cast `CL_WIN_WRITER` to `IF_WRITER` and return the object.

One clear advantage is that the client program doesn't need to know anything about `WINDOWS_SPECIFIC_STUFF`. All it cares about is getting a writer targeting the current OS without having to mess around with boring details. Our code achieves this by having OS-specific stuff managed outside the common ground.

In `IF_FACTORY~GET_EXECUTER`, we perform many of the same steps. However, because we don't have any Windows-specific operations in the `CL_WIN_EXE` class, we can simply create and cast the object.

The Unix factory doesn't contain any surprises; it is quite similar to the Windows factory, as can be seen in Listing 2.9.

```
 CLASS zcl_unix_factory DEFINITION
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.
    INTERFACES zif_factory.
  PRIVATE SECTION.
  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_unix_factory IMPLEMENTATION.

  METHOD zif_factory~get_writer.
    DATA(lo_unix_writer) = NEW zcl_unix_writer( ).
    lo_unix_writer->unix_specific_stuff( ).
    lo_unix_writer->unix_specific_further_stuff( ).
    ro_wri ?= lo_unix_writer.
  ENDMETHOD.

  METHOD zif_factory~get_executer.
    DATA(lo_unix_executer) = NEW zcl_unix_exe( ).
    lo_unix_executer->enter_admin_mode( ).
```

```
    ro_exe ?= lo_unix_executer.
  ENDMETHOD.

ENDCLASS.
```

Listing 2.9 Unix Factory Class

Finally, we will bring everything together by creating a sample client program that takes advantage of the abstract factory design pattern, as seen in Listing 2.10.

```
REPORT zclient.

PARAMETERS:
  p_os TYPE zbcd_os,
  p_path TYPE char30.

PERFORM main.

FORM main.
  DATA lo_factory TYPE REF TO zif_factory.

* Create factory object
  CASE p_os.
    WHEN 'WINDOWS'.
      DATA(lo_win_fact) = NEW zcl_win_factory( ).
      lo_factory ?= lo_win_fact.
    WHEN 'UNIX'.
      DATA(lo_unix_fact) = NEW zcl_unix_factory( ).
      lo_factory ?= lo_unix_fact.
  ENDCASE.

* From this point on, we don't care about the underlying OS.
* Get writer & executer from factory
  DATA(lo_writer) = lo_factory->get_writer( ).
  DATA(lo_executer) = lo_factory->get_executer( ).

* Write and execute
  lo_writer->write_file( p_path ).
  lo_executer->execute_app( ).

ENDFORM.
```

Listing 2.10 Client Program

The program is not perfect: the creation of the factory object (`LO_FACTORY`) has been coded into the program. In practice, you would do this in a distinct class, possibly in the form of a static method. For our purposes, we avoided complicating things

any more than necessary. However, it is important to keep in mind that `LO_FACTORY` should probably be created in another class—so we don't have to repeat the `CASE/WHEN` statements in each and every client program when using similar code in the real world.

Note

The names of subclasses (`ZCL_WIN_FACTORY`, `ZCL_UNIX_FACTORY`) are hardcoded in this example. See Appendix B, which covers subclass determination, for alternative approaches.

Objects returned by the factory method should probably be marked as `CREATE PRIVATE`. Otherwise, client programs can create objects directly, bypassing the valuable logic in the factory method. By marking the objects with `CREATE PRIVATE`, we ensure that the factory method is called to create an object instance.

2.2 Related Patterns

A strong, hands-on understanding of the factory and builder design patterns is highly recommended before attempting to use the abstract factory. When choosing which design pattern to use, if you feel like abstract factory will add an obsolete creational layer, you can keep things simple and get away by using factory or builder.

You should also note that an application usually needs only one concrete factory object during runtime. Therefore, implementing factory objects using singleton design pattern (Chapter 8) is a good idea.

2.3 Summary

Although the factory design pattern is sufficient for most cases, you might need an additional level of abstraction for cases where the factory needs to behave differently in various situations. That's where the abstract factory shines. Applications targeting different operating systems are typical examples.

The builder design pattern is typically preferred over other creational patterns when you need to create a modular object in a step-by-step approach. You can imagine building a Lego model: On each step, you pick one dynamically selected object and attach others. In the end, you end up having a complete model, mostly composed of distinct objects.

3 Builder

The builder design pattern comes in handy when you need to create a complex object. The idea is to split the creation process into small manageable steps, and let a builder method run through them one at a time. As a result, code repetition is prevented—instead of repeating the complex object creation code every time we need an instance, we simply call a central builder method that will provide us with the object instance we need.

In this chapter, we will begin by looking at case study to illustrate how the builder design pattern works. Then, we will examine some criteria to help us decide when to use this design pattern before briefly touching upon the need to mark the object returned by this pattern as `CREATE PRIVATE`.

3.1 Case Study: Jobs for Text Files

Our goal in this chapter will be to write a code that periodically reads data from various sources and prepares text files with that data. Our solution will require up to set up the following jobs:

- Job 1 runs daily, reads stock values, prepares a tab-delimited file, and saves the file to the application server.
- Job 2 runs weekly, reads stock values, prepares a CSV file, and emails the file to some manager.
- Job 3 runs weekly, reads sales values, prepares a CSV file, and saves the file to the application server.

- Job 4 runs monthly, reads sales values, prepares a tab-delimited file, and emails the file to some third-party company.

Figure 3.1 provides a flowchart of these jobs and their assigned tasks.

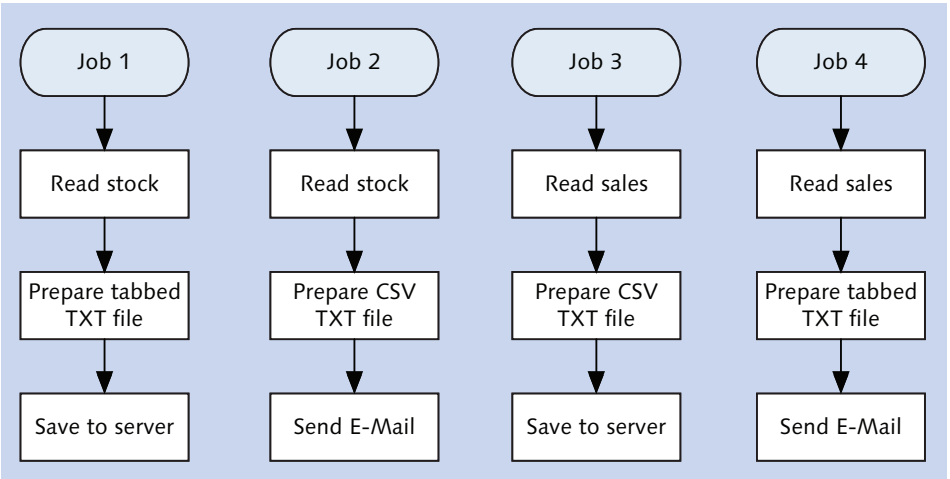


Figure 3.1 Job Flowcharts

Using a classical approach (which, throughout the book, we use to mean without design patterns or object-oriented ABAP), we would have to write four distinct programs covering the four different jobs. Listing 3.1 demonstrates what the first of these programs would look like.

```
REPORT zjob1.

" Some data definitions
" Maybe a selection screen

START-OF-SELECTION.
  PERFORM read_stock.
  PERFORM prepare_tabbed_txt.
  PERFORM download_file.
END-OF-SELECTION.

" Some form definitions
```

Listing 3.1 Job 1 in Procedural ABAP

The second program would be as seen in Listing 3.2.

```
REPORT zjob2.

" Some data definitions
" Maybe a selection screen

START-OF-SELECTION.
  PERFORM read_stock.
  PERFORM prepare_csv_txt.
  PERFORM send_email.
END-OF-SELECTION.

" Some form definitions
```

Listing 3.2 Job 2 in Procedural ABAP

The third program would be as seen in Listing 3.3.

```
REPORT zjob3.

" Some data definitions
" Maybe a selection screen

START-OF-SELECTION.
  PERFORM read_sales.
  PERFORM prepare_csv_txt.
  PERFORM download_file.
END-OF-SELECTION.

" Some form definitions
```

Listing 3.3 Job 3 in Procedural ABAP

Last, but not least, the fourth program would be as seen in Listing 3.4.

```
REPORT zjob4.

" Some data definitions
" Maybe a selection screen

START-OF-SELECTION.
  PERFORM read_sales.
  PERFORM prepare_tabbed_txt.
  PERFORM send_email.
END-OF-SELECTION.

" Some form definitions
```

Listing 3.4 Job 4 in Procedural ABAP

You may be getting the feeling that all four programs have something in common, and you'd be right. Figure 3.2 demonstrates the flow logic common to all four programs.

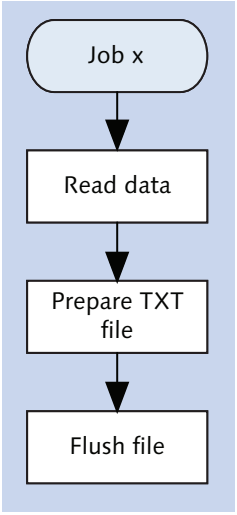


Figure 3.2 Common Flow Logic

Having four distinct programs with very similar algorithms is not the best idea ever. Why? Imagine that you have a new requirement that says that, if the programs are executed in the foreground, an ALV report should be displayed before flushing the file. You would have to implement the ALV code into four distinct programs, which would be time consuming. Worse, you could have forty programs to be modified—but the principle is the same.

Code repetition in distinct programs is also a risk. A smart programmer would write common subroutines to read sales data, stock data, etc., and call the right subroutines for the corresponding jobs. However, the design doesn't force the programmer to reuse existing subroutines. A junior programmer with the task of writing `ZJOB5` might simply copy and paste the code, resulting in two distinct spots to manage the logic of reading sales data.

This is where the builder design pattern can make our lives easier.

As you saw in Figure 3.2, each and every one of our programs has the exact same skeleton algorithm. The only thing that changes is the achievement of each step. The overview of our jobs can be seen in Table 3.1.

Job	Reads via	Generates via	Flushes via
1	Stock reader	Tab text generator	Downloader
2	Stock reader	CSV text generator	Emailer
3	Sales reader	CSV text generator	Downloader
4	Sales reader	Tab text generator	Emailer

Table 3.1 Overview of Jobs

Therefore, if we enter the object-oriented world and transform the components in Table 3.1 into classes, we would end up having the following classes:

- Stock reader
- Sales reader
- Tab text generator
- CSV text generator
- Downloader
- Emailer

It is even possible to group those classes by their purposes, as follows:

- Readers (implementing `if_reader`)
  - Stock reader
  - Sales reader
- Generators (implementing `if_generator`)
  - Tab text generator
  - CSV text generator
- Flushers (implementing `if_flusher`)
  - Downloader
  - Emailer

You can probably see where this is going. If we can abstract the flow into a reader interface, a generator interface, and a flusher interface, life becomes simpler

because we can get away with having a single program for the entire algorithm. Listing 3.5 shows this in action.

```
REPORT zjob_pseudo.

  DATA:
    go_reader TYPE REF TO zif_reader,
    go_generator TYPE REF TO zif_generator,
    go_flusher TYPE REF TO zif_flusher.

" Some data definitions
" Maybe a selection screen

START-OF-SELECTION.
  " Some code to dynamically create objects
  DATA(gt_data) = go_reader->get_data( ).
  DATA(gt_file) = go_generator->generate( gt_data ).
  go_flusher->flush( gt_file ).
END-OF-SELECTION.

" Some form definitions
```

Listing 3.5 Pseudodynamic Program

As long as we make sure that the objects are created correctly, `ZJOB` will run perfectly. Luckily for us, creating objects correctly is the exact purpose of the builder design pattern. Let's start the transformation by checking out the entire Unified Modeling Language (UML) diagram shown in Figure 3.3.

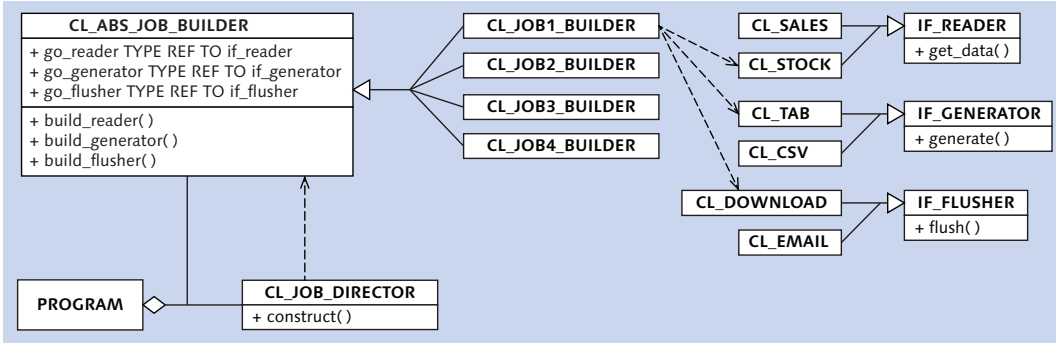


Figure 3.3 Entire System Overview

There's a lot going on in this diagram: Basically, we have an abstract job builder class named as `CL_ABS_JOB_BUILDER`, which is the central class of the entire system. Job builder classes, named as `CL_JOBx_BULDER` for example, are implement-

ing the abstract class to create specialized job classes. Other elements, such as readers, generators, and flushers, are going to be used as the building blocks of the job objects.

Let's start by inspecting the first interface in Listing 3.6, IF_READER, and its subclasses.

```
INTERFACE zif_reader
  PUBLIC .

  METHODS get_data
    RETURNING
      VALUE(rr_itab) TYPE REF TO data.

ENDINTERFACE.
```

Listing 3.6 Reader Interface

Why didn't we use a static data type and used TYPE REF TO data instead? The reason is that different implementations (such as CL_SALES and CL_STOCK) will read and return internal tables with different fields. For the sake of flexibility, the interface will deal with a data reference only.

Moving forward, let's see what CL_SALES might look like in Listing 3.7.

```
CLASS zcl_sales DEFINITION
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.
    INTERFACES zif_reader.
  PRIVATE SECTION.
    DATA gt_sales TYPE ztt_sales.
    " Some helpful private methods
  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_sales IMPLEMENTATION.

  METHOD zif_state~get_data.
    read_sales_data( ). " Fills gt_sales
    rr_itab = REF #( gt_sales ).
  ENDMETHOD.

  " Some further methods

ENDCLASS.
```

Listing 3.7 Sales Reader

CL_STOCK isn't too different, as you can see in Listing 3.8.

```
CLASS zcl_stock DEFINITION
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.
    INTERFACES zif_reader.
  PRIVATE SECTION.
    DATA gt_stock TYPE ztt_stock.
    " Some helpful private methods
  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_stock IMPLEMENTATION.

  METHOD zif_state~get_data.
    read_stock_data( ). " Fills gt_stock
    rr_itab = REF #( gt_stock ).
  ENDMETHOD.

  " Some further methods

ENDCLASS.
```

Listing 3.8 Stock Reader

At this point, we have a reader interface and two classes implementing it. Moving forward, we will have the same logic for IF_GENERATOR. According to our requirements, we need two generators: CL_TAB and CL_CSV. Let's fast forward and merge all three objects in Listing 3.9, as they all follow the same logic as IF_READER.

```
INTERFACE zif_generator
  PUBLIC .

  METHODS generate
    IMPORTING
      !ir_data_itab TYPE REF TO data
    RETURNING
      VALUE(rr_file_itab) TYPE REF TO data.

ENDINTERFACE.

CLASS zcl_tab DEFINITION
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.
    INTERFACES zif_generator.
  PRIVATE SECTION.
    DATA gt_file TYPE ztt_file.
```

```
    " Some helpful private methods
  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_tab IMPLEMENTATION.

  METHOD zif_generator~generate.
    prepare_tab_file( ir_data_tab ). " Fills gt_file
    rr_itab = REF #( gt_file ).
  ENDMETHOD.

  " Some further methods
ENDCLASS.

CLASS zcl_csv DEFINITION
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.
    INTERFACES zif_generator.
  PRIVATE SECTION.
    DATA gt_file TYPE ztt_file.
    " Some helpful private methods
  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_csv IMPLEMENTATION.

  METHOD zif_generator~generate.
    prepare_csv_file( ir_data_tab ). " Fills gt_file
    rr_itab = REF #( gt_file ).
  ENDMETHOD.

  " Some further methods
ENDCLASS.
```

Listing 3.9 Generator Interface and Classes

`IF_FLUSHER` has much the same logic as `IF_GENERATOR`: one interface, two concrete classes, as can be seen in Listing 3.10.

```
INTERFACE zif_flusher
  PUBLIC .

  METHODS flush
    IMPORTING
      !ir_file_itab TYPE REF TO data.

ENDINTERFACE.
```

```
CLASS zcl_download DEFINITION
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.
    INTERFACES zif_flusher.
  PRIVATE SECTION.
    " Some helpful private methods
  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_download IMPLEMENTATION.

  METHOD zif_flusher~flush.
    download_file( ir_file_tab ). " Uses fld.symbols, writes file
  ENDMETHOD.

  " Some further methods
ENDCLASS.

CLASS zcl_email DEFINITION
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.
    INTERFACES zif_flusher.
  PRIVATE SECTION.
    " Some helpful private methods
  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_email IMPLEMENTATION.

  METHOD zif_flusher~flush.
    send_mail( ir_file_tab ). " Uses fld.symbols & sends E-Mail
  ENDMETHOD.

  " Some further methods
ENDCLASS.
```

Listing 3.10 Flusher Interface and Classes

In a real-world situation, the methods would look a bit more advanced. For example, the interface methods would definitely raise exceptions if a subclass failed. However, we don't need all the bells and whistles to properly demonstrate the builder pattern.

Where do we stand now? So far, we have prepared our library, as demonstrated in Figure 3.4.

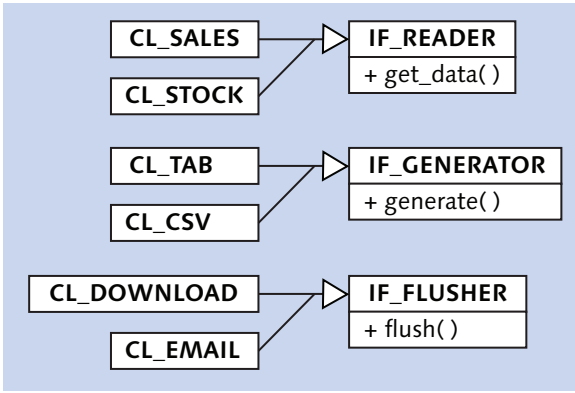


Figure 3.4 Library of ABAP Elements

As you can see, when zoomed in, this UML locally looks like the strategy design pattern implemented three times, as follows:

- **Reader strategy**
  Defined in `IF_READER` and implemented by `CL_SALES` and `CL_STOCK`
- **Generator strategy**
  Defined in `IF_GENERATOR` and implemented by `CL_TAB` and `CL_CSV`
- **Flusher strategy**
  Defined in `IF_FLUSHER` and implemented by `CL_DOWNLOAD` and `CL_EMAIL`

If, at some point, we need a new data source (such as planned production orders), all we need to do is define a new class implementing `IF_READER`. We don't need to modify anything else within the system. The same applies to other interfaces as well. If we need to generate a PDF file, for instance, all we need to do is to define a new class implementing `IF_GENERATOR`. Following the same logic, if we need to flush the file into SAP ERP's Document Management System (DMS) one day, implementing `IF_FLUSHER` into a new class will be enough.

At this time, we have three interfaces and six classes in our pocket. Remember that we need to bind them in different combinations for each distinct job, as follows:

- Job 1: Read from `CL_STOCK`, generate with `CL_TAB`, flush using `CL_DOWNLOAD`
- Job 2: Read from `CL_STOCK`, generate with `CL_CSV`, flush using `CL_EMAIL`
- Job 3: Read from `CL_SALES`, generate with `CL_CSV`, flush using `CL_DOWNLOAD`
- Job 4: Read from `CL_SALES`, generate with `CL_TAB`, flush using `CL_EMAIL`

This is the exact spot where the builder design pattern comes forward. So far, we have built the library that the builder will use. Now, in Listing 3.11, we will see what the builder looks like.

```
CLASS zcl_abs_job_builder DEFINITION
  PUBLIC ABSTRACT CREATE PUBLIC.

  PUBLIC SECTION.
    DATA:
      go_reader TYPE REF TO zif_reader,
      go_generator TYPE REF TO zif_generator,
      go_flusher TYPE REF TO zif_flusher.

    METHODS:
      build_reader ABSTRACT,
      build_generator ABSTRACT,
      build_flusher ABSTRACT.
  PRIVATE SECTION.
  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_abs_job_builder IMPLEMENTATION.

ENDCLASS.
```

Listing 3.11 Abstract Builder Class

Because `CL_ABS_JOB_BUILDER` doesn't contain any code at all, we could have also defined an interface. However, in more advanced real-world situations, the builder usually includes some generic code or some utility methods for subclasses to use. Therefore, an abstract class often makes more sense.

Finally, let's build our first concrete job class! The code in Listing 3.12 is quite simple and straightforward.

```
CLASS zcl_job1_builder DEFINITION
  INHERITING FROM zcl_abs_job_builder
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.
    METHODS build_reader REDEFINITION.
    METHODS build_generator REDEFINITION.
    METHODS build_flusher REDEFINITION.
  PRIVATE SECTION.
  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_job1_builder IMPLEMENTATION.
```

```
  METHOD build_reader.
    go_reader ?= NEW zcl_stock( ).
  ENDMETHOD.

  METHOD build_generator.
    go_generator ?= NEW zcl_tab( ).
  ENDMETHOD.

  METHOD build_flusher.
    go_flusher ?= NEW zcl_download( ).
  ENDMETHOD.

ENDCLASS.
```

Listing 3.12 Builder Class for Job 1

The same logic applies to the second job, only with different library classes, as demonstrated in Listing 3.13.

```
CLASS zcl_job2_builder DEFINITION
  INHERITING FROM zcl_abs_job_builder
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.
    METHODS build_reader REDEFINITION.
    METHODS build_generator REDEFINITION.
    METHODS build_flusher REDEFINITION.
  PRIVATE SECTION.
  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_job2_builder IMPLEMENTATION.

  METHOD build_reader.
    go_reader ?= NEW zcl_stock( ).
  ENDMETHOD.

  METHOD build_generator.
    go_generator ?= NEW zcl_csv( ).
  ENDMETHOD.

  METHOD build_flusher.
    go_flusher ?= NEW zcl_email( ).
  ENDMETHOD.

ENDCLASS.
```

Listing 3.13 Builder Class for Job 2

Jobs 3 and 4 are fairly similar, so we will merge them in Listing 3.14.

```
CLASS zcl_job3_builder DEFINITION
  INHERITING FROM zcl_abs_job_builder
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.
    METHODS build_reader REDEFINITION.
    METHODS build_generator REDEFINITION.
    METHODS build_flusher REDEFINITION.
  PRIVATE SECTION.
  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_job3_builder IMPLEMENTATION.

  METHOD build_reader.
    go_reader ?= NEW zcl_sales( ).
  ENDMETHOD.

  METHOD build_generator.
    go_generator ?= NEW zcl_csv( ).
  ENDMETHOD.

  METHOD build_flusher.
    go_flusher ?= NEW zcl_download( ).
  ENDMETHOD.

ENDCLASS.

CLASS zcl_job4_builder DEFINITION
  INHERITING FROM zcl_abs_job_builder
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.
    METHODS build_reader REDEFINITION.
    METHODS build_generator REDEFINITION.
    METHODS build_flusher REDEFINITION.
  PRIVATE SECTION.
  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_job4_builder IMPLEMENTATION.

  METHOD build_reader.
    go_reader ?= NEW zcl_sales( ).
  ENDMETHOD.

  METHOD build_generator.
    go_generator ?= NEW zcl_tab( ).
```

```
  ENDMETHOD.

  METHOD build_flusher.
    go_flusher ?= NEW zcl_email( ).
  ENDMETHOD.

ENDCLASS.
```

Listing 3.14 Builder Class for Job 3 and 4

What we have done is create a new class and cast it to an interface. In a typical real-world application, you would usually have to do more—like reading some customizing tables, doing authorization checks, etc., though our example skips these steps in order to keep things simple. At this point, we have covered most of the design, as can be seen in Figure 3.5.

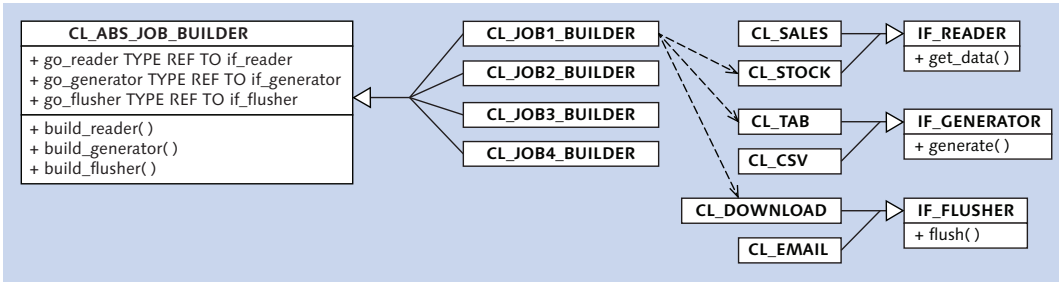


Figure 3.5 Design Mostly Covered

We now have three interfaces and six classes in our library. We also have four builders, each corresponding to a distinct job requirement. Approaching the final stage of the development, we will write the director class. The director is responsible of calling each building step to create our object. Let's take a look at Listing 3.15 and see what the director looks like.

```
CLASS zcl_director DEFINITION
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.
    METHODS construct
      CHANGING
        !co_builder TYPE REF TO zcl_abs_job_builder.
  PRIVATE SECTION.
  PROTECTED SECTION.
ENDCLASS.
```

```
CLASS zcl_director IMPLEMENTATION.

  METHOD construct.
    co_builder->build_reader( ).
    co_builder->build_generator( ).
    co_builder->build_flusher( ).
  ENDMETHOD.

ENDCLASS.
```

Listing 3.15 Director Class

As you can see, the entire class consists of a single method: `CONSTRUCT` (not to be confused with a constructor). If the class was always as simple as this, we could have moved `CONSTRUCT` to another class (such as `CL_ABS_JOB_BUILDER`) so that we could have one less class. However, often enough, the director needs to do further checks, prepare some initial data, etc. Therefore, we have left it as a distinct class.

As our final step, let's take a look at Listing 3.16 and see what our job program looks like.

```
REPORT zjob.

PARAMETERS: p_bldcls TYPE seoclassname OBLIGATORY.
PERFORM main.

FORM main.
  DATA:
    lo_obj TYPE REF TO object,
    lo_builder TYPE REF TO cl_abs_job_builder.

*   Create builder object
    CREATE OBJECT lo_obj TYPE (p_bldcls).
    lo_builder ?= lo_obj.

*   Let the director construct everything
    NEW zcl_job_director( )->construct(
      CHANGING co_builder = lo_builder
    ).

*   Run steps of the job given in P_BLDCLS
    DATA(lr_raw) = lo_builder->go_reader->get_data( ).
    DATA(lr_file) = lo_builder->go_generator->generate( lr_raw ).
    lo_builder->go_flusher->flush( lr_file ).
ENDFORM.
```

Listing 3.16 Job Application

If we run/schedule this job by typing CL_JOB1_BUILDER into P_BLDCLS, the following will happen:

- LO_BUILDER will be an instance of CL_JOB1_BUILDER.
- LO_BUILDER->GO_READER will be an instance of CL_STOCK. Therefore, LO_BUILDER->GO_READER->READ will execute CL_STOCK~READ.
- LO_BUILDER->GO_GENERATOR will be an instance of CL_TAB. Therefore, LO_BUILDER->GO_GENERATOR->GENERATE will execute CL_TAB~GENERATE.
- LO_BUILDER->GO_FLUSHER will be an instance of CL_DOWNLOAD. Therefore, LO_BUILDER->GO_FLUSHER->FLUSH will execute CL_DOWNLOAD~FLUSH.

If we run/schedule this job by typing CL_JOB2_BUILDER into P_BLDCLS, the following will happen:

- LO_BUILDER will be an instance of CL_JOB2_BUILDER.
- LO_BUILDER->GO_READER will be an instance of CL_STOCK. Therefore, LO_BUILDER->GO_READER->READ will execute CL_STOCK~READ.
- LO_BUILDER->GO_GENERATOR will be an instance of CL_CSV. Therefore, LO_BUILDER->GO_GENERATOR->GENERATE will execute CL_CSV~GENERATE.
- LO_BUILDER->GO_FLUSHER will be an instance of CL_EMAIL. Therefore, LO_BUILDER->GO_FLUSHER->FLUSH will execute CL_EMAIL~FLUSH.

If we run/schedule this job by typing CL_JOB3_BUILDER into P_BLDCLS, the following will happen:

- LO_BUILDER will be an instance of CL_JOB3_BUILDER.
- LO_BUILDER->GO_READER will be an instance of CL_SALES. Therefore, LO_BUILDER->GO_READER->READ will execute CL_SALES~READ.
- LO_BUILDER->GO_GENERATOR will be an instance of CL_CSV. Therefore, LO_BUILDER->GO_GENERATOR->GENERATE will execute CL_CSV~GENERATE.
- LO_BUILDER->GO_FLUSHER will be an instance of CL_DOWNLOAD. Therefore, LO_BUILDER->GO_FLUSHER->FLUSH will execute CL_DOWNLOAD~FLUSH.

If we run/schedule this job by typing CL_JOB4_BUILDER into P_BLDCLS, well, you can try to fill the blanks by yourself:

- LO_BUILDER will be an instance of _____.
- LO_BUILDER->GO_READER will be an instance of _____. Therefore, LO_BUILDER->GO_READER->READ will execute _____.

- `LO_BUILDER->GO_GENERATOR` will be an instance of _____. Therefore, `LO_BUILDER->GO_GENERATOR->GENERATE` will execute _____.
- `LO_BUILDER->GO_FLUSHER` will be an instance of _____. Therefore, `LO_BUILDER->GO_FLUSHER->FLUSH` will execute _____.

As you can see, the builder design pattern provides immense flexibility and extendibility when you have objects with a common skeleton (like reader/generator/flusher) with alternative functionalities (sales/stock reader, TAB/CSV generator, file/email flusher). Builder can be seen as a hybrid between factory and strategy (Chapter 4 and Chapter 25, respectively), where you build objects by using predefined objects (sharing the same interfaces) from a library.

Note

The names of the subclasses (`ZCL_SALES`, `ZCL_CSV`, etc.) are hardcoded in this example. See Appendix B on subclass determination for alternative approaches.

3.2 When to Use

If the creation process of an object is simple, then using the builder pattern could be overkill. As basically a less error-prone alternative for a constructor, the builder pattern would come into question only if you foresee complexity regarding creating a new instance of an object.

If your object feels like a Lego toy or like Ikea furniture where you need to build a big structure out of smaller components, then the builder design pattern will probably be a good choice.

3.3 Privacy

Unlike most other creational design patterns, the objects to be returned by the builder design pattern should not be marked as `CREATE PRIVATE`. If the objects were marked as `CREATE PRIVATE`, the builder classes would be unable to create the objects via the `NEW` or `CREATE OBJECT` commands. However, if a class has its own factory method, you can safely mark the class as `CREATE PRIVATE`—builders would call the factory method.

3.4 Summary

Being a composite design pattern, builder might be more challenging than others to understand initially. Now that you have a complete picture, you can restart the section and read again if needed. You may find that the logic is much clearer after a second read.

The builder design pattern shines when it comes to building complex objects typically containing other objects. Builder provides the logic to create a complex object in a step-by-step approach, assembling building blocks like a Lego toy. For simple cases, builder might be overkill—the factory design pattern, discussed in Chapter 4, is the more basic form.

Factory, one of the most popular design patterns, centralizes the process of creating an object. Basically, factory is a static method that creates and returns an object instance. It is primarily used when the underlying concrete class must be determined dynamically or when the steps to create the object are similar in complexity.

4 Factory

The factory design pattern is probably one of the most popular design patterns in the history of object-oriented programming (OOP). The idea behind this pattern is pretty simple, however.

Sometimes, the creation of an object is so simple that you don't even need a constructor. You can simply use the `NEW` command, and that's it. In such cases, you don't need any design pattern. Other times, creating an object requires some parameters and a simple initialization code. In such cases, you would typically create a constructor and put the initialization code into it. To create a new instance, the `NEW` command is used, and parameters are passed.

However, there are cases where object creation is a rather complex process. For example, you may need to decide which concrete class instance to create and cast onto a common interface. In such cases, you simply create a *factory method*, which takes all the steps to create and return an object instance. This is where the factory design pattern comes in.

In this chapter, we will create SAP ERP Financial Accounting (FI) documents for different parties making use of the factory design pattern to do so. Then, we will discuss some of the primary advantages of this design pattern and look at a few related patterns.

4.1 Case Study: FI Documents for Parties

In this example, we will create FI documents for different parties: vendors, customers, and employees. We need a single program that is capable of creating FI

documents for all of these parties individually. Following the strategy design pattern (discussed in detail in Chapter 25), we can come up with a solution, as shown in Figure 4.1.

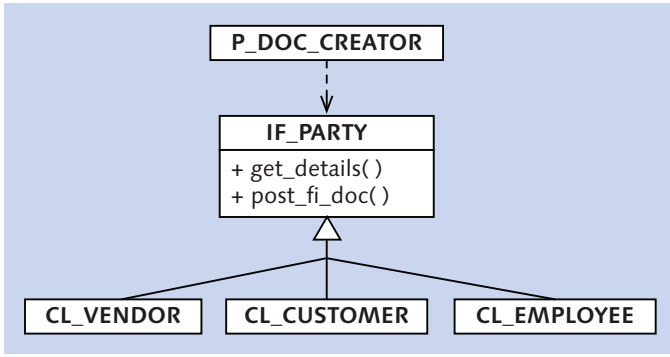


Figure 4.1 Class Hierarchy for Parties

In the Unified Modeling Language (UML) diagram in Figure 4.1, we see that `IF_PARTY` is the common interface for any party type. `CL_VENDOR`, `CL_CUSTOMER`, and `CL_EMPLOYEE` are responsible of implementing `GET_DETAILS` and `POST_FI_DOC`. Although what `CL_VENDOR`, `CL_CUSTOMER`, and `CL_EMPLOYEE` do differs, the interface is the same. The common program `P_DOC_CREATOR` doesn't care which concrete class it's dealing with. As long as the class has an implementation of `IF_PARTY`, it can be used with `P_DOC_CREATOR`.

The next question we have to resolve is how are we going to let the user choose which class to use? One approach, seen in Listing 4.1, is to ask the user to name a class directly.

```
REPORT zdoccreator.

DATA:
  go_obj TYPE REF TO object,
  go_party TYPE REF TO zif_party.
" Some additional data definitions

PARAMETERS: p_clsname TYPE seoclsname OBLIGATORY.
" Some additional select options

START-OF-SELECTION.
  CREATE OBJECT go_obj TYPE (p_clsname).
  go_party ?= go_obj.
  " Main logic of the program, using go_party
```

```
END-OF-SELECTION.

" Some form definitions
```

Listing 4.1 Requesting Class Names from the User

The problem is that the user might not understand what the program is asking for when it requests a "technical class." The user may also simply not know the technical name for the class. In our example, we have rather simple and intuitive class names, such as `CL_VENDOR`. But that's not always the case—you might end up having a class name like `CL_CSKKPPGT` where no user can tell what it does.

Instead of expecting the user to know class names, you can create a Z-table to store class names with a correspondence to a more readable label, like those in Table 4.1.

LABEL	CLSNAME
VENDOR	CL_VENDOR
CUSTOMER	CL_CUSTOMER
EMPLOYEE	CL_EMPLOYEE

Table 4.1 Structure of a Custom Containing Class Names

Once this is done, all you need to do is ask the user for the label. Afterwards, you can read the Z-table for the technical class name and create the corresponding object, as demonstrated in Listing 4.2.

```
REPORT zdoccreator.

DATA:
  go_obj TYPE REF TO object,
  go_party TYPE REF TO zif_party.

" Some additional data definitions

PARAMETERS p_label TYPE zlabel-label OBLIGATORY.
" Some additional select options

START-OF-SELECTION.
  SELECT SINGLE clsname INTO @DATA(gv_clsname)
    FROM zlabel
    WHERE label EQ @p_label.

  CREATE OBJECT go_obj TYPE (gv_clsname).
  go_party ?= go_obj.
```

```
  " Main logic of the program, using go_party
END-OF-SELECTION.

" Some form definitions
```

Listing 4.2 Requesting Understandable Labels from the User

We are making good progress, as long as you overlook the fact that we used unnecessary global variables. While not the best practice in the real world, global variables will do for our simple example of how to combine the strategy and MVC (model–view–controller) design patterns.

Now, imagine that we suddenly need two more client programs to take advantage of `IF_PARTY` and its subclasses. The design for this program would be as shown in Figure 4.2.

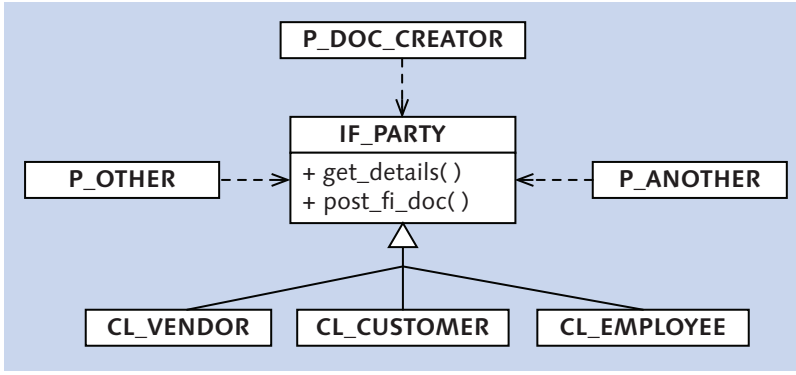


Figure 4.2 Further Client Programs

Having two further programs is actually no hassle—the programs will reuse the `CL_VENDOR`, `CL_CUSTOMER`, and `CL_EMPLOYEE` classes. However, the problem is on the level of object creation. When there was a single program, we needed the following three steps to create an object:

1. Read `ZLABEL` to determine technical class name.
2. Create an object instance.
3. Cast the instance to the interface.

Are we going to repeat those steps each and every time we need an `IF_PARTY` instance? What if we had twenty steps and ten programs? What if we need to modify the object creation logic afterwards? Are we going to modify ten programs,

while trying our best not to make a mistake in any of them? The factory design pattern can step in here. The object creation logic will reside in a separate method, and programs will call that method instead of using the CREATE OBJECT/NEW commands directly.

Listing 4.3 demonstrates what such a method looks like.

```
CLASS zcl_factory DEFINITION
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.
    CLASS-METHODS get_party_instance
      IMPORTING !iv_label TYPE zlabel-label
      RETURNING VALUE(ro_party) TYPE REF TO zif_party.
  PRIVATE SECTION.
  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_factory IMPLEMENTATION.

  METHOD get_party_instance.
    DATA lo_obj TYPE REF TO object.

    SELECT SINGLE clsname INTO @DATA(lv_clsname)
      FROM zlabel
      WHERE label EQ @iv_label.

    CREATE OBJECT lo_obj TYPE (lv_clsname).
    ro_party ?= lo_obj.
  ENDMETHOD.

ENDCLASS.
```

Listing 4.3 Factory Method in Factory Class

Even if it took twenty steps to create an instance, we could store it all in GET_PARTY_INSTANCE. From now on, every time a client program needs an instance of IF_PARTY, it only needs to call this factory method. Let's check out Listing 4.4 to see what our sample program will look like now.

```
REPORT zdoccreator.

" Some data definitions

PARAMETERS p_label TYPE zlabel-label OBLIGATORY.
" Some additional select options

START-OF-SELECTION.
```

```
  DATA(go_party) = zcl_factory=>get_party_instance( p_label ).
  " Main logic of the program, using go_party
END-OF-SELECTION.

" Some form definitions
```

Listing 4.4 Sample Program Consuming Factory Method in Factory Class

So elegant, yet so simple. You may notice that the code is also more comprehensible than before. When a guest developer looks at this code, he/she would understand instantly what is going on. If the developer is curious, he/she could always double-click `GET_PARTY_INSTANCE` and see how exactly the instance is forged.

In this example, we have created a distinct factory class to return an object instance. However, another common method is to have a static method within a class that returns an instance of itself. As an example of this, take a look at Listing 4.5.

```
CLASS zcl_sample DEFINITION
  PUBLIC FINAL CREATE PRIVATE.

  PUBLIC SECTION.
    CLASS-METHODS get_instance
      IMPORTING !iv_dummy TYPE string
      RETURNING VALUE(ro_sample) TYPE REF TO zcl_sample.
  PRIVATE SECTION.
  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_sample IMPLEMENTATION.

  METHOD get_instance.
    ro_sample = NEW #( ).
    " Initialization code
  ENDMETHOD.

ENDCLASS.
```

Listing 4.5 Factory Method in Own Class

What you might notice here is that we have marked the class as private by adding the code `CREATE PRIVATE`. As a result, an instance of this class can no longer be created using `CREATE OBJECT` or `NEW` commands. The only option left is to use the factory method `ZCL_SAMPLE=>GET_INSTANCE`—which is a far more preferable situation. We want the initialization logic in `GET_INSTANCE` to run every time an instance is created, and marking the class itself as private ensures that.

Note

The names of subclasses (ZCL_VENDOR, ZCL_CUSTOMER, etc.) are determined using a Z-table in this example. See Appendix B on subclass determination for alternative approaches.

4.2 Advantages

The factory design pattern will prevent you from recoding the complex object initialization logic in multiple places. Instead, you will be centralizing the code to construct the object, and all other classes will call this central method (called the factory method) to get an instance.

As a result, your classes will be more developer friendly. A new developer in the team would be happy to find factory methods returning the objects he/she needs. Because the developer doesn't need to learn the steps involved in creating a complex object, the time to start reusing existing classes is reduced.

Having centralized methods for object creation also reduces test time. Once the object creation method is well tested, we don't need to worry about it in further steps as we can safely assume that the object we get has been correctly created.

Extensibility is another advantage of the factory design pattern. Factory methods often return an object implementing a certain interface. If you create a new class having this interface, the only place you would need to modify would be the factory method itself. Even if you have dozens of client applications, you wouldn't have to modify a single client application if you used the factory design pattern.

4.3 Related Patterns

Commonly, a software design starts off with a factory method, which is simple and effective. Later on, the design could evolve toward abstract factory, builder, or prototype patterns, depending on the degree and type of flexibility needed.

If you end up in a situation where certain circumstances (such as the underlying OS) force your factory method to behave significantly differently in different situations, you can add another layer of abstraction and lean towards the abstract factory design pattern (Chapter 2).

If you sense that you need to construct a composite object out of other building blocks (such as other objects), you can transform your factory into a builder (Chapter 3).

If you observe that runtime to construct a new object is too high, you can lean towards the prototype design pattern (Chapter 7) where you start creating the new object by cloning an existing one, thus bypassing the initialization code.

4.4 Summary

The factory design pattern is used in cases where object creation is a rather complex process but we don't want to repeat code in every single client application. Centralizing the initialization code in a factory method prevents code duplication.

In most cases, the constructor method of a class with a factory method should be marked as private. As a result, client applications are forced to call the factory method to create an instance so the necessary steps are taken correctly.

One of the performance-oriented design patterns, lazy initialization postpones the creation of an object until the first time it is accessed. Chances are, if the object is not accessed at all, the runtime cost to create the object can be avoided totally.

5 Lazy Initialization

Although not one of the "traditional" design patterns, lazy initialization provides a neat approach that allows for memory efficiency and performance improvement. This pattern is based upon the idea of postponing the creation of an object until the first time it is accessed. As might be guessed, the key motive behind using this pattern is that you can avoid creating the object altogether if you never need it.

In this chapter, we will first look at an example of lazy initialization in practice using logging errors, before briefly detailing some advantages of using lazy initialization and some related patterns.

5.1 Case Study: Logging Errors

In this example, we are going to have an imaginary class, which we'll call `cl_critical`, performing a critical task. The context of the task is not relevant to this use case. This class will contain a "nice object" that is responsible for error management. The nice object is capable of logging errors into the application log, sending emails to the appropriate administrators, and even submitting tickets into a third-party system using a RESTful API.

However, the problem is that this nice object has a huge memory footprint, and creating the object has a high performance cost.

Listing 5.1 demonstrates the structure of the `cl_critical` class without the logic of lazy initialization in place.

```
CLASS zcl_critical DEFINITION
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.
    " Some type definitions
    METHODS constructor.
    METHODS do_critical_stuff.
    METHODS get_error_obj
      RETURNING VALUE(ro_obj) TYPE REF TO zcl_error.

    " Some irrelevant methods

  PRIVATE SECTION.
    DATA go_error TYPE REF TO zcl_error.
    " Some hidden variables
    " Some irrelevant methods
  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_critical IMPLEMENTATION.

  METHOD constructor.
    go_error = NEW #( ).
  ENDMETHOD.

  METHOD do_critical_stuff.
    DATA: lt_bapiret2 TYPE bapiret2_tab,
          lv_error_occurred TYPE abap_bool.

    " Some complex code

    IF lv_error_occurred EQ abap_true.
      go_error->add_messages( lt_bapiret2 ).
    ENDIF.
  ENDMETHOD.

  METHOD get_error_obj.
    ro_obj = go_error.
  ENDMETHOD.

ENDCLASS.
```

Listing 5.1 Class without Lazy Initialization

The program that makes use of this class looks like Listing 5.2.

```
REPORT ZP_APP.

" Some data definitions
" Maybe a selection screen
```

```
PERFORM main.

FORM main.

  " Some data definitions

  DATA(lo_crit) = NEW zcl_critical( ).
  lo_crit->do_critical_stuff( ).

  CHECK lo_crit->has_error( ) EQ abap_true.
  DATA(lo_error) = lo_crit->get_error_obj( ).
  lo_error->write_to_application_log( ).
  lo_error->send_emails( ).
  lo_error->create_ticket( ).

  " Some code to display results to user

ENDFORM.
```

Listing 5.2 Sample Program Using Our Class

Figure 5.1 contains the Unified Modeling Language (UML) diagram for the entire application for a better overview.

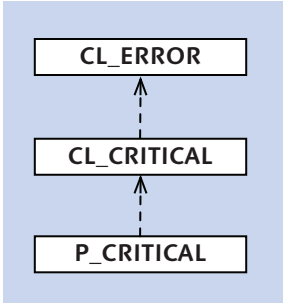


Figure 5.1 UML Overview

Assuming that `CL_ERROR` doesn't raise any exceptions, we have done well so far. However, we're not quite done, as there is one point we still need to consider: How often do we expect errors to occur? On every execution? Not likely. Occasionally? Not likely either.

In a good design, errors should occur pretty rarely. Therefore, most of the time, the costly `GO_ERROR` object will not be required at all. Therefore, creating this object in the constructor of `CL_CRITICAL` is probably a performance and memory drain.

This cost can easily be prevented by using the lazy initialization design pattern, though. With lazy initialization, we will postpone an object's creation until the first time the object is needed. If no error occurs and GO_ERROR is not needed at all, we won't create the object and will significantly save on memory and execution time.

To implement this logic, we only need to make a small change in CL_CRITICAL. The creation of GO_ERROR no longer takes place in the constructor, but instead, we create the object the first time it is accessed. If it is not accessed at all, it won't be created either. See the modified code in Listing 5.3.

```
CLASS zcl_critical DEFINITION
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.
    " Some type definitions
    METHODS constructor.
    METHODS do_critical_stuff.
    METHODS get_error_obj RETURNING VALUE(ro_obj) TYPE REF TO zcl_
error.

    " Some irrelevant methods

  PRIVATE SECTION.
    DATA go_error TYPE REF TO zcl_error.
    " Some hidden variables
    " Some irrelevant methods
  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_critical IMPLEMENTATION.

  METHOD constructor.
    " We are NOT creating GO_ERROR here!
  ENDMETHOD.

  METHOD do_critical_stuff.
    DATA: lt_bapiret2 TYPE bapiret2_tab,
          lv_error_occurred TYPE abap_bool.

    " Some complex code

    IF lv_error_occurred EQ abap_true.
      get_error_obj( ). " We don't need to receive
      go_error->add_messages( lt_bapiret2 ).
    ENDIF.
  ENDMETHOD.
```

```
  METHOD get_error_obj.
    IF go_error IS INITIAL.
      go_error = NEW #( ). " Object is created on first access
    ENDIF.
    ro_obj = go_error.
  ENDMETHOD.

ENDCLASS.
```

Listing 5.3 Class with Lazy Initialization

We don't need to modify the client program(s) at all in order to use the lazy initialization pattern—it all happens in the class. With the modifications to the code in Listing 5.3, we are done!

5.2 Advantages

The decision to use the lazy initialization design pattern is similar to the decision to use the `BOXED` command in data definitions. If you are not sure if an object will be needed during runtime, postponing the creation process to the first time the object is needed is a good idea in terms of performance. That way, you don't create the instance if no one uses it.

5.3 Related Patterns

To make this pattern work, there should be no direct access to the lazy object. The object should only be returned by a method. If you make the lazy object a globally visible variable, other classes can access it while it is still initial. Often, the method to return the lazy object can be implemented as a singleton (Chapter 8) or multiton (Chapter 6) design pattern. If that's not the case, you will probably end up implementing a factory (Chapter 4) or builder (Chapter 3) method to return the object.

If the object is really costly to create, you can consider using both lazy initialization and the prototype design pattern (Chapter 7) simultaneously. First, access can be managed via lazy initialization so that the object is not created at all if it's not needed. If more objects of the same type is needed, creating the new object can be done by cloning the existing object—which is the logic of prototype design pattern.

5.4 Summary

Although extremely simple, lazy initialization can be an invaluable asset for performance improvement and is based on the idea of postponing an object creation until the first time the object is accessed.

Make sure that the postponed object is marked as private; you wouldn't want client programs to access the object directly.

Multiton is a more advanced version of the singleton design pattern. For each object key (such as a material number), a static object instance of a class (such as a material class) is kept in a central location. Whenever a client requests an object corresponding to the object key, the existing object is returned instead of creating a new one.

6 Multiton

Multiton is essentially an extension of the singleton design pattern discussed in Chapter 8. Therefore, we strongly recommended that you understand the singleton pattern before moving on to multiton.

Singleton involves keeping a static instance of a class and returning it whenever requested. That way, we make sure that only one instance of the class is stored in the memory, no matter how many times the object creation method is called. This approach can save memory significantly and improve performance.

Multiton takes the same principle and adds one more thing on top of it: Instead of keeping a single static object, the design pattern keeps a static array of objects for each key. Clients requesting an object provide a key value and access the corresponding static object.

Here is a more detailed explanation: While requesting an object, you need to pass a key. If an object corresponding to that key is being requested for the first time, the multiton method creates the object, stores it in a static hashed table, and returns it. If an object corresponding to the same key is requested again, the object is not created again—instead, the instance stored in the static hashed table is returned, saving us both the runtime to re-create the object and the memory to store it.

Note that, since the same object is returned for the same key, the state is shared among all clients—one modification will affect all remaining clients. Therefore, multiton would typically be preferred for objects with immutable states, such as master data classes.

This approach is simpler than it sounds, actually—as simple as singleton itself. Let's now look at a concrete example of multiton in practice before moving on to some criteria for when to use multiton, when to avoid it, and a brief discussion of state modification.

6.1 Case Study: Vendor Balance

In this example, we are going to create a custom application where the user can enter planned payments to vendors. In a table control, the user is going to manually enter vendor numbers (LIFNR), payment dates, and payment amounts. The same vendor can be entered multiple times for different dates.

Sounds simple so far. The catch is that, every time the user enters a vendor number, he/she would like to see the balance of the vendor immediately in a read-only field on the table control. Table 6.1 represents how the table would look.

Date	Vendor	Amount	Balance
May 12	100235	750	4,500
May 17	102334	600	2,700
May 18	100333	940	1,000
May 19	100235	600	4,500

Table 6.1 Sample View within the Application GUI

The balance column shows a real-time calculated value based on tables `BSAK` and `BSIK`. We can assume the value in the balance column to be the balance at the moment the application is executed. In a real-world application, the user would probably want to see a running balance as well, but we don't want to complicate things for the sake of our example.

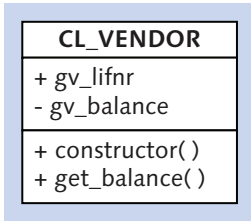


Figure 6.1 Vendor Class

In our problem, every time the user enters a new line, we need to validate the existence of the vendor and read its balance. For this purpose, we can create a new class called CL_VENDOR, which will be used by the application. You can see the basic structure of this class in Figure 6.1.

The source code of this class is demonstrated in Listing 6.1.

```
CLASS zcl_vendor DEFINITION
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.
    DATA gv_lifnr TYPE lifnr READ-ONLY.

    METHODS constructor
      IMPORTING
        !iv_lifnr TYPE lifnr.

    METHODS get_balance
      RETURNING
        VALUE(rv_balance) TYPE dmbtr.

  PRIVATE SECTION.
    DATA:
      gv_balance TYPE dmbtr,
      gv_balance_read TYPE abap_bool.

  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_vendor IMPLEMENTATION.

  METHOD constructor.
    " Check if IV_LIFNR exists in LFA1 and raise error if not
    gv_lifnr = iv_lifnr.
  ENDMETHOD.

  METHOD get_balance.
    IF gv_balance_read EQ abap_false.
      " Read vendor balance from BSIK, BSAK, etc into gv_balance
      gv_balance_read = abap_true.
    ENDIF.
    rv_balance = gv_balance.
  ENDMETHOD.

ENDCLASS.
```

Listing 6.1 Basic Vendor Class

Listing 6.2 contains a code snippet from the main program, which shows how to call CL_VENDOR to read the balance of a given vendor.

```
FORM new_line
  USING iv_lifnr TYPE lifnr
  CHANGING cs_line TYPE t_line.

  DATA(lo_vendor) = NEW zcl_vendor( iv_lifnr ).
  cs_line-balance = lo_vendor->get_balance( ).
ENDFORM.
```

Listing 6.2 Code Snippet Using Vendor Class

Now, let's turn back to the data being entered into our program, as seen in Table 6.1. If you look at lines 1 and 4, you can see that the same vendor has been entered twice. In our current architecture, the application will enter NEW_LINE twice for vendor 100235 and read tables BSIK and BSAK twice for the exact same vendor. This performance problem can be solved using the multiton design pattern.

In the same way that we can write a static factory method in singleton (Chapter 8), we are going to do the same here. There will be one difference, however: Created instances will be stored inside a static internal table and reused if necessary. Listing 6.3 demonstrates what CL_VENDOR would look like with this logic.

```
CLASS zcl_vendor DEFINITION
  PUBLIC FINAL CREATE PRIVATE.

  PUBLIC SECTION.
    DATA gv_lifnr TYPE lifnr READ-ONLY

    CLASS-METHODS get_instance
      IMPORTING !iv_lifnr TYPE lifnr
      RETURNUNG VALUE(ro_obj) TYPE REF TO zcl_vendor.

    METHODS get_balance
      RETURNING
        VALUE(rv_balance) TYPE dmbtr.

  PRIVATE SECTION.

   TYPES:
      BEGIN OF t_multiton,
        lifnr TYPE lifnr,
        obj TYPE REF TO zcl_vendor,
      END OF t_multiton,

      tt_multiton TYPE HASHED TABLE OF t_multiton
        WITH UNIQUE KEY primary_key COMPONENTS lifnr.
```

```
    CLASS-DATA gt_multiton TYPE tt_multiton.

    DATA:
      gv_balance TYPE dmbtr,
      gv_balance_read TYPE abap_bool.

    METHODS constructor
      IMPORTING
        !iv_lifnr TYPE lifnr.

  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_vendor IMPLEMENTATION.

  METHOD constructor.
    " Check if IV_LIFNR exists in LFA1 and raise error if not
    gv_lifnr = iv_lifnr.
  ENDMETHOD.

  METHOD get_instance.

    ASSIGN gt_multiton[ KEY primary_key
      COMPONENTS lifnr = iv_lifnr ]
      TO FIELD-SYMBOL(<ls_multiton>).

    IF sy-subrc NE 0.
      " Check if IV_LIFNR exists in LFA1 and raise error if not
      DATA(ls_multiton) = VALUE t_multiton( lifnr = iv_lifnr ).
      ls_multiton-obj = NEW #( iv_lifnr ).
      INSERT ls_multiton INTO TABLE gt_multiton
        ASSIGNING <ls_multiton>.
    ENDIF.

    ro_obj = <ls_multiton>-obj.

  ENDMETHOD.

  METHOD get_balance.
    IF gv_balance_read EQ abap_false.
      " Read vendor balance from BSIK, BSAK, etc into gv_balance
      gv_balance_read = abap_true.
    ENDIF.
    rv_balance = gv_balance.
  ENDMETHOD.

ENDCLASS.
```

Listing 6.3 Vendor Class Using Multiton

GT_MULTITON is our static object pool. Here, we store instances of CL_VENDOR for each LIFNR, making sure that a single instance of CL_VENDOR exists for each LIFNR. Every time a new object is requested in GET_INSTANCE, we check if this LIFNR exists in GT_MULTITON and return the existing object if it does. If not, we create a new instance of CL_VENDOR, add it to GT_MULTITON, and return the object.

In this way, if you request a CL_VENDOR for LIFNR = 100235 twice, you get the exact same object. If you ask for it 100 times, you will get the exact same object 100 times. This approach saves lots of memory—and runtime as well: Once the balance of 100235 has been calculated, at some point in the application, it doesn't need to be calculated again. If the user enters "100235" five times, the balance will only be calculated on the first occurrence. On the following four occurrences, GET_INSTANCE will return the exact same object (containing the calculated balance), and GET_BALANCE will simply return the value from the cache.

6.2 When to Use

As mentioned, we use this pattern frequently if we have an object corresponding to a master data type, such as material, vendor, etc.—as long as the state of the object can safely be shared.

This approach may look especially tempting if you need to do an immutable, performance-hungry calculation per key. In our case study, we had to calculate the balance of each vendor only once. That way, the balance is calculated only the first time the user has entered a vendor number—not every time.

Note that the object to be returned by this pattern should be marked as CREATE PRIVATE. In the example in Section 6.1, the constructor method has been marked private as well. Otherwise, other classes might skip the creational method and create the object via NEW or CREATE OBJECT commands, which is not desirable because they would be bypassing the multiton method completely and creating distinct object instances instead of sharing a single object for the given key.

6.3 When to Avoid

Before using multiton, you should consider if multiple instances of the same key are really going to be requested. If you have an application where the same

material number can be entered multiple times, managing material objects using the multiton design pattern would probably save some memory and runtime.

However, if a material can't be entered multiple times, each material object will be needed and requested only once. Therefore, the overhead cost of using the multiton pattern, which involves hashed table management and an extra class in memory, won't pay off. In such a case, creating material objects using the `NEW` command or a factory method would be a better idea.

6.4 State Modification

Multiton is especially useful if you need to create objects corresponding to master data, such as clients, materials, etc. Since master data is relatively constant, you can safely reuse the same object in most cases. Be careful though: If you modify the data in a multiton object, that data is changed across all references within your program because the state is singular—just like the object itself. If you want distinct objects with distinct states, multiton is not the right choice.

6.5 Summary

Multiton can be seen as singleton on steroids. This design pattern is based on the idea of keeping static objects per object key; whenever a client shows up with an object key, the static object is returned. This approach can dramatically reduce memory consumption because only one object per key is stored in the memory.

Multiton is typically used in, but not limited to, cases where you have a class corresponding to a master data type. Classes subject to multiton should be marked as private. Otherwise, you can't prevent client programs from creating redundant identical objects.

Be mindful of the fact that object singularity also means state singularity.

Using the prototype design pattern is essentially a clone operation. Prototype clones an existing object to create a new object, and then you may modify the new state as needed. This design pattern is significantly useful for cases where the initialization of the class is performance hungry. Cloning an existing object avoids that cost.

7 Prototype

Simple, yet effective, the prototype design pattern is used to create a new object simply by cloning an existing one. Under circumstances where the process of creating an instance is too costly, you can significantly save on runtime by cloning an already-created object.

Prototype can easily be imagined like the mitotic division of a cell, which results in two identical cells. You can make minor changes to the new cell after it is cloned.

In this chapter, we will first look at a practical example of the prototype design pattern and then discuss how to change class properties, look at some of the implications of clone operations, and examine a few related design patterns.

7.1 Case Study: Item Clone

In this example, we will work on an imaginary purchase requisition application where the user has to enter the vendor code (LIFNR), material number (MATNR), unit price (PREIS and WAERS), and quantity (MENGE and MEINS) into the GUI. We will want it to be possible to enter multiple line items.

Listing 7.1 demonstrates what the model class of such an application would roughly look like.

```
CLASS zcl_purchase DEFINITION
  PUBLIC FINAL CREATE PUBLIC.
```

```
  PUBLIC SECTION.
    " Some type definitions

    METHODS add_line IMPORTING !is_line TYPE t_line.
    METHODS load_from_db IMPORTING !iv_docno TYPE zdocno.
    METHODS save_to_db.

  PRIVATE SECTION.
    " Some hidden variables
    " Some boring methods
  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_purchase IMPLEMENTATION.

  METHOD add_line.
    APPEND VALUE #(
      obj_material = NEW zcl_material( is_line-matnr )
      lifnr = is_line-lifnr
      matnr = is_line-matnr
      price = VALUE #(
        preis = is_line-preis
        waers = is_line-waers )
      quan = VALUE #(
        menge = is_line-menge
        meins = is_line-meins )
      ) TO gt_line.
  ENDMETHOD.

  METHOD load_from_db.
    " Some boring code containing SELECT, etc.
  ENDMETHOD.

  METHOD save_to_db.
    " Some boring code containing MODIFY, etc.
  ENDMETHOD.

ENDCLASS.
```

Listing 7.1 Purchase Requisition Class

So far, so good. As you see, `GT_LINE` is a nested internal table with five fields: one object (`OBJ_MATERIAL`); two variables (`LIFNR`, `MATNR`); and two work areas (`PRICE`, `QUAN`). At this point, we will assume that creating a new `CL_MATERIAL` has a high runtime cost, which means that `CL_MATERIAL~CONSTRUCTOR` contains costly code snippets that take quite some time to execute.

Listing 7.2 demonstrates what the `CL_MATERIAL` class looks like.

```
CLASS zcl_material DEFINITION
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.
    " Some type definitions

    METHODS constructor IMPORTING !iv_matnr TYPE matnr.
    " Some further methods

  PRIVATE SECTION.
    " Some hidden methods
  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_material IMPLEMENTATION.

  METHOD constructor.
    " Some very slow but unavoidable code
  ENDMETHOD.

  " Some more methods
ENDCLASS.
```

Listing 7.2 Material Class

If the user enters a new line item containing a brand new material, we must call `ADD_LINE` again. Thus, because we will be creating a new instance of `CL_MATERIAL` we will have to endure high runtime cost. On the other hand, what if the user enters a material number that he/she had entered before? Would we still create a brand new instance of `CL_MATERIAL`? With our current code, we would. So we must ask ourselves, "Is there a better way?"

Luckily there is! What we will do is clone an existing material object instead of creating a new one—the core logic of the prototype design pattern. Let's look at the Unified Modeling Language (UML) diagram in Figure 7.1 to see what this should look like.

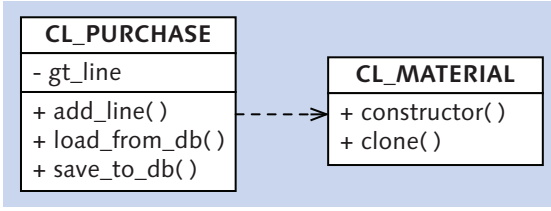


Figure 7.1 Relationship between Purchase Order and Material Classes

The UML diagram has only one extra method: CLONE. This method will do exactly what it says: clone the object. That's the easy part; the relatively tricky part will transpire in CL_PURCHASE. Let's start simple and see what CL_MATERIAL looks like in Listing 7.3.

```
CLASS zcl_material DEFINITION
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.
    " Some type definitions

    METHODS constructor IMPORTING !iv_matnr TYPE matnr.
    METHODS clone RETURNING VALUE(ro_material) TYPE REF TO zcl_material.
    " Some further methods

  PRIVATE SECTION.
    " Some hidden methods
  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_material IMPLEMENTATION.

  METHOD constructor.
    " Some very slow but unavoidable code
  ENDMETHOD.

  METHOD clone.
    SYSTEM-CALL OBJMGR CLONE me TO ro_material.
  ENDMETHOD.

  " Some more methods
ENDCLASS.
```

Listing 7.3 Material Class with Prototype

Normally, within the CLONE method, we would create the object RO_MATERIAL (avoiding the slow code in the constructor using flag variables and conditions) and copy the entire state of ME to RO_MATERIAL. However, we used a cheat code. What you see is a system call to clone ME, which is not meant for external use. No one gets hurt by little hacks like this, as long as the unit tests are done well.

Now, back to CL_PURCHASE. Instead of creating a new CL_MATERIAL object every time ADD_LINE is called, we will clone an existing material object whenever possible. The code to do so can be seen in Listing 7.4.

```
METHOD add_line.
  DATA lo_material TYPE REF TO zcl_material.

  ASSIGN gt_line[ matnr = is_line-matnr ]
    TO FIELD-SYMBOL(<ls_line>).
  IF sy-subrc EQ 0.
    lo_material = <ls_line>-obj_material->clone( ).
  ELSE.
    lo_material = NEW zcl_material( is_line-matnr ).
  ENDIF.

  APPEND VALUE #(
    obj_material = lo_material
    lifnr = is_line-lifnr
    matnr = is_line-matnr
    price = VALUE #(
      preis = is_line-preis
      waers = is_line-waers )
    quan = VALUE #(
      menge = is_line-menge
      meins = is_line-meins )
    ) TO gt_line.
ENDMETHOD.
```

Listing 7.4 Adding a New Line with Material Clone

Brilliant, isn't it? Instead of re-calling the constructor every time the same material is added, we simply cloned an existing object and avoided the initialization runtime cost.

7.2 Changing Class Properties

When using the prototype pattern, one thing to take into account is that you'll probably want to change some properties of the new class. For example, imagine that you have a class that reads and stores the stock quantity of a given material and its storage location. If you clone such an object but need to change the storage location or material code, you would need to read the stock of the new LGORT/MATNR from scratch. Doing so makes the cloning process less profitable.

Using this pattern makes the most sense if the construction of the object is somewhat costly and if whatever you build in the constructor won't change too much once it is prepared—even after the object is cloned.

7.3 Clone Operations

When using the prototype pattern, you must also consider how deep or shallow a clone should be. In a deep clone operation, all variable values within the source class are cloned to the target class. In a shallow clone operation, the values are cloned partially. Since each and every programming operation has a runtime cost, these considerations will partially determine how much using the prototype pattern will benefit you.

If your application contains numerous cloneable classes in a hierarchy of inheritance or composition, creating a distinct class with static methods of cloning objects might be a good idea. An alternative approach is to let each and every cloneable class have its own method called `CLONE`.

Be careful though: If your cloned object contains data or class references, they will be cloned as well. However, what you have cloned will be an address in memory. Therefore, the data source or object that they point to will remain the same. When you change `GO_OBJ_OLD->GO_SUBOBJ`, you also change `GO_OBJ_NEW->GO_SUBOBJ` because they point to the same memory address. Instead, you might need to create new instances of variables/objects, and make the clone pointers refer to the new instances.

7.4 Related Patterns

If you are cloning a class and don't expect changing anything at all, multiton (Chapter 6) could be a better approach. By using prototype, you are assigning an ad-hoc memory space for each object instance, which can be prevented in multiton.

If the class you want to clone has a partially fixed and partially variable state, the flyweight design pattern (Chapter 15), could be a better approach. That way, instead of assigning ad-hoc memory space for parts of the objects that are exactly the same, you keep them in one place and only assign memory space for the variable part.

7.5 Summary

The idea behind prototype is to clone an existing object to create a new one. This design pattern makes sense in cases where the constructor of an object has a demanding runtime. Cloning and modifying an existing object prevents the re-execution of the slow code.

A basic decision for using prototype is about how deep or shallow a clone should be. You also need to be mindful of references in the source object. You sometimes may need to create new instances in the clone so that they don't point the same object/variable.

Before using prototype, check if multiton or flyweight is more suitable for the case at hand.

Sometimes you need to ensure that a single instance of a class is shared among the entire call stack. This requirement may arise due to performance concerns or from the need to share a singular state throughout the entire system. Whatever the reason may be, the singleton design pattern ensures the existence of a single shared object among all involved clients.

8 Singleton

The term *singleton* is one of the most widely known concepts in the world of design patterns. Even if a developer has a shallow knowledge of design patterns, he/she probably heard about singleton—along with MVC (model–view–controller, Chapter 1), of course. Singleton's purpose is to ensure that only one instance of a class exists. This instance will be reused among all programs, functions, methods, etc. whenever a new instance is requested.

This approach not only contributes to the overall performance, it also provides a certain degree of consistency. When a single instance is shared among different code snippets, you make sure that the state (variables, etc.) of the common object is also shared. Sometimes, this is exactly what you need.

In this chapter, we will look at a practical example of the singleton design pattern before discussing some the advantages and disadvantages of choosing singleton as well as some related patterns.

8.1 Case Study: Subcomponents

In this example, we will have a system with four elements: a main program (in Transaction SE38) and three classes (in Transaction SE24) that have various purposes, as shown in Figure 8.1.

In Figure 8.1, `CL_READ` is responsible for reading data from SAP tables and functions. `CL_CALCULATE` is responsible for performing calculations, and `CL_POST` is

responsible for posting data by creating a new document. They are distinct classes because other programs can use them separately, if the need arises.

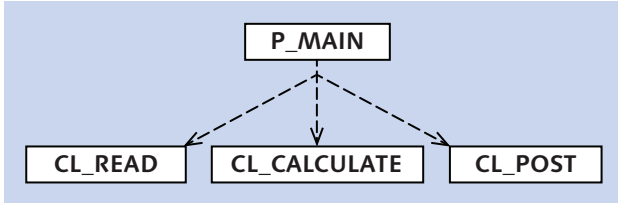


Figure 8.1 Basic Architecture

To make things more clear, Listing 8.1 demonstrates what the main flow of `P_MAIN` looks like.

```
* Get data from reader
  DATA(lo_read) = NEW zcl_read( ).
  lo_read->read_data( ).
  DATA(lt_raw) = lo_read->get_data( ).

* Do calculations
  DATA(lo_calc) = NEW zcl_calculate( ).
  lo_calc->calculate( EXPORTING it_raw_data = lt_raw ).
  DATA(lt_final) = lo_calc->get_calculated_data( ).

* Create documents
  DATA(lo_post) = NEW zcl_post( ).
  cl_post->create_documents( EXPORTING it_calc_data = lt_final ).
```

Listing 8.1 The Main Program

Although extremely simplified, this code snippet demonstrates clearly what the application does as a whole: It gets the data from the reader, does the calculation, and creates documents. Now here is the tricky part: This application makes use of bills of materials (BOMs). To be more clear, `CL_READ`, `CL_CALCULATE`, and `CL_POST` all need to *explode* (expand to access individual items) the BOMs of products to get some data from their components.

The first approach you might try would be creating a BOM-related class. This class would have an appropriate method to explode the BOM using the function module `CS_BOM_EXPL_MAT_V2` and to cache the MATNR and BOM data in case the same material is called more than once. Figure 8.2 displays the Unified Modeling Language (UML) diagram for this approach, where the central BOM class is accessed by multiple client classes.

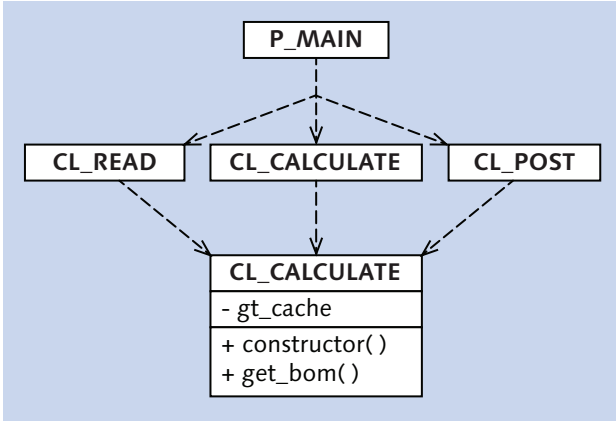


Figure 8.2 BOM Class Placed into the Architecture

Listing 8.2 demonstrates what `CL_BOM` might look like.

```
CLASS zcl_bom DEFINITION
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.
    " Some type definitions

    METHODS constructor.

    METHODS get_bom
      IMPORTING
        !iv_matnr TYPE matnr
      RETURNING
        VALUE(rt_bom) TYPE zt_bom.

  PRIVATE SECTION.
    TYPES: BEGIN OF t_cache,
      matnr TYPE matnr,
      bom TYPE zt_bom,
    END OF t_cache,

    tt_cache TYPE HASHED TABLE OF t_cache
      WITH UNIQUE KEY primary_key COMPONENTS matnr.

    DATA gt_cache TYPE tt_cache.

    " Some further hidden data
    " Some hidden methods

  PROTECTED SECTION.
```

```
ENDCLASS.

CLASS zcl_bom IMPLEMENTATION.

  METHOD constructor.
    " Some initialization
  ENDMETHOD.

  METHOD get_bom.

    ASSIGN gt_cache[ KEY primary_key COMPONENTS matnr = iv_matnr ]
      TO FIELD-SYMBOL(<ls_cache>).

    IF sy-subrc NE 0.
      INSERT VALUE #(
        matnr = iv_matnr
        bom = read_bom_from_scratch( iv_matnr )
        ) INTO TABLE gt_cache ASSIGNING <ls_cache>.
    ENDIF.

    rt_bom[] = <ls_cache>-bom[].

  ENDMETHOD.

  " Some further methods

ENDCLASS.
```

Listing 8.2 The BOM Class without Singleton

To make the code less complicated, we excluded the details of the hidden (private) method `READ_BOM_FROM_SCRATCH`. Logically, this method would call the function `CS_BOM_EXPL_MAT_V2` and return the components in a clean format.

Everything looks good so far. We have created a central class to deal with the dreadful BOM explosion process, and it even has a caching mechanism! Even if the same material is requested more than once, the BOM explosion only takes place the first time. On future requests, the cache from `GT_CACHE` is used. This might remind you of the lazy initialization design pattern (Chapter 5), if you have read that chapter already.

However, this code can be further improved. At this point, `CL_READ`, `CL_CALCULATE`, and `CL_POST` contain their own distinct instances of `CL_BOM`. During a sample runtime flow, the following actions take place:

1. `CL_READ` requires the components of material M1001. `CL_READ~GO_BOM` reads them from the scratch.

2. CL_READ requires the components of material M1001 again. CL_READ~GO_BOM returns them from the cache.
3. CL_CALCULATE requires the components of material M1001. CL_CALCULATE ~GO_BOM reads them from the scratch.
4. CL_CALCULATE requires the components of material M1001 again. CL_CALCULATE~GO_BOM returns them from the cache.
5. CL_POST requires the components of material M1001. CL_POST~GO_BOM reads them from the scratch.
6. CL_POST requires the components of material M1001 again. CL_POST~GO_BOM returns them from the cache.

Clearly, we have read the BOM data of M1001 three times, despite using a cache mechanism within CL_BOM. Without any cache, we could have read it six times, which would have been worse, but three times is still worse than reading the BOM of M1001 only once.

Yes, you heard that right. There is a way of making CL_READ, CL_CALCULATE, and CL_POST share the same instance (and cache) of CL_BOM, thus reading the BOM only once for M1001 during the entire runtime. This functionality is achieved by making use of the singleton design pattern. Let's take a look of the UML diagram in Figure 8.3 first.

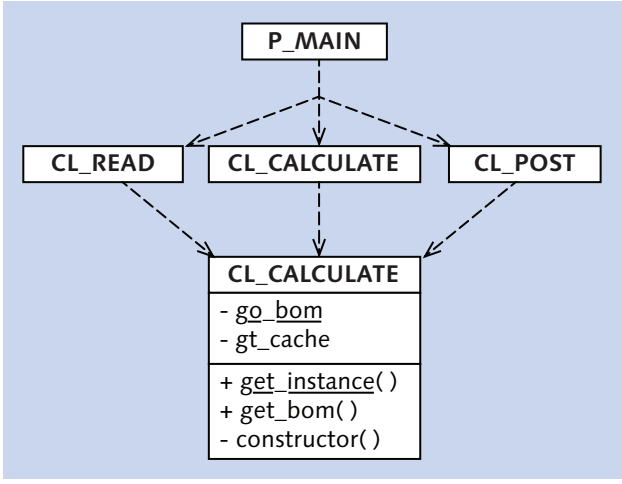


Figure 8.3 BOM Class Turned into Singleton

This diagram is similar to the one in Figure 8.1. The differences in CL_BOM are small, but vital, as follows:

- CONSTRUCTOR is private now. This means, you can no longer create an instance of CL_BOM using the command CREATE OBJECT or NEW.
- CL_BOM stores a static instance of itself, called GO_BOM.
- CL_BOM has a static public method called GET_INSTANCE, which should create and return GO_BOM on the first call and just return GO_BOM on further calls.

This sounds more complicated than it looks, so let's move on to Listing 8.3 to see CL_BOM in action.

```
CLASS zcl_bom DEFINITION
  PUBLIC FINAL CREATE PRIVATE.

  PUBLIC SECTION.
    " Some type definitions

    CLASS-METHODS get_instance
      RETURNING
        VALUE(ro_bom) TYPE REF TO zcl_bom.

    METHODS get_bom
      IMPORTING
        !iv_matnr TYPE matnr
      RETURNING
        VALUE(rt_bom) TYPE zt_bom.

  PRIVATE SECTION.
    " Same definitions as before; plus...
    CLASS-DATA go_bom TYPE REF TO zcl_bom.
    METHODS constructor.

  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_bom IMPLEMENTATION.

  METHOD get_instance.
    IF go_bom IS INITIAL.
      go_bom = NEW #( ).
    ENDIF.

    ro_bom = go_bom.
  ENDMETHOD.

  METHOD constructor.
    " Some initialization
```

```
  ENDMETHOD.

  METHOD get_bom.
    " Same as before
  ENDMETHOD.
ENDCLASS.
```

Listing 8.3 The BOM Class with Singleton

From now on, you need to call `CL_BOM=>GET_INSTANCE` to get a new instance of `CL_BOM`. This method always returns the static variable `CL_BOM=>GO_BOM`.

This means that, even if `CL_READ`, `CL_CALCULATE`, and `CL_POST` call `CL_BOM=>GET_INSTANCE` independently and multiple times, they all will get the same object instance, which is the static variable `CL_BOM=>GO_BOM`.

As a result, our sample application would run as follows:

1. `CL_READ` requires the components of material M1001. `CL_READ~GO_BOM` (which is equal to `CL_BOM=>GO_BOM`) reads them from the scratch.
2. `CL_READ` requires the components of material M1001 again. `CL_READ~GO_BOM` (which is equal to `CL_BOM=>GO_BOM`) returns them from the cache.
3. `CL_CALCULATE` requires the components of material M1001. `CL_CALCULATE~GO_BOM` (which is equal to `CL_BOM=>GO_BOM`) returns them from the cache.
4. `CL_CALCULATE` requires the components of material M1001 again. `CL_CALCULATE~GO_BOM` (which is equal to `CL_BOM=>GO_BOM`) returns them from the cache.
5. `CL_POST` requires the components of material M1001. `CL_POST~GO_BOM` (which is equal to `CL_BOM=>GO_BOM`) returns them from the cache.
6. `CL_POST` requires the components of material M1001 again. `CL_POST~GO_BOM` (which is equal to `CL_BOM=>GO_BOM`) returns them from the cache.

As you see, the singleton design pattern has ensured that only one instance of `CL_BOM` is used throughout the entire application, and the cache is shared. M1001's BOM was exploded only once during the entire runtime.

Note

The object to be returned by this pattern should be marked as `CREATE PRIVATE`. Otherwise, other classes might skip the singleton method and create the object via `NEW` or `CREATE OBJECT` commands, which is not desirable. The very purpose of singleton design pattern is to share a single object instance globally, and skipping the singleton method leaves open the possibility to create distinct instances.

8.2 Advantages and Disadvantages

The most significant advantage of singleton is that it decreases the memory footprint of your application. Instead of having multiple instances of the same class, you can get away with creating a single class that shares its state among multiple objects. Having only one class is the second significant advantage.

However, this can be a disadvantage as well. If your objects need different states instead of a shared one, singleton is probably not the answer you are looking for. (Hint: the flyweight design pattern might be what you are looking for. See Chapter 15 for more information.)

8.3 Related Patterns

The singleton design pattern is a good match with the factory design pattern (Chapter 4). Logically, you need a global static access point to get a singleton instance. This access point is often a factory method. If you need a more advanced method of creating a singleton instance, you can pick another creational pattern as well, such as builder (Chapter 3) or abstract factory (Chapter 2).

In a typical case, the factory method in question will be implementing the lazy initialization design pattern (Chapter 5). Lazy initialization will delay the singleton instantiation until the first time it is required.

Singleton is used when the system only needs one instance of a class. However, there might also be a case where you need one instance for a given key value. For example, you might need a single instance for each KUNNR (entered by the user). For such cases, singleton has a cousin, called multiton. We recommend that you take a look at and understand the multiton design pattern (Chapter 6) to complete your mental picture of the single instance paradigm.

8.4 Summary

Singleton is about making sure that a class has only one shared object instance among all client applications during the entire runtime. This is achieved by marking the singleton object as private and returning a single static instance through a factory method. Singleton will decrease the memory consumption of your application. Lazy initialization is a good companion of singleton. Multiton is a more advanced version of singleton.

PART III

Structural Design Patterns

We all have used cable converters of some sort in our lives. Some convert HDMI to USB, while others convert 2.5mm jacks to 3.5mm. If we implement this logic to the object-oriented world, we come to the adapter design pattern. This pattern is used when two classes with incompatible interfaces need to work together.

9 Adapter

We all know what physical adapters do: They take one interface and convert it to another. US-to-European plug adapters are a typical example of this concept. Initially, an electrical device you purchase from the US can't be plugged into a European wall socket. However, if you have an adapter in between, there's no problem—you can use whatever US device you like in Europe.

In our virtual object-oriented world, the adapter design pattern has a similar functionality. When you have two distinct classes with incompatible interfaces, all you need to do is to develop an adapter class to make a conversion.

In this chapter, we will first look at an in-depth example of what the adapter design pattern looks like in practice. We will then take a look at a few ancillary topics: glue code, two-way adapters, and legacy classes.

9.1 Case Study: Project Management Tools

In this example, we create a class (`cl_ps`) capable of creating work breakdown structures (WBSs) in SAP ERP Project System (PS). The initial model class would look like Listing 9.1.

```
CLASS zcl_ps DEFINITION
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.

    METHODS create_project
```

```
      IMPORTING
      !is_proj TYPE zs_proj.

    METHODS create_wbs
      IMPORTING
        !is_wbs TYPE zs_wbs.

    METHODS create_activity
      IMPORTING
        !is_activity TYPE zs_activity.

    " Some further methods

  PRIVATE SECTION.
    " Some type & data definitions + methods

  PROTECTED SECTION.
    " Possibly some type & data definitions + methods

ENDCLASS.

CLASS zcl_ps IMPLEMENTATION.

  METHOD create_project.
    " Some code to create a new PS project using BAPI, BDC, etc.
  ENDMETHOD.

  METHOD create_wbs.
    " Some code to create a new PS WBS element using BAPI, BDC, etc
.
  ENDMETHOD.

  METHOD create_activity.
    " Some code to create a new PS activity using BAPI, BDC, etc.
  ENDMETHOD.

  " Some further methods

ENDCLASS.
```

Listing 9.1 Class Targeting PS

This simplified class gives you the basic idea of its purpose. The class is capable of creating a project and adding WBS elements and activities underneath. Next, if we had to write a program to create WBS elements, we would need the architecture in Figure 9.1.

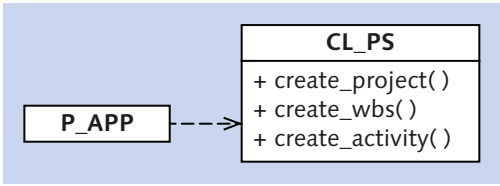


Figure 9.1 Basic Unified Modeling Language (UML) Structure

The code for the program to create WBS elements would look like Listing 9.2, though we have used some pseudocode for the sake of simplicity.

```
REPORT ZP_APP.

" Some data definitions
" Maybe a selection screen

PERFORM main.

FORM main.

  " Some data definitions
  " Some code to get PS data, possibly from a screen or CSV file

  DATA(lo_ps) = NEW zcl_ps( ).

  lo_ps->create_project( gs_proj ).

  LOOP AT gt_wbs ASSIGNING FIELD-SYMBOL(<ls_wbs>).
    lo_ps->create_wbs( <ls_wbs> ).
  ENDLOOP.

  LOOP AT gt_activity ASSIGNING FIELD-SYMBOL(<ls_activity>).
    lo_ps->create_activity( <ls_activity> ).
  ENDLOOP.

  " Some code to display results to user

ENDFORM.
```

Listing 9.2 Sample Program Using the SAP PS Class

At this point, we have implemented the MVC (model–view–controller) logic (as described in Chapter 1), where `CL_PS` is the model class, `P_APP` is the controller, and the SAP GUI components are our views.

Out of nowhere, the company decides to buy a third-party project management tool: Microsoft Project (MS Project). Management would like to create MS Project

files using the P_APP program because all the employees involved are already familiar with P_APP.

The engineers at Microsoft were kind enough to provide a custom ABAP class that has the ability to create and maintain MS Project files. The relevant part of structure of the custom class provided by Microsoft would look like Figure 9.2.

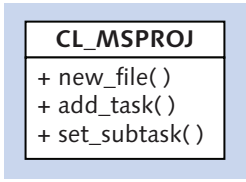


Figure 9.2 MS Project Class

If we inspect the methods from Figure 9.2 one by one, we see that each performs the follow actions:

- NEW_FILE creates a new MS Project file.
- ADD_TASK adds a new task to the file, using some import parameter.
- SET_SUBTASK marks TASK1 as a subtask of TASK2.

Technically, the class looks fine. However, let's not forget that we need to use this class from within P_APP, which was designed to work with CL_PS. Unfortunately, CL_MSPROJ looks nothing like CL_PS, despite the fact that it has similar methods with similar purposes. Check out Figure 9.3 to see the similarities and differences of the classes in UML format.

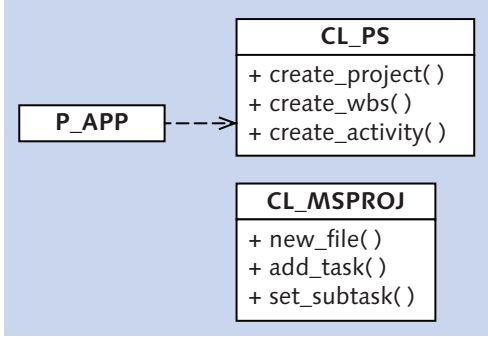


Figure 9.3 MS Project Class Hovering in the Architecture

Using the classical approach, one could solve the problem by adding conditions and conversion rules into P_APP, turning it into a bloated program. Listing 9.3 shows what a sample implementation using the classical approach would look like.

```
REPORT ZP_APP.

" Some data definitions

PARAMETERS:
  p_sapps RADIOBUTTON GROUP rg1,
  p_msprj RADIOBUTTON GROUP rg1.

" Maybe additional selection screen parameters

PERFORM main.

FORM main.

  " Some data definitions
  " Some code to get PS data, possibly from a screen or CSV file

  IF p_sapps EQ abap_true.

    DATA(lo_ps) = NEW zcl_ps( ).

    lo_ps->create_project( gs_proj ).

    LOOP AT gt_wbs ASSIGNING FIELD-SYMBOL(<ls_wbs>).
      lo_ps->create_wbs( <ls_wbs> ).
    ENDLOOP.

    LOOP AT gt_activity ASSIGNING FIELD-SYMBOL(<ls_activity>).
      lo_ps->create_activity( <ls_activity> ).
    ENDLOOP.

  ELSEIF p_msproj EQ abap_true.

    PERFORM convert_sap_ps_to_ms_proj.

    DATA(lo_ms) = NEW zcl_msproj( ).

    lo_ms->new_file( gs_ms_proj ).

    LOOP AT gt_ms_wbs ASSIGNING FIELD-SYMBOL(<ls_ms_wbs>).
      lo_ms->add_task( <ls_ms_wbs> ).
    ENDLOOP.

    LOOP AT gt_ms_activity
```

```
      ASSIGNING FIELD-SYMBOL(<ls_ms_activity>).

      lo_ms->add_task( <ls_ms_activity> ).

      lo_ms->set_subtask(
        iv_parent = <ls_ms_activity>-wbs_code
        iv_child = <ls_ms_activity>-activity_code ).
    ENDLOOP.

  ENDIF.

  " Some code to display results to user

ENDFORM.
```

Listing 9.3 Conversion within the Program

Listing 9.3 is an ugly and inflexible piece of code, that's for sure. What if the project management team decides to purchase additional project management tools such as Primavera and Jira? Should we add more and more conversions and `IF`s into `P_APP`? No, this is a bad idea. If you follow this road, you are making `P_APP` into one of those applications that everyone hates and doesn't dare to touch and that turns out to be a liability rather than an asset to the IT department.

The adapter design pattern suggests a simple but effective solution to the problem. The idea is to keep everything (namely, `P_APP` and `CL_PS`) almost exactly as they are today and to develop an adapter class to add support for `CL_MSPROJ`.

The UML diagram for a program making use of the adapter design pattern would look like Figure 9.4.

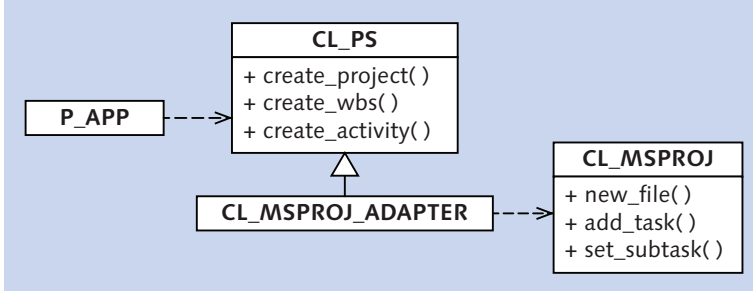


Figure 9.4 MS Project Adapter

When implementing this model, we need to take the following steps:

1. Create the class CL_MSPROJ_ADAPTER as a subclass of CL_PS, ensuring that all methods are derived.
2. Override/redefine all methods in CL_MSPROJ_ADAPTER, ensuring that they call the data conversions and call the appropriate CL_MSPROJ methods.
3. Modify P_APP so that it can work with either CL_PS or any subclass derived from it.

The third step ensures that we have the flexibility for future project platforms, whatever they may be. When a new project platform is added, all we need to do is create a new adapter class and that's it.

Let's get to work! Now that we know what CL_PS and CL_MSPROJ should look like, the code structure of CL_MSPROJ_ADAPTER should not contain any surprises, as can be seen in Listing 9.4.

```
CLASS zcl_msproj_adapter DEFINITION
  INHERITING FROM zcl_ps
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.

    METHODS create_project REDEFINITION.
    METHODS create_wbs REDEFINITION.
    METHODS create_activity REDEFINITION.

  PRIVATE SECTION.
    DATA go_ms TYPE REF TO zcl_msproj.
    " Some type & data definitions + methods

  PROTECTED SECTION.

ENDCLASS.

CLASS zcl_msproj_adapter IMPLEMENTATION.

  METHOD create_project.
    DATA(ls_ms_file) = convert_proj_to_ms_file( is_project ).
    go_ms = NEW zcl_msproj( ).
    go_ms->new_file( ls_ms_file ).
  ENDMETHOD.

  METHOD create_wbs.
    DATA(ls_ms_task) = convert_wbs_to_ms_task( is_wbs ).
    go_ms->add_task( ls_ms_task ).
  ENDMETHOD.
```

```
  METHOD create_activity.
    DATA(ls_ms_task_child) = convert_activity_to_ms_task(
      is_activity ).
    go_ms->add_task( ls_ms_task_child ).
    go_ms->set_subtask(
      iv_parent = is_activity-wbs_code
      iv_child = is_activity-activity_code ).
  ENDMETHOD.

  " Some further methods

ENDMETHOD.

ENDCLASS.
```

Listing 9.4 Adapter Class

Since `CL_MSPROJ_ADAPTER` is derived from `CL_PS`, it can be used virtually everywhere `CL_PS` can be used. Importantly, this includes `P_APP`—which brings us to the last step where we need to modify `P_APP`, as showcased in Listing 9.5.

```
REPORT ZP_APP.

" Some data definitions
PARAMETERS: p_clsname TYPE seoclsname OBLIGATORY.
" Maybe additional selection screen parameters

PERFORM main.

FORM main.

  DATA:
    lo_obj TYPE REF TO object,
    lo_ps TYPE REF TO zcl_ps.

  " Some data definitions
  " Some code to get PS data, possibly from a screen or CSV file

  CREATE OBJECT lo_obj TYPE (p_clsname).
  lo_ps ?= lo_obj.

  lo_ps->create_project( gs_proj ).

  LOOP AT gt_wbs ASSIGNING FIELD-SYMBOL(<ls_wbs>).
    lo_ps->create_wbs( <ls_wbs> ).
  ENDLOOP.

  LOOP AT gt_activity ASSIGNING FIELD-SYMBOL(<ls_activity>).
    lo_ps->create_activity( <ls_activity> ).
```

```
  ENDLOOP.

  " Some code to display results to user

ENDFORM.
```

Listing 9.5 Program Using the Adapter Class

In this semi-pseudocode, we preferred to get the class name from the user to keep things simple. Keeping class names in a Z-table and asking the user for a more human-readable data source is a better idea, but let's keep it simple at this time.

Note

See Appendix B on subclass determination for alternative approaches for getting the class name.

If the user feeds `CL_PS` into `P_CLSNAME`, the following will happen:

1. `LO_PS` will be an instance of `CL_PS`.
2. `LO_PS->CREATE_PROJECT` will call `CL_PS~CREATE_PROJECT` and create a PS project.
3. `LO_PS->CREATE_WBS` will call `CL_PS~CREATE_WBS` and create a WBS element.
4. `LO_PS->CREATE_ACTIVITY` will call `CL_PS~CREATE_ACTIVITY` and create an SAP activity.

If the user feeds `CL_MSPROJ_ADAPTER` into `P_CLSNAME`, the following will happen:

1. `LO_PS` will be an instance of `CL_MSPROJ_ADAPTER`.
2. `LO_PS->CREATE_PROJECT` will call `CL_MSPROJ_ADAPTER~CREATE_PROJECT`, which will create a new MS Project file by calling `CL_MSPROJ~NEW_FILE`.
3. `LO_PS->CREATE_WBS` will call `CL_MSPROJ_ADAPTER~CREATE_WBS`, which will create a task in the MS Project file by calling `CL_MSPROJ~ADD_TASK`.
4. `LO_PS->CREATE_ACTIVITY` will call `CL_MSPROJ_ADAPTER~CREATE_ACTIVITY`, which will do the following:
   - Create a new task in the MS Project file by calling `CL_MSPROJ~ADD_TASK`.
   - Set the new task as a subtask by calling `CL_MSPROJ~SET_SUBTASK`.

As you can see, adapters do more than simple data conversion; they also adapt method differences between classes. In our example, to create an activity, one method call is enough in `CL_PS`. However, `CL_MSPROJ` requires two sequential

method calls to achieve the same functionality. `CL_MSPROJ_ADAPTER` does all the dirty work for us.

9.2 Glue Code

Code within the adapter class is sometimes referred to as *glue code* (also known as *binding code*). As the name suggests, the adapter class shouldn't contain any business logic, calculation, or computation. Instead, it should only do the thing it is meant to do: convert one class interface to another one. Business logic should be left to client applications.

The general principles of object-oriented programming (OOP), described in detail in Appendix C, suggest that each class should have a single responsibility and be really good at it. A class should not attempt to perform further tasks. Instead, one class can be coordinated with other classes for improved functionality.

The adapter design pattern is no exception to this principle. The adapter class is created for a single purpose: interface conversion. Any further functionality violates the basic principle. Mixing business logic with glue code is a typical case of bad design.

9.3 Two-Way Adapters

If two incompatible classes need to interact in both directions, you can consider implementing a two-way adapter. Such an adapter class would wrap class A in such a way that makes it compatible for class B; it would also wrap class B in such a way that makes it compatible for class A. Since ABAP doesn't support multiple inheritance at this time, you would need to store private instances of classes A and B inside the adapter and rewrite appropriate methods to achieve this. Depending on the case, you may prefer having two separate inherited adapter classes.

Let's see this approach in a small example.

At one side, we have application A, which accepts a certain data in tab-delimited format. Listing 9.6 demonstrates how the data provider interface and class would look like.

```
INTERFACE zif_tab_data_provider.
  PUBLIC .

 METHODS get_tab_data
    IMPORTING
      !is_param TYPE zbcs_param
    RETURNING
      VALUE(rt_tab) TYPE zbctt_tab.

ENDINTERFACE.

CLASS zcl_tab_data_default DEFINITION
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.
    INTERFACES zif_tab_data_provider.
  PROTECTED SECTION.
  PRIVATE SECTION.

ENDCLASS.

CLASS zcl_tab_data_default IMPLEMENTATION.

  METHOD zif_tab_data_provider~get_tab_data.
    " Produce tab-delimited data with values in IS_PARAM
    " Return RT_TAB
  ENDMETHOD.

ENDCLASS.
```

Listing 9.6 Sample Interface and Class for Tab-Delimited Data Creation

On the other side, we have application B, which accepts similar data in XML format. Listing 9.7 demonstrates the interface and class for this purpose.

```
INTERFACE zif_xml_data_provider.
  PUBLIC .

 METHODS get_xml_data
    IMPORTING
      !is_param TYPE zbcs_param
    RETURNING
      VALUE(rt_xml) TYPE zbctt_xml.

ENDINTERFACE.

CLASS zcl_xml_data_default DEFINITION
  PUBLIC FINAL CREATE PUBLIC.
```

```
  PUBLIC SECTION.
    INTERFACES zif_xml_data_provider.
  PROTECTED SECTION.
  PRIVATE SECTION.

ENDCLASS.

CLASS zcl_xml_data_default IMPLEMENTATION.

  METHOD zif_xml_data_provider~get_xml_data.
    " Produce XML data with values in IS_PARAM
    " Return RT_XML
  ENDMETHOD.

ENDCLASS.
```

Listing 9.7 Sample Interface and Class for XML Data Creation

Let's summarize what we have at this point:

- Application A will read tab-delimited files over `IF_TAB_DATA_PROVIDER`, which is implemented by `CL_TAB_DATA_PROVIDER`.
- Application B will read XML files over `IF_XML_DATA_PROVIDER`, which is implemented by `CL_XML_DATA_PROVIDER`.

The need for a two-way adapter would pop up when the applications need to use their mutual classes. In other words, application A would need to import XML files, while application B would need to import tab-delimited files.

A possible solution is to implement two distinct adapters. The first adapter would convert `CL_TAB_DATA_PROVIDER` to `CL_XML_DATA_PROVIDER` so application A can import XML files. The second adapter would convert `CL_XML_DATA_PROVIDER` to `CL_TAB_DATA_PROVIDER` so application B can import tab-delimited files.

In the case of the two-way adapter, we would have a single adapter, usable by both applications, which can convert tab-delimited files into XML files and XML files into tab-delimited files. Listing 9.8 demonstrates a two-way adapter class.

```
CLASS zcl_two_way_adapter DEFINITION
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.
    INTERFACES:
      zif_tab_data_provider,
      zif_xml_data_provider.
```

```
    METHODS constructor.

  PROTECTED SECTION.
  PRIVATE SECTION.

    DATA:
      go_tab TYPE REF TO zcl_tab_data_default,
      go_xml TYPE REF TO zcl_xml_data_default.

ENDCLASS.

CLASS zcl_two_way_adapter IMPLEMENTATION.

  METHOD constructor.
    go_tab = NEW #( ).
    go_xml = NEW #( ).
  ENDMETHOD.

  METHOD zif_tab_data_provider~get_tab_data.
    DATA(lt_xml) = go_xml->zif_xml_data_provider~get_xml_data(
      is_param
    ).
    " Convert lt_xml to tab-delimited format into RT_TAB
    " Return RT_TAB
  ENDMETHOD.

  METHOD zif_xml_data_provider~get_xml_data.
    DATA(lt_tab) = go_tab-> zif_tab_data_provider~get_tab_data(
      is_param
    ).
    " Concert lt_tab to XML format into RT_XML
    " Return RT_XML
  ENDMETHOD.

ENDCLASS.
```

Listing 9.8 Sample Two-Way Adapter Class

Now, this class is very versatile. When application A calls `CL_TWO_WAY_ADAPTER->ZIF_TAB_DATA_PROVIDER~GET_TAB_DATA`, it actually gets the XML data—but converted into a tab-delimited format. In a similar logic, when application B makes a call to the method `CL_TWO_WAY_ADAPTER->ZIF_XML_DATA_PROVIDER~GET_XML_DATA`, it actually gets the tab-delimited data—but converted into XML format.

This simplified example gives you the basic idea behind two-way adapters. In a real-world situation, a two-way adapter would probably contain many more methods—but the basic idea will remain the same.

9.4 Legacy Classes

Although adapter seems like a good solution in terms of reusability, it needs to be handled with care—especially if you are using it to link a legacy class to a new system. You will not only be taking over the algorithm, but you will be taking on potential performance problems, bugs, glitches, etc. as well.

Legacy classes are chunks of code that have lived in a system for a long time. These classes typically perform rather complex tasks, and project management usually wants to retain these classes. Therefore, we developers often have to bend new applications in such a way that the applications can reuse the code within legacy classes.

The adapter design pattern looks like a good approach in such a case. Even if the legacy class is incompatible with our new interface, we can easily develop an adapter and make it compatible.

However, just because a legacy class performs a certain task correctly, doesn't mean that it always does so quickly or perfectly. When you are expected to overtake an untouchable chunk of historical code, you will unfortunately be responsible for the misbehavior of that code as well.

If the legacy class contains bugs, users and project managers will find those bugs in your application. "But I only wrote the adapter" will not save you. Or, if the legacy class has performance problems, guess whose application will be blamed for being slow?

Long story short, if you are unsure about a legacy class, your test scenarios need to be more intensive than usual to detect potential problems before going live.

9.5 Summary

Adapter classes work like US-to-European plug adapters, converting the interface of a class to another one and enabling you to add a class with an initially incompatible interface into your architecture. However, you should handle legacy classes with care.

The bridge design pattern is typically used to reduce the number of objects within the runtime scope. If you expect a large number of objects with splittable states, bridge may be the key to reducing the memory footprint dramatically.

10 Bridge

Sometimes, you may feel like you need to create a large number of classes. As a metaphor, imagine that you have three colors (red, green, blue) and three shapes (circle, triangle, square). If you created a class for each combination, you would have nine classes: red circle, red triangle, red square, green circle, green triangle, green square, blue circle, blue triangle, and blue square.

This complexity only gets worse the larger the number of options you have. What if you need to implement an additional color and shape? You would have sixteen classes. Add an additional option to each side, and suddenly you have twenty-five classes. That's a huge load and hard to maintain. For such cases, the bridge design pattern offers a neat solution to keep the class population under control.

In this chapter, we will examine a case study about a messaging framework, where we need to send emails and SMS messages to various third parties. After the case study, we will discuss how the bridge design pattern can decrease the number of classes dramatically.

10.1 Case Study: Messaging Framework

Every company requires messaging, sending emails, and sometimes SMS (text) messages, to their clients, vendors, employees, or intercompany contacts. In this example, we will create a common messaging framework that can be used by all custom applications.

Using this program, individual applications will no longer need to tinker with tables `ADRC`/`ADRP`/etc. to get email addresses or cellphone numbers and won't

contain custom code to send emails or SMS messages either. These applications will simply provide the identifier of the recipient and the message contents, and the framework will take care of the rest.

Listing 10.1 demonstrates what a message type should look like.

```
TYPES:
  BEGIN OF t_msg,
    recipient TYPE char10,
    message TYPE string,
  END OF t_msg.
```

Listing 10.1 Message Type

The variable `recipient` can contain a customer number (KUNNR), vendor number (LIFNR), employee number (PERNR), or an intercompany contact ID (ZINNR), all of which you can store in a Z-table.

Without using the bridge design pattern, you could create this program using a chain of classes. Figure 10.1 showcases what this would look like.

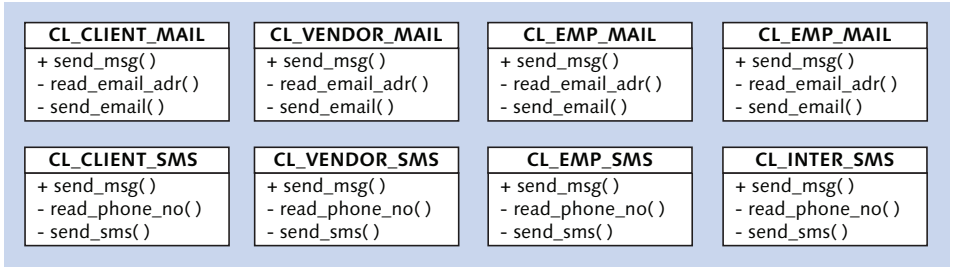


Figure 10.1 Chain of Classes without Bridge Logic

This design can be improved even further by using inheritance; you can move all the classes into a hierarchy, as seen in Figure 10.2.

At the top (most abstract) level, you would have an interface (`IF_SENDER`). At the intermediate level, you would have abstract classes (`CL_MAIL_SENDER`, `CL_SMS_SENDER`) implementing the interface. At the bottom level, you would have your concrete classes, which would perform the necessary operations.

Although this design is better than classical ABAP and will work, there are some troubling aspects. First, it relies heavily on inheritance. In the domain of design patterns, we tend to prefer composition over inheritance (for a more in-depth explanation as to why this is preferred, see Appendix A).

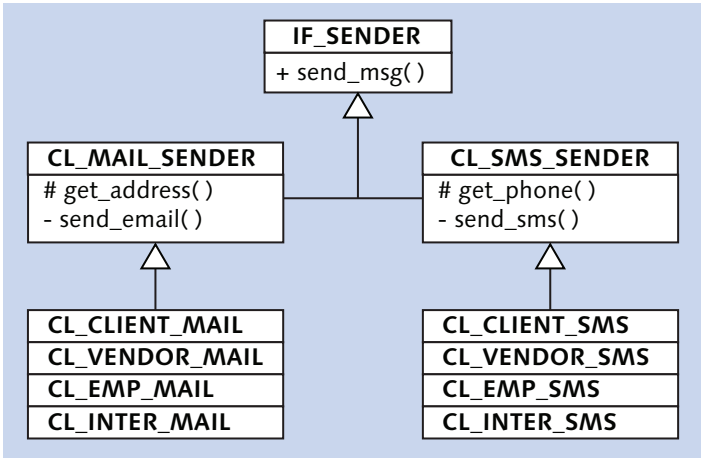


Figure 10.2 Hierarchy of Classes without Bridge Logic

Second, and this is key to deciding to use the bridge design pattern, the number of concrete classes to be created is too high. If you look closely, we have two categories of classes, as follows:

- Contact type (client, vendor, employee, intercompany)
- Message type (email, SMS)

Regardless of the hierarchy you use, you will end up having 8 (4 × 2) concrete classes (the Cartesian product, which equals the product of the two categories). What's worse is that, if you get a brand new message type in the future (such as fax), the number will increase to 12 (4 × 3) concrete classes. If you get a new contact type on top of that, the number of concrete classes you need to create will be 15 (5 × 3). That's a lot of concrete classes and future maintenance to take on. In short, whenever you feel like you need to create a high number of classes, it might just be the time to use the bridge design pattern.

The bridge design pattern allows the two categories to live in their own independent, abstract worlds, and then you combine them over a bridge structure whenever needed. In our example, our categories are the contact type and message type.

Let's start with the contact type and take a look at its structure in Figure 10.3.

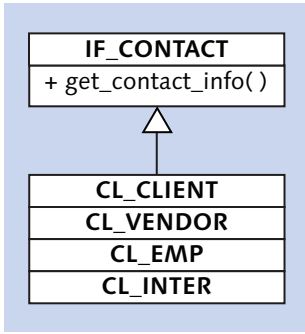


Figure 10.3 Contact Interface with Implementing Classes

For our contact types, we will have one common interface and four concrete classes. Note that we are not dealing with anything related to sending messages here. On this side of the bridge, all that we care about is to getting the contact information for different contact types. Listing 10.2 demonstrates what the interface might look like.

```
INTERFACE zif_contact.
  PUBLIC .

  TYPES:
    BEGIN OF t_info,
      phone TYPE char30,
      email TYPE string,
    END OF t_info.

 METHODS get_contact_info
    IMPORTING
      !iv_recipient TYPE char10
    RETURNING
      VALUE(rs_info) TYPE t_info.

ENDINTERFACE.
```

Listing 10.2 Contact Interface

The concrete classes implementing `IF_CONTACT` are quite simple and look similar to each other. Let's take a look at `CL_CLIENT` in Listing 10.3.

```
CLASS zcl_client DEFINITION
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.
```

```
    INTERFACES zif_contact.

  PRIVATE SECTION.
PROTECTED SECTION.
ENDCLASS.

CLASS zcl_client IMPLEMENTATION.

  METHOD zif_contact~get_contact_info.
    " Read contact data from ADRC, etc and fill RS_INFO.
  ENDMETHOD.

ENDCLASS.
```

Listing 10.3 Client Class

The other three classes (`CL_VENDOR`, `CL_EMP`, and `CL_INTER`) will be similar, so we won't present the pseudocode for their implementations here.

Next, with the diagram in Figure 10.4, we are moving to the other side of the bridge where the different message types reside. We see an abstract message class here (`CL_MESSAGE`), and two concrete message types are derived from it (`CL_EMAIL`, `CL_SMS`). The common part of our concrete classes is centralized in the abstract class, and each concrete class has its own operational code within the method `SEND_MSG`.

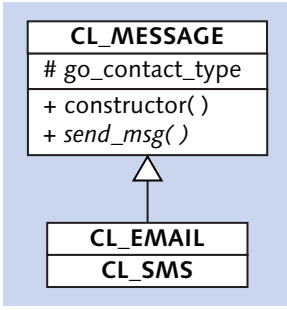


Figure 10.4 Message Interface with Implementing Classes

This diagram will become more clear once we take a look at the abstract class `CL_MESSAGE`, seen in Listing 10.4.

```
CLASS zcl_message DEFINITION
  PUBLIC ABSTRACT CREATE PUBLIC.
```

```
  PUBLIC SECTION.

    METHODS constructor
      IMPORTING
        !io_contact_type TYPE REF TO zif_contact.

    METHODS send_msg ABSTRACT
      IMPORTING
        !iv_recipient TYPE char10
        !iv_message TYPE string.

  PRIVATE SECTION.

  PROTECTED SECTION.
    DATA go_contact_type TYPE REF TO zif_contact.
ENDCLASS.

CLASS zcl_message IMPLEMENTATION.

  METHOD constructor.
    go_contact_type = io_contact_type.
  ENDMETHOD.

ENDCLASS.
```

Listing 10.4 Message Class

What we're doing in Listing 10.4 is importing the contact type in question into the constructor. We will be expecting an implementation of `IF_CONTACT`, which might be an instance of `CL_CLIENT`, `CL_VENDOR`, `CL_EMP`, or `CL_INTER`. If we need to engage different contact types in the future, that's no problem—as long as `IF_CONTACT` is implemented, we are good to go and don't need to modify anything in `CL_MESSAGE` or its subclasses.

Now, let's take a look at `CL_EMAIL`, the class in charge of email messaging, as seen in Listing 10.5.

```
CLASS zcl_email DEFINITION
  INHERITING FROM zcl_message
  PUBLIC ABSTRACT CREATE PUBLIC.

  PUBLIC SECTION.
    METHODS send_msg REDEFINITION.
  PRIVATE SECTION.
  PROTECTED SECTION.
ENDCLASS.
```

```
CLASS zcl_email IMPLEMENTATION.

  METHOD send_msg.
    DATA(lv_email) = go_contact_type->get_contact_info( iv_recipient )-
email.
    " Send IV_MESSAGE to LV_EMAIL using SAP methods
  ENDMETHOD.

ENDCLASS.
```

Listing 10.5 Email Class

CL_SMS will look similar, as demonstrated in Listing 10.6.

```
CLASS zcl_sms DEFINITION
  INHERITING FROM zcl_message
  PUBLIC ABSTRACT CREATE PUBLIC.

  PUBLIC SECTION.
    METHODS send_msg REDEFINITION.
  PRIVATE SECTION.
  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_sms IMPLEMENTATION.

  METHOD send_msg.
    DATA(lv_phone) = go_contact_type->get_contact_info( iv_recipient )-
phone.
    " Send IV_MESSAGE to LV_PHONE using an external Web service
  ENDMETHOD.

ENDCLASS.
```

Listing 10.6 SMS Class

Let's take a step back and examine what we have achieved so far. We made sure that contact type and message type exist without any hard-wire dependency in between. These message type classes are like two distinct parties on opposite sides of a river. We have built a bridge between them by passing an instance of IF_CONTACT to CL_MESSAGE, as can be seen in Figure 10.5.

Please note that CL_MESSAGE could have been an interface as well. You can pick the approach that best suits your requirements.

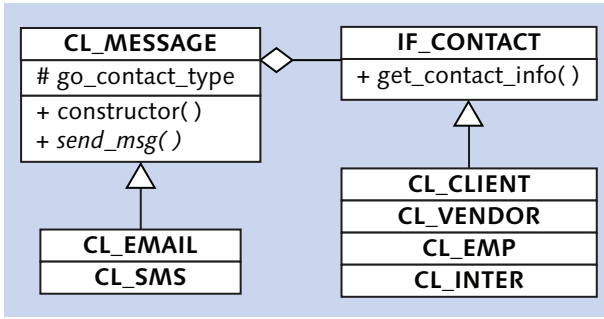


Figure 10.5 Bridge Diagram

Finally, we can look at a sample client application (Listing 10.7) and see how easy it is to send a message now. In this instance, we will be sending an email to a client.

```
DATA lo_contact TYPE REF TO zif_contact.

lo_client = NEW zcl_client( ).
lo_contact ?= lo_client.

lo_email = NEW zcl_email( lo_contact ).
lo_email->send_msg(
  iv_recipient = gv_kunnr
  iv_message = gv_mail_body
).
```

Listing 10.7 Sample Application Sending Email

Before we were using the bridge design pattern, we had to create eight concrete classes. By using the bridge design pattern, we only had to create six concrete classes. The savings only increase as the number of items in each category grows. Adding one more message type and contact type, for example, would require the following number of concrete classes:

- Without bridge design pattern: 15 concrete classes
- With bridge design pattern: 8 concrete classes

The more combinations you have, the more advantageous the bridge design pattern becomes.

Note

The names of the subclasses (`ZCL_CLIENT`, `ZCL_EMAIL`, etc.) are hardcoded in this example. See Appendix B on subclass determination for alternative approaches.

10.2 Advantages

As the number of cases on each side of the bridge increases, the advantage of using the bridge design pattern becomes more and more significant. If you have 10 options in each category, you would need to create 10 × 10 = 100 classes if you don't use the bridge design pattern. If you use the pattern, the number of classes you need to create is only 10 + 10 = 20. This difference is huge and provides a great advantage.

Another advantage is that classes on different sides of the bridge exist independently. In our example, the classes used to send emails and SMSs exist independently from classes for client, vendor, employee, and intercompany communications. Thus, each class can be reused by other applications as well. If we need to send an SMS message for a completely different purpose, we can simply reuse `CL_SMS`. Additionally, if we need the contact information of an employee for a different purpose, we can simply reuse `CL_EMP`.

10.3 Summary

If you feel like you need to create a high number of classes, which is typically the Cartesian product of two class types, the bridge design pattern can dramatically decrease the number of classes. This reduction is achieved by creating a composite structure of distinct class types, where the main class type contains the secondary class type as an attribute.

The bridge design pattern also contributes to the goal of making the classes less interdependent, because classes on different sides of the bridge can operate independently.

The composite design pattern deals with parent–child relationships. While composite may be overkill for simple header–item cases, for anything more complex, composite is human-friendly way to represent and browse nodes. Organizational hierarchies, SAP ERP Project System (PS) structures, and bill of materials (BOM) relations are typical examples.

11 Composite

The composite design pattern is used in cases you want to have a recursive parent–child relationship of objects. Typical examples of such relations within SAP are BOM structures, work breakdown structure (WBS) elements, and HR organizations. Both can be visually imagined as a tree structure. Therefore, we can say that whenever you have a tree structure, it can be programmatically represented using the composite design pattern.

When we talk about trees, we would immediately imagine branches and leaves. Ironically, this is close to the traditional terminology of composite design pattern. An element without further subelements (children) is called a *leaf object*. A parent element with subelements (children) is called a *composite object*. We will see both in action within our case study in Section 11.2.

To be able to use the composite design pattern, a good understanding of tree structures and experience with recursive programming are absolute musts. In the following sections, we will begin with a refresher on recursive programming, before diving into a practical example of the composite design pattern using human resource positions. We will close out the chapter with a discussion of the advantages and disadvantages of this pattern, some suggestions on when to use it, and its most closely related design patterns.

11.1 Recursive Programming: A Refresher

A recursive method is a method that calls itself. This technique is essential for tree structures. When you have a database table storing a tree structure, it is impossible

to know in advance how deep the tree goes. You can't simply assume that the tree goes (for instance) five levels deep because the users may add the sixth level anytime, without notice. To ensure that you access all elements within an unknown level of depth, you should go beyond traditional `LOOP`s and get busy with recursive methods.

To see how recursive methods are necessary for tree structures, let's begin by assuming that we have the following PS structure in hand:

- 100: Technical
  - 101: Production
    - 1011: Assembly
    - 1012: Quality check
  - 1013: Packaging
  - 102: Maintenance
    - 1021: Scheduled maintenance
    - 1022: Defect maintenance
- 200: Managerial
  - 201: Local
  - 202: Global

This logical structure (or tree structure) would be stored in the database, as well as in an internal table, as seen in Table 11.1.

ID	TEXT	PARENT_ID
100	Technical	
101	Production	100
1011	Assembly	101
1012	Quality check	101
1013	Packaging	101
102	Maintenance	100
1021	Scheduled maintenance	102
1022	Defect maintenance	102

Table 11.1 Simplified SAP Storing a PS Tree

ID	TEXT	PARENT_ID
200	Managerial	
201	Local	200
202	Global	200

Table 11.1 Simplified SAP Storing a PS Tree (Cont.)

To continue this example, let's assume that we need to write a method that returns the top-most parent of any given node. Therefore, this method should return the following values:

- `GET_TOP_PARENT( 1012 )` should return 100. The immediate parent of 1012 is 101, but we want the top-most parent.
- `GET_TOP_PARENT( 202 )` should return 200.
- `GET_TOP_PARENT( 100 )` should return 100 because 100 is a top parent itself.

The problem is that, when the method `GET_TOP_PARENT` is called with a node ID, we don't know where exactly it stands within the tree structure. The node may be a top-most parent itself; it may be just one level down or somewhere really deep within the tree. Therefore, you can't simply read the internal table a finite number of times and guarantee that you will pinpoint the top-most parent. Since master data is subject to change, we really can't assume how deep or shallow the tree is. Just when you assume that the tree has 3 levels, the user may add another child, which increases the depth to level 4.

Situations like these are exactly where a recursive method shines. Since the method will be calling itself until the top-most parent is found, we don't need to provide the exact number of runs—the method will simply recursively call itself until the task is done.

Let's see this in action in Listing 11.1 for a better understanding.

```
CLASS zcl_recursive DEFINITION
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.
    METHODS get_top_parent
      IMPORTING !iv_id TYPE zt_node_id
      RETURNING VALUE(rv_top_id) TYPE zt_node_id.
    " Some further method definitions
  PROTECTED SECTION.
```

```
  PRIVATE SECTION.
    DATA gt_node TYPE STANDARD TABLE OF zt_node WITH DEFAULT KEY.
    " Some further data definitions
    " Some further method definitions
ENDCLASS.

CLASS zcl_recursive IMPLEMENTATION.
  METHOD get_top_parent.
    ASSIGN gt_node[ id = iv_id ] TO FIELD-SYMBOL(<ls_node>).
    IF <ls_node>-parent_id IS INITIAL.
      rv_top_id = <ls_node>-id.
    ELSE.
      rv_top_id = get_top_parent( <ls_node>-parent_id ).
    ENDIF.
  ENDMETHOD.
ENDCLASS.
```

Listing 11.1 Simple Recursive Method

There you go! The significant line regarding recursion is marked in bold. Now, let's do a little hypothetical debugging to see exactly how this recursion works.

For the method `GET_TOP_PARENT( 1012 )`, the following actions will take place:

1. Line 4 is found.
2. It has a parent (101), so we call `GET_TOP_PARENT( 101 )`.
3. Line 2 is found.
4. It has a parent (100), so we call `GET_TOP_PARENT( 100 )`.
5. Line 1 is found.
6. It has no parent, so we know that we found the top parent!
7. Return 100.

As you see, we have found the top parent in two recursions. In other words, `GET_TOP_PARENT` has called itself recursively twice. As the tree goes deeper, the number of recursive calls will increase.

Please note that we have left out any kind of defensive programming for the sake of simplicity. In a real-world application, you should take measures against infinite loops and data defects. Nevertheless, this example should be enough for a basic understanding of recursive methods. Let's now move forward to the actual subject of this chapter: the composite design pattern.

11.2 Case Study: HR Positions

A typical example of the composite design pattern would be the storage of HR organizational units. As you know, they are stored in a parent–child relationship, as can be seen in Figure 11.1.

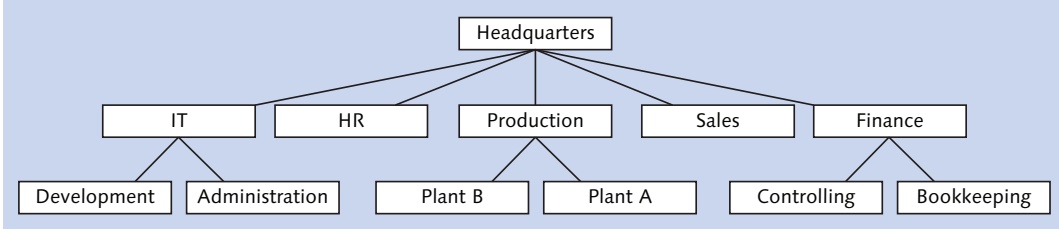


Figure 11.1 Sample Organizational Structure

SAP ERP Human Capital Management (SAP ERP HCM) keeps the definitions of the organizational objects in table `HRP1000`; and relationships, in table `HRP1001`. Our goal in this example will be to store the entire HR organizational structure somewhere in our application. A solution using classical ABAP would look like the code snippet in Listing 11.2.

```
SELECT * INTO TABLE gt_hrp1000 FROM hrp1000.
SELECT * INTO TABLE gt_hrp1001 FROM hrp1001.
```

Listing 11.2 HR Data Selection

Simply selecting everything from the database is workable, yet blunt and inefficient, solution. Why? Imagine that the user clicked a button and made us provide him/her with the subdepartments of the organizational unit "Production." Using classical ABAP, we would need to create a complicated recursive `LOOP` involving the fields OTYPE, OBJID, BEGDA, and ENDDA.

Having to write a complicated `LOOP` is only the first problem. Imagine that, after four years, the company decided to make use of the PLVAR field. You would have to modify many `LOOP`s to take that into account, which is an unnecessary and risky effort, given the possibility for error. Additionally, guest programmers without extensive HR knowledge may have to access our object, again increasing the risk of error.

Instead of working on raw internal tables, the composite design pattern suggests an alternative approach: to abstract the data structure using class hierarchies. Figure 11.2 shows what an abstraction of `HRP1000` and `HRP1001` would look like.

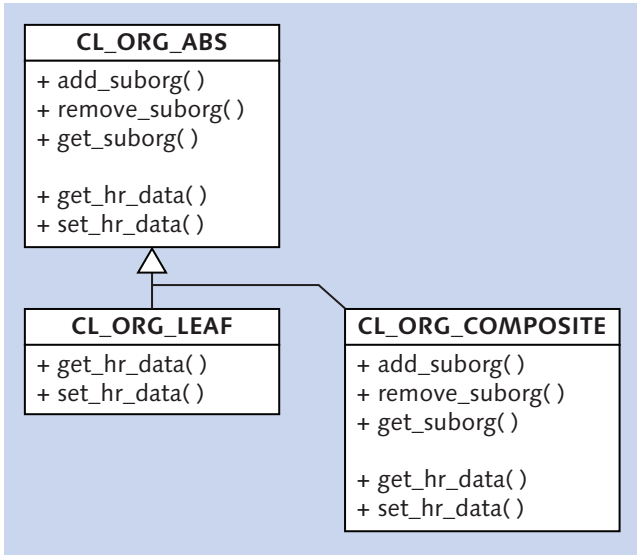


Figure 11.2 Composite Structure

In the diagram in Figure 11.2, you can see the following:

- `CL_ORG_ABS` is an abstract class, representing any record in `HRP1000`. `GET_HR_DATA` and `SET_HR_DATA` are merely sample additional methods; `CL_ORG_ABS` would typically have other methods as well.
- `CL_ORG_LEAF` is a concrete class, representing a record without children. In our sample chart from Figure 11.1, HR, Sales, Development, Administration, Plant A, Plant B, Controlling, and Bookkeeping would be represented using this class.
- `CL_ORG_COMPOSITE` is a concrete class, representing a record with children. In our sample chart from Figure 11.1, Headquarters, IT, Production, and Finance would be represented using this class.

This structure is quite intuitive. Anyone with an understanding of tree structures and recursive programming can understand and access the organizational structure easily, without knowing anything about the data structure of SAP ERP HCM tables.

Additionally, if we need to modify `LOOP` and `READ TABLE` statements related to `HRP1000` and `HRP1001`, the only place we need to modify is `CL_ORG_COMPOSITE`. Other applications using `CL_ORG_*` are not affected at all because the composite logic is abstracted from the underlying data structure.

Let's take a closer look at the classes seen in Figure 11.2, starting with the abstract class in Listing 11.3.

```
CLASS zcl_org_abs DEFINITION
  PUBLIC ABSTRACT CREATE PUBLIC.

  PUBLIC SECTION.

    " Some constants, etc.
    " Some data definitions

    METHODS add_suborg ABSTRACT
      IMPORTING !is_hrp1000 TYPE hrp1000

    METHODS remove_suborg ABSTRACT
      IMPORTING !iv_objid TYPE hrp1000-objid.

    METHODS get_suborg ABSTRACT
      IMPORTING
        !iv_objid TYPE hrp1000-objid
      RETURNING
        VALUE(ro_org) TYPE REF TO zcl_org_abs.

    METHODS get_hr_data FINAL
      RETURNING
        VALUE(rs_hrp1000) TYPE hrp1000.

    METHODS set_hr_data FINAL IMPORTING !is_hrp1000 TYPE hrp1000.

    " Some further methods

  PRIVATE SECTION.
    " Some further methods
  PROTECTED SECTION.
    DATA gs_hrp1000 TYPE hrp1000.
ENDCLASS.

CLASS zcl_org_abs IMPLEMENTATION.

  METHOD get_hr_data.
    rs_hrp1000 = gs_hrp1000.
  ENDMETHOD.

  METHOD set_hr_data.
    gs_hrp1000 = is_hrp1000.
  ENDMETHOD.

ENDCLASS.
```

Listing 11.3 Abstract Class for Organizational Units

Note that the abstract methods don't have any implementations. Those will (and must) be implemented by the subclasses derived from CL_ORG_ABS.

The first implementer of this abstract class is CL_ORG_LEAF, which has no suborganizations (i.e. those organizational units with no children), as demonstrated in Listing 11.4.

```
CLASS zcl_org_leaf DEFINITION
  INHERITING FROM zcl_org_abs
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.
    METHODS add_suborg REDEFINITION.
    METHODS remove_suborg REDEFINITION.
    METHODS get_suborg REDEFINITION.
  PRIVATE SECTION.
  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_org_leaf IMPLEMENTATION.

  METHOD add_suborg.
    " Raise an exception, because we don't have children
  ENDMETHOD.

  METHOD remove_suborg.
    " Raise an exception, because we don't have children
  ENDMETHOD.

  METHOD get_suborg.
    " Raise an exception, because we don't have children
  ENDMETHOD.

ENDCLASS.
```

Listing 11.4 Leaf Class

In the leaf class, all suborganization methods must raise exceptions because this class corresponds to the bottom-level organizational units. We have left out the exception definitions for the sake of simplicity. Moving on to CL_ORG_COMPOSITE, we will implement the class that handles organizations containing suborganizations (i.e. any organizational units that have children), as seen in Listing 11.5.

```
CLASS zcl_org_composite DEFINITION
  INHERITING FROM zcl_org_abs
  PUBLIC FINAL CREATE PUBLIC.
```

```
  PUBLIC SECTION.
    METHODS add_suborg REDEFINITION.
    METHODS remove_suborg REDEFINITION.
    METHODS get_suborg REDEFINITION.
  PRIVATE SECTION.

    TYPES: BEGIN OF t_suborg,
      objid TYPE hrp1000-objid,
      org TYPE REF TO zcl_org_abs,
    END OF t_suborg,

    tt_suborg TYPE STANDARD TABLE OF t_org WITH DEFAULT KEY.

    DATA gt_suborg TYPE tt_suborg.

  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_org_composite IMPLEMENTATION.

  METHOD add_suborg.
    DATA(lo_org) = NEW zcl_org_abs( ).
    lo_org->set_hr_data( is_hrp1000 ).

    APPEND VALUE #(
      objid = is_hrp1000-objid
      org = lo_org ) to gt_suborg.
  ENDMETHOD.

  METHOD remove_suborg.
    DELETE gt_suborg WHERE objid EQ iv_objid.
  ENDMETHOD.

  METHOD get_suborg.
    ro_org = gt_suborg[ objid = iv_objid ]-org.
  ENDMETHOD.

ENDCLASS.
```

Listing 11.5 Composite Class

Next, let's build our data structure using the composite pattern, as seen in the code snippet in Listing 11.6.

```
CLASS zcl_sample DEFINITION
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.

    " Some constants, etc.
```

```
    DATA go_head TYPE REF TO zif_org.

    METHODS constructor.

    " Some more methods, etc.

  PRIVATE SECTION.

    DATA:
      gt_hrp1000 TYPE STANDARD TABLE OF hrp1000 WITH DEFAULT KEY,
      gt_hrp1001 TYPE STANDARD TABLE OF hrp1001 WITH DEFAULT KEY.

    METHODS add_suborgs
      IMPORTING
        !is_parent_objid TYPE hrp1000-objid
      CHANGING
        !co_parent_obj TYPE REF TO zif_org.

    " Some further methods
  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_sample IMPLEMENTATION.

  METHOD constructor.

*   Get raw data from DB
    SELECT * INTO TABLE gt_hrp1000 FROM hrp1000
      WHERE otype EQ @c_otype_org.
    SELECT * INTO TABLE gt_hrp1001 FROM hrp1001
      WHERE otype EQ @c_otype_org.

*   Build composite structure using raw data
    DATA(ls_top_def) = get_top_def( lt_Def ).
    go_head = NEW #( ).
    go_head->set_hr_data( ls_top_def ).

    add_suborgs(
      EXPORTING iv_top_objid = ls_top_Def-objid
      CHANGING co_parent_obj = go_head ).

  ENDMETHOD.

  METHOD add_suborgs.

    DATA lo_org_abs TYPE REF TO zcl_org_abs.

    LOOP AT gt_hrp1001 ASSIGNING FIELD-SYMBOL(<ls_hrp1001>)
      WHERE objid EQ iv_top_objid.
```

```
      ASSIGN gt_hrp1000[ objid = <ls_hrp1001>-objid ]
        TO FIELD-SYMBOL(<ls_hrp1000>).

      IF does_org_have_children( <ls_hrp1000>-objid
        ) EQ abap_true.

        DATA(lo_org_composite) = NEW zcl_org_composite( ).
        lo_org_abs ?= lo_org_composite.
        lo_org_abs->set_hr_data( <ls_hrp1000> ).

        add_suborgs(
          EXPORTING iv_top_objid = <ls_hrp1000>-objid
          CHANGING co_parent_obj = lo_org_abs ).

      ELSE.
        DATA(lo_org_leaf) = NEW zcl_org_leaf( ).
        lo_org_abs ?= lo_org_leaf.
        lo_org_abs->set_hr_data( <ls_hrp1000> ).
      ENDIF.

      co_parent_obj->add_suborg( lo_org_abs ).
    ENDLOOP.

  ENDMETHOD.

  " Some further methods

ENDCLASS.
```

Listing 11.6 Sample Class Building a Composite Structure

Using simple, standard recursive calls, we store the entire HR structure inside `CL_SAMPLE~GO_HEAD`. To simplify the code, we have assumed that the only key we need is OBJID and ignored other important fields such as PLVAR, RELAT, BEGDA, ENDDA, etc.

At this point, we can play around with the code to make it possible to access departments, for example (see Listing 11.7).

```
* Get text of top department
DATA(lv_top_text) = go_head->get_hr_data( )->stext.

* Access IT department
DATA(lo_it) = go_head->get_suborg( c_objid_it ).

* Access development department
DATA(lo_dev) = lo_it->get_suborg( c_objid_dev ).
```

Listing 11.7 Code Snippets Accessing Departments

You can add extra methods to access all subobjects, get data in a special ITAB format, etc. All is possible at this point.

11.3 Advantages

The most significant advantage of the composite design pattern is that the client program can access and extract data from the hierarchy without knowing anything about the underlying data structure. In other words, the code in the client program wouldn't need to specify the keys of the underlying internal tables to access subcomponents. This relationship is coded only once within the bowels of the composite classes, and that's it. Guest programmers can enjoy this convenience, and if your data structure changes at some point (imagine adding a new key field), you only need to change the composite classes. You won't have to run a huge where-used list and modify dozens of different client programs.

Additionally, the client program can deal with primitive objects (leaves) and composite objects uniformly because they share a common interface. The advantage? Using a common interface reduces code written in the client program and prevents code duplication.

11.4 Disadvantages

Experience with recursive programming is virtually a must for the composite design pattern. If your programmers don't have experience in that area, you can expect some trouble.

Another disadvantage is that the leaf classes have to create some empty methods, which do nothing or raise an exception. Leaf and composite classes are derived from the same parent class, which contains methods for subcomponent management such as `ADD_SUB`, `REMOVE_SUB`, `GET_SUB`, etc. These methods are needed in the composite class; however, they are obsolete for the leaf class, since it doesn't have any subcomponents. Generally, due to inheritence, the leaf class must still implement them even if they do nothing but raise an exception.

What might look like a redundant workload is a small price to pay for uniformity.

11.5 When to Use

You might have to decide whether to endure the trouble of implementing the composite design pattern or whether you could get away with merely using internal tables to store your hierarchy. Your choice will depend on the situation, and there are no hard-and-fast rules.

If you have a simple and solid one-level hierarchy, such as header–item relations within SAP ERP Sales and Distribution (SD) and SAP ERP Financials (FI) documents, you can get away with internal tables. In such a case, you could do the following:

- You could create a single internal table with a nested structure. Each header line would contain an internal table of items.
- You could create two internal tables (header and item), ensuring that the item table stores all the keys of the header table as well.
- You could create three internal tables (header, item, and relations). The relation table would store key pairs corresponding to the header/item lines. For new records, the relation in between can be stored with help of substitute GUIDs.

However, the hierarchy is not always that simple. You may need to store BOM relations, HR structures, order–delivery–invoice relations, etc. As the complexity of the hierarchy increases, the composite design pattern gets more and more attractive. When there is a recursive relationship (like a tree), we can definitely favor using the composite pattern.

11.6 Related Patterns

The composite design pattern is in perfect harmony with the factory design pattern, as we saw in our example in Section 11.2. Creation of `GO_HEAD` is a complex progress; therefore, it makes sense to delegate this task to a distinct factory class and then call `CL_FACTORY~GET_ORG_INSTANCE` any time we need to access the organizational structure. See Chapter 4 for more information about the factory design pattern.

Composite and leaf objects often make great flyweights. Therefore, reading and understanding Chapter 15 on the flyweight pattern is recommended. If you have

a tree with many elements, they most probably have partially similar states. In such a case, using flyweight can reduce memory usage and improve your performance.

11.7 Summary

The composite design pattern is the typical approach to tree structures with parent–child relations. The purpose of this pattern is to mask the complicated data relations, providing a friendly interface to browse the tree easily. Recursive programming is a prerequisite though.

Composite might be overkill for single-level, head-item relations. Multilevel tree structures are where it really shines.

A composite design pattern is a good opportunity to apply the factory design pattern. Composite and leaf classes also make great flyweights.

A data access object (DAO) is essentially a strategy design pattern that specializes in multiple data sources. If there are different data sources for the same object type, a data access object isolates the data operation logic from the business logic through a common data interface. This isolation enables you to juggle different data sources without affecting any other layer.

12 Data Access Object

A data access object is not one of the traditional design patterns. It is widely associated with Java Platform, Enterprise Edition (Java EE) applications where an application has to work with multiple data sources such as relational databases, XML files, etc.

In a typical ABAP application, we have a single database system, and the entire program is based on that single database. In fact, having a single centralized database is also one of the core advantages of using an SAP ERP system. However, there might be cases where we have to deal with multiple data sources. For such cases, using a data access object will bring many advantages.

In this chapter, we will look at a real-world example of the data access object design pattern, where potential customers in a third-party customer relationship management (CRM) system need to be merged with real customers in SAP. We will also take a quick look at preventing redundant code and a few related patterns.

12.1 Case Study: Potential Customers

As you probably know already, customers in SAP ERP are maintained using the XD* transaction codes and stored in transparent tables such as `KNA1`, `KNB1`, `KNVV`, etc. In our case study, we will assume that our company has a third-party CRM system, where the data for potential customers is stored. These individuals won't be created as customers in SAP until they are accepted as real customers. The

CRM system provides a web service (SOAP—Simple Object Access Protocol) interface to access potential customers.

Our goal will be to develop an ABAP application that lists actual and potential customers together. If you look closely, we have two distinct data sources, as follows:

- The central SAP database for real customers
- The third-party CRM system for potential customers

Under these circumstances, the first design pattern to think of would be a data access object. If we have multiple data sources representing the same data type, DAO will help us isolate the database layer from the business logic, improving our flexibility and decreasing the dependency on a certain database product.

Implementing DAO is similar to implementing the strategy design pattern (Chapter 25), which we recommend you master before proceeding, if you haven't done so already.

The first step is to create a class to store customer data, as shown in Figure 12.1. This is a simple class with three significant attributes: `GV_POTENTIAL` indicates if the customer is potential or real, `GV_CODE` stores the customer number, and `GV_NAME` stores the customer name.

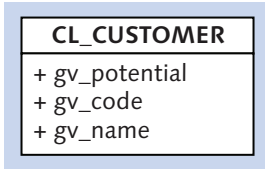


Figure 12.1 Customer Class

A sample implementation of the customer class is demonstrated in Listing 12.1.

```
CLASS zcl_customer DEFINITION
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.
    DATA:
      gv_potential TYPE abap_bool,
      gv_code TYPE kunnr,
      gv_name TYPE name1_gp

  PRIVATE SECTION.
  PROTECTED SECTION.
```

```
ENDCLASS.

CLASS zcl_customer IMPLEMENTATION.

ENDCLASS.
```

Listing 12.1 Object to Store Customer Data

So far, so good. Since the class `CL_CUSTOMER` doesn't have any methods, you could also create a simple structure/type instead of a class. However, in a real-life situation, the class would have methods. Therefore, let's follow tradition and proceed by creating a class.

The second step is to create an interface for data access objects, as shown in Figure 12.2. The interface will have a single method to return a list of customers from its corresponding system.

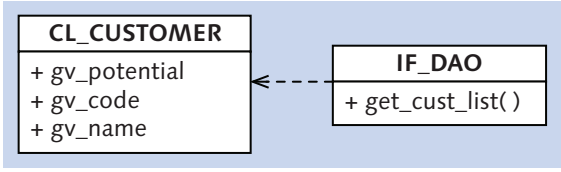


Figure 12.2 Data Access Interface

Sample code corresponding to the data access interface is demonstrated in Listing 12.2.

```
INTERFACE zif_dao.
  PUBLIC .

  TYPES:
    BEGIN OF t_customer,
      code TYPE kunnr,
      obj TYPE REF TO zcl_customer,
    END OF t_customer,

    tt_customer TYPE STANDARD TABLE OF t_customer
      WITH DEFAULT KEY.

 METHODS get_cust_list
    RETURNING
      VALUE(rt_customer) TYPE tt_customer.

ENDINTERFACE.
```

Listing 12.2 Data Access Interface

This interface requires that, for each and every data source, we need to return a customer list. Using this interface, we will create two implementations: one for the SAP database and another for the CRM system's web service, as shown in Figure 12.3.

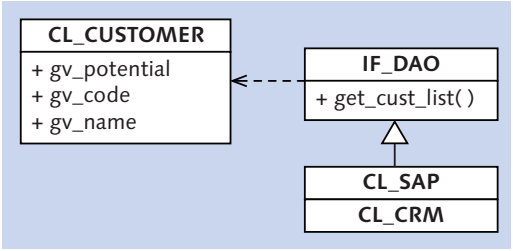


Figure 12.3 Complete Architecture

Moving forward to Listing 12.3, let's see what `CL_SAP` should look like. The class will be reading real customers in SAP's database and return them in the format specified by the interface `IF_DAO`.

```
CLASS zcl_sap DEFINITION
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.
    INTERFACES zif_dao.

  PRIVATE SECTION.
  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_sap IMPLEMENTATION.

  METHOD zif_dao~get_cust_list.

    SELECT kunnr, name1 INTO TABLE @DATA(lt_kna1) FROM kna1.

    LOOP AT lt_kna1 ASSIGNING FIELD-SYMBOL(<ls_kna1>).
      APPEND VALUE #( code = <ls_kna1>-kunnr ) TO rt_customer
        ASSIGNING FIELD-SYMBOL(<ls_customer>).

      <ls_customer>-obj = NEW #( ).
      <ls_customer>-obj->gv_potential = abap_false.
      <ls_customer>-obj->gv_code = <ls_kna1>-kunnr.
      <ls_customer>-obj->gv_name = <ls_kna1>-name1.
    ENDLOOP.
```

```
  ENDMETHOD.

ENDCLASS.
```

Listing 12.3 SAP Data Class

CL_CRM won't look too different from CL_SAP. However, we will leave the complex SOAP code out of our example, writing placeholder comments instead, as the specifics are not necessary for our purposes. See Listing 12.4 for the code for CL_CRM.

```
CLASS zcl_crm DEFINITION
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.
    INTERFACES zif_dao.

  PRIVATE SECTION.
  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_crm IMPLEMENTATION.

  METHOD zif_dao~get_cust_list.

    " Call Web service and put returned data into lt_crm

    LOOP AT lt_crm ASSIGNING FIELD-SYMBOL(<ls_crm>).
      APPEND VALUE #( code = <ls_crm>-cuscode ) TO rt_customer
        ASSIGNING FIELD-SYMBOL(<ls_customer>).

      <ls_customer>-obj = NEW #( ).
      <ls_customer>-obj->gv_potential = abap_true.
      <ls_customer>-obj->gv_code = <ls_crm>-cuscode.
      <ls_customer>-obj->gv_name = <ls_kna1>-cusname.
    ENDLOOP.

  ENDMETHOD.

ENDCLASS.
```

Listing 12.4 CRM Data Class

CUSCODE and CUSNAME are imaginary fields; we assume they will be returned from the CRM system.

At this point, we have two distinct data classes implementing the same interface and, thus, returning the same object type. Next, let's take a look at Listing 12.5 to see what our client application, which lists all the clients together, will look like.

```
DATA:
  lo_dao TYPE REF TO zif_dao,
  lo_obj TYPE REF TO object,
  lt_customer TYPE zif_dao=>tt_customer.

* Build list of DAO classes
  SELECT clsname INTO TABLE @DATA(lt_cls)
    FROM seometarel
    WHERE refclsname eq 'ZIF_DAO'.

* Read customer data from each DAO class
  LOOP AT lt_cls ASSIGNING FIELD-SYMBOL(<ls_cls>).
    CREATE OBJECT lo_obj TYPE (<ls_cls>-clsname).
    lo_dao ?= lo_obj.
    APPEND LINES OF lo_dao->get_cust_list( ) to lt_customer.
  ENDLOOP.

* Display list of customers
  LOOP AT lt_customer ASSIGNING FIELD-SYMBOL(<ls_cust>).
    NEW-LINE.
    WRITE:
      <ls_cust>-obj->gv_code,
      <ls_cust>-obj->gv_name,
      <ls_cust>-obj->gv_potential.
  ENDLOOP.
```

Listing 12.5 Client Application

Listing customers via `WRITE` is not the best idea ever in a real-world situation. Displaying the data in ALV is the least we can do. However, for the sake of simplicity, `WRITE` fulfills our purposes.

Note

In this example, the names of subclasses (`ZCL_CRM`, `ZCL_SAP`) are determined by reading the class relations in the table `SEOMETAREL`. See Appendix B on subclass determination for alternative approaches.

12.2 Redundant Code Prevention

In this chapter, we have assumed that we will conduct `READ` operations using the DAO interface. In a SCRUD (select/create/read/update/delete) situation, the value of using DAO would be far greater because low-level persistence operations would be stored in DAO classes and won't be mixed with the business rules at all.

In the same way that using `WRITE` for client information is independent from the underlying data technology, more complex business rules would be independent from the underlying data technology, thus preventing redundant and complex code.

12.3 Related Patterns

DAO is, more or less, a data source implementation of the strategy design pattern. As you will see in Chapter 25, strategy is used when you have multiple algorithms sharing the same interface. You create a common interface and then create a class for each algorithm that implements the interface.

The same applies to DAO: If you expect the ABAP code to interact with distinct data sources as distinct algorithms, you can consolidate them under a common interface so they can be used interchangeably. Just like alternative algorithms in the strategy design pattern!

12.4 Summary

A data access object is a specialized variation of strategy design pattern, which is used in cases where the application has to work with multiple data sources. You would typically create a data access interface and create a distinct class for each data source. That way, persistence operations are isolated from the business logic.

The decorator design pattern is about "decorating" an object by modifying its state. A common example is an online pizza store: Your web application has a pizza object with image, ingredients, and total price attributes. As the visitor adds new toppings, you "decorate" the pizza object by running it through the selected topping object, updating as you go.

13 Decorator

Decorator is one of the most commonly used design patterns in SAP and is preferred when the state of a single object needs to be dynamically modified for multiple purposes.

If your core object is a box, you can imagine the decorator classes as the wrapping paper and ribbon to "decorate" it and turn it into a gift. However, you don't know what kind of paper and ribbon you need until the client tells you. Therefore, you keep multiple papers and ribbons ready, and whenever you know which pair is needed, you decorate the box accordingly.

In this chapter, we will use user exits as a real-world case study to see the decorator pattern in action. We will also cover some of the advantages of using decorator, as well as some of the challenges. We will finish up the chapter with a discussion of the most relevant related patterns.

13.1 Case Study: User Exit

User exits are one of the best spots to apply decorator design pattern. Why? Often, a popular user exit gets more and more complicated over the years. New requirements come up, different developers modify the same include, and the code ends up being a mess no one dares to touch. You see nested loops, conditions—it all looks like a bowl of spaghetti.

Another problem is that an error on the part of a developer in one place can affect the rest of the user exit. For example, imagine a junior developer using the `CHECK`

command—this would ensure that the rest of the code after the CHECK command doesn't get executed at all. While it's possible that this is the desired functionality, more than often, it is not.

A typical user exit, without making use of the decorator design pattern, looks like Listing 13.1.

```
FUNCTION SAMPLE_USER_EXIT.
*"----------------------------------------------------------------------
*"*"Local Interface:
*"  CHANGING
*"     REFERENCE(CS_DATA) TYPE  SOME_TYPE
*"----------------------------------------------------------------------

INCLUDE zinclude.

ENDFUNCTION.
```

Listing 13.1 Sample User Exit Function Module

The include program would look like Listing 13.2, after modification by the initial developer.

```
* Make discount if this is a special customer
IF zcl_sd_toolkit=>is_customer_special( cs_data-kunnr )
  EQ abap_true.

  cs_data-wrbtr = cs_data-wrbtr * zcl_sd_toolkit=>get_rate(
    cs_data-kunnr ).

ENDIF.
```

Listing 13.2 Initial User Exit Implementation

So far everything looks clean and understandable. However, things start to get complicated after more developers jump in (see Listing 13.3). As new requirements arise, each developer will add more code to the user exit; possibly making the program longer and harder to maintain.

```
* DEV 1: Make discount if this is a special customer
IF zcl_sd_toolkit=>is_customer_special( cs_data-kunnr )
  EQ abap_true.

  cs_data-wrbtr = cs_data-wrbtr * zcl_sd_toolkit=>get_rate(
    cs_data-kunnr ).

ENDIF.
```

```
* DEV 2: Convert to company code currency
DATA(lv_comp_curr) = zcl_fi_toolkit=>get_comp_curr(
  cs_data-bukrs ).

IF cs_data-waers NE lv_comp_curr.
  zcl_fi_toolkit=>conv_curr(
    EXPORTING
      iv_src = cs_data-waers
      iv_tar = lv_comp_curr
    CHANGING
      cv_amount = cs_data-wrbtr ).
  cs_data_waers = lv_comp_curr.
ENDIF.

* DEV 3: If quantity > safety stock, decrease quantity
" Some complicated code here

* DEV 4: If posting date < start of month, correct it
" Some complicated code here
```

Listing 13.3 Various User Exit Implementations

This is only the beginning, and the file will keep growing over time. Using include files to split functionality is a popular approach; however, the risk remains that an error in an earlier include will affect the functionality of a later include.

Unsurprisingly, the decorator design pattern provides a safer, more flexible, and scalable approach, as seen in Figure 13.1.

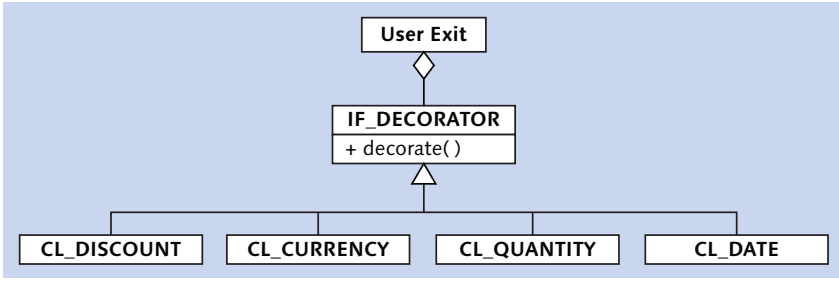


Figure 13.1 Decorator Architecture

In this approach, we have an interface (`IF_DECORATOR`) targeting the user exit. For each requirement, a new class is derived. The user exit itself simply detects the concrete classes and runs `CS_DATA` through them. Listing 13.4 demonstrates what the interface would look like.

```
INTERFACE zif_decorator.
  PUBLIC .

 METHODS decorate
    CHANGING
      !cs_data TYPE some_type.

ENDINTERFACE.
```

Listing 13.4 Decorator Interface

In a real-world situation, we would want to assign at least one exception class to the method `DECORATE` because the subclasses might encounter errors and need to report back. However, as usual, we are keeping things tidy and simple and have therefore left any defensive programming out.

Time to see our concrete decorator classes! Let's start with `CL_DISCOUNT`, which will apply the discount rate and do nothing else (see Listing 13.5). A common best practice for the decorator design pattern (and for object-oriented programming [OOP] in general), one code unit should do a single thing and nothing more.

```
CLASS zcl_discount DEFINITION
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.
    INTERFACES zif_decorator.

  PRIVATE SECTION.
PROTECTED SECTION.
ENDCLASS.

CLASS zcl_discount IMPLEMENTATION.

  METHOD zif_decorator~decorate.
    CHECK zcl_sd_toolkit=>is_customer_special( cs_data-kunnr )
      EQ abap_true.

    cs_data-wrbtr = cs_data-wrbtr * zcl_sd_toolkit=>get_rate(
      cs_data-kunnr ).
  ENDMETHOD.

ENDCLASS.
```

Listing 13.5 Discount User Exit Class

As you can see, we put the first "unit" from the original user exit code into the class but with one major difference: We are able to use the `CHECK` command now

because the command won't stop other classes from being called. This wasn't the case in the classical approach, thus giving us more freedom when using the decorator design pattern.

The other three classes have the exact same logic, so we will simply consolidate their code in Listing 13.6.

```
CLASS zcl_currency DEFINITION
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.
    INTERFACES zif_decorator.

  PRIVATE SECTION.
PROTECTED SECTION.
ENDCLASS.

CLASS zcl_currency IMPLEMENTATION.

  METHOD zif_decorator~decorate.
    DATA(lv_comp_curr) = zcl_fi_toolkit=>get_comp_curr(
      cs_data-bukrs ).

    CHECK cs_data-waers NE lv_comp_curr.

    zcl_fi_toolkit=>conv_curr(
      EXPORTING
        iv_src = cs_data-waers
        iv_tar = lv_comp_curr
      CHANGING
        cv_amount = cs_data-wrbtr ).

    cs_data_waers = lv_comp_curr.
ENDMETHOD.

ENDCLASS.

CLASS zcl_quantity DEFINITION
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.
    INTERFACES zif_decorator.

  PRIVATE SECTION.
PROTECTED SECTION.
ENDCLASS.

CLASS zcl_quantity IMPLEMENTATION.
```

```
  METHOD zif_decorator~decorate.
    " Some complex code here
  ENDMETHOD.

ENDCLASS.

CLASS zcl_date DEFINITION
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.
    INTERFACES zif_decorator.

  PRIVATE SECTION.
PROTECTED SECTION.
ENDCLASS.

CLASS zcl_date IMPLEMENTATION.

  METHOD zif_decorator~decorate.
    " Some complex code here
  ENDMETHOD.

ENDCLASS.
```

Listing 13.6 Further User Exit Classes

We have now successfully split the functionality of the "spaghetti"-like include file into four distinct classes. We're not quite done, however, as we still need to deal with the include itself. Take a look at Listing 13.7 to see how we'll handle this.

```
DATA:
  lo_obj TYPE REF TO object,
  lo_decorator TYPE REF TO zif_decorator,
  lt_clsname TYPE STANDARD TABLE OF seometarel-clsname.

* Detect classes implementing our interface
SELECT clsname INTO TABLE lt_clsname
  FROM seometarel
  WHERE refclsname eq 'ZIF_DECORATOR'.

* Loop through classes, letting each class modify the data
LOOP AT lt_clsname ASSIGNING FIELD-SYMBOL(<lv_clsname>).
  CREATE OBJECT lo_obj TYPE (<lv_clsname>).
  lo_decorator ?= lo_obj.
  lo_decorator->decorate( CHANGING cs_data = cs_data ).
ENDLOOP.
```

Listing 13.7 User Exit Include File Using Decorators

That's all there is to it. As you know already, the table SEOMETAREL holds the link between interfaces and classes. When we run a query against it, we find all classes implementing IF_DECORATOR (i.e. our concrete classes).

In our next step, we run through our concrete classes and let them modify CS_DATA. During the loop, the following happens:

1. In the first run, <LV_CLSNAME> would be ZCL_CURRENCY. Therefore, LO_DECORATOR would be an instance of ZCL_CURRENCY, and DECORATE would execute ZCL_CURRENCY~DECORATE.
2. In the second run, <LV_CLSNAME> would be ZCL_DATE. Therefore, LO_DECORATOR would be an instance of ZCL_DATE, and DECORATE would execute ZCL_DATE~DECORATE.
3. In the third run, <LV_CLSNAME> would be ZCL_DISCOUNT. Therefore, LO_DECORATOR would be an instance of ZCL_DISCOUNT, and DECORATE would execute ZCL_DISCOUNT~DECORATE.
4. In the fourth run, <LV_CLSNAME> would be ZCL_QUANTITY. Therefore, LO_DECORATOR would be an instance of ZCL_QUANTITY, and DECORATE would execute ZCL_QUANTITY~DECORATE.

Note

The names of subclasses (ZCL_DISCOUNT, ZCL_CURRENCY, etc.) are determined by reading the database table SEOMETAREL in this example. See Appendix B on subclass determination for alternative approaches. You might particularly need an alternative approach if the execution order of decorators is important. When you read SEOMETAREL, the order is determined by the alphabetical order of records in the database table SEOMETAREL. If you have to run through your decorators in a special order, you would need to have your own Z-table containing class names and execution order and read your Z-tables instead of SEOMETAREL.

13.2 Advantages and Challenges

The decorator design pattern gives us the flexibility to add new decorator classes in the future. Additionally, you can have multiple developers working on distinct decorators at the same time, which is a great asset when you have a competitive deadline.

Another advantage is the ability to determine the decorators to be called dynamically during runtime. An error in one decorator also won't affect the rest of the decorators in any way, as long as the error handling mechanism of the model class is solid.

The biggest challenge you'll face when using the decorator pattern is deciding whether or not to create a new decorator class. Sometimes, it is clear that you need a new decorator. However, sometimes, you'll be tempted to add some conditional code into an existing decorator to modify its behavior—resulting in a single decorator class acting as two decorators, depending on the condition.

More an art than a science, the best advice is to make sure that you don't go against with the very idea of using this design pattern—which is to distribute distinct decoration tasks to distinct classes. If you feel that a decorator class is fulfilling two tasks, then it might be time to define a new class and split the tasks.

13.3 Related Patterns

If you have a lot of decorators with similar behavior, you might consider using the template method design pattern to cover the code common to similar decorators. Otherwise, you might have a large number of similar classes, which will be both confusing and frustrating to understand and maintain for your successor, or yourself in the future.

If you need to share variables among decorators, you can consider using the property container design pattern. Chapter 16 on the property container design pattern contains a similar example to this chapter that demonstrates perfectly how to apply a property container to decorators.

If you need to do more than simply sharing variables and need a specific execution order or to apply business logic, calling your decorators within the scope of a mediator design pattern (Chapter 20) is advisable.

13.4 Summary

Decorator is useful in cases where the state of a single object needs to be modified for multiple purposes. Each modification is isolated in a distinct class; therefore,

they don't interfere with each other and can be accessed dynamically and independently.

The ease of adding new classes and the ability to have separate developers work on them are some of the significant perks of this pattern.

Similar decorators can be subclassed under a common template class, and variables can be shared via a property container class.

The goal of façade is to simplify the lives of developers by providing a simple interface for a complex operation. There are cases where you need to access multiple classes for a commonly required functionality. Instead of forcing all developers to understand the complex details of your design, you can provide them a shallow method on the surface that deals with all the details in the subsystem.

14 Façade

You have probably already used the façade design pattern at some point without knowing that it was called "façade." Its main purpose is to take a complicated system and provide an easy-to-understand method to do all the dirty work—making the lives of other programmers easier.

Although they don't belong to the object-oriented world, Business Application Programming Interfaces (BAPIs) are semantically perfect examples of the façade approach. For example, without batch input technology, modifying an existing sales document is a complex task. If you had to do it yourself programmatically, you would have to call a handful of functions in a certain order, maybe even modify dozens of tables, and you still wouldn't be sure that you didn't break anything in SAP's standard logic. Instead, you can use the BAPI `BAPI_SALESORDER_CHANGE`—basically a façade function—which provides a nice, simple interface and does all the dirty work.

We will start with the case study of a bonus calculation framework, where multiple classes need to operate together to do a complex calculation. The façade design pattern will save the day and make the calculation process easy to use for all developers. We will follow with the discussion on when to use this pattern, and what other patterns are related to it.

14.1 Case Study: Bonus Calculation

In this example, we will be dealing with a bonus calculation application, which has one simple purpose: calculating annual bonus values for each client based on sales values.

The application has four major components: a GUI, a class to calculate the annual sales values for each client, a class to detect blocked customers, and a class to calculate bonus earned by each client.

Whenever the user hits the "calculate" button in our application, it will take the steps sketched in Figure 14.1. The code will read current sales values, filter out the blocked clients, and calculate the earned bonus value of each client.

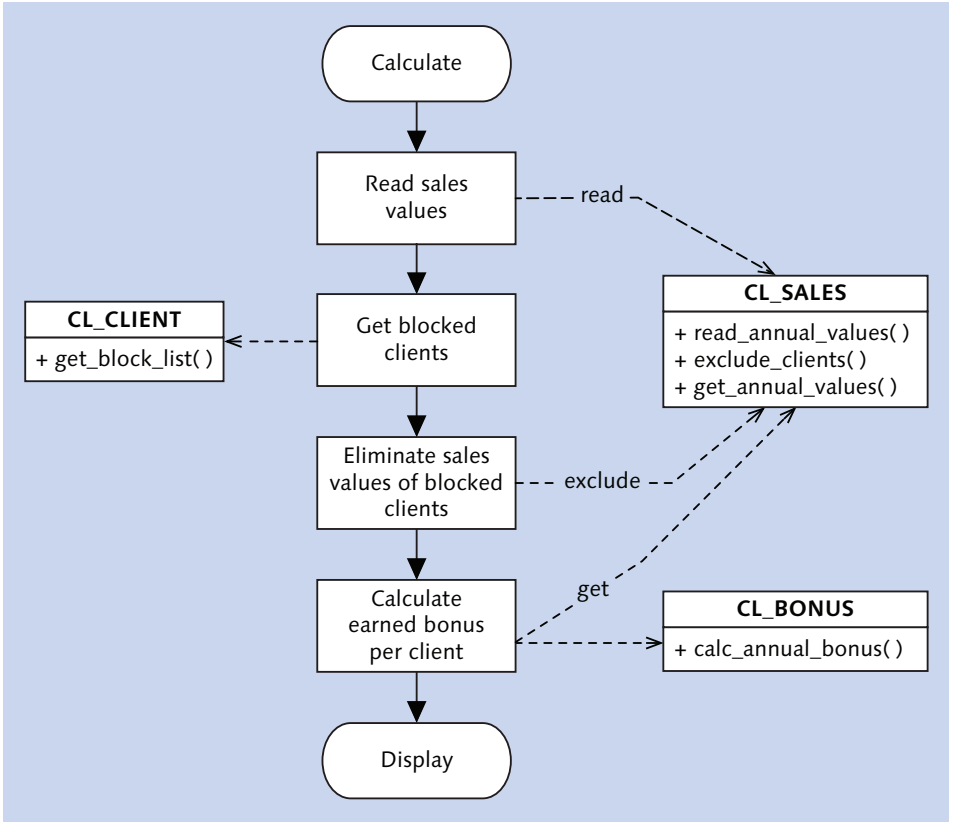


Figure 14.1 Bonus Calculation Flowchart

Our code for the GUI application—in MVC (model–view–controller) terms, the "controller"—would roughly look like Listing 14.1.

```
DATA(lo_sales) = NEW zcl_sales( ).
lo_sales->read_annual_values( ).

DATA(lt_blocked_client) = NEW zcl_client( )->get_block_list( ).
lo_sales->exclude_clients( lt_blocked_client ).

DATA(lt_annual) = lo_sales->get_annual_values( ).
DATA(lt_bonus) = NEW zcl_bonus( )->calc_annual_bonus( lt_annual ).

display_alv( lt_bonus ).
```

Listing 14.1 GUI Application Code Snippet

In Listing 14.1, we have different classes doing different things, and the client application (the GUI) binds together the different classes to achieve the desired functionality.

What if you were told you that a different application needed the annual bonus values as well? For example, an RFC function (remote function call) that needs to send the values to a foreign system, a key performance indicator (KPI) report calculating forecast bonus values, or simply an alternative GUI in SAPUI5/Web Dynpro ABAP. What would you do? There seem to be two solutions here: to copy and paste your code to the new application or to use the façade design pattern.

You can probably see why it is a bad idea to copy and paste the code in Listing 14.1 to the new application: If any of the methods change, the corresponding where-used list gets more complicated than necessary. Another unpleasant consequence is that, if you have to call an additional method in between, you would need to modify multiple spots (instead of one) and could easily miss one.

The better solution is to take advantage of the façade pattern. Using the façade pattern, we will encapsulate the complex method call sequence into a single method. Any application that requires the annual bonus values can simply call the façade method and let it do the dirty work—without having to know anything about the underlying methods or flow logic. Figure 14.2 showcases the architecture for this method.

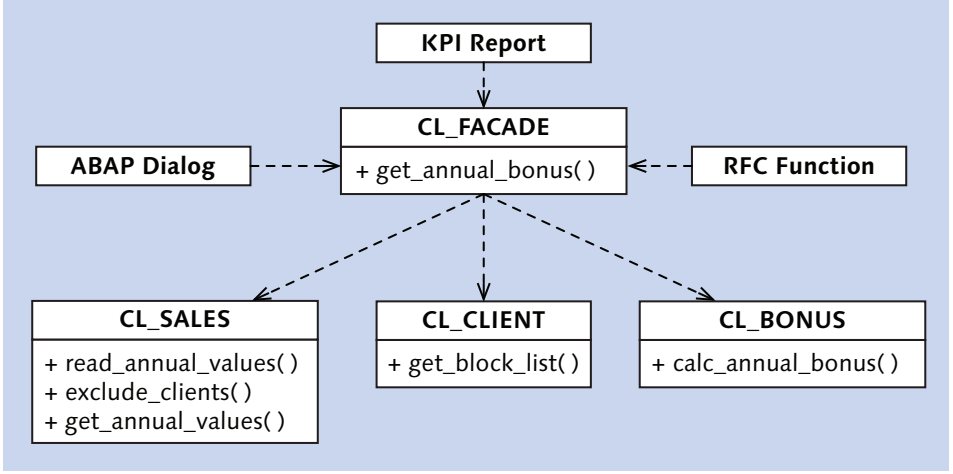


Figure 14.2 Multiple Façade Clients

In the Unified Modeling Language (UML) diagram in Figure 14.2, all three clients will share the same single line of code (taking advantage of the façade method), as follows:

```
DATA(lt_bonus) = NEW zcl_facade( )->get_annual_bonus( ).
```

Please note that you don't actually need to define a new class to contain a façade method. In our example, we could have easily put the method `GET_ANNUAL_BONUS` into `CL_BONUS`. For the sake of our example, we have defined a new class to make the diagram easier to understand.

14.2 When and Where to Use

Whenever you have a particular functionality that requires deep knowledge of your complex class hierarchy, don't hesitate to provide a façade method, which will save you time. Also, other programmers will be grateful because a façade method will greatly reduce the learning curve for your successors.

If you encounter a poorly designed class hierarchy and spend a lot of time trying to understand its structure, you might also consider writing a façade class to make the system easier to use for all (current and future) developers.

The following steps can be taken to decide where to put a façade method:

- If the façade logic is complicated and requires multiple private methods/definitions to complete its task, we would define a new class and put the method there. A bloated class is one of the least liked things in the object-oriented world, and you should avoid them as much as possible. If you have a complicated façade method, instead of entering a host and making it grow in size, having a separate class for the purpose is more appropriate in terms of object-oriented principles.
- If the façade logic fits into a single method and we have a corresponding class of the subject in question, that's where we would put the façade method—probably as a static method. In the example in Section 14.1, `CL_BONUS` calculates bonus values; logically, `GET_ANNUAL_BONUS` can be placed there without creating problems.
- If we can't find a host class for the simple façade method, instead of making an awkward placement, we would put it inside a toolkit class. Typically, every system has toolkit classes where miscellaneous methods are placed. If we had a class like `CL_SD_UTILITIES`, that could host the façade method.

Note

Don't be tempted to fill the entire system with façade methods though. It only makes sense when a certain functionality (method) would be called several times from various points. Façade methods have a maintenance cost: Whenever you change something in your class hierarchy, you might have to modify involved façade methods as well. However, modifying one façade method is far better than drowning in a composite where-used list of twenty-five foreign objects.

Another point to consider is that the façade method should not be the only access point for the application's subsystem. Otherwise, the flexibility would be greatly reduced.

If you have designed the system out of independent classes, other developers can use those classes for different purposes in the future. We have used the analogy of Lego blocks before—each piece can be used for something else as well. If you let them use your pieces independently as well, they will have more building blocks to build great applications too.

The façade design pattern is an easy-to-use method used to fulfill common development requirements, but shouldn't be considered an all-knowing guru object or a gatekeeper.

14.3 Related Patterns

If your façade is an object, it is often represented as a singleton because it is likely that only one façade object is required. If your façade is a method, it is often represented as a static method due for similar reasons.

If you are only using the façade class to wrap the underlying object, you may consider using the proxy (Chapter 17) or adapter (Chapter 9) pattern instead. Although the difference would be minor, one or the other may more accurately reflect your intent. The purpose of façade is almost always to build an easy access point for something more complicated, which is not the core purpose of proxy or adapter.

14.4 Summary

The goal of façade design pattern is to provide a simplified access point to a complex subsystem of classes. The application of façade reduces code duplication, development time, and new developer onboarding time. However, future maintenance cost should of façade methods also be considered.

Depending on the complexity of the underlying functionality, façade can be designed as a new singleton class, a static method of an existing model class, or a static method of a toolkit class.

Flyweight, a performance-oriented design pattern, can decrease the memory footprint of your applications dramatically. If the state of your class has an immutable part, you can represent it as a shared static object. If you have a large number of objects, having partially shared states will decrease their memory consumption.

15 Flyweight

If we consider singleton to be the most basic way of sharing objects, multiton would be the next step where a static array of objects for each object key is kept and can be reused as many times as needed. To take it a step further, flyweight can be seen as a more advanced version of multiton. Therefore, if you don't feel comfortable with singleton and multiton yet, we highly recommend taking another look at Chapter 8 and Chapter 6, respectively.

At a basic level, if you imagine a multiton design pattern with the ability to change some properties of the shared object, you more or less arrive to the flyweight design pattern.

Our journey to flyweight will start with a case study, where we will upload an Excel file containing the same material code multiple times. Instead of creating a distinct material object for each line, we will take advantage of the flyweight design pattern, thus reducing the memory footprint of our application. The case study will be followed by a discussion regarding the disadvantages, appropriate use cases, and related patterns of flyweight.

15.1 Case Study: Negative Stock Forecast

In this example, we will create an application that is initiated by uploading an Excel file. The file will contain thousands of lines of raw material requirements sorted by date. We will then need to run a simulation and calculate the dates on which we might face a negative stock situation. The uploaded file will be formatted like Table 15.1.

DATE	MATERIAL	QUANTITY	UoM
03.02.2016	M1	50	PC
03.02.2016	M2	22	PC
03.02.2016	M3	4	PC
04.02.2016	M1	3	PC
04.02.2016	M2	16	PC
04.02.2016	M3	7	PC
05.02.2016	M1	8	PC
05.02.2016	M2	8	PC
05.02.2016	M3	4	PC
…			

Table 15.1 Sample Upload Data

Using object-oriented design, but not making use of design patterns, we might conclude that we need to create an object to represent the stock quantity of each date and material pair, as shown in Figure 15.1.

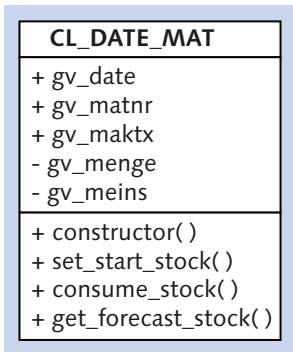


Figure 15.1 Date and Material Class

Using this design, the client application presented in Listing 15.1 would have an array of `CL_DATE_MAT` and `CONSUME_STOCK( )` for each line in the uploaded file.

```
TYPES:
  BEGIN OF t_obj,
    index TYPE i,
    obj TYPE REF TO zcl_date_mat,
  END OF t_obj,
```

```
  tt_obj TYPE STANDARD TABLE OF t_obj WITH DEFAULT KEY.

DATA gt_obj TYPE t_obj.

PERFORM upload_file.

LOOP AT gt_file ASSIGNING FIELD-SYMBOL(<ls_file>).
  APPEND VALUE #( index = sy-tabix ) TO gt_obj
    ASSIGNING FIELD-SYMBOL(<ls_obj>).

  <ls_obj>-obj = NEW #(
    iv_matnr = <ls_file>-matnr
    iv_date = <ls_file>-date ).

  " If this is the first instance, set start stock from MARD
  " Else, set start stock from previous day
  <ls_obj>-obj->consume_stock( <ls_file>-menge ).
ENDLOOP.
```

Listing 15.1 Initial Client Application

Although we could improve code quality by using hashed keys and including some defensive programming, this code would work but is not ideal memory-wise. Imagine that the text file contains 3,000 lines (1,000 days × 3 materials). Under those circumstances, we would have an array of 3,000 objects in which the following was true:

- `GV_MATNR`, `GV_MAKTX` and `GV_MEINS` of M1 are duplicated 1,000 times.
- `GV_MATNR`, `GV_MAKTX` and `GV_MEINS` of M2 are duplicated 1,000 times.
- `GV_MATNR`, `GV_MAKTX` and `GV_MEINS` of M3 are duplicated 1,000 times.

Even this simple example has a lot of avoidable memory consumption. How do we avoid this memory consumption?

By using the flyweight design pattern, of course! We start off by creating an interface for our core object, material, as shown in Figure 15.2, a simple interface, containing only getter/setter methods.

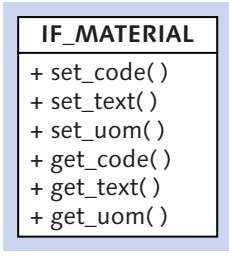


Figure 15.2 Material Interface

The code of the material interface is demonstrated in Listing 15.2.

```
INTERFACE zif_material.
  PUBLIC .

  METHODS set_code IMPORTING !iv_matnr TYPE matnr.
  METHODS set_text IMPORTING !iv_maktx TYPE maktx.
  METHODS set_uom IMPORTING !iv_meins TYPE meins.
  METHODS get_code RETURNING VALUE(rv_matnr) TYPE matnr.
  METHODS get_text RETURNING VALUE(rv_maktx) TYPE maktx.
  METHODS get_uom RETURNING VALUE(rv_meins) TYPE meins.

ENDINTERFACE.
```

Listing 15.2 Material Interface

Because we have such a simple interface, which won't change from material to material, we could have created a concrete material class directly. However, in a real-world situation, an interface is rarely that simple—it usually contains some operational methods, which might be implemented differently among distinct concrete subclasses. As such, we have created an interface to align more closely with what this would look like in practice.

The next step follows logically: creating (at least one) implementation for the given interface, as shown in Figure 15.3.

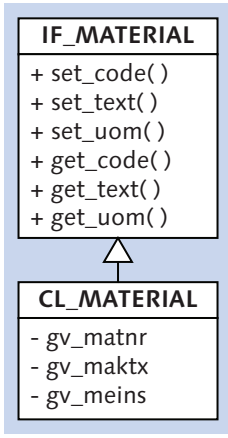


Figure 15.3 Material Class

The code of the material class is demonstrated in Listing 15.3.

```
CLASS zcl_material DEFINITION
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.
    INTERFACES zif_material.

  PRIVATE SECTION.
    DATA:
      gv_matnr TYPE matnr,
      gv_maktx TYPE maktx,
      gv_meins TYPE meins.
  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_material IMPLEMENTATION.

  METHOD zif_material~set_code.
    gv_matnr = iv_matnr.
  ENDMETHOD.

  METHOD zif_material~set_text.
    gv_maktx = iv_maktx.
  ENDMETHOD.

  METHOD zif_material~set_uom.
    gv_meins = iv_meins.
  ENDMETHOD.

  METHOD zif_material~get_code.
    rv_matnr = gv_matnr.
  ENDMETHOD.

  METHOD zif_material~get_text.
    rv_maktx = gv_maktx.
  ENDMETHOD.

  METHOD zif_material~get_uom.
    rv_meins = gv_meins.
  ENDMETHOD.

ENDCLASS.
```

Listing 15.3 Material Class

All that we do in this (oversimplified) class is to set and get private values. As we stated earlier, a typical class in a real-world situation would have much more to it, but the purpose of our example is to understand the design pattern—not to produce fancy code.

Our next step will be an implementation of the multiton design pattern. For an in-depth look at the logic of multiton, see Chapter 6; for now, we will simply demonstrate the class. Figure 15.4 demonstrates the structure of the material factory class (refer to Chapter 4 for more information).

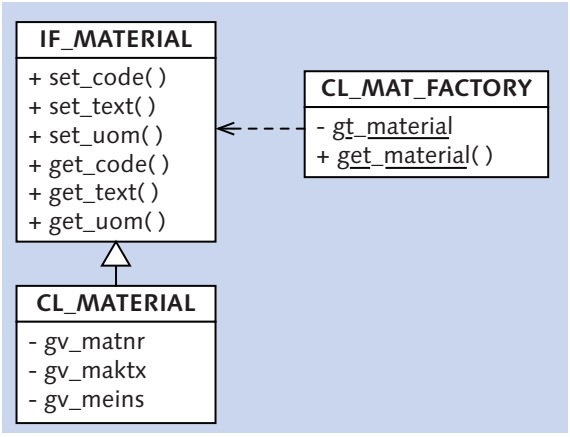


Figure 15.4 Material Factory

The code of the material factory class is demonstrated in Listing 15.4.

```
CLASS zcl_mat_factory DEFINITION
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.
    CLASS-METHODS get_material
      IMPORTING
        !iv_matnr TYPE matnr
      RETURNING
        VALUE(ro_mat) TYPE REF TO zif_material.

  PRIVATE SECTION.
    TYPES:
      BEGIN OF t_mat,
        matnr TYPE matnr,
        obj TYPE REF TO zif_material,
      END OF t_mat,

      tt_mat TYPE HASHED TABLE OF t_mat
        WITH UNIQUE KEY primary_key COMPONENTS matnr.

    CLASS-DATA gt_mat TYPE tt_mat.
```

```
  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_mat_factory IMPLEMENTATION.

  METHOD get_material.

    ASSIGN gt_mat[ KEY primary_key COMPONENTS matnr = iv_matnr ]
      TO FIELD-SYMBOL(<ls_mat>).

    IF sy-subrc NE 0.

      SELECT SINGLE mara~matnr, mara~meins, makt~maktx
        INTO @DATA(ls_db)
        FROM mara
        LEFT JOIN makt ON makt~matnr EQ mara~matnr
          AND makt~spras EQ @sy-langu
        WHERE mara~matnr EQ @iv_matnr.

      DATA(ls_mat) = VALUE t_mat( matnr = iv_matnr ).
      DATA(lo_mat) = NEW zcl_material( ).
      ls_mat-obj ?= lo_mat.
      ls_mat-obj->set_code( ls_db-matnr ).
      ls_mat-obj->set_text( ls_db-maktx ).
      ls_mat-obj->set_uom( ls_db-meins ).
      INSERT ls_mat INTO TABLE gt_mat ASSIGNING <ls_mat>.
    ENDIF.

    ro_mat = <ls_mat>-obj.

  ENDMETHOD.

ENDCLASS.
```

Listing 15.4 Material Factory

Note

The name of the subclass (`ZCL_MATERIAL`) is hardcoded in this example. See Appendix B on subclass determination for alternative approaches.

By implementing the multiton logic, we have ensured that we get a single instance for each material—therefore, nothing is duplicated. Even if every material is repeated 1,000 times in the file, the memory will still only contain three instances of `MATNR`, `MAKTX`, and `MEINS` (for materials M1, M2, and M3, respectively). As a result, we will store 9 (3 × 3) variables in the memory, which is a lot

less than what we had before. In the traditional terminology, MATNR, MAKTX, and MEINS would be the variables contained in the *intrinsic state* of the material object.

What about date and quantity values? We have not forgotten them! Since date and quantity are changeable, we will evaluate them as extrinsic and handle them in a separate class, as shown in Figure 15.5.

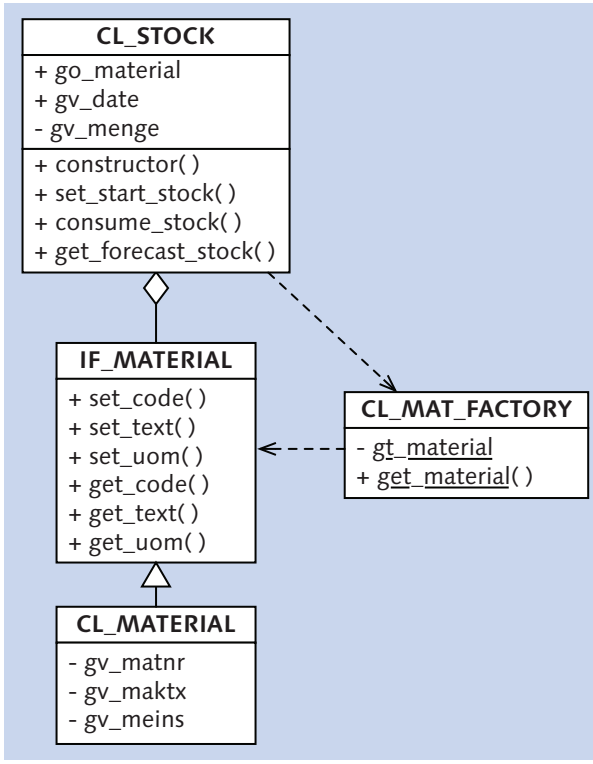


Figure 15.5 Stock Class

What we have in CL_STOCK is very similar to our first raw class CL_DATE_MAT. The difference is that repeated material master data (intrinsic state) is kept in a multiton class (CL_MAT_FACTORY) and is not repeated at all—this saves a lot of memory. Let's take a look at the code of CL_STOCK in Listing 15.5 to get a better understanding of the difference.

```
CLASS zcl_stock DEFINITION
  PUBLIC FINAL CREATE PUBLIC.
```

```
  PUBLIC SECTION.
    DATA:
      go_material TYPE REF TO zif_material READ-ONLY,
      gv_date TYPE dats READ-ONLY.

    METHODS constructor
      IMPORTING
        !iv_matnr TYPE matnr
        !iv_date TYPE dats.

    METHODS set_start_stock IMPORTING !iv_menge TYPE zmenge.
    METHODS consume_stock IMPORTING !iv_menge TYPE zmenge.
    METHODS get_forecast_stock
      RETURNING VALUE(rv_menge) TYPE zmenge.

  PRIVATE SECTION.
    DATA gv_menge TYPE zmenge.
  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_stock IMPLEMENTATION.

  METHOD constructor.
    go_material = zcl_mat_factory=>get_material( iv_matnr ).
    gv_date = iv_date.
  ENDMETHOD.

  METHOD set_start_stock.
    gv_menge = iv_menge.
  ENDMETHOD.

  METHOD consume_stock.
    SUBTRACT iv_menge FROM gv_menge.
  ENDMETHOD.

  METHOD get_forecast_stock.
    rv_menge = gv_menge.
  ENDMETHOD.

ENDCLASS.
```

Listing 15.5 Stock Class

The `CONSTRUCTOR` method is where the magic happens. Because `IF_MATERIAL` instances are reused, we will have only three `IF_MATERIAL` instances in memory—corresponding to M1, M2, and M3 in the file. Even if the file has 3,000 lines, we will have only three instances of `IF_MATERIAL`—a dramatic memory improvement.

Note

As you have seen in our example, each flyweight object has two states. The intrinsic state (material) is the shared object and is instance independent. The extrinsic state (date and stock) is instance dependent. The intrinsic state contains the variables shared among all instances, and the ability to share this state is what makes this pattern efficient. The extrinsic state contains the ad-hoc variables of each instance. Those variables are not shared, and this state distinguishes instances from each other.

Because the date and quantity changes on every line, we can't reduce memory usage much there, so we will still have 3,000 instances of `CL_STOCK`. However, the memory footprint of 3,000 × `CL_STOCK` is much lower than the memory footprint of 3,000 × `CL_DATE_MAT`.

The code of the client application, on the other hand, is the same as our original approach—as presented in Listing 15.6.

```
TYPES:
  BEGIN OF t_obj,
    index TYPE i,
    obj TYPE REF TO zcl_stock,
  END OF t_obj,

  tt_obj TYPE STANDARD TABLE OF t_obj WITH DEFAULT KEY.

DATA gt_obj TYPE t_obj.

PERFORM upload_file.

LOOP AT gt_file ASSIGNING FIELD-SYMBOL(<ls_file>).
  APPEND VALUE #( index = sy-tabix ) TO gt_obj
    ASSIGNING FIELD-SYMBOL(<ls_obj>).

  <ls_obj>-obj = NEW #(
    iv_matnr = <ls_file>-matnr
    iv_date = <ls_file>-date ).
  " If this is the first instance, set start stock from MARD
  " Else, set start stock from previous day
  <ls_obj>-obj->consume_stock( <ls_file>-menge ).
ENDLOOP.
```

Listing 15.6 Client Application after Flyweight

Like many other design patterns, flyweight does its magic under the hood, deep within the private section of its class. The client application using the class containing a flyweight doesn't really need to know that there is a flyweight involved.

That's good news in terms of development—even if you decide to add a flyweight to your class later on, you probably wouldn't need to change the client applications much, as shown in our case study. You can make a before-and-after comparison between Listing 15.1 and Listing 15.6 to see that not much has been changed.

15.2 Disadvantages

Although using the flyweight object is efficient in terms of memory, it comes with a cost: dependency. The extrinsic part of each object is independent. However, the intrinsic parts of objects are totally dependent. When you change a variable within the intrinsic part of any object, you are changing the common memory area, and this change will be inherited/reflected by all other flyweight objects. The intrinsic part doesn't even contain separate variables; it simply contains pointers addressing the same memory area.

In short, the most significant downside of this pattern is that all instances of the flyweight class will be related; therefore, single instances of the class can't behave independently from others. This downside is hardly prohibitive: it might be the very reason you would pick flyweight as your design pattern of choice. If instances need to be completely independent, flyweight is not your answer. If this kind of a dependency works for you, then by all means, go ahead and use the flyweight design pattern, as it will save you a lot of memory.

15.3 When to Use

Using flyweight only makes sense if you foresee a large number of similar (but not completely identical) objects. Flyweight might be overkill if you are going to have one completely identical instance for each key. In such a case, multiton offers a simpler solution.

Deciding to use flyweight is like deciding to use a database index. If your table has only a few entries, you probably don't bother creating an index at all. If you have millions of records though, that's a different story. You might want to use indexes to increase performance.

A similar approach applies to internal tables as well. If you have only a few entries in your internal table, you may not bother using performance improvement

methods, such as hashed/sorted tables, to read data. However, as the size of your tables increase, the necessity of such methods becomes evident.

The same principle applies to the decision to use the flyweight design pattern. If you don't foresee a large number of objects, you may skip the idea of using the flyweight design pattern. However, in cases where you may end up with a lot of similar objects, using this pattern could decrease your memory footprint dramatically.

15.4 Related Patterns

Flyweight objects are usually created using a factory method, and the factory method either is a static method or belongs to a singleton class. This approach ensures that the intrinsic state of the flyweight object is shared across the entire system.

In order to share the intrinsic state of the pattern among all instances, we simply define that part as a static value, which is the very purpose of static data definitions anyway. By marking the intrinsic state as static, we ensure that the system does not create the intrinsic state repeatedly. We ensure that only one instance exists among the entire system.

Where does that lead us? Well, this sounds a lot like the singleton or multiton design patterns, discussed in Chapter 8 and Chapter 6, respectively. As you will remember, those patterns are based on the idea of storing and using a single object instance among the entire system. This approach is a natural fit for the intrinsic state of the flyweight object.

Note

Creating a factory method to return the flyweight object, which will store the intrinsic state as a singleton or multiton object, is generally considered a best practice. Note that we followed this path in our example.

15.5 Summary

Flyweight is an advanced version of multiton, where the intrinsic state of the object is independent and the extrinsic state is dependent. By sharing the intrinsic

state among a large number of flyweight objects, the memory footprint can be dramatically reduced—although at the cost of limiting the freedom of flyweight objects by making them related.

If you don't foresee a very large number of objects, applying flyweight might be overkill. Flyweight objects are typically created using a factory method supporting singleton or multiton logic.

A property container can be imagined as a collection of variables to be tossed around. The same bag is accessed by multiple objects, and each object can typically read, write, or add variables. Containers in SAP Business Workflow serve much the same purpose as the property container design pattern.

16 Property Container

The property container design pattern is a simple yet extremely useful design pattern. Chances are that you have used it in the past without knowing its name.

If you are familiar with SAP Business Workflow, you will find this pattern familiar. In SAP Business Workflow, each workflow has a central container that stores all of the common variables for tasks to use and share. The property container design pattern follows the same logic, and we'll create a central class to store all the common variables for other classes to use and share.

To give another example, the registry of the Windows operating system can also be imagined as a huge, OS-wide property container from which all programs read and write.

Focusing on object-oriented ABAP, we will start by inspecting a real-life scenario where a large enhancement is made to a standard SAP program. This enhancement will call multiple methods of various classes, and variable sharing between the methods will be conducted easily via a property container. The chapter will continue with a discussion of common practices, including static and instance containers and how to preserve the independence of classes and the uniqueness of variables. We will also see how a property container can be used with related patterns.

16.1 Case Study: Enhancing an SAP Program

For this example, in addition to the property contain design pattern, we will also be making use of the decorator design pattern (Chapter 13). Our goal will be to

alter standard SAP code to allow us modify the contents of the internal table XKONP. Instead of viciously editing delicate German code, creating an enhancement point instead is advisable.

For small enhancements, such as setting a break point or writing data into a Z-table, code can directly be written into the enhancement point itself. However, for more advanced operations, you should write the code into a class and have the enhancement point only contain the method call.

If the same enhancement point must contain several operations, our first instinct should be to use the decorator design pattern. The Unified Modeling Language (UML) diagram containing the runtime logic for this enhancement point would look like Figure 16.1.

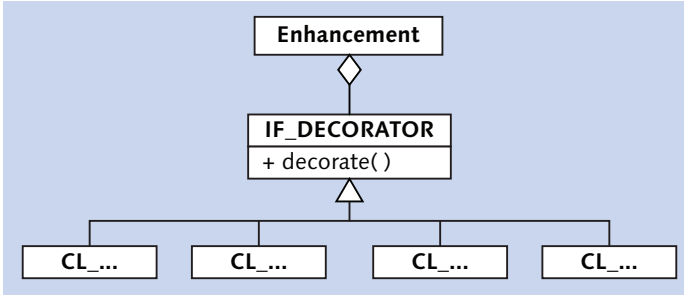


Figure 16.1 Basic Decorator Diagram

Note

If you are not familiar with the decorator design pattern, don't worry. At this time, it is enough to know that we will make sequential method calls from a set of objects. The enhancement will loop among objects and let them decorate the variables of the enhancement point sequentially. If you would like to learn more about the decorator design pattern, see Chapter 13.

Listing 16.1 demonstrates what the code for the enhancement point might look like.

```
TYPES: tt_dec TYPE STANDARD TABLE OF zif_decorator.

DATA:
  lo_obj TYPE REF TO object,
  lo_decorator TYPE REF TO zif_decorator,
  lt_clsname TYPE STANDARD TABLE OF seometarel-clsname,
```

```
  lt_dec TYPE tt_dec.

* Detect classes implementing our interface & build table of classes
SELECT clsname INTO TABLE lt_clsname
  FROM seometarel
  WHERE refclsname eq 'ZIF_DECORATOR'.

LOOP AT lt_clsname ASSIGNING FIELD-SYMBOL(<lv_clsname>).
  CREATE OBJECT lo_obj TYPE (<lv_clsname>).
  lo_decorator ?= lo_obj.
  APPEND lo_decorator TO lt_dec.
ENDLOOP.

* Loop through classes, letting each class modify the data
LOOP AT lt_dec ASSIGNING FIELD-SYMBOL(<lo_dec>).
  <lo_dec>->decorate( CHANGING ct_konp = xkonp[] ).
ENDLOOP.
```

Listing 16.1 Sample Decorator Enhancement Point

In this example, the names of subclasses are determined by reading the database table SEOMETAREL. As you can see, the enhancement modifies the contents of table XKONP.

Note

See Appendix B on subclass determination for alternative approaches.

The decorator interface would look like Listing 16.2.

```
INTERFACE zif_decorator.
  PUBLIC .

TYPES tt_konp TYPE STANDARD TABLE OF konp WITH DEFAULT KEY.

 METHODS decorate
    CHANGING
      !ct_konp TYPE tt_konp.

ENDINTERFACE.
```

Listing 16.2 Decorator Interface

Every class implementing this interface will look like Listing 16.3.

```
CLASS zcl_sample DEFINITION
  PUBLIC FINAL CREATE PUBLIC.
```

```
  PUBLIC SECTION.
    INTERFACES zif_decorator.

  PRIVATE SECTION.
PROTECTED SECTION.
ENDCLASS.

CLASS zcl_sample IMPLEMENTATION.

  METHOD zif_decorator~decorate.
    " Some code to mess with CT_KONP
  ENDMETHOD.

ENDCLASS.
```

Listing 16.3 Sample Decorator Class

You may have one, two, five, or twenty classes like this that need to perform a distinct task related to the internal table `XKONP`. However, our example does not end here. Perhaps these classes need to share some information. Maybe the third class wants to set a (red) flag for all subsequent classes to indicate that those classes shouldn't touch certain records. Maybe one of the classes makes a performance-hungry calculation and wants to share the results with the subsequent classes so that they don't repeat the same process, thus saving runtime.

In classical ABAP, one solution is to use the commands `EXPORT TO MEMORY` and `IMPORT FROM MEMORY`. These commands could work, but using a property container is a more elegant alternative solution because it can help prevent some drawbacks from shared ABAP memory, which include the following:

- If the memory ID is hardcoded, you might have a difficult time discovering where it was exported/imported. If you use a property container, a simple where-used list of the property class will reveal its access points.
- If the source/target memory types are not identical, you may encounter application errors. Often, an unfamiliar ABAP developer modifies a variable without noticing that the variable is imported/exported. When you use a property container, you have the chance to assign shared variables to generic field symbols, reducing the risk of such semantic errors.
- If you are debugging through a complex architecture, you may lose track of exported/imported variables. A property container, on the other hand, serves as a central variable repository where you can see and track all variables easily.

- Once you change the value of an imported variable, you'll need to re-export it to make the change public to other points of interest—which has a runtime cost. When you use a property container, you are dealing with reference variables; thus, when you change the value, changing it once changes it globally.

What is a property container? It is a class to store key-value pairs. Listing 16.4 demonstrates a sample property container class, taken from a live SAP system.

```
CLASS zcl_bag DEFINITION
  PUBLIC
  FINAL
  CREATE PUBLIC .

  PUBLIC SECTION.

    TYPES:
      BEGIN OF t_var,
        name    TYPE string,
        value   TYPE REF TO data,
      END OF t_var,

      tt_var TYPE HASHED TABLE OF t_var
        WITH UNIQUE KEY primary_key COMPONENTS  name.

  METHODS get_var
      IMPORTING
        !iv_name TYPE string
      RETURNING
        VALUE(rr_val) TYPE REF TO data
      RAISING
        zcx_bc_container_var.

  METHODS set_var
      IMPORTING
        is_var TYPE t_var.

  PRIVATE SECTION.

    DATA gt_var TYPE tt_var.

  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_bag IMPLEMENTATION.

  METHOD get_var.

    ASSIGN gt_var[ KEY primary_key
                   COMPONENTS name = iv_name ]
```

```
      TO FIELD-SYMBOL(<ls_var>).

    IF <ls_var> IS NOT ASSIGNED.
      RAISE EXCEPTION TYPE zcx_bc_container_var
        EXPORTING
          name    = iv_name
          textid  = zcx_bc_container_var=>var_missing.
    ENDIF.

    rr_val = <ls_var>-value.
  ENDMETHOD.

  METHOD set_var.

    ASSIGN gt_var[ KEY primary_key
      COMPONENTS name = is_var-name ]
      TO FIELD-SYMBOL(<ls_var>).

    IF <ls_var> IS NOT ASSIGNED.
      INSERT VALUE #( name = is_var-name )
        INTO TABLE gt_var ASSIGNING <ls_var>.
    ENDIF.

    <ls_var>-value = is_var-value.

  ENDMETHOD.

ENDCLASS.
```

Listing 16.4 Sample Property Container Class

With this class, you can set and get parameter values. Let's inspect the components of this class and see what they do for us, as follows:

- `T_VAR`
  Line type for a variable name and its value. Please note that we are storing the data reference only, not the value itself. Holding a reference enables us to support values of any kind, making the property container extremely flexible and adaptable.
- `TT_VAR`
  Table type for `T_VAR`.
- `GT_VAR`
  A private internal table to store the values set by client applications.
- `SET_VAR`
  Method that basically writes a name + value (reference) pair into `GT_VAR`.

- GET_VAR
  Method that returns the value (reference) of the given variable name from GT_VAR.

When we put this class into play, the UML diagram of the enhancement solution evolves into the sketch in Figure 16.2.

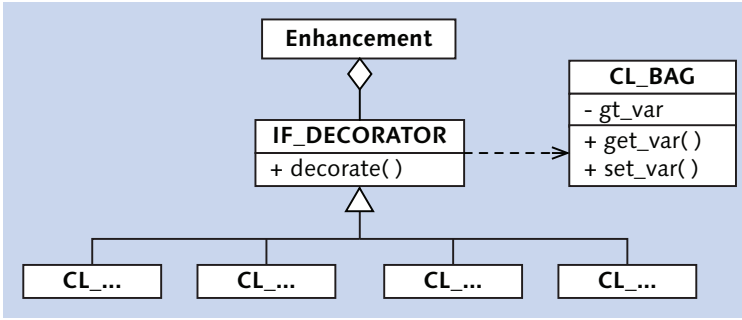


Figure 16.2 Decorator with Property Container

The decorator will expect the client application (enhancement point) to create and pass a property container that is usable by any subclass to share values. Therefore, the interface will now look like Listing 16.5.

```
INTERFACE zif_decorator.
  PUBLIC .

TYPES tt_konp TYPE STANDARD TABLE OF konp WITH DEFAULT KEY.

 METHODS decorate
    IMPORTING
      !io_bag TYPE REF TO zcl_bag
    CHANGING
      !ct_konp TYPE tt_konp.

ENDINTERFACE.
```

Listing 16.5 Decorator Interface

Since we aren't going to change the memory address of the CL_BAG instance, we made IO_LOG an import variable.

Next, we have to create and pass the property container instance from the enhancement point, as demonstrated in Listing 16.6.

```
TYPES: tt_dec TYPE STANDARD TABLE OF zif_decorator.

DATA:
  lo_obj TYPE REF TO object,
  lo_decorator TYPE REF TO zif_decorator,
  lt_clsname TYPE STANDARD TABLE OF seometarel-clsname,
  lt_dec TYPE tt_dec.

* Detect classes implementing our interface & build table of classes
SELECT clsname INTO TABLE lt_clsname
  FROM seometarel
  WHERE refclsname eq 'ZIF_DECORATOR'.

LOOP AT lt_clsname ASSIGNING FIELD-SYMBOL(<lv_clsname>).
  CREATE OBJECT lo_obj TYPE (<lv_clsname>).
  lo_decorator ?= lo_obj.
  APPEND lo_decorator TO lt_dec.
ENDLOOP.

* Create property bag instance
DATA(lo_bag) = NEW zcl_bag( ).

* Loop through classes, letting each class modify the data
LOOP AT lt_dec ASSIGNING FIELD-SYMBOL(<lo_dec>).
  <lo_dec>->decorate(
    EXPORTING
      io_bag = lo_bag
    CHANGING
      ct_konp = xkonp[]
  ).
ENDLOOP.
```

Listing 16.6 Decorator Enhancement Point with Property Container

Since every subclass of `IF_DECORATOR` will access the same instance of `CL_BAG`, they can freely set and share values. Let's assume that a subclass of `IF_DECORATOR` has an algorithm like that in Listing 16.7, where a certain calculation is done and results are written into the property container under the name `CALCU`.

```
METHOD zif_decorator~decorate.
    " Some code containing calculations
    " Calculated re-usable values reside in GT_CALCU
    io_bag->set_var(
      iv_name = 'CALCU'
      iv_val = REF #( GT_CALCU )
      ).
ENDMETHOD.
```

Listing 16.7 Sample Decorator Class Writing into Property Container

In a real-life situation, we wouldn't set the literal CALCU like this. Instead, we would use some constant stored in a suitable neutral class. However, we have taken this shortcut for the sake of simplicity in our example.

Note

This version of the property container works with data references. Data references prevent the kernel from copying the same dataset to multiple locations in the memory and allow subsequent classes to access a single point in the memory. If a subsequent class modifies the internal table GT_CALCU, it will be modified for all other classes as well.

Next, let's see how another class would access the calculation in Listing 16.8. This class needs to read the value corresponding to CALCU and use that value to fulfill a further task.

```
METHOD zif_decorator~decorate.

    FIELD-SYMBOLS: <lt_calcu> TYPE ztt_calcu.

    TRY.
        DATA(lr_calcu) = io_bag->get_var( 'CALCU' ).
        CHECK lr_calcu IS NOT INITIAL.

        ASSIGN lr_calcu->* to <lt_calcu>.
        CHECK <lt_calcu> IS ASSIGNED.

        " Some code to take advantage of the precalculated data

      CATCH zcx_bc_container_var.
        " Some code to execute in case CALCU is not set
    ENDTRY.

ENDMETHOD.
```

Listing 16.8 Sample Decorator Class Reading from a Property Container

In order to keep the classes loosely coupled, it is crucial not to assume that the property container will contain any particular value. As you can see in Listing 16.8, we have taken precautions in case CALCU is not set yet. In other words, we can't simply assume that the property container would definitely have a value stored under the name CALCU. In many cases, the variable could be missing. For example, an earlier class may have been removed, or its order may have changed so it now comes after our example class.

In Listing 16.8, we have simply used CHECK commands to ensure that the variable is present—if it's not, the method will simply exit. In a real-world situation, you can go further and implement alternative code blocks for cases where your expected variable can or can't be found. Or, if finding the variable is inevitably a must, you can raise an exception about the missing variable.

16.2 Static vs. Instance Containers

In our example in Section 16.1, we created an instance container and shared it among different classes. However, you can also create a static container object and let any class access it, anywhere.

From a technical standpoint, both approaches will work fine, and they share a lot of common ground. In both cases, you will end up having a property container class with an internal table to store values, a getter method, and a setter method. However, their runtime scopes will be different.

An instance property container will have its own private state—in case you create multiple property containers, the variables won't mix up. This was true in our case study as well. Check Listing 16.9 to see how an instance property container is defined—the central internal table will be an instance variable.

```
CLASS zcl_bag DEFINITION
  PUBLIC
  FINAL
  CREATE PUBLIC .

  PUBLIC SECTION.
    " Some definitions
  PRIVATE SECTION.
    DATA gt_var TYPE tt_var.
  PROTECTED SECTION.
ENDCLASS.
```

Listing 16.9 Instance Property Container

A static property container will share its state throughout the entire runtime. Even if you create multiple instances of a static property container, the state will be the same. In other words, you will see the exact same variables in each property container instance. Check Listing 16.10 to see how a static property container is defined—the central internal table will be a static variable.

```
CLASS zcl_bag DEFINITION
  PUBLIC
  FINAL
  CREATE PUBLIC .

  PUBLIC SECTION.
    " Some definitions
  PRIVATE SECTION.
    CLASS-DATA gt_var TYPE tt_var.
  PROTECTED SECTION.
ENDCLASS.
```

Listing 16.10 Static Property Container

When a new developer looks at a property container, he/she will understand the original developer's intention by his/her choice between a static or instance property container.

When we see an instance container, we should assume that the variables are going to be accessed within a controlled environment. The classes created and controlled by the main program will use and share the container, nothing more.

On the other hand, when we see a static container, we should assume that the variables are going to be accessed by a much wider scope of classes, possibly exceeding the controllable area of the main program. For example, the main program can be submitting another program, which might have ten subclasses, one of which might process a batch input. If you need to share your container among all that mess, then a static container is the answer.

16.3 Sharing Variables

One significant challenge with property containers is that you can unintentionally make the classes interdependent. Imagine you have two decorator classes, A and B. If B is using a variable set from A, the two variables are not loosely coupled any more—B is dependent on A from now on. If you remove A from the system, B is probably not going to work the way you imagined. Additionally, since a property container is an object that stores dynamically assigned variables, the relation between A and B will not show up in your typical where-used list.

You also need to be aware of the principles of the *n*-tiered architecture when applying a property container. Just as an OS can't be dependent on the applications the

user might install later on, low-level classes shouldn't depend on high-level classes a programmer might develop later. In fact, low-level classes shouldn't even be aware of high-level concrete classes—the communication between different levels should be performed via abstract interfaces.

The same principle applies to the property container design pattern as well. A variable set by a low-level class can be read by a high-level class because the high-level class is usually aware of and can be allowed to depend on the low-level class. This principle is similar to the way that a program is aware of and depends on the OS and can read its settings.

Note

One important point to understand: Variables can be shared, but they shouldn't turn into a point of interdependence—there is a fine line in between.

16.4 Variable Uniqueness

As systems grow, new subclasses will be added—possibly by new developers. The challenge is that you can store a variable called `DATE1` within the container and a new developer may not be aware of it. If he/she overwrites your value with another date and names it `DATE1` as well, the rest of the system might collapse.

Multiple solutions are available to solve this problem. One solution is to plant some defensive code into the container class to ensure that each variable name can be created only once—much like a hashed table.

Another solution is to store the context name of the variable with the variable name together. If you do that, `CLASS1~DATE1` and `CLASS2~DATE1` will never overwrite each other. The identifier does not need to be the class name itself—you may pick what is best for your situation.

16.5 Related Patterns

As we have already seen, the property container design pattern is a natural companion of the decorator design pattern (Chapter 13). However, don't blindly apply a property container to every decorator. If the interface of the decorator is

well defined and you don't feel like any explicit variable sharing will be needed, it's best to keep things simple—one less class is one less hassle.

Decorator is not the only use for property containers, of course—it is simply a typical example. Decorator can be used in conjunction with other patterns as well: builder, bridge, chain of responsibility, mediator, observer, and strategy are some of the patterns where a property container could be useful.

In the builder design pattern (Chapter 3), we often build a new object out of other objects. If you need to share variables between them while preserving their independency, using a property container can help.

In the bridge design pattern (Chapter 10), we have two categories of classes that need to operate together. Using a central property container to store common data will keep the application flexible.

A chain of responsibility (Chapter 18) suggests building a chain of objects dynamically and run a request through them until one of them can handle it. Since we can't know in advance which classes we might encounter in the chain, variable sharing can be conducted by setting a standard property container and running it through the involved objects.

In the case of a mediator (Chapter 20), we have a central class responsible of managing the relationship between objects—much like SAP Process Integration (PI). If the mediator class determines the objects dynamically, you can't always assume what kind of objects you'll be dealing with. Therefore, instead of setting concrete variables, using a property container to share variables between managed objects can be useful.

The observer design pattern (Chapter 22) is like Twitter—whenever something important happens in the source class, it "tweets" about it, and client classes "following" the source class will be notified about it. In many cases, the source class determines what kind of variables will be contained in the notification. However, sometimes you need to share variables between "follower" classes. Adding a common property container to the notification object can help you with that.

The strategy design pattern (Chapter 25) is about having exchangeable algorithms and dynamically determining the algorithm to use. Sometimes, you pick one algorithm and use it during the entire runtime. However, there are cases where you need to swap algorithms during a single runtime. If you need to share variables between swapped algorithm classes, a property container can be helpful.

16.6 Summary

A property container is a bag of variables passed around between classes. You need to decide between a static or instance container before going forward with the property container class.

Be careful to make the classes indirectly interdependent over the container class as well as to preserve variable uniqueness within the bag.

A property container is a natural companion to the decorator design pattern but can be used to support other patterns as well.

A virtual wrapper, a proxy is useful when you must preserve the interface of a class for the sake of compatibility, but you want the methods to behave differently. In such a case, the proxy design pattern wraps the legacy class with a new class in which some legacy methods are simply overwritten with new code.

17 Proxy

The proxy design pattern wraps an existing class with a new one. The interface of the inner class and the wrapper class remain exactly the same. This design pattern is typically used when you want to keep the interface of a class as is but want its methods to behave differently from how they originally behaved. The proxy design pattern is also known as the *surrogate design pattern*.

In this chapter, we will cover a case study involving classes reading classified employee salaries. We can't take the risk of a faulty ABAP code exposing salary information. Therefore, we will wrap the original salary class into a safe proxy class and let our programmers use the safe class only. The chapter will end with discussion of the advantages of proxy and its related patterns.

17.1 Case Study: Sensitive Salary Information

In this example, we will be dealing with SAP ERP Human Capital Management (SAP ERP HCM). As you know, salary information is among the most sensitive data a company must store. As programmers, one of our most essential responsibilities is to keep such sensitive data confidential.

Keeping this principle in mind, our goal in this section will be to create a new class within SAP ERP HCM that can return the address data, employment date, and salary data of any given employee. Let's call this class `CL_EMPLOYEE`.

For this example, we need to develop two programs: The first one (`P_RISKY`) will be developed by you and will display the salary data of the given employee. The

second one (P_SAFE), on the other hand, we will assume is being developed by a junior developer, as is often the case in the real world. P_SAFE will display the employment date and address data of the given employee. Both will use the same class.

Our first thought will likely be to make use of the MVC (model–view–controller) design pattern (Chapter 1). The Unified Modeling Language (UML) diagram for CL_EMPLOYEE is shown in Figure 17.1.

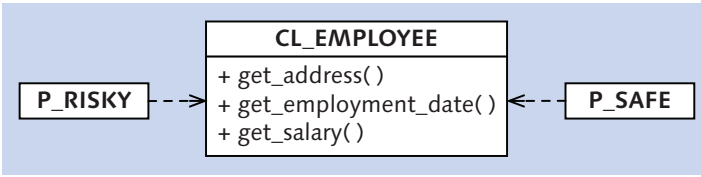


Figure 17.1 Two Applications, One Class

The code for CL_EMPLOYEE would roughly look like Listing 17.1. Basically, we have a class that can return the address, the employment date, or the salary of a given employee. Of these, the salary method is the sensitive component here.

```
CLASS zcl_employee DEFINITION
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.

    METHODS get_address
      IMPORTING !iv_pernr TYPE persno
      EXPORTING !e_data TYPE zs_address.

    METHODS get_employment_date
      IMPORTING !iv_pernr TYPE persno
      EXPORTING !e_data TYPE begda.

    METHODS get_salary
      IMPORTING !iv_pernr TYPE persno
      EXPORTING !e_data TYPE zs_salary.

  PRIVATE SECTION.
    " Some type & data definitions + methods

  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_employee IMPLEMENTATION.
```

```
  METHOD get_address.
    " Some code to get address data & put into e_data
  ENDMETHOD.

  METHOD get_employment_date.
    " Some code to get employment data & put into ev_date
  ENDMETHOD.

  METHOD get_salary.
    " Some code to read salary & put into e_data
  ENDMETHOD.

  " Some further methods

ENDMETHOD.

ENDCLASS.
```

Listing 17.1 Employee Class

While we won't present the full code here, the relevant part of `P_RISKY` would look like Listing 17.2.

```
DATA(lo_emp) = NEW zcl_employee( ).

lo_emp->get_salary(
  EXPORTING iv_pernr = gv_pernr
  IMPORTING e_data = DATA(ls_salary)
).

PERFORM display_salary USING ls_salary.
```

Listing 17.2 Risky Code Snippet

Likewise, `P_SAFE` would contain a code snippet as in Listing 17.3.

```
DATA(lo_emp) = NEW zcl_employee( ).

lo_emp->get_address(
  EXPORTING iv_pernr = gv_pernr
  IMPORTING e_data = DATA(ls_adr)
).

lo_emp->get_employment_date(
  EXPORTING iv_pernr = gv_pernr
  IMPORTING e_data = DATA(lv_begda) ).

PERFORM display_employee USING ls_adr lv_begda.
```

Listing 17.3 Safe Code Snippet

The rest of the code should be standard: P_RISKY calls CL_EMPLOYEE~GET_SALARY, and P_SAFE calls CL_EMPLOYEE~GET_ADDRESS and CL_EMPLOYEE~GET_EMPLOYMENT_DATA.

However, to highlight the usefulness of the proxy design pattern, let's assume that the junior developer included an error (our apologies to junior developers everywhere). In this example, he/she has used dynamic method calls while developing P_SAFE and has left a huge security gap that enables a sly user to call and display GET_SALARY. See Listing 17.4 for the code containing this error.

```
" Some dynamic data related code

DATA(lo_emp) = NEW zcl_employee( ).

CALL METHOD lo_emp->(gv_dynamic_method_name)
  EXPORTING iv_pernr = gv_pernr
  IMPORTING e_data = <fs_data>.

PERFORM display_dynamic_data USING <fs_data>.
```

Listing 17.4 Dynamic Method Call in Safe Application

If the user finds a way to modify GV_DYNAMIC_METHOD_NAME and to get the value GET_SALARY (via debugging, for example), he/she could display anyone's salary.

Another possibility is that the junior makes a coding error in such a critical place that a short dump is displayed to the user. Because of Murphy's Law, such a short dump would certainly contain the salary values.

In the world of programming, paranoia is your friend. In a real-world situation, you need to accept the fact that you can't foresee everything. The combination of a junior developer (or, to be fair to junior developers, an undercaffeinated senior developer) plus a sly user can result in unfavorable consequences. The proxy design pattern can provide a useful solution to this typical case. If there is the possibility of error on the part of the developer accessing your object or if you're dealing with sensitive information that a user should not have access to, you simply don't give access to the sensitive method. If they don't have the code at all, it can't be exploited.

Using the proxy design pattern, Figure 17.2 represents what a relatively safer UML diagram would look like.

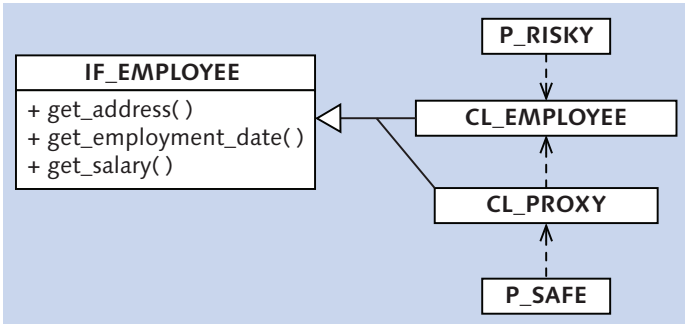


Figure 17.2 Proxy Application

The ideas behind this pattern are simple, yet effective, as you can see in the following:

- Create an interface (`IF_EMPLOYEE`) that contains the methods needed.
- Inherit a class (`CL_EMPLOYEE`) and implement all methods as usual.
- Inherit a secondary class (`CL_PROXY`), which will use `CL_EMPLOYEE` in the background but which will also change the functionality of certain methods (in our case: `GET_SALARY`).
- Pick the class to use depending on the scope of the application. In our example, `P_RISKY` (developed by Mr./Ms. Senior) would use `CL_EMPLOYEE` directly, while `P_SAFE` (developed by Mr./Ms. Junior) would use `CL_PROXY` instead.

To start the implementation, we will define `IF_EMPLOYEE`, which will look quite similar to our former `CL_EMPLOYEE` class, actually (see Listing 17.5).

```
INTERFACE zif_employee.
 PUBLIC.

    METHODS get_address
      IMPORTING !iv_pernr TYPE persno
      EXPORTING !e_data TYPE zs_address.

    METHODS get_employment_date
      IMPORTING !iv_pernr TYPE persno
      EXPORTING !e_data TYPE begda.

    METHODS get_salary
      IMPORTING !iv_pernr TYPE persno
      EXPORTING !e_data TYPE zs_salary.

ENDINTERFACE.
```

Listing 17.5 Employee Interface

The implementation of the new CL_EMPLOYEE is no major hassle either (see Listing 17.6).

```
CLASS zcl_employee DEFINITION
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.
    INTERFACES zif_employee.

  PRIVATE SECTION.
    " Some type & data definitions + methods

  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_employee IMPLEMENTATION.

  METHOD zif_employee~get_address.
    " Some code to get address data & put into e_data
  ENDMETHOD.

  METHOD zif_employee~get_employment_date.
    " Some code to get employment data & put into e_data
  ENDMETHOD.

  METHOD zif_employee~get_salary.
    " Some code to read salary & put into e_data
  ENDMETHOD.

  " Some further methods

ENDCLASS.
```

Listing 17.6 Employee Class Implementing Interface

Now for the interesting part! The core logic of the proxy design pattern is demonstrated in Listing 17.7.

```
CLASS zcl_proxy DEFINITION
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.
    INTERFACES zif_employee.

  PRIVATE SECTION.
    DATA go_emp TYPE REF TO zcl_employee.
    " Some type & data definitions + methods
```

```
  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_proxy IMPLEMENTATION.

  METHOD zif_employee~get_address.
    " Some code to ensure that GO_EMP is created
    go_emp->zif_employee~get_address(
      EXPORTING iv_pernr = iv_pernr
      IMPORTING e_data = e_data ).
  ENDMETHOD.

  METHOD zif_employee~get_employment_date.
    " Some code to ensure that GO_EMP is created
    go_emp->zif_employee~get_employment_date(
      EXPORTING iv_pernr = iv_pernr
      IMPORTING e_data = e_data ).
  ENDMETHOD.

  METHOD zif_employee~get_salary.
    RETURN.
  ENDMETHOD.

  " Some further methods

ENDCLASS.
```

Listing 17.7 Proxy Class

If you inspect the code, you will see that `GET_ADDRESS` and `GET_EMPLOYMENT_DATE` reflect the functionality of `CL_EMPLOYEE` directly. However, the sensitive method `GET_SALARY` simply returns nothing. There is no code to call the salary calculation at all.

When we look at the client applications, Listing 17.8 represents the code snippet of `P_RISKY`, which is the same as before.

```
DATA(lo_emp) = NEW zcl_employee( ).

lo_emp->get_salary(
  EXPORTING iv_pernr = gv_pernr
  IMPORTING e_data = DATA(ls_salary)
).

PERFORM display_salary USING ls_salary.
```

Listing 17.8 Risky Code Snippet after Proxy

However, P_SAFE will change a bit, as evident in Listing 17.9.

```
DATA(lo_emp) = NEW zcl_proxy( ).

lo_emp->get_address(
  EXPORTING iv_pernr = gv_pernr
  IMPORTING e_data = DATA(ls_adr)
).

lo_emp->get_employment_date(
  EXPORTING iv_pernr = gv_pernr
  IMPORTING e_data = DATA(lv_begda)
).

PERFORM display_employee USING ls_adr lv_begda.
```

Listing 17.9 Safe Code Snippet after Proxy

As you can see, P_SAFE is using an instance of CL_PROXY now, which contains absolutely no code within the method GET_SALARY. Even if a dynamic method call takes place, the risk is minimized, as demonstrated in Listing 17.10.

```
" Some dynamic data related code

DATA(lo_emp) = NEW zcl_proxy( ).

CALL METHOD lo_emp->(gv_dynamic_method_name)
  EXPORTING iv_pernr = gv_pernr
  IMPORTING e_data = <fs_data>.

PERFORM display_dynamic_data USING <fs_data>.
```

Listing 17.10 Dynamic Method Call after Proxy

Assuming that the user finds a way of changing GV_DYNAMIC_METHOD_NAME and making it contain the value GET_SALARY, we are still secure because the GET_SALARY in CL_PROXY is empty.

17.2 When to Use

We have seen that the proxy design pattern can be used in cases where you want to limit the access to the inner class due to security reasons. Such proxies are called *protection proxies*. Beyond our simple example, there are many other cases where you could find a wrapper class useful.

You can use the proxy design pattern to redirect the request to a remote machine. For example, you might need your `CALL_BAPI` method to call a remote web service instead of posting a document to SAP. Such proxies are called *remote proxies*.

Performance might be another reason to use a proxy class—you can delay some performance-hungry operations of the inner class until they are actually needed and possibly prevent them entirely if they are never needed at all. Such proxies are called *virtual proxies*.

17.3 Related Patterns

The proxy design pattern is often compared to the adapter design pattern (Chapter 9), but the differences are not difficult to spot. Proxy provides the same interface as the inner class. Adapter provides a different interface.

If you are dealing with a virtual proxy, you should consider taking advantage of the lazy initialization design pattern (Chapter 5).

Using the proxy design pattern can encourage you to take advantage of inheritance, where you will derive most of the methods of the inner class and override a few methods only. In our case study, we have used the composition approach. `CL_EMPLOYEE` into `CL_PROXY` share the same interface.

Although possible from a technical point of view, inheritance is usually not the recommended approach in the world of design patterns. If you want to use inheritance and to change only a few methods so that they behave a little differently under different conditions, check out Chapter 24 to see if using the state design pattern is a better idea.

17.4 Summary

The proxy design pattern is about wrapping an existing class with a new one, where the wrapper preserves the interface but behaves differently. There are three typical cases to use a proxy.

You might want to limit the scope of a risky class over a proxy—this is called a protection proxy. You also might want to redirect a request to a different system—this is called a remote proxy. You might want to improve the performance of a slow class by adding lazy initialization—this is called a virtual proxy.

Say you have an event and a list of potential classes to handle the event. However, since the classes are determined dynamically, you can't be sure which class would handle the situation best. Chain of responsibility offers a practical solution: build a chain of objects and pass the event from object to object. The first object to successfully handle the event will break the chain and return the result.

18 Chain of Responsibility

Sometimes you have multiple potential classes to handle an event or request within your application, and you can't determine in advance which one can actually handle the event/request. Using the chain of responsibility design pattern, you can give more than one object the chance to handle the request until you find the right one.

We will start with a case study that surely has been encountered by most ABAP developers at some point in their careers: purchase order approver determination via a user exit. Complex business rules are involved in this situation, and trying to fit them all into a single user exit might end in tears. This case study will reveal how a chain of responsibility can make such cases easier to code and maintain, thus reducing the likelihood of coding errors. We will continue with the discussion of risks involved and the related patterns.

18.1 Case Study: Purchase Order Approver Determination

In this example, we will step into the world of workflows. Our client needs to approve purchase orders before they are released and has a rather complex agent determination logic involving a priority system. The rule set looks as follows:

- Priority 1: If purchasing organization is ORG1, then George and Mary should approve.

- Priority 2: If purchasing organization is ORG2 or ORG3, the following rules come into play:
  - If the amount is less than $1,000, then John should approve.
  - If the amount is greater than $1,000, the relevant department manager from the HR structure should approve first. Next, the financial manager from the HR structure should approve.
- Priority 3: If purchasing organization is ORG4, the following rules come into play:
  - For plant PLT1, the department manager and financial manager should approve.
  - For plant PLT2, the immediate superior should approve.
  - For all other plants, the department manager should approve.

Of course, this wouldn't stop here. If you have developed workflows before, you know that rules change over time, and you need a flexible code structure to keep up with changes. Writing fixed workflow rules will save the day today but won't get you too far when things change.

Although there are multiple solutions in terms of flexibility, for the sake of our example, we will focus on the chain of responsibility design pattern, which provides a nice, smooth, flexible, and extendable solution to such problems. Whenever you see a set of prioritized rules, chain of responsibility is one of the first design patterns that should spring to mind.

Luckily, a set of prioritized rules is exactly what we have here. We have three distinct rules. If rule 1 applies to the situation, we can determine the agents and return to the main application with our answer. If it doesn't apply, then we move on to rule 2 to see if it applies the situation. If it doesn't, then we move on to the next rule, and so on. Basically, we build a chain of responsibility (pun intended) and stop on the first link that can respond to the requirement.

The development usually starts by building an abstract class (`cl_agent_rule`), which will be the parent of each and every rule (in our example, priority 1, priority 2, priority 3), as seen in Figure 18.1.

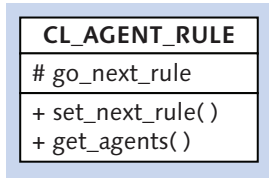


Figure 18.1 Agent Rule Abstract Class

The code of the `cl_agent_rule` class is demonstrated in Listing 18.1.

```
CLASS zcl_agent_rule DEFINITION
  PUBLIC ABSTRACT CREATE PUBLIC.

  PUBLIC SECTION.

    TYPES:
      tt_agent TYPE STANDARD TABLE OF swhactor WITH DEFAULT KEY,
      tt_ekpo TYPE STANDARD TABLE OF ekpo WITH DEFAULT KEY.

    METHODS set_next_rule FINAL
      IMPORTING
        !io_rule TYPE REF TO zcl_agent_rule.

    METHODS get_agents ABSTRACT
      IMPORTING
        !is_ekko TYPE ekko
        !it_ekpo TYPE tt_ekpo
      RETURNING
        VALUE(rt_agent) TYPE tt_agent.

  PRIVATE SECTION.
  PROTECTED SECTION.
    DATA go_next_rule TYPE REF TO zcl_agent_rule.
ENDCLASS.

CLASS zcl_agent_rule IMPLEMENTATION.

  METHOD set_next_rule.
    go_next_rule = io_rule.
  ENDMETHOD.

ENDCLASS.
```

Listing 18.1 Agent Rule Abstract Class

We have two major methods in this abstract class, as follows:

- SET_NEXT_RULE will be used to determine the link order in the chain. This results in the following:
  - In rule 1, this method will be used to make its GO_NEXT_RULE to point to rule 2.
  - In rule 2, this method will be used to make its GO_NEXT_RULE to point to rule 3.
  - On the last rule, GO_NEXT_RULE will be initial.
- GET_AGENTS is the abstract class and will be filled within the concrete classes only. This method will actually read the SAP tables to return workflow approvers.

To get a better understanding, we need to move forward to the concrete classes, as sketched in Figure 18.2.

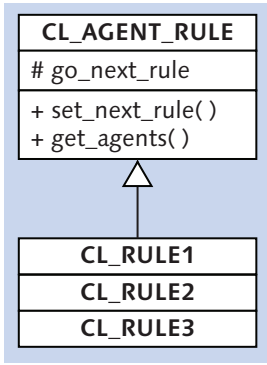


Figure 18.2 Agent Rule Concrete Classes

Let's start with our first rule. To keep things simple, we will use hardcoded values (ignoring that this is very bad practice) and focus on the pattern. See Listing 18.2 for the code contained in the first rule.

```
CLASS zcl_rule1 DEFINITION
  INHERITING FROM zcl_agent_rule
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.
    METHODS get_agents REDEFINITION.

  PRIVATE SECTION.
  PROTECTED SECTION.

ENDCLASS.
```

```
CLASS zcl_rule1 IMPLEMENTATION.

  METHOD get_agents.
    IF is_ekko-ekorg EQ 'ORG1'.
      rt_agent = VALUE #(
        ( OTYPE = 'US' OBJID = 'GEORGE' )
        ( OTYPE = 'US' OBJID = 'MARY' ) ).
    ELSE.
      CHECK go_next_rule IS NOT INITIAL.
      rt_agent = go_next_rule->get_agents(
        is_ekko = is_ekko
        it_ekpo = it_ekpo ).
    ENDIF.
  ENDMETHOD.

ENDCLASS.
```

Listing 18.2 Class for Rule 1

This code does not do anything unexpected. If we can handle the request within this rule (`EKORG EQ 'ORG1'`), we fulfill the request and return the agent. Otherwise, we pass the request to the next rule, which is `RULE2` and which should have been set over `SET_NEXT_RULE` beforehand. We haven't performed that task yet—the client application is responsible of that kind of stuff.

Following the same logic, let's take a look at rule 2 in Listing 18.3.

```
CLASS zcl_rule2 DEFINITION
  INHERITING FROM zcl_agent_rule
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.
    METHODS get_agents REDEFINITION.

  PRIVATE SECTION.
  PROTECTED SECTION.

ENDCLASS.

CLASS zcl_rule2 IMPLEMENTATION.

  METHOD get_agents.
    IF is_ekko-ekorg EQ 'ORG2' OR is_ekko-ekorg EQ 'ORG3'.
      " Calculate the amount sum in IT_EKPO
      " Determine agents based on sum and put into RT_AGENT
    ELSE.
      CHECK go_next_rule IS NOT INITIAL.
      rt_agent = go_next_rule->get_agents(
```

```
        is_ekko = is_ekko
        it_ekpo = it_ekpo ).
    ENDIF.
  ENDMETHOD.

ENDCLASS.
```

Listing 18.3 Class for Rule 2

The agent determination logic in rule 2 is rather complex and so has been skipped over so that we can focus on understanding the logic of the chain of responsibility pattern.

The code for rule 3 will look very similar to the previous two, as seen in Listing 18.4.

```
CLASS zcl_rule3 DEFINITION
  INHERITING FROM zcl_agent_rule
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.
    METHODS get_agents REDEFINITION.

  PRIVATE SECTION.
  PROTECTED SECTION.

ENDCLASS.

CLASS zcl_rule3 IMPLEMENTATION.

  METHOD get_agents.
    IF is_ekko-ekorg EQ 'ORG4'.
      " Determine the plant
      " Determine agents based on the plant and put into RT_AGENT
    ELSE.
      CHECK go_next_rule IS NOT INITIAL.
      rt_agent = go_next_rule->get_agents(
        is_ekko = is_ekko
        it_ekpo = it_ekpo ).
    ENDIF.
  ENDMETHOD.

ENDCLASS.
```

Listing 18.4 Class for Rule 3

Now that we have our chain of rules, we can move on to the client application that will determine the actual agent by following the links of the chain.

First of all, we need a Z-table to determine the priority of the classes. We will simply call this table ZCHAIN (see Table 18.1).

PRIORITY	CLSNAME
1	ZCL_RULE1
2	ZCL_RULE2
3	ZCL_RULE3

Table 18.1 Sample of Rules, Ordered by Priority

The client application needs to follow these steps:

1. Read EKKO and EKPO to build our decision base.
2. Read ZCHAIN to build the list of rules.
3. Link the rules like a chain.
4. Start the agent determination process by calling the first rule.
5. Return agents.

Let's see this code in action in Listing 18.5.

```
TYPES:
  BEGIN OF t_chain,
    priority TYPE zchain-priority,
    clsname TYPE zchain-clsname,
    obj TYPE REF TO zcl_agent_rule,
  END OF t_chain,

  tt_chain TYPE STANDARD TABLE OF t_chain WITH DEFAULT KEY.

DATA:
  lt_chain TYPE tt_chain,
  lt_ekpo TYPE zcl_agent_rule=>tt_ekpo,
  ls_ekko TYPE ekko,
  lv_obj TYPE REF TO object.

* Read EKKO + EKPO to build our decision base
  SELECT SINGLE * INTO ls_ekko FROM ekko WHERE ebeln EQ iv_ebeln.
  SELECT * INTO TABLE lt_ekpo FROM ekpo WHERE ebeln EQ iv_ebeln.

* Read ZCHAIN to build the list of rules
  SELECT * INTO CORRESPONDING FIELDS OF TABLE lt_chain
    FROM zchain.

  LOOP AT lt_chain ASSIGNING FIELD-SYMBOL(<ls_chain>).
```

```
    CREATE OBJECT lv_obj TYPE (<ls_chain>-clsname).
    <ls_chain>-obj ?= lv_obj.
  ENDLOOP.

* Link the rules like a chain
  LOOP AT lt_chain ASSIGNING <ls_chain>.
    ASSIGN lt_chain[ sy-tabix + 1 ] TO FIELD-SYMBOL(<ls_next>).
    CHECK sy-subrc EQ 0.
    <ls_chain>-obj->set_next_rule( <ls_next>-obj ).
  ENDLOOP.

* Start agent determination process by calling the first rule
  ASSIGN lt_chain[ 1 ] TO <ls_chain>.
  DATA(lt_agent) = <ls_chain>-obj->get_agents(
    is_ekko = ls_ekko
    it_ekpo = lt_ekpo ).

* Return agents
  " Some magic workflow code
```

Listing 18.5 Client Application Using Chain of Responsibility

As usual, we have left out all the defensive programming and bells and whistles, leaving the pure overview of the design pattern in place. Here is what will happen on runtime:

1. `CL_RULE1~GET_AGENTS` will be called. If agents are determined, the method exits.
2. If not, `CL_RULE2~GET_AGENTS` will be called. If agents are determined, the method exits.
3. If not, `CL_RULE3~GET_AGENTS` will be called. Because this is the last link in the chain, the method exits whether agents are determined or not.

Note

The names of subclasses (`ZCL_RULE1`, `ZCL_RULE2`, etc.) are determined using a Z-table in this example. See Appendix B on subclass determination for alternative approaches. However, having a Z-table gives us a great deal of flexibility in this case. Consultants and key users can change the order of rules by going to Transaction SM30 and then altering `ZCHAIN` without even touching a single line of ABAP code.

Another advantage is that, if additional rules pop up over time, all you need to do is to create a new class and put it into `ZCHAIN`. Or, if you need to deactivate a rule temporarily, you can just remove it from `ZCHAIN` and that's it—no need to comment out ABAP codes.

A little piece of advice: Instead of passing EKKO and EKPO as two distinct variables in the rule class, we recommend you wrap them together into a structure—or even better, an object. Doing so is not a technical requirement but does make the code look less complicated.

18.2 Risks

When using the chain of responsibility design pattern, there might be a case when none of the handlers (the rules, in our example) can process your request. Therefore, you should be prepared for the request not being handled at all. In such a case, you can have a default handler and pass the request to that object. Another alternative is to raise an exception and let the client application decide what to do about it. In any case, it is not recommended to leave the request unprocessed.

A broken chain is another risk. In this pattern, we expect each handler class to handle the request if it can and pass it forward if it can't. If, for some reason, one of the developers forgets to pass it forward, the chain would be broken and the entire application might fail. To prevent such mistakes, you can add more responsibilities to the abstract handler class. The abstract class would have an abstract method to handle the request and a concrete method to pass it forward and let the subclasses fill the handler code only. That way, the part where the request is passed from hand to hand among potential handlers can never be forgotten because it is no longer be the responsibility of subclasses.

18.3 Related Patterns

The chain of handlers can often be implemented as a singleton (Chapter 8). The exception would be when you build the handler chain dynamically depending on the situation.

If you combine chain of responsibility with the composite design pattern (Chapter 11), you can make each component's parent act as the next in line for a request. This approach can be useful when the handlers have a hierarchy from general (lowest-level parent) to special (highest-level child). You can move from child to parent, ensuring that the more special handlers have priority over handlers that are more general.

If each request is handled by one handler or the client knows the request object relationship in advance, a chain of responsibility would add an obsolete layer that increases complexity and decreases performance. In such cases, you can pick a simpler pattern—such as strategy (Chapter 25).

18.4 Summary

A chain of responsibility is typically used when you have many potential classes to handle a request but can't determine in advance which one would best handle the case. Each class inspects the request and either handles it or forwards it to the next candidate.

Remember to have a safety net for cases where no candidate class can deal with the case and manage the risks from a chain break. A chain of handlers is often built as a singleton.

PART IV

Behavioral Design Patterns

The command design pattern helps when you have a handful of tasks, which need to be called in different orders depending on the situation. In other words, you need to pick the correct flow dynamically. In this pattern, a collection of tasks is stored in a receiver class, and various command classes perform those tasks in distinct orders.

19 Command

Sometimes a handful of functionalities, such as Business Application Programming Interfaces (BAPIs), need to be called in different orders and different ways under different conditions. To do so, you could write a method for each BAPI and then write a central method filled with `IF`s—we will see this approach in action at the start of Section 19.1. However, the command design pattern provides a more flexible and manageable approach to this problem. By allowing you to pick the correct flow dynamically, you will have much cleaner code.

Speaking of BAPIs, our case study will highlight an SAP ERP Sales and Distribution (SD) application where we need to post different kinds of documents in different cases. Using the command design pattern, we will build a clean and flexible structure. Even if further posting rules come into play later on, we can add them without modifying other classes. This chapter will follow the case study with a discussion on when to use/avoid the command design pattern, and significant related patterns will be highlighted.

19.1 Case Study: SD Documents

In this example, we will be dealing with an application capable of creating SD documents. In most SD environments, there are three major document types involved: an order, a delivery, and an invoice.

Our goal is to be able to create a different set of documents under different circumstances. For instance, we may want to be able to do the following:

- For customers of type A, we need to create an order document, then a delivery document, and finally an invoice, finishing the entire chain.
- For customers of type B, we only need to create an order.
- For customers of type C, we need to create an order document and then an invoice.

Using classic ABAP, you could develop a façade-like subroutine (see Chapter 14 for more on façade) to handle this requirement, as demonstrated in Listing 19.1.

```
FORM post_docs USING is_info.

  CASE is_info-ktokd.
    WHEN 'A'.
      PERFORM: post_ord USING is_info,
               post_dlv USING is_info,
               post_inv USING is_info.
    WHEN 'B'.
      PERFORM post_ord USING is_info.
    WHEN 'C'.
      PERFORM: post_ord USING is_info,
               post_inv USING is_info.
  ENDCASE.

ENDFORM.
```

Listing 19.1 Subroutine for Document Creation

Although this subroutine will work and solve our current problem, it has limitations. If the requirements expand to include additional customer types or if the number of actions increases, we would need to modify the central method—which means a new user test organization for all former cases.

The same applies to a new document type. If we suddenly need to post a goods issue as well, we will be in the same situation where we need to modify already-tested code.

Another disadvantage is that the chain of cases resides in a closed subroutine. If another application needs to create an order and then create an invoice in the future, the same logic needs to be copied—and copying code instead of reusing it is an anti-pattern (for more information on anti-patterns, see Appendix C).

The command design pattern can address these issues. We start off by creating the receiver class. The receiver class contains the pool of methods to create orders, deliveries, and invoices. Note that the class doesn't know anything about

the business logic or the order of methods. It is simply a method container class—similar to a toolkit class. The structure of the receiver class (`cl_receiver`) is sketched in Figure 19.1.

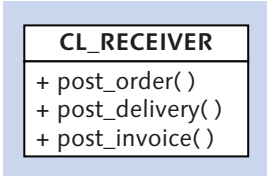


Figure 19.1 Receiver Class

The code of the `cl_receiver` class is demonstrated in Listing 19.2.

```
CLASS zcl_receiver DEFINITION
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.

    METHODS post_order IMPORTING !is_info TYPE zs_info.
    METHODS post_delivery IMPORTING !is_info TYPE zs_info.
    METHODS post_invoice IMPORTINT !is_info TYPE zs_info.

  PRIVATE SECTION.
    " Some type & data definitions + methods

  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_receiver IMPLEMENTATION.

  METHOD post_order.
    " Call BAPI to create order
  ENDMETHOD.

  METHOD post_delivery.
    " Call BAPI to create delivery
  ENDMETHOD.

  METHOD post_invoice.
    " Call BAPI to create invoice
  ENDMETHOD.

  " Some further methods

ENDCLASS.
```

Listing 19.2 Receiver Class

At this point, we have a group of methods to be used however we'd like. Our next step is to create the command classes. The general logic is that we must have a distinct command class for each combination of method calls from the receiver class. In our example, each customer type requires that SD document creation happen in a different order. Therefore, we will create a distinct command class for each customer type. The command classes will be derived from a common interface or abstract class so they can be used interchangeably—as showcased in Figure 19.2.

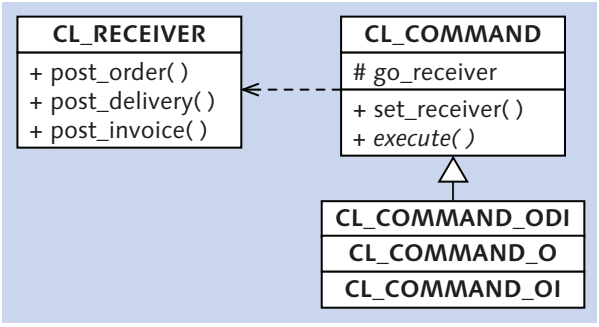


Figure 19.2 Command Classes

In this case, we have derived the command classes from an abstract class `CL_COMMAND`. In Figure 19.2, you can see the following:

- `CL_COMMAND_ODI` is the class to create an *o*rder, a *d*elivery, and an *i*nvoice. For customers of type A, we will be using this class.
- `CL_COMMAND_O` is the class to create an *o*rder. For customers of type B, we will be using this class.
- `CL_COMMAND_OI` is the class to create an *o*rder and an *i*nvoice. For customers of type C, we will be using this class.

Let's take a look at `CL_COMMAND` in Listing 19.3 and see what the abstract class looks like. The abstract class would have, at a minimum, two methods: `SET_RECEIVER` to set the receiver object and `EXECUTE` to execute the command itself. Please note that `EXECUTE` should be an abstract method as it will be filled by subclasses.

```
CLASS zcl_command DEFINITION
  PUBLIC ABSTRACT CREATE PUBLIC.

  PUBLIC SECTION.
```

```
    METHODS set_receiver FINAL
      IMPORTING
        !io_receiver TYPE REF TO zcl_receiver.

    METHODS execute ABSTRACT.

  PRIVATE SECTION.
  PROTECTED SECTION.
    DATA go_receiver TYPE REF TO zcl_receiver.
ENDCLASS.

CLASS zcl_command IMPLEMENTATION.

  METHOD set_receiver.
    go_receiver = io_receiver.
  ENDMETHOD.

ENDCLASS.
```

Listing 19.3 Abstract Command Class

Next, let's take a look at `CL_COMMAND_ODI` in Listing 19.4. As a subclass of `CL_COMMAND`, `CL_COMMAND_ODI` needs to implement the abstract method `EXECUTE` to perform its required tasks.

```
CLASS zcl_command_odi DEFINITION
  INHERITING FROM zcl_command
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.
    METHODS execute REDEFINITION.
  PRIVATE SECTION.
  PROTECTED SECTION.

ENDCLASS.

CLASS zcl_command_odi IMPLEMENTATION.

  METHOD execute.
    go_receiver->post_order( ).
    go_receiver->post_delivery( ).
    go_receiver->post_invoice( ).
  ENDMETHOD.

ENDCLASS.
```

Listing 19.4 Command Class CL_COMMAND_ODI

Due to the nature of abstract classes, `SET_RECEIVER` and `GO_RECEIVER` were inherited by `CL_COMMAND_ODI`. Implementing the `EXECUTE` method is all there is left to

do for the command classes. All other command classes will share the same logic; see Listing 19.5 for this last bit of coding.

```
CLASS zcl_command_o DEFINITION
  INHERITING FROM zcl_command
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.
    METHODS execute REDEFINITION.
  PRIVATE SECTION.
  PROTECTED SECTION.

ENDCLASS.

CLASS zcl_command_o IMPLEMENTATION.

  METHOD execute.
    go_receiver->post_order( ).
  ENDMETHOD.

ENDCLASS.

CLASS zcl_command_oi DEFINITION
  INHERITING FROM zcl_command
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.
    METHODS execute REDEFINITION.
  PRIVATE SECTION.
  PROTECTED SECTION.

ENDCLASS.

CLASS zcl_command_oi IMPLEMENTATION.

  METHOD execute.
    go_receiver->post_order( ).
    go_receiver->post_invoice( ).
  ENDMETHOD.

ENDCLASS.
```

Listing 19.5 Command Classes CL_COMMAND_O and CL_COMMAND_OI

Note that this is an oversimplified example, as usual. In a real-world application, the `EXECUTE` method would be much more crowded. At this point, we already have lots of flexibility available. If, in the future, we get new customer types requiring the same chain of documents, we can reuse the command classes `CL_COMMAND_ODI`, `CL_COMMAND_O`, and `CL_COMMAND_OI`.

The next step is to bring the receiver and command classes together to make the magic happen. This connection happens in a separate class, traditionally called the *invoker*, which is represented in Figure 19.3. If we summarize the invoker's architecture, CL_RECEIVER acts like a pool of useful methods—much like a toolkit class. CL_COMMAND and its subclasses contain distinct algorithms that make use of the receiver methods. CL_INVOKER is the new member of the "team" and is actually responsible for executing the command.

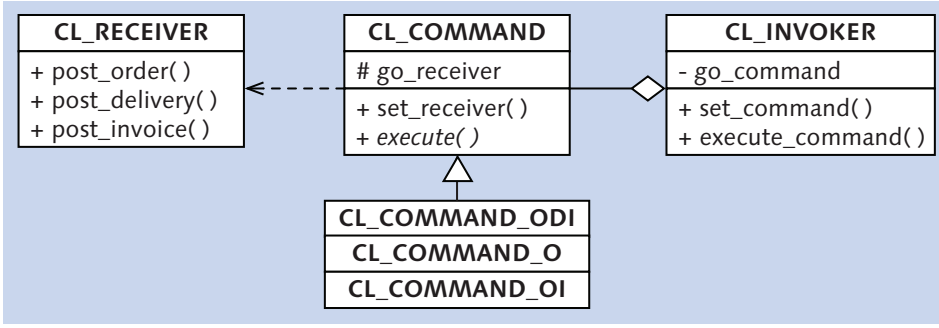


Figure 19.3 Invoker Class

The code for the invoker class is shown in Listing 19.6.

```
CLASS zcl_invoker DEFINITION
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.

    METHODS set_command
      IMPORTING !iv_command TYPE REF TO zcl_command.

    METHODS execute_command.

  PRIVATE SECTION.
    DATA go_command TYPE REF TO zcl_command.
    " Some private methods
  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_invoker IMPLEMENTATION.

  METHOD set_command.
    go_command = iv_command.
  ENDMETHOD.
```

```
  METHOD execute_command.
    " Do some common validations; such as data integrity, etc
    " Do some common checks; such as authorization, etc
    " Do some common initialization, etc
    go_command->execute( ).
  ENDMETHOD.

ENDCLASS.
```

Listing 19.6 Invoker Class

The purpose of the invoker is hidden in the comments of the `EXECUTE_COMMAND` method. The invoker contains some common initialization and validation code that needs to be conducted before actually executing the command. If you are merely calling `go_command->execute( )` within `EXECUTE_COMMAND`, you can technically leave out the invoker. However, design pattern traditionalists may raise their eyebrows.

We now have everything we need to build our application. Let's take a look at the client application in Listing 19.7 and see what this looks like.

```
FORM post_docs USING is_info.

  DATA:
    lo_command TYPE REF TO zcl_command,
    lo_obj TYPE REF TO object.

* Get the name of the corresponding command class
  SELECT SINGLE clsname INTO @DATA(lv_clsname)
    FROM zsdt_command
    WHERE ktokd EQ @is_info-ktokd.

* Create objects
  DATA(lo_receiver) = NEW zcl_receiver( ).

  CREATE OBJECT lo_obj TYPE (lv_clsname).
  lo_command ?= lo_obj.
  lo_command->set_receiver( lo_receiver ).

  DATA(lo_invoker) = NEW zcl_invoker( ).
  lo_invoker->set_command( lo_command ).

* Create documents
  lo_invoker->execute_command( ).

ENDFORM.
```

Listing 19.7 Client Application Using Command

Note

The names of subclasses (`ZCL_COMMAND_ODI`, `ZCL_COMMAND_O`, and `ZCL_COMMAND_OI`) are determined by reading a Z-table in this example. We assumed that the class name of each customer type is stored in the database table `ZSDT_COMMAND`.

See Appendix B on subclass determination for alternative approaches to using Z-tables.

19.2 When to Use or Avoid

If you have a collection of functionalities (within `CL_RECEIVER`) that need to be called in various orders under different circumstances, the command design pattern is a good approach.

However, if you have only a couple of simple functionalities, using the command design pattern might be overkill. In short, command pays off in complex scenarios but can bring excess complexity to simple cases.

19.3 Related Patterns

In this chapter, we have witnessed the flexibility the command design pattern can provide. Command often comes with an additional functionality: undo. If you keep the previous state and provide an additional method to return to this state, your application will be supporting the undo operation in no time. If you keep an array of previous states, you can let the user undo as much as he/she likes. The memento design pattern (Chapter 21) is a natural supplemental functionality. Combining command and memento will provide you the necessary framework to implement the undo operation easily. To duplicate the state to put into the undo list, you can take advantage of the prototype design pattern (Chapter 7).

If some of your commands share a similar logic, you can use intermediate abstract classes to avoid code duplication and derive each similar command class from the abstract class—which means you can take advantage of the template method design pattern (Chapter 26).

If you feel like you need to share variables among different commands, the property container (Chapter 16) might be the answer you are looking for. Compared

to using `EXPORT TO MEMORY`/`IMPORT FROM MEMORY` commands, a property container is a cleaner and more manageable solution.

19.4 Summary

The command design pattern can be used when you have a collection of methods that need to be called in a different order and fashion under different circumstances. Just make sure that you are not overengineering and adding excess complexity.

Especially in GUI applications, undo is often a natural companion for the command design pattern. If this is the case, you can add the memento design pattern as a supplement to be responsible for undo operations.

The purpose of the mediator design pattern is clear: to act as a central point of governance for object interaction. Instead of making classes directly refer to each other for a certain coordinated functionality, we make them blind to each other. Each class notifies a central mediator class of significant events, and the mediator decides what to do and calls other classes as needed.

20 Mediator

The purpose of the mediator design pattern is to provide a central point that administers how objects interact. Instead of making objects refer to each other, they send and receive messages to/from the mediator class. That way, the interactions are more manageable, and error handling and maintenance are easier.

Are you familiar with SAP Process Integration (SAP PI)? What SAP PI does for independent systems is done by the mediator class for independent classes. The logic is the same, and so are the advantages (and disadvantages too, of course).

Another typical example that illuminates how mediator logic works is a chat room. When a guest in a chat room sends a message, that message is not sent directly to the computers of other guests. Instead, it is sent to the chat server, and the server distributes this message to other guests. In this case, the chat server takes the role of a mediator. In scope of our design pattern, the mediator class takes the same central role for other classes.

In this chapter, we will start with a case study of a stock movement simulation application, where we create an object for each storage location. As the user simulates movements, storage location objects need to interact. Instead of making them refer each other directly, we will consolidate the relations in a mediator and see how helpful this is. We will continue by discussing the appropriate times to use a mediator and its potential disadvantages.

20.1 Case Study: Stock Movement Simulation

In this example, we will create an application where the user is able to simulate stock movements between storage locations. The user will have a GUI where he/she can transfer stock values between storage locations and see what happens. A sketch of the GUI in question is available in Figure 20.1.

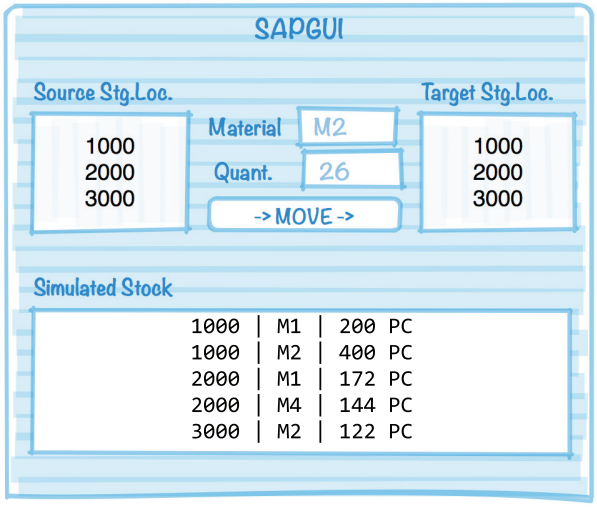


Figure 20.1 Sketch of Stock Movement GUI

For each store location, we are going to need a distinct object storing the stock values that is also able to make stock movements. Figure 20.2 represents the interface (`if_lgort`) we will use for this purpose.

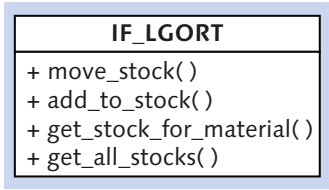


Figure 20.2 Storage Location Interface

The code for this storage location interface is demonstrated in Listing 20.1.

```
INTERFACE zif_lgort.
 PUBLIC.
```

```
    TYPES:
      BEGIN OF t_stock,
        matnr TYPE matnr,
        menge TYPE menge_d,
      END OF t_stock,

      tt_stock TYPE HASHED TABLE OF t_stock
        WITH UNIQUE KEY primary_key COMPONENTS matnr.

    METHODS move_stock
      IMPORTING
        !iv_target TYPE lgort_d
        !is_stock TYPE t_stock
      RAISING
        zcx_insufficient_stock.

    METHODS add_to_stock IMPORTING !is_stock TYPE t_stock.

    METHODS get_stock_for_material
      IMPORTING
        !iv_matnr TYPE matnr
      RETURNING
        VALUE(rv_menge) TYPE menge_d.

    METHODS get_all_stocks
      RETURNING VALUE(rt_stock) TYPE tt_stock.

ENDINTERFACE.
```

Listing 20.1 Storage Location Interface

Let's inspect the components of the storage location interface:

- `T_STOCK` is the line type to store the stock per material.
- `TT_STOCK` is the table type for `T_STOCK`.
- `MOVE_STOCK` is the method to remove stock from one storage location and to add it to another storage location.
- `ADD_STOCK` is the method to add stock to the storage location.
- `GET_STOCK_FOR_MATERIAL` will return the current simulated stock for the given material.
- `GET_ALL_STOCKS` will return the current simulated stock of all materials.

The model class will contain an object reference for every storage location we have, and the corresponding data definition will look like Listing 20.2.

```
TYPES:
  BEGIN OF t_stloc,
    lgort TYPE lgort_d,
    obj TYPE REF TO zif_lgort,
  END OF t_stloc,

  tt_stloc TYPE HASHED TABLE OF t_stloc
    WITH UNIQUE KEY primary_key COMPONENTS lgort.

DATA gt_stloc TYPE tt_stloc.
```

Listing 20.2 Model Class Data Definition Snippet

For our basic example, we can get away with a single storage location implementation, as sketched in Figure 20.3.

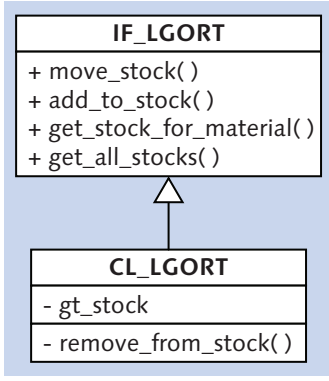


Figure 20.3 Storage Location Class

The code representing this basic class contains no surprises, as can be seen in Listing 20.3.

```
CLASS zcl_lgort DEFINITION
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.
    INTERFACES zif_lgort.

  PRIVATE SECTION.
    DATA gt_stock TYPE tt_stock.

    METHODS remove_from_stock
      IMPORTING
        !is_stock TYPE t_stock
      RAISING
```

```
        zcx_insufficient_stock.

  PROTECTED SECTION.

ENDCLASS.

CLASS zcl_lgort IMPLEMENTATION.

  METHOD zif_lgort~move_stock.
    " Coming soon, be patient
  ENDMETHOD.

  METHOD zif_lgort~add_to_stock.
    COLLECT is_stock INTO gt_stock.
  ENDMETHOD.

  METHOD get_stock_for_material.
    TRY.
        rv_menge = gt_stock[ KEY primary_key
          COMPONENTS matnr = iv_matnr]-menge.
      CATCH cx_sy_itab_line_not_found ##NO_HANDLER.
    ENDTRY.
  ENDMETHOD.

  METHOD get_all_stocks.
    rt_stock[] = gt_stock[].
  ENDMETHOD.

  METHOD remove_from_stock.
    DATA(lv_current) = get_stock_for_material( iv_matnr ).
    IF lv_current LT iv_menge.
      RAISE EXCEPTION TYPE zcx_insufficient_stock.
    ENDIF.
    COLLECT VALUE #( matnr = iv_matnr menge = iv_menge * -1 )
      INTO gt_stock.
  ENDMETHOD.

ENDCLASS.
```

Listing 20.3 Incomplete Storage Location Class

To keep things simple, we have made some modifications to the real-life example. The most significant point is that we have assumed that all stock values will have the same unit of measure, leaving the field MEINS out.

So far, so good—we have a storage location class in the pocket at this point. Please note that we haven't implemented the method `MOVE_STOCK` yet. What we want to happen is that every time the user clicks the Move button, the application will call

the MOVE_STOCK method of the corresponding object. The object will to check if it has enough stock; if this is not the case, it will raise an exception. If the stock is enough, the source object will reduce its own stock and increase the stock of the target object by calling the ADD_TO_STOCK method of the other object.

How is this possible? After all, the object corresponding to storage location 1000 doesn't know anything about the object corresponding to storage location 2000. How are we going to make them interact?

One solution would be to make the objects mutually dependent—meaning, we can make the objects refer each other. This dependency is possible from a technical perspective; however, it contradicts with the basic design principle of making objects only loosely coupled. Objects referring each other directly should be avoided as much as possible.

Another solution would be to pass the target object as a changing parameter. Although this is also technically possible, you would be exposing the target object too much. All you want to allow is the stock value to change, nothing else. If you pass the target object as a changing parameter and expose the entire object, the code in the source object might perform unsafe operations and modify the target object more than it should.

Here, the mediator design pattern steps in. Instead of making the objects refer each other implicitly, we create a secondary mediator class to manage the relationships between objects. Every time an IF_LGORT instance needs to address another one, it contacts the mediator class for this purpose, as sketched in Figure 20.4.

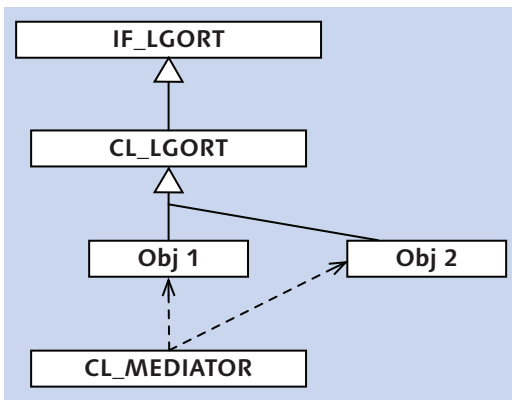


Figure 20.4 Mediator Class

The mediator class is the central management class for all the relationships between objects. To make the concept more tangible, imagine this: if all of your SAP and non-SAP systems were objects, SAP PI would be your mediator class.

Let's see what the mediator class looks like in Listing 20.4.

```
CLASS zcl_mediator DEFINITION
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.

    CLASS-METHODS get_instance_singleton
      EXPORTING !eo_mediator TYPE REF TO zcl_mediator.

    METHODS register_lgort
      IMPORTING
        !iv_lgort TYPE lgort_d
        !ir_objref TYPE REF TO data.

    METHODS send_stock
      IMPORTING
        !iv_target TYPE lgort_d
        !is_stock TYPE zif_lgort~t_stock.

  PRIVATE SECTION.
    TYPES: BEGIN OF t_stloc,
      lgort TYPE lgort_d,
      objref TYPE REF TO data,
    END OF t_stloc,

    tt_stloc TYPE HASHED TABLE OF t_stloc
      WITH UNIQUE KEY primary_key COMPONENTS lgort.

    DATA: gt_stloc TYPE tt_stloc.

    " Some further definitions regarding singleton, etc.

  PROTECTED SECTION.

ENDCLASS.

CLASS zcl_mediator IMPLEMENTATION.

  METHOD get_instance_singleton.
    " Some code to return a singleton instance
  ENDMETHOD.

  METHOD register_lgort.
    INSERT VALUE #( lgort = iv_lgort objref = ir_objref )
```

```
      INTO TABLE gt_stloc.
  ENDMETHOD.

  METHOD send_stock.

    FIELD-SYMBOLS: <lo_lgort> TYPE REF TO zif_lgort.

    ASSIGN gt_stloc[
      KEY primary_key
      COMPONENTS lgort = iv_lgort
    ] TO FIELD-SYMBOL(<ls_stloc>).

    CHECK sy-subrc EQ 0.

    ASSIGN <ls_stloc>-objref->* to <lo_lgort>.

    <lo_lgort>->add_to_stock( is_stock ).

  ENDMETHOD.

ENDCLASS.
```

Listing 20.4 Mediator Class

The model class is expected to maintain a (singleton) instance of `CL_MEDIATOR` and register each and every `IF_LGORT` instance into it via the method `REGISTER_LGORT`. If you are not sure about the singleton design pattern, see Chapter 8.

Let's inspect the components of the mediator class in detail:

- `T_STLOC` is the line type to store a storage location object by storage location code.
- `TT_STLOC` is the table type of `T_STLOC`.
- `GT_STLOC` will keep our list and objects of storage locations.
- `GET_INSTANCE_SINGLETON` will return an instance of `CL_MEDIATOR`.
- `REGISTER_LGORT` will add a new storage location object into `GT_STLOC`.
- `SEND_STOCK` will send a new stock quantity to the designated `CL_LGORT` object.

At this point, we can finally fill the method `MOVE_STOCK` of `CL_LGORT`. The entire class is going to look like Listing 20.5.

```
CLASS zcl_lgort DEFINITION
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.
```

```
    INTERFACES zif_lgort.

  PRIVATE SECTION.
    DATA gt_stock TYPE tt_stock.

    METHODS remove_from_stock
      IMPORTING
        !is_stock TYPE t_stock
      RAISING
        zcx_insufficient_stock.

  PROTECTED SECTION.

ENDCLASS.

CLASS zcl_lgort IMPLEMENTATION.

  METHOD zif_lgort~move_stock.

    remove_from_stock( is_stock ).

    zcl_mediator=>get_instance_singleton(
      IMPORTING eo_mediator = DATA(lo_mediator)
    ).

    lo_mediator->send_stock(
      iv_target = iv_target
      is_stock = is_stock
    ).

  ENDMETHOD.

  METHOD zif_lgort~add_to_stock.
    COLLECT is_stock INTO gt_stock.
  ENDMETHOD.

  METHOD get_stock_for_material.
    TRY.
        rv_menge = gt_stock[ KEY primary_key
          COMPONENTS matnr = iv_matnr]-menge.
      CATCH cx_sy_itab_line_not_found ##NO_HANDLER.
    ENDTRY.
  ENDMETHOD.

  METHOD get_all_stocks.
    rt_stock[] = gt_stock[].
  ENDMETHOD.

  METHOD remove_from_stock.
    DATA(lv_current) = get_stock_for_material( iv_matnr ).
```

```
    IF lv_current LT iv_menge.
      RAISE EXCEPTION TYPE zcx_insufficient_stock.
    ENDIF.
    COLLECT VALUE #( matnr = iv_matnr menge = iv_menge * -1 )
      INTO gt_stock.
  ENDMETHOD.

ENDCLASS.
```

Listing 20.5 Complete Storage Location Class

Now, we need to focus on the method `MOVE_STOCK` to understand what's going on. Remember that the purpose of this method is to remove the stock from the source storage location and add it to the target storage location. Currently, we are acting from the standpoint of the source storage location.

The first step is simple. To remove the stock, we simply call the method `REMOVE_FROM_STOCK`, and that's it—`GT_STOCK` is modified to have a lower value.

The second step requires us to send the stock to the target storage location. Instead of "touching" the target object directly, we have created a mediator instance (`LO_MEDIATOR`) and called its method `SEND_STOCK`.

Now, how is that useful? We can quickly think of a handful of cases where running through a mediator is a good idea:

- The mediator can keep a log of movements, managing a Z-table of transaction histories.
- The mediator can split the stock between two target storage locations in special cases.
- The mediator can make use of a lock object to ensure that the storage location objects are not accessed simultaneously by multiple clients.
- The mediator can centrally make some ad-hoc authorization checks and refuse the operation if the user lacks authorization.

The list can be expanded even further, but you get the idea. Centralizing such tasks ensures the maintainability of our application and also makes the task of error tracing easier. If storage location objects refer to each other directly, pinpointing the cause of a relational error can be quite difficult. However, when we centralize everything inside a mediator, we have a single place to look/debug in case of an error.

20.2 When to Use

Using the mediator design pattern promotes loose coupling between objects. Objects don't refer each other directly, and to change an interaction rule, you likely wouldn't need to change the objects at all—all you have to do is to modify the mediator class. When you are trying to decide whether or not you should use a mediator class, imagine deciding using middleware for integration.

If you have two simple classes with minimum interaction, implementing a mediator class might be overkill—just like installing complicated middleware software for a simple integration between two systems. Having an extra link in the chain without any value-add is not advisable as it adds complexity to the system. Instead, the observer design pattern (Chapter 22) might be used to preserve loose coupling and avoid using a mediator.

However, as the number of classes or interactions increase, the advantages of using a mediator class also increase—just like the advantages of centralizing system integrations with middleware. Since we are integrating classes, why not use a middleware-like class to manage their relations?

20.3 Disadvantages

One potential disadvantage of using the mediator class is complexity. Mediator classes tend to get bloated and complex over time. If the number of interactions is dramatically high, you could consider having multiple mediators. Each mediator would have its own domain of responsibility. In that case, having an abstract mediator class in which other mediators would be derived from a common object (which covers the basic mediator functionality) might be a good idea.

20.4 Summary

A mediator is basically a central class that administers the interactions between independent objects. It centralizes interaction rules and reduces system complexity, making error tracking and future maintenance easier.

For the most simple cases, mediator might be overkill though. For more complex cases, you might consider having multiple mediators with different responsibilities.

Memento focuses on one thing only: undo operations. If, in any given case, you need to perform an undo within your application, memento is your first stop. Memento also makes it possible to reverse the process to support redo operations as well.

21 Memento

The memento design pattern is an object-oriented undo operation. Whenever you have an object, for which you may need an "undo" operation, the memento design pattern is probably going to solve your problem.

Calculators make a good analogy for memento. Advanced calculators let you "undo" the latest calculation, returning to the previous value. That's exactly what we want.

Memento is a suggested practice to add a "checkpoint" capability to your state as well. If you encounter a failure after certain operations, you can return the state to the previously saved checkpoint.

Memento can also be imagined as an object-oriented rollback functionality. Especially if you are using the command design pattern (Chapter 19), an undo functionality is often needed. Memento is a natural supplement.

In this chapter, we will demonstrate memento with a case study of a budget planning application. As the user enters financial values into the application, he/she will be able to "undo" the operations and return to the previous state. The chapter will continue with a discussion involving the risks and with an evaluation of a possible redo operation. We will also discuss related patterns of memento.

21.1 Case Study: Budget Planning

In this example, we are going to create an application where the user can enter budget values for cost elements. This application is similar to your typical SAP

GUI application with a table control to enter data. Figure 21.1 represents a simple sketch of such a UI.

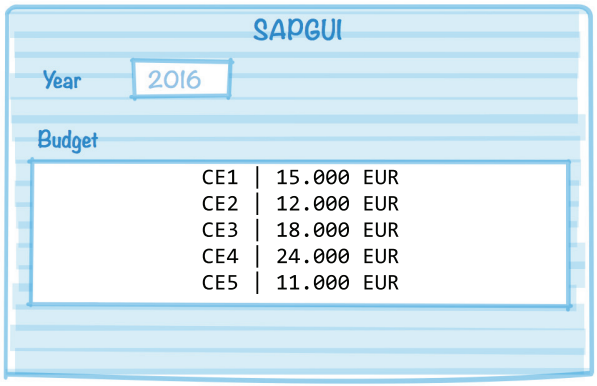


Figure 21.1 Basic GUI Sketch

However, there's a catch: The user requires the ability to undo his/her modifications on the table control. After playing around a bit, he/she might think that the former figure looked better and want to return to the previous state. Figure 21.2 represents the same UI with an undo button.

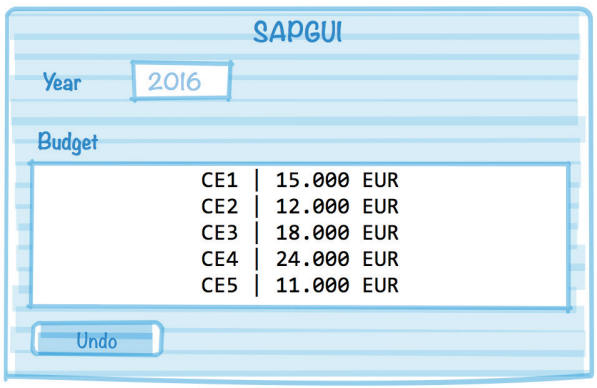


Figure 21.2 Basic GUI Sketch with Undo Button

Before bringing memento into the equation, let's start with the foundation of the application. This application will be an MVC (model–view–controller) (see Chapter 1) application with a GUI and a model class, as you can see in Figure 21.3.

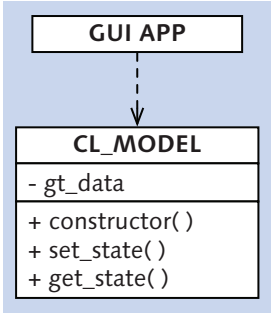


Figure 21.3 MVC Overview

The method SET_STATE sets the data in the GUI to the model class. GET_STATE returns the data in the model class to the application. Typically, the model class would contain further methods (or objects) to save the data to the database, read from the database, do authority checks, etc., though we have kept things simple in our example. Listing 21.1 demonstrates what CL_MODEL would look like.

```
CLASS zcl_model DEFINITION
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.
    METHODS constructor.
    METHODS set_state IMPORTING !it_data TYPE ztt_data.
    METHODS get_state RETURNING VALUE(rt_data) TYPE ztt_data.

  PRIVATE SECTION.
    DATA gt_data TYPE ztt_data.
    " Some data definitions
    " Some method definitions
  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_model IMPLEMENTATION.

  METHOD constructor.
    " Initialization
  ENDMETHOD.

  METHOD set_state.
    " Validate incoming data
    gt_data[] = it_data[].
  ENDMETHOD.

  METHOD get_state.
```

```
    rt_data[] = gt_data[].
  ENDMETHOD.

  " Some more methods
ENDCLASS.
```

Listing 21.1 Model Class

Let's inspect the components of the model class:

- GT_DATA is the internal table to store the financial values. Our state consists of merely GT_DATA.
- CONSTRUCTOR is self-explanatory.
- SET_STATE is our setter method, which will validate the financial values entered by the user and store it inside GT_DATA.
- GET_DATA is out getter method, which will return the financial values in GT_DATA.

Listing 21.2 displays a small code snippet from the main application.

```
FORM data_changed.
  go_model->set_state( gt_gui_data ).
ENDFORM.
```

Listing 21.2 Client Application Code Snippet

Let's now move on to actually implementing the memento design pattern, which will enable the undo function. We will start by creating a memento class, which is able to hold the state of CL_MODEL and will be able to hold a copy of GT_DATA. You can see the structure of the memento class in Figure 21.4.

In our example, CL_MODEL and CL_MEMENTO look similar. However, in a real-world application, CL_MODEL would have much more to it, while CL_MEMENTO would usually stay as simple as it looks right now. Listing 21.3 demonstrates what the CL_MEMENTO class looks like.

```
CLASS zcl_memento DEFINITION
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.
    METHODS constructor IMPORTING !it_data TYPE ztt_data.
    METHODS get_state RETURNING VALUE(rt_data) TYPE ztt_data.

  PRIVATE SECTION.
```

```
    DATA gt_data TYPE ztt_data.
  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_memento IMPLEMENTATION.

  METHOD constructor.
    gt_data[] = it_data[].
  ENDMETHOD.

  METHOD get_state.
    rt_data[] = gt_data[].
  ENDMETHOD.

ENDCLASS.
```

Listing 21.3 Memento Class

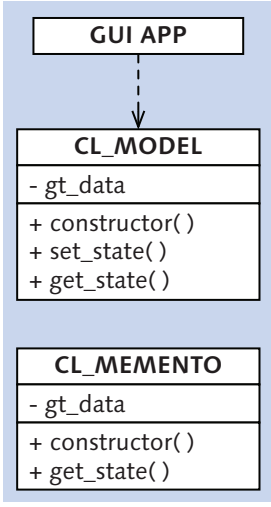


Figure 21.4 Memento Class

In our example, all the data we need to undo lies within `GT_DATA`. Therefore, `CL_MEMENTO` can get away with storing `GT_DATA` only. In a real-world application, you would probably have multiple variables, and your `CL_MEMENTO` would have to store all of them.

`CL_MEMENTO` clearly saves a single state of `CL_MODEL`. Therefore, if we use `CL_MEMENTO` alone, we can only perform a single step of undo. However, we need the

ability to perform the undo operation multiple times. For this purpose, a second class called CL_CARETAKER is required. All that the caretaker does is to store an array of CL_MEMENTO and return the appropriate instance of CL_MEMENTO when asked, as shown in Figure 21.5.

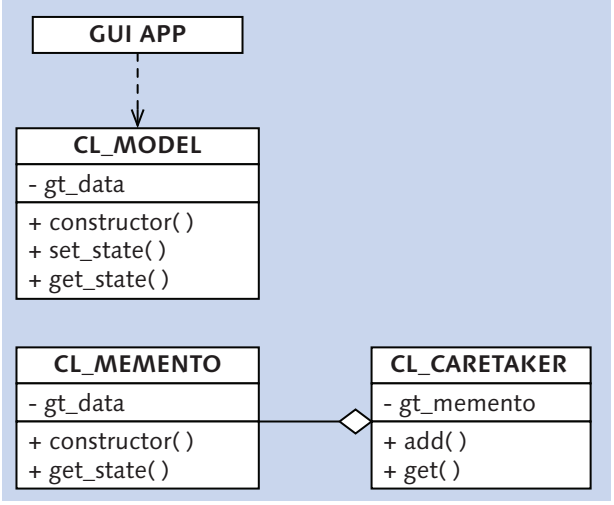


Figure 21.5 Caretaker Class

Let's take a look at the code of CL_CARETAKER in Listing 21.4.

```
CLASS zcl_caretaker DEFINITION
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.
    METHODS add IMPORTING !io_memento TYPE REF TO zcl_memento.

    METHODS get
      IMPORTING
        !iv_index TYPE i
      RETURNING
        VALUE(ro_memento) TYPE REF TO zcl_memento.

  PRIVATE SECTION.
    TYPES:
      BEGIN OF t_memento_list,
        index TYPE i,
        memento TYPE REF TO zcl_memento,
      END OF t_memento_list,
```

```
      tt_memento_list TYPE STANDARD TABLE OF t_memento_list.

    DATA gt_memento TYPE tt_memento_list.

  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_caretaker IMPLEMENTATION.

  METHOD add.
    APPEND VALUE(
      index = LINES( gt_memento ) + 1
      memento = io_memento ) TO gt_memento.
  ENDMETHOD.

  METHOD get.
    ro_memento = gt_memento[ index = iv_index ]-memento.
  ENDMETHOD.

ENDCLASS.
```

Listing 21.4 Caretaker Class

Let's inspect the components of the caretaker class to have a better understanding of its functionality:

- `T_MEMENTO_LIST` is the line type to store historic memento objects, `INDEX` is the sequence number, and `MEMENTO` is the object.
- `TT_MEMENTO_LIST` is the table type of `T_MEMENTO_LIST`.
- `GT_MEMENTO` is the internal table to store memento objects.
- `ADD` is the method to add a new checkpoint. This method will append the passed memento object to `GT_MEMENTO` with a new `INDEX` value.
- `GET` will return the memento object from `GT_MEMENTO` that corresponds to the passed `INDEX` value.

Improving the code in this class can be done is several ways. Using a GUID instead of an index variable is usually a better idea, as indexes (like `SY-TABIX`) tend to mingle when an unexpected `SORT` command is executed. Another point is that performance could be improved with a hashed table. Nonetheless, you can see how the caretaker class takes care of a list of memento objects.

In the next step, we will add memento support to our model class. Basically, we will add two methods to the model class—as sketched in Figure 21.6.

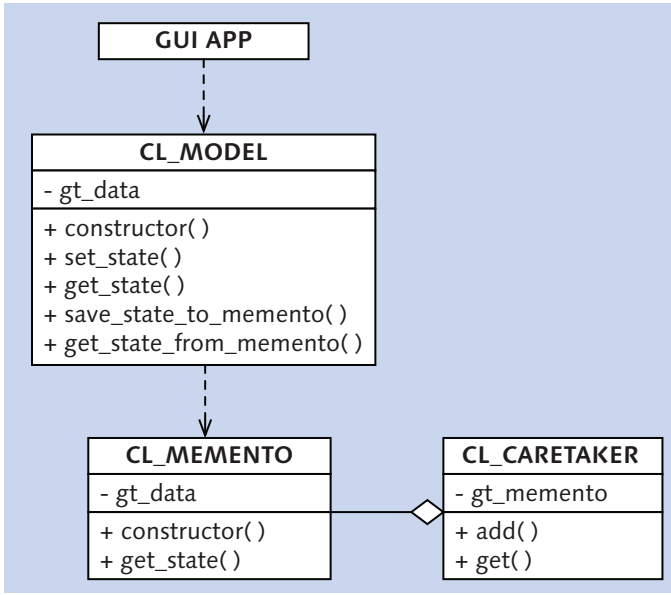


Figure 21.6 Complete Memento Architecture

Note

The model class is traditionally called the *originator class* in the world of memento.

SAVE_STATE_TO_MEMENTO is responsible for saving the current state of CL_MODEL (which means GT_DATA) into a new memento object and returning it. GET_STATE_FROM_MEMENTO is responsible for building the state of CL_MODEL (which means GT_DATA) from the passed memento object. You can see the model class with memento support in Listing 21.5.

```
CLASS zcl_model DEFINITION
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.
    METHODS constructor.
    METHODS set_state IMPORTING !it_data TYPE ztt_data.
    METHODS get_state RETURNING VALUE(rt_data) TYPE ztt_data.

    METHODS save_state_to_memento
      RETURNING
        VALUE(ro_memento) TYPE REF TO zcl_memento.

    METHODS get_state_from_memento
```

```
      IMPORTING
        !io_memento TYPE REF TO zcl_memento.

  PRIVATE SECTION.
    DATA gt_data TYPE ztt_data.
    " Some data definitions
    " Some method definitions
  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_model IMPLEMENTATION.

  METHOD constructor.
    " Initialization
  ENDMETHOD.

  METHOD set_state.
    " Validate incoming data
    gt_data[] = it_data[].
  ENDMETHOD.

  METHOD get_state.
    rt_data[] = gt_data[].
  ENDMETHOD.

  METHOD save_state_to_memento.
    ro_memento = NEW #( get_state( ) ).
  ENDMETHOD.

  METHOD get_state_from_memento.
    set_state( io_memento->get_state( ) ).
  ENDMETHOD.

  " Some more methods
ENDCLASS.
```

Listing 21.5 Model Class with Memento Support

Finally, we can code our main application to support the undo functionality. We won't be coding the entire application for example. Instead, we will inspect the most relevant parts of the application in Listing 21.6.

```
DATA:
  go_caretaker TYPE REF TO zcl_caretaker,
  go_model TYPE REF TO zcl_model.

" Some unrelated code

FORM data_changed.
```

```
  go_model->set_state( gt_gui_data ).
  go_caretaker->add( go_model->save_state_to_memento( ) ).
ENDFORM.

FORM undo.
  " Determine the index of previous step and store into lv_index
  go_model->get_state_from_memento(
    go_caretaker->get( lv_index )
  ).

  PERFORM refresh_screen.
ENDFORM.
```

Listing 21.6 Client Application Code Snippets with Memento

As we can see, we have instances for the caretaker and model classes.

Whenever the user changes the data in the GUI, the application would call the form `DATA_CHANGED`, which sets the state into the model object `GO_MODEL` and creates a checkpoint in `GO_CARETAKER`. We assumed here that each and every change that a user makes needs to be undoable—therefore, we create a new checkpoint for every movement.

Whenever the user clicks the Undo button, the application would call the form `UNDO`, which will get the previous state from `GO_CARETAKER` and set the previous state into the model class `GO_MODEL` as the current state, followed by a GUI refresh.

As you can see, the memento design pattern is not complicated at all. After creating the memento and caretaker classes, the extra coding on the GUI (controller) side is minimal.

21.2 Risks

In the example in Section 21.1, we have forced the memento/caretaker classes to store the entire state of the model class (`GT_DATA`). You might have a case where the state is too big, and replacing the entire state might be a performance hog. If the state is too big, you can get away by storing the delta only. In such a case, you would need to store what has changed, and the undo operation will not replace the entire state. Instead, the operation will only replace the delta. (Hint: The data you need to store in your memento class will probably look like `CDHDR` and `CDPOS`.)

Another risk is that the caretaker can consume a lot of memory if you don't limit the number of undo operations. If you record every single change without any limitation, the caretaker might have to store thousands of historical states. We recommend limiting the number of possible undo operations by a reasonable amount.

Note that doing an undo is not always an easy task. If you have created an SAP ERP Financial Accounting (FI) document, your undo operation might need to visit Transaction FB08 and do a reverse posting, which might complicate things in a way you couldn't foresee—be careful in that area. If the state involves posting/cancelling documents, we recommend you reconsider allowing users to undo those operations.

21.3 Redo

If you extend the functionality of your classes, you can also implement a redo operation. For redo operations, you need to implement the functionality of moving forward and backward between states.

Let's add the redo functionality to the client application in our case study.

```
DATA:
  go_caretaker TYPE REF TO zcl_caretaker,
  go_model TYPE REF TO zcl_model.

" Some unrelated code

FORM data_changed.
  go_model->set_state( gt_gui_data ).
  go_caretaker->add( go_model->save_state_to_memento( ) ).
ENDFORM.

FORM undo.
  " Determine the index of previous step and store into lv_index
  go_model->get_state_from_memento(
    go_caretaker->get( lv_index )
  ).

  PERFORM refresh_screen.
ENDFORM.

FORM redo.
  " Determine the index of the next step and store into lv_index
  go_model->get_state_from_memento(
```

```
    go_caretaker->get( lv_index )
  ).

  PERFORM refresh_screen.
ENDFORM.
```

Listing 21.7 Client Application with Redo Functionality

As you can see, we did something similar in the form `UNDO`—with one difference though.

When the user clicks the Undo button, we would decrease the state index by 1 to find out the *previous* index and get that state from the caretaker.

When the user clicks the Redo button, we would increase the state index by 1 to find out the *next* index and get that state from the caretaker.

Of course, this simple example leaves out any defensive programming. You are responsible for ensuring that the user can't click Undo after the earliest checkpoint is reached or that he/she can't click Redo after the latest checkpoint is reached.

21.4 Summary

Memento is the standard approach for undo/redo functionality. However, performance risks should be taken into account. If the state is too big, memento should store the delta only. Limiting the number of undo operations is also a good idea.

Memento and the command design pattern (Chapter 19) often go hand-in-hand.

If the observer design pattern had been invented yesterday, it would probably be called the Twitter design pattern. This design pattern has classes "follow" each other to be notified of significant events. Upon receiving a new "tweet," each "follower" class evaluates the situation and takes action as needed.

22 Observer

The observer design pattern, one of the more popular patterns, informs other classes when something important happens in the source class. You can imagine it like Twitter: You follow a person you care about, and whenever something significant happens, he/she posts a tweet that is seen by all the followers. The tweeter doesn't need to know anything about his/her followers; the follower has the responsibility to read and react to each tweet. Gathering all interested parties in the same place to make announcements would be too complicated.

The same logic applies to the object-oriented world in the form of the observer design pattern. Imagine having a central class (called the *subject* in the technical world), where interesting things happen—like the creation of a new document, the deletion of a line item, a significant change to master data, etc. If these changes affect other applications (classes), you make them "follow" the central class so they are informed about whatever is going on. Each "follower" (called *observers* in the technical world) is free to filter, ignore, or act upon incoming information, and that's none of the central class' business.

Now, we will move forward to a case study that deals with target sales values. As the user updates the target figures, we need to notify other applications. The entire communication will run over the observer design pattern. We will continue with a discussion regarding the advantages/disadvantages of observer and an evaluation of related patterns.

22.1 Case Study: Target Sales Values

In this example, we will have a custom-developed application where the user enters or modifies the target sales values of dealers. This custom application will be composed of a GUI program (developed in Transaction SE38), a model class (developed in Transaction SE24), and some Z-tables to store the data (defined in Transaction SE11). The central table would look like Table 22.1.

LIFNR	PERIOD	TARGET_AMOUNT	CURRENCY
L00015	01.2015	40,000	USD
L00015	02.2015	38,000	USD
L00015	03.2015	37,500	USD
L00016	01.2015	22,000	USD
L00016	02.2015	24,000	USD
L00016	03.2015	22,000	USD

Table 22.1 Sample Data in the Central Database Table

Let's label this application `ZTARGET`. For this example, we will also have an additional application to keep track of changes in significant sales data, just like the functionality in `CDHDR` and `CDPOS`—but with a more advanced logic. For example, this application may send emails to key people when a significant change happens. Let's label this application `ZCHANGE`.

Your job as a programmer will be to make the two applications talk. Whenever something changes in `ZTARGET`, you want `ZCHANGE` to take action and make the appropriate people receive warning messages.

Now, using the classical approach, your code in `ZTARGET` would look like Listing 22.1.

```
METHOD save_targets.

  write_to_db( ).

  DATA(lo_chg) = NEW zcl_bc_change( ).
  lo_chg->detect_recipients( gv_vkorg ).

  LOOP AT gt_data ASSIGNING FIELD-SYMBOL(<ls_data>).
    lo_chg->add_change( |New target for { <ls_data>-lifnr }| ).
  ENDLOOP.
```

```
  lo_chg->flush( ).

ENDMETHOD.
```

Listing 22.1 Bad Code in Model Class

This code is subpar for various reasons, as follows:

- You forced ZTARGET to learn and implement the standards of ZCHANGE. It's just like making the finance department purchase their own supplies when you actually have a purchasing department.
- ZTARGET and ZCHANGE are no longer loosely coupled—they are more hard wired. If, one day, the structure of ZCHANGE changes, you'll have to modify ZTARGET as well, which costs time and effort and is a risky thing to do. You may end up planting bugs into ZTARGET, which will try to hide until the most inappropriate moment.
- Inevitably, more applications will be interested in ZTARGET changes in the future. This may require you to add more and more foreign code (like ZCHANGE) and end up having an overly complicated system.

Luckily, we have the observer design pattern to avoid all this and can develop accordingly.

Figure 22.1 represents the Unified Modeling Language (UML) diagram of the solution to our problem using the observer design pattern.

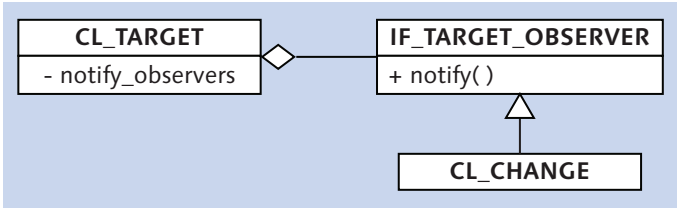


Figure 22.1 Observer Architecture

In this solution, we'll make use of an interface called IF_TARGET_OBSERVER. Whenever data is modified in CL_TARGET, CL_TARGET will find all implementations of IF_TARGET_OBSERVER and call the method NOTIFY. Being the first and only implementer of the interface, CL_CHANGE will be following CL_TARGET and act whenever a significant data change happens.

This approach eliminates most, if not all, of the disadvantages of using the classical approach, as follows:

- ZTARGET will do what it does best; so will ZCHANGE—just like two separate departments in a company. The two applications won't try to overtake each other's tasks—they will simply communicate.
- ZTARGET and ZCHANGE are loosely coupled. If anything changes in ZCHANGE, the change will be maintained in ZCHANGE alone. Most of the time, no maintenance would be needed on ZTARGET.
- If further applications are interested in ZTARGET changes, all they have to do is to implement IF_TARGET_OBSERVER—no need to modify ZTARGET at all.

Now, let's move forward and look at the code—starting with the interface in Listing 22.2.

```
INTERFACE zif_target_observer.
  PUBLIC .

  CLASS-METHODS notify
    IMPORTING
      !iv_vkorg TYPE vkorg
      !it_data TYPE zsdtt_target.

ENDINTERFACE.
```

Listing 22.2 Observer Interface

You may be wondering why we preferred to use a static method instead of an instance method. The reason is that ZTARGET doesn't know anything about its followers. We can't simply call the command CREATE OBJECT for each follower because each follower might expect different constructor parameters, which we can't know in advance. Therefore, using static methods in observer interfaces is usually a good idea. On the other hand, if you have a software standard where all observers are obliged to implement a certain constructor (or no constructor at all), feel free to use instance methods in the interface.

Be careful though because static methods can't be redefined in ABAP. If you want to derive observers from each other or from an abstract class, using an instance method becomes a necessity.

In other words, if observer class A is the parent of observer class B, you can't redefine (override) any static methods of class A in class B. Therefore, any method

in A that you want to redefine in B must be an instance method. Because of that, you'll likely prefer an instance method over a static method in a real-world application where inheritance might be an option.

Let's take a look at the new ZTARGET in Listing 22.3.

```
METHOD save_targets.

  DATA lt_clsname TYPE STANDARD TABLE OF seoclsname.

  write_to_db( ).

  SELECT clsname INTO TABLE lt_clsname
    FROM seometarel
    WHERE refclsname eq 'ZIF_TARGET_OBSERVER'.

  LOOP AT lt_clsname ASSIGNING FIELD-SYMBOL(<lv_clsname>).
    CALL METHOD (<lv_clsname>)=>zif_target_observer~notify
      EXPORTING
        iv_vkorg = gv_vkorg
        it_data = gt_data.
  ENDLOOP.

ENDMETHOD.
```

Listing 22.3 Notifying Observers in Model Class

To get into greater detail for the code in Listing 22.3, first we see the following:

```
  write_to_db( ).
```

This code writes the data stored in GT_DATA is to the Z-table. Next, we have these three lines:

```
  SELECT clsname INTO TABLE lt_clsname
    FROM seometarel
    WHERE refclsname eq 'ZIF_TARGET_OBSERVER'.
```

Note

The names of subclasses are determined using SEOMETAREL in this example. See Appendix B on subclass determination for alternative approaches.

Using SEOMETAREL is a nice trick to find out all implementations of a certain interface. Whenever a class implements an interface, a corresponding record is written into the database table SEOMETAREL. That way, we have built a list of all of our followers. Be careful, however, because if we want to inform the observers in a certain

order or if only some of the observers have to be informed in certain cases, then we need to bypass SEOMETAREL and create our own Z-table storing all this data. However, in such cases, we would probably be using another pattern—like the mediator design pattern (Chapter 20). Most of the time, SEOMETAREL is good enough for the observer pattern.

Finally, now that we have a list of observers (followers), we keep them informed using the following lines from Listing 22.3:

```
LOOP AT lt_clsname ASSIGNING FIELD-SYMBOL(<lv_clsname>).
  CALL METHOD (<lv_clsname>)=>zif_target_observer~notify
    EXPORTING
      iv_vkorg = gv_vkorg
      it_data = gt_data.
ENDLOOP.
```

This technique will call a method of a class whose name is dynamically determined. For each class in SEOMETAREL, the method NOTIFY is called.

Our last step for this example will be to inspect ZCHANGE in Listing 22.4 and see what the relevant part of the code looks like.

```
CLASS zcl_bc_change DEFINITION
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.
    INTERFACES zif_target_observer.
    "Some data definitions
    "Some method definitions
  PRIVATE SECTION.
    "Some data definitions
    "Some method definitions
  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_bc_change IMPLEMENTATION.

  METHOD zif_target_observer~notify.
    detect_recipients( iv_vkorg ).

    LOOP AT it_data ASSIGNING FIELD-SYMBOL(<ls_data>).
      add_change( |New target for vendor { <ls_data>-lifnr }| ).
    ENDLOOP.

    flush( ).

  ENDMETHOD.
```

```
  "Some more methods
ENDCLASS.
```

Listing 22.4 Observer Class

With this code, whatever happens in `ZCHANGE` stays in `ZCHANGE`, and no other party needs to get involved.

22.2 Advantages

Following our example, let's underline some of the advantages of using the observer design pattern in such cases:

- Instead of making the central class do all the work, responsibilities (and code) are distributed to the appropriate classes, which makes the system less complicated and easier to maintain.
- Once the pattern is up and running, you rarely have to modify the delicate central class to add more functionality. Just add a new follower, and it's done.
- Followers don't affect each other at all—a fault in follower A usually doesn't affect the functionality of follower B.
- You can make developers work in parallel on various classes.

22.3 Disadvantages

One potential disadvantage of this design pattern is performance. The source class doesn't know much about its observers. Therefore, every time something important happens within the subject, it loops through the observers and notifies them. Out of three observers, two might ignore the message, and only one of them might be interested in it. From a technical standpoint, this is fine. However, each irrelevant observer consumes a certain runtime to evaluate the message and reject it. That's no big deal if you have 3 observers, but if you have 300 observers, this might start to be a performance problem. In such a case, you can store the class names of your observers in a Z-table and add some columns containing your business rules. That way, for each message, the source class can select, for example, only the 5 out of 300 classes that are really interested in the event and notify only them.

Another potential performance gap is the data traffic between the source class and observers. If the observer needs to pass a few simple variables only, this is no big deal. However, if you have huge internal tables with millions of lines of data, you might want to start considering if passing all the data to observers is a good idea or not.

Regarding this source class, there are two common models:

- **The push model**
  The source class simply passes all relevant data to objects. Theoretically, the source class doesn't know anything about its observers or what kind of data they need. Therefore, the source class needs to pass all the data to them. The relatively simpler way of the two models, the push model may be preferable if the data and expected observer volume is low.
- **The pull model**
  The source class doesn't pass any data to objects, instead providing an instance of itself while having a bunch of `GET_*` methods. Each observer "pulls" the data it needs from those methods and no more. This model may be preferable if the data and expected observer volume is high.

22.4 Multiple Subjects

Occasionally, you might have a case where an observer class needs to observe multiple subjects. In that case, the observer class would need to differentiate the source of the message. Adding the ID of the source subject—or even better, a reference to the subject itself—makes the task easier for observers.

Here is the notification interface from our case study again.

```
INTERFACE zif_target_observer.
  PUBLIC .

  CLASS-METHODS notify
    IMPORTING
      !iv_vkorg TYPE vkorg
      !it_data TYPE zsdtt_target.

ENDINTERFACE.
```

Listing 22.5 Case Study Notification Interface

In our initial case study, this kind of notification would only be produced by the sample application ZTARGET.

What if we need to raise the same notification from an alternative application, ZTARGET2? The catch is that we may need different observers with different purposes. Some observers might need to observe ZTARGET only, some might need to observer ZTARGET2 only, and some might need to observer them both but differentiate the source of the notification.

To fulfill this requirement, we'll simply add the source of the notification to the interface, as demonstrated in Listing 22.6.

```
INTERFACE zif_target_observer.
  PUBLIC .

  CLASS-METHODS notify
    IMPORTING
      !iv_src TYPE zd_source
      !iv_vkorg TYPE vkorg
      !it_data TYPE zsdtt_target.

ENDINTERFACE.
```

Listing 22.6 Notification Source Added to the Interface

When ZTARGET produces a notification, it will simply pass the value ZTARGET to IV_SRC. When ZTARGET2 produces a notification, it will simply pass the value ZTARGET2 to IV_SRC. That way, the observer can choose its behavior based on the source of the notification. Depending on its purpose, an observer can either evaluate or ignore this information.

22.5 Related Patterns

If the behaviors of some observers are very similar, you can take advantage of the template method design pattern (Chapter 26) and derive similar observers from a common abstract class that implements your observer interface.

If the behaviors of your observers are complex and you need them to interact, you can combine observer with other design patterns. If the interaction is limited to sharing variables, you can implement the property container design pattern (Chapter 16) and make the container object part of the observer interface. That

way, all observers can share their variables as needed. If the interaction is more complex than that and involves a certain execution order, you can implement the mediator design pattern (Chapter 20) and let the mediator manage the observer traffic based on business rules.

22.6 Summary

Observer is the Twitter of design patterns. If events in a class concern some other classes as well, the observer classes can follow to the source class to get notifications. This approach ensures that classes don't perform tasks beyond their purpose and reduces their interdependency.

As the number of observers increase, performance-related measures might need to be taken. Filtering unrelated observers out is a simple but effective approach. If the data traffic volume is too large, we recommend using the pull model to reduce the memory footprint.

Similar observers can typically be derived from a template abstract class. Data can be shared among observers via a property container. The subject class often makes a good singleton.

Web services like SOAP (Simple Object Access Protocol) or REST are web-based applications with preannounced interfaces where requests from clients wait. Whenever a call is made, the service performs the task as programmed. The servant design pattern follows a similar logic. By building a class to provide a certain functionality, client classes can possess that functionality by consuming the servant class.

23 Servant

To grasp the necessity and usefulness of the servant design pattern, think about web services. A web service is basically a collection of methods that reside on a web server, ready to be called. Whenever a client calls the service using the provided interface, the web service executes some operation and optionally (but often) returns a value.

With this approach, you ensure that the web service is independent from the clients; the service doesn't need to know anything about them. All you need to do is provide an interface, and whoever needs to perform that specific task can call the service.

The servant design pattern follows the same logic: Whenever you have several methods that need to be shared among multiple unrelated classes, you can put them inside of a servant class instead of copying and pasting them into each and every client. In other words, instead of building a web service, you build a servant class that is accessible by any client in need.

If the functionality is rather simple, you can get away with writing static methods into a toolkit class. However, if you need multiple methods to achieve your desired functionality, providing a servant class is a cleaner approach. A servant class also allows you to give the class a name that describes what it does. Toolkit classes can easily get bloated in no time, and programmers may have a hard time finding a certain method.

We will start off with the case study of an address builder, where we need to create preformatted addresses for customers, vendors, and SAP users. Instead of

repeating the address formatting code in various classes, we will centralize the functionality in a servant class. We will also discuss the possible extensions of the case study, followed by related patterns.

23.1 Case Study: Address Builder

In this example, we need to build a program that takes raw address data and transforms it into nicely formatted text in three formats: short, medium, and long. However, we need this functionality for customer, vendor, and user addresses.

A programmer without an awareness of the servant design pattern could code three methods into the customer class as demonstrated in Listing 23.1.

```
CLASS zcl_customer DEFINITION
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.
    " Some irrelevant data & method definitions
    METHOD get_short_address RETURNING VALUE(rv_adr) TYPE string.
    METHOD get_medium_address RETURNING VALUE(rv_adr) TYPE string.
    METHOD get_long_address RETURNING VALUE(rv_adr) TYPE string.
  PRIVATE SECTION.
  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_customer IMPLEMENTATION.

  METHOD get_short_address.
    " Reads KNA1, ADR*, etc.
    " Builds rv_adr in the short format
  ENDMETHOD.

  METHOD get_medium_address.
    " Reads KNA1, ADR*, etc.
    " Builds rv_adr in the medium format
  ENDMETHOD.

  METHOD get_long_address.
    " Reads KNA1, ADR*, etc.
    " Builds rv_adr in the long format
  ENDMETHOD.

ENDCLASS.
```

Listing 23.1 Customer Class without a Servant Class

Now that the customer class is finished, he/she would move forward to the vendor class and mostly copy and paste the code (as you can see in Listing 23.2), changing only a few lines.

```
CLASS zcl_vendor DEFINITION
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.
    " Some irrelevant data & method definitions
    METHOD get_short_address RETURNING VALUE(rv_adr) TYPE string.
    METHOD get_medium_address RETURNING VALUE(rv_adr) TYPE string.
    METHOD get_long_address RETURNING VALUE(rv_adr) TYPE string.
  PRIVATE SECTION.
  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_vendor IMPLEMENTATION.

  METHOD get_short_address.
    " Reads LFA1, ADR*, etc.
    " Builds rv_adr in the short format - similar to ZCL_CUSTOMER
  ENDMETHOD.

  METHOD get_medium_address.
    " Reads LFA1, ADR*, etc.
    "Builds rv_adr in the medium format - similar to ZCL_CUSTOMER
  ENDMETHOD.

  METHOD get_long_address.
    " Reads LFA1, ADR*, etc.
    " Builds rv_adr in the long format - similar to ZCL_CUSTOMER
  ENDMETHOD.

ENDCLASS.
```

Listing 23.2 Vendor Class without a Servant Class

Finally, he/she would do the same for the user class, as demonstrated in Listing 23.3.

```
CLASS zcl_user DEFINITION
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.
    " Some irrelevant data & method definitions
    METHOD get_short_address RETURNING VALUE(rv_adr) TYPE string.
    METHOD get_medium_address RETURNING VALUE(rv_adr) TYPE string.
    METHOD get_long_address RETURNING VALUE(rv_adr) TYPE string.
  PRIVATE SECTION.
```

```
  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_user IMPLEMENTATION.

  METHOD get_short_address.
    " Reads USR*, ADR*, etc.
    " Builds rv_adr in the short format - similar to ZCL_CUSTOMER
  ENDMETHOD.

  METHOD get_medium_address.
    " Reads USR*, ADR*, etc.
    "Builds rv_adr in the medium format - similar to ZCL_CUSTOMER
  ENDMETHOD.

  METHOD get_long_address.
    " Reads USR*, ADR*, etc.
    " Builds rv_adr in the long format - similar to ZCL_CUSTOMER
  ENDMETHOD.

ENDCLASS.
```

Listing 23.3 User Class without a Servant

For those of you who have internalized the principles of object-oriented design, this architecture should be extremely disturbing. The algorithm for building `RV_ADR` is repeated in every single class, which is an undesirable case of copy-and-paste programming.

To come at this problem in another way, you could also plant three methods into a toolkit class (like `BUILD_SHORT_ADDRESS`, `BUILD_MEDUIM_ADDRESS`, and `BUILD_LONG_ADDRESS`). In this solution, each client would call the corresponding method. For example, `ZCL_CUSTOMER` would look like Listing 23.4.

```
CLASS zcl_customer DEFINITION
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.
    " Some irrelevant data & method definitions
    METHOD get_short_address RETURNING VALUE(rv_adr) TYPE string.
    METHOD get_medium_address RETURNING VALUE(rv_adr) TYPE string.
    METHOD get_long_address RETURNING VALUE(rv_adr) TYPE string.
  PRIVATE SECTION.
  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_customer IMPLEMENTATION.
```

```
  METHOD get_short_address.
    " Reads KNA1, ADR*, etc.
    " Transform the data into a common structure
    rv_adr = zcl_toolkit=>build_short_address( ls_common ).
  ENDMETHOD.

  METHOD get_medium_address.
    " Reads KNA1, ADR*, etc.
    " Transform the data into a common structure
    rv_adr = zcl_toolkit=>build_medium_address( ls_common ).
  ENDMETHOD.

  METHOD get_long_address.
    " Reads KNA1, ADR*, etc.
    " Transform the data into a common structure
    rv_adr = zcl_toolkit=>build_long_address( ls_common ).
  ENDMETHOD.

ENDCLASS.
```

Listing 23.4 Customer Class Using a Toolkit

`ZCL_VENDOR` and `ZCL_USER` would be modified the same way. While this approach is one degree better than repeating the same code in all three classes, it still has the following disadvantages:

- Toolkit classes are useful for small utility methods. However, if you overuse them, toolkit classes tend to transform into bloated static method containers, and the methods tend to have crowded and confusing interfaces. This approach is not a good idea when you are looking ways to decentralize responsibilities and build small, virtual Lego pieces to build applications. Overused toolkit classes result in increases in the interdependency of objects, which is not advisable.
- Assume that you have a new address formats in the future: extremely short and very long addresses. You would need to write two new toolkit methods and two new client methods each for customer, vendor, and user class. This approach produces a lot of avoidable work.

With two inferior solutions behind us, let's now move on to the servant design pattern. Returning to our comparison between servant and web services, if building addresses was a functionality we would like to provide to multiple applications on multiple platforms, we would write a web service and publish it on a web server. The service would have three methods:

- BUILD_SHORT_ADDRESS
- BUILD_MEDIUM_ADDRESS
- BUILD_LONG_ADDRESS

Any application that needed a formatted address would call one of those methods, pass the raw address data, and get a nicely formatted address string.

In the servant design pattern, we will semantically do the same. However, we will write a (servant) class instead of a web service. Listing 23.5 demonstrates what the servant class would look like.

```
CLASS zcl_address_servant DEFINITION
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.

    METHOD constructor
      IMPORTING !io_resident TYPE REF TO zif_resident.

    METHOD build_short_address
      RETURNING VALUE(rv_adr) TYPE string.

    METHOD build_medium_address
      RETURNING VALUE(rv_adr) TYPE string.

    METHOD build_long_address
      RETURNING VALUE(rv_adr) TYPE string.

  PRIVATE SECTION.
  PROTECTED SECTION.
    DATA go_resident TYPE REF TO zif_resident.

ENDCLASS.

CLASS zcl_address_servant IMPLEMENTATION.

  METHOD constructor.
    go_resident = io_resident.
  ENDMETHOD.

  METHOD build_short_address.
    DATA(ls_raw) = go_resident->get_raw_address_data( ).
    " Builds rv_adr in the short format
  ENDMETHOD.

  METHOD build_medium_address.
```

```
    DATA(ls_raw) = go_resident->get_raw_address_data( ).
    " Builds rv_adr in the medium format
  ENDMETHOD.

  METHOD build_long_address.
    DATA(ls_raw) = go_resident->get_raw_address_data( ).
    " Builds rv_adr in the long format
  ENDMETHOD.

ENDCLASS.
```

Listing 23.5 Servant Class

You probably have noticed that the class is taking advantage of an interface: `IF_RESIDENT`. Just like a web service has a WSDL (Web Service Definition Language)/SOAP interface to pass data between the server and client, the servant class also needs an interface to pass data between itself and client classes. `IF_RESIDENT` is our interface in this scenario, which would look like Listing 23.6.

```
INTERFACE zif_resident
  PUBLIC .

  TYPES: BEGIN OF t_raw_address,
    name TYPE string,
    phone TYPE string,
    street TYPE string,
    " Some further fields
  END OF t_raw_address.

  METHODS get_raw_address_data
    RETURNING
      VALUE(rs_raw) type t_raw_address.

ENDINTERFACE.
```

Listing 23.6 Resident interface

`CL_CUSTOMER`, `CL_VENDOR`, and `CL_USER` will implement this interface. After that, we can build customer, vendor, and user addresses using `CL_ADDRESS_SERVANT`. This class will be the central service to transform address data, and the code to transform address data strings will reside there—instead of being duplicated in client classes or stored in potentially bloated toolkits.

For a better overview of this program, see the Unified Modeling Language (UML) diagram in Figure 23.1.

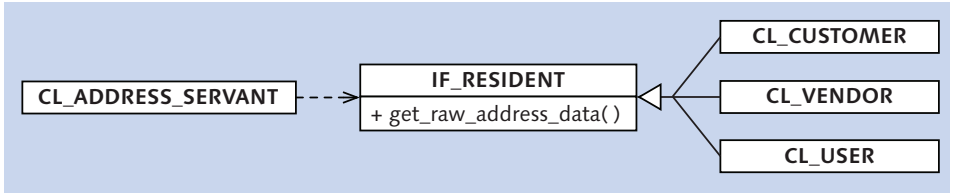


Figure 23.1 Servant Architecture

In a real-world situation, the interface would probably contain more methods. However, as always, we want to keep things simple to improve your focus on the main subject.

Now let's take a look at the new version of `CL_CUSTOMER` in Listing 23.7.

```
CLASS zcl_customer DEFINITION
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.
    " Some irrelevant data & method definitions
    INTERFACES zif_resident.
  PRIVATE SECTION.
  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_customer IMPLEMENTATION.

  METHOD zif_resident~get_raw_address_data.
    " Reads KNA1, ADR*, etc.
    " Write data into rs_raw
  ENDMETHOD.

ENDCLASS.
```

Listing 23.7 Customer Class Implementing Resident Interface

This class is easy to follow and contains no code repetition at all. The same will apply to `CL_VENDOR`, as you can see in Listing 23.8.

```
CLASS zcl_vendor DEFINITION
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.
    " Some irrelevant data & method definitions
    INTERFACES zif_resident.
  PRIVATE SECTION.
  PROTECTED SECTION.
```

```
ENDCLASS.

CLASS zcl_vendor IMPLEMENTATION.

  METHOD zif_resident~get_raw_address_data.
    " Reads LFA1, ADR*, etc.
    " Write data into rs_raw
  ENDMETHOD.

ENDCLASS.
```

Listing 23.8 Vendor Class Implementing Resident Interface

Unsurprisingly, we will have a similar situation in `CL_USER`, as demonstrated in Listing 23.9.

```
CLASS zcl_user DEFINITION
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.
    " Some irrelevant data & method definitions
    INTERFACES zif_resident.
  PRIVATE SECTION.
  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_user IMPLEMENTATION.

  METHOD zif_resident~get_raw_address_data.
    " Reads USR*, ADR*, etc.
    " Write data into rs_raw
  ENDMETHOD.

ENDCLASS.
```

Listing 23.9 User Class Implementing Resident Interface

A client application that needs to transform a customer address will make a simple call, as shown in Listing 23.10.

```
DATA lo_resident TYPE REF TO zif_resident.

DATA(lo_customer) = NEW zcl_customer( gv_kunnr ).
lo_resident ?= lo_customer.

DATA(lo_servant) = NEW zcl_address_servant( lo_resident ).
DATA(lv_address) = lo_servant->build_medium_address( ).
```

Listing 23.10 Accessing Customer and Servant Classes

As you can see, the class CL_ADDRESS_SERVANT is very similar to a web service and results in the following:

- Any class with address information can implement IF_RESIDENT and make itself available to CL_ADDRESS_SERVANT.
- Any client that needs to transform an address can call ZCL_ADDRESS_SERVANT just like a web service and make use of this functionality.

Following this logic, the same structure can be applied for the vendor and user classes, as you can see in Listing 23.11.

```
DATA lo_resident TYPE REF TO zif_resident.

* Sample for vendor
DATA(lo_vendor) = NEW zcl_vendor( gv_lifnr ).
lo_resident ?= lo_vendor.

DATA(lo_servant) = NEW zcl_address_servant( lo_resident ).
DATA(lv_address) = lo_servant->build_medium_address( ).

* Sample for user
DATA(lo_user) = NEW zcl_user( gv_bname ).
lo_resident ?= lo_user.

lo_servant = NEW zcl_address_servant( lo_resident ).
lv_address = lo_servant->build_medium_address( ).
```

Listing 23.11 Accessing Vendor, User, and Servant Classes

Note

The names of subclasses (ZCL_CUSTOMER, ZCL_VENDOR, and ZCL_USER) are hardcoded in this example. See Appendix B on subclass determination for alternative approaches.

The sample code in Listing 23.11 shows how easily servant classes can be consumed. Even if we need further resident classes in the future, such as CL_EMPLOYEE or CL_PLANT, the simple code template to format their addresses will remain the same.

23.2 Extensions

Now that you have grasped the core logic of this design pattern, let's look at ways to extend the functionality it provides.

You can have multiple servants accepting the same interface and even change them during runtime. Just as CL_ADDRESS_SERVANT accepts IF_RESIDENT, you can have further classes like CL_ZIP_VALIDATOR and CL_SMS_SENDER that also accept IF_RESIDENT. That way, you can add further services without touching CL_CUSTOMER, CL_VENDOR, and CL_USER.

If you need even more flexibility, a servant can also accept multiple interfaces. In our simple example, CL_ADDRESS_SERVANT accepts one single interface: IF_RESIDENT. Everything in the class is based on the data returned by IF_RESIDENT. However, nothing bars a servant class from accepting multiple interfaces. For instance, CL_ADDRESS_SERVANT might have accepted a secondary interface called IF_TAX_PAYER to provide a method like VALIDATE_TAX_CODE. In that case, CL_VENDOR and CL_CUSTOMER would have implemented IF_TAX_PAYER to return their tax data, and any client might call the validation method whenever they need to—just like calling a web service. Please remember that the methods in the interface aren't limited to returning data—they can perform tasks as well.

23.3 Related Patterns

If the functionality and structures of two classes overlap to a large extent, the template method design pattern can be used to prevent recoding very similar methods in each concrete class. If the functionality overlaps but the structures are not similar at all, the overlapping functionality can be coded through a servant class. In this case, each concrete class (which differs from each other) can implement a common interface and share the functionality.

The visitor design pattern (Chapter 27) may seem to have a similar purpose to the servant pattern at first. However, each is actually preferred for different cases. If you want to add further functionality to a single class, use the visitor design pattern. If you want to share a single functionality among multiple classes, use the servant design pattern.

23.4 Summary

The servant design pattern can be considered the class equivalent of a web service. Certain operations are hosted by a servant class, and client classes can consume those operations just like consuming a web service.

For simple one-method operations, the static methods of a toolkit class can be sufficient. Servant really shines in more complex situations, where multiple related methods are involved.

You can have multiple servants sharing the same interface for similar but different purposes, thus reducing complexity in client classes. In fact, a single servant can accept multiple interfaces for different purposes. The flexibility works both ways.

The state design pattern focuses on getting rid of repetitive conditions in multiple methods of a class. If you find yourself typing the same `IF` *or* `CASE` *conditions over and over again, chances are you'll find a more flexible approach using the state design pattern. In this design pattern, you define a new class for each condition, all of which share the same interface.*

24 State

As one of the most popular design patterns, state provides immense simplification to any eligible class. If you are repeating the same `IF…ELSE…ENDIF` or `CASE` statements in various methods of a class, switching to the state design pattern might be a good choice. State saves you from creating lots of identical conditional code.

We will start off with a case study where our central class needs to behave differently based on the authorization level of the user. Instead of checking the authorization level in each relevant method of the class, we will simplify the task by taking advantage of a state interface. We will continue with the discussion of the advantages of state and its related patterns.

24.1 Case Study: Authorization-Based Class Behavior

Often, you will have a global flag, and the functionality of various methods would depend on how the flag is set. A typical example is a class with behavior dependent on the authorization level of the user. Listing 24.1 demonstrates what the relevant part of such a class would look like.

```
METHOD read_data.

  CASE gv_auth_level.
    WHEN c_auth_high. read_all_data( ).
    WHEN c_auth_low. read_small_data( ).
    WHEN c_auth_none. RETURN.
```

```
    ENDCASE.

  ENDMETHOD.

  METHOD save_data.

    CASE gv_auth_level.
      WHEN c_auth_high. save_all_data( ).
      WHEN c_auth_low. save_small_data( ).
      WHEN c_auth_none. RETURN.
    ENDCASE.

  ENDMETHOD.
```

Listing 24.1 Authorization-Dependent Class Behavior

In Listing 24.1, the behaviors of `READ_DATA` and `SAVE_DATA` depend on the authorization level of the user, as follows:

- If the user has high authorization, he/she can read and save everything.
- If the user has low authorization, he/she can read and save only part of the data.
- If the user has no authorization, he/she can't read or save anything at all.

In the code in Listing 24.1, we have achieved this functionality by placing a `CASE` statement into corresponding methods. However, this approach is inefficient due to the following reasons:

- The same `CASE` logic appears in multiple methods. If a new authorization level pops up someday, the developer needs to modify them all, raising the development time and the risk of ruining something that was already working well.
- Inflation of methods can easily occur. Instead of having a single `READ_DATA` method, we have three: `READ_DATA`, `READ_ALL_DATA`, and `READ_SMALL_DATA`. As a result, the class is more complicated than necessary. Not defining the extra methods and turning `READ_DATA` into a big ugly method is even worse, and the same applies to `SAVE_DATA`. If we had ten methods corresponding to each authorization level, we would end up having thirty methods in total.

How do we fix this? As you have probably guessed, the state design pattern is going to save the day. To apply the pattern, we need to take three simple steps:

1. Define an interface containing the methods that should act differently depending on the state. In our example, the interface will contain two methods: `READ_DATA` and `SAVE_DATA`.

2. Create an implementation of the interface for each possible state. In our example, we will have three implementations—CL_HIGH, CL_LOW, and CL_NONE—each corresponding to an authorization state.
3. Set up the main class so it uses the corresponding state class depending on the state itself.

This solution might be hard to imagine, but you will understand perfectly once you see the Unified Modeling Language (UML) diagram in Figure 24.1 and the sample code for each component.

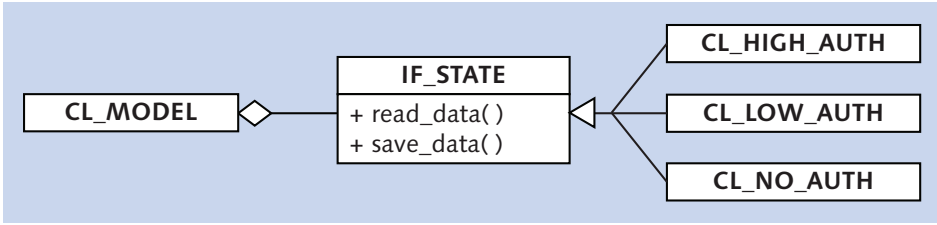


Figure 24.1 State Design Pattern Architecture

Listing 24.2 demonstrates what the interface will look like.

```
INTERFACE zif_state
  PUBLIC .

  METHODS read_data
    RETURNING
      VALUE(rt_data) TYPE cl_model=>tt_data.

  METHODS save_data
    IMPORTING
      !it_data TYPE cl_model=>tt_data.

ENDINTERFACE.
```

Listing 24.2 State Interface

Have you noticed what this interface is missing? Exceptions. Normally, READ_DATA and SAVE_DATA should contain at least one exception each, which can be raised by the implementation classes if anything goes wrong. However, we have intentionally omitted exceptions for the sake of simplicity.

Now, we will create an implementation for each possible state. Listing 24.3 demonstrates the class for users that have a high authorization level.

```
CLASS zcl_high_auth DEFINITION
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.
    INTERFACES zif_state.
  PRIVATE SECTION.
  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_high_auth IMPLEMENTATION.

  METHOD zif_state~read_data.
    "Read "all" data from the database table and return
  ENDMETHOD.

  METHOD zif_state~save_data.
    "Replace "all" data in the database table with IT_DATA
  ENDMETHOD.

ENDCLASS.
```

Listing 24.3 State Class for High-Authorization Users

To simplify the example and make it more understandable, we have replaced huge chunks of executable code with simple placeholder comments. Following the same approach, let's move forward and inspect the class in Listing 24.4, which will work for users with a low authorization level.

```
CLASS zcl_low_auth DEFINITION
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.
    INTERFACES zif_state.
  PRIVATE SECTION.
  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_low_auth IMPLEMENTATION.

  METHOD zif_state~read_data.
    " Read "partial" data from the database table and return
  ENDMETHOD.

  METHOD zif_state~save_data.
    " Replace "partial" data in the database table with IT_DATA
  ENDMETHOD.

ENDCLASS.
```

Listing 24.4 State Class for Low-Authorization Users

As you might expect, next comes the class for those with no authorization at all, as shown in Listing 24.5.

```
CLASS zcl_no_auth DEFINITION
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.
    INTERFACES zif_state.
  PRIVATE SECTION.
  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_no_auth IMPLEMENTATION.

  METHOD zif_state~read_data.
    RETURN.
  ENDMETHOD.

  METHOD zif_state~save_data.
    RETURN.
  ENDMETHOD.

ENDCLASS.
```

Listing 24.5 State Class for No-Authorization Users

In a real-world scenario, you would actually raise an exception if the user didn't have any authorization. To keep things simple, we have left the `RAISE EXCEPTION` code out.

At this point, we have a distinct class for each authorization state, each of which behaves completely differently. You can see that, instead of filling up `CL_MODEL` with methods or chunks of code, we have distributed the load to the corresponding state classes, which share the same interface. How are we going to manage these state classes in the model class? You'll find the answer in Listing 24.6.

```
CLASS zcl_model DEFINITION
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.

    " Some constants, etc.

    DATA go_state TYPE REF TO zif_state.

    METHODS set_state
      IMPORTING
        !iv_auth_level TYPE zbcd_auth_level.
```

```
    " Some more methods, etc.

  PRIVATE SECTION.
  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_model IMPLEMENTATION.

  METHOD set_state.

    DATA: lv_clsname TYPE seoclsname,
          lo_object TYPE REF TO object.

    CASE iv_auth_level.
      WHEN c_auth_high. lv_clsname = 'ZCL_HIGH_AUTH'.
      WHEN c_auth_low. lv_clsname = 'ZCL_LOW_AUTH'.
      WHEN c_auth_none. lv_clsname = 'ZCL_NO_AUTH'.
    ENDCASE.

    CREATE OBJECT lo_object TYPE (lv_clsname).
    go_state ?= lo_object.

  ENDMETHOD.

ENDCLASS.
```

Listing 24.6 Stateful Model Class

The names of the subclasses (`ZCL_*_AUTH`) are hardcoded in this example. In a real-world situation, we definitely wouldn't use hardcoded class names. The class names were provided as literals simply to make the code more understandable. Instead, in the real world, we could read the class name from a customizing table, which would look like Table 24.1.

AUTH_LEVEL	CLASS_NAME
HIGH	ZCL_HIGH_AUTH
LOW	ZCL_LOW_AUTH
NONE	ZCL_NO_AUTH

Table 24.1 State Class per Authorization Level Stored in a Custom Table

Note

See Appendix B on subclass determination for alternative approaches to hardcoding the names of subclasses.

After we call `SET_STATE` for a given authorization level, `GO_STATE` will point to the class containing the code for that level. As a result, the following is true:

- If we run `SET_STATE( C_AUTH_HIGH )`,
  - `GO_STATE->READ_DATA` will execute `ZCL_HIGH_AUTH~READ_DATA`.
  - `GO_STATE->SAVE_DATA` will execute `ZCL_HIGH_AUTH~SAVE_DATA`.
- If we run `SET_STATE( C_AUTH_LOW )`,
  - `GO_STATE->READ_DATA` will execute `ZCL_LOW_AUTH~READ_DATA`.
  - `GO_STATE->SAVE_DATA` will execute `ZCL_LOW_AUTH~SAVE_DATA`.
- If we run `SET_STATE( C_AUTH_NONE )`, the user has no authorizations, and
  - `GO_STATE->READ_DATA` will execute `ZCL_NO_AUTH~READ_DATA`.
  - `GO_STATE->SAVE_DATA` will execute `ZCL_NO_AUTH~SAVE_DATA`.

This is a very powerful functionality indeed.

Note

Who said that an object could have only one state? In more complicated situations, you may implement multiple states to your object. For example, you may have one state regarding authorization level (high, low, etc.) and one state regarding the type of your document (order, delivery, invoice, etc.).

24.2 Advantages

This pattern will kill off conditional statements and reduces duplication as well. Having the same `if/then` statements over and over again can lead to an increased risk of creating an ugly bug—eventually, a future developer will forget to add his/her new condition to one of those statements.

This pattern also makes it easier to extend the functionality of the class. Got a new condition? Not a problem—just add a new state class, and you are good to go. No need to touch the well-tested legacy class at all; if something goes wrong in the new state, former states probably won't be affected, which is great in terms of damage control.

Some further advantages include:

- The size of the main class shrinks, making things simpler and more manageable.

- Developers can work in parallel on different classes.
- A technical difficulty regarding an authorization level (in our example) doesn't affect users with different authorization levels at all.

24.3 Related Patterns

Remember that states are classes. Therefore, the general principles of object-oriented programming (OOP) apply such as inheritance, encapsulation, and polymorphism, discussed in Appendix C. If some of your concrete state classes are very similar, feel free to centralize their common functionality by using the template method design pattern (Chapter 26), which should implement your state interface.

As you read and learn, the state design pattern may look similar to strategy design pattern (Chapter 25). There is some truth to this, but the most significant difference is that a strategy object is usually bound only once and remains where it is until the end of the execution. State, on the other hand, provides a flexible structure where the binding can (and probably will) be changed dynamically during runtime.

Although not a general rule, state objects are often singletons (Chapter 8). If the number of state objects is very high and they have partially shared states, you can consider applying the flyweight design pattern (Chapter 15) to reduce the memory footprint.

If you have multiple states and need to make them work in harmony, check if bridge design pattern (Chapter 10) can help.

24.4 Summary

The state design pattern is useful when you have to repeat the same conditional `IF` or `CASE` chain in multiple methods of the same class. Instead of having repetitive conditions, you end up having a distinct class for each condition.

This approach not only simplifies the code; it also simplifies managing errors and addressing new conditions. The same model class may have multiple states. Similar states can centralize common functionality into parent abstract classes.

Strategy comes in handy when you have alternative algorithms in your pocket and need to pick one during runtime. This pattern suggests creating a common interface and coding each algorithm into a class implementing the interface, which will enable you to dynamically cast any of the classes against the interface during runtime.

25 Strategy

Strategy is one of the design patterns you will probably end up using frequently. This pattern is used when you have multiple algorithms sharing the same interface. Once implemented correctly, your application can determine which algorithm to execute during runtime, thus empowering your application to juggle algorithms as needed.

If you have five alternative algorithms, you are going to need to create an interface and five concrete classes. Your central application will dynamically create an object referring to the appropriate class and cast it onto the interface, and the rest of the code will deal with the interface without knowing or caring about what kind of code lies beneath.

That way, you will have a loosely coupled collection of interchangeable classes in your pocket and can use the appropriate class depending on the situation. Adding new classes to the collection is as easy as implementing a new class using the same interface.

Note

The strategy design pattern is sometimes called the *policy design pattern*.

We will start off with a case study where we need to send master data from SAP to various external systems. Each system will naturally have its own distinct integration standard, but we will overcome this difficulty by taking advantage of the strategy design pattern. The chapter will continue with discussions about the advantages of the pattern, ways of passing data to the strategy object, the use of optional strategies, the power of abstract classes in strategies, and related patterns.

25.1 Case Study: Sending Material Master Data

Our example will deal with an integration program where you have to send material master data to various systems. However, the systems differ in terms of content and format: One system expects you to send MATNR and MEINS in the form of a CSV file, while the other one expects you to send MATNR, MATKL, and MEINS in the form of an XML file.

Using the classical approach, you would develop a custom program where the user selects the target system via radio buttons on the selection screen. The program would read material master data from `MARA` and contain a `FORM` for each target system. Listing 25.1 demonstrates what this program and its code would look like.

```
PARAMETERS:
   p_sys01 RADIOBUTTON GROUP rg1 DEFAULT 'X',
   p_sys02 RADIOBUTTON GROUP rg1.

 PERFORM main.

 FORM main.

   DATA lt_mara TYPE tt_mara.

   SELECT matnr matkl meins
     INTO CORRESPONDING FIELDS OF TABLE lt_mara
     FROM mara.

   IF p_sys01 EQ abap_true.
     PERFORM send_to_sys01 USING lt_mara.
   ENDIF.

   IF p_sys02 EQ abap_true.
     PERFORM send_to_sys02 USING lt_mara.
   ENDIF.

 ENDFORM.
```

Listing 25.1 Classical Program to Send Material Data

The problem with this approach is that you have to modify your central program for each and every new system you might have, and a coding error for a new system might also have a negative impact on existing systems. Another problem is that you would have a hard time assigning different systems to different developers. Although these assignments might work with separate include files, you can't activate and test the main program easily before everything else is finished.

That's where the strategy design pattern comes forward. If you look carefully, you will notice that SEND_TO_SYS01 and SEND_TO_SYS02 share the same interface—in other words, they both import an internal table of the same type (TT_MARA). The only thing that differs is what they do internally. SEND_TO_SYS01 will convert the raw data into a CSV file and send it to system 1, while SEND_TO_SYS02 will convert the raw data into an XML file and send to system 2.

In general, if you have different algorithms sharing the same interface, you should consider the implementation of a strategy design pattern. Let's take a look at the Unified Modeling Language (UML) diagram of our solution incorporating this design pattern in Figure 25.1.

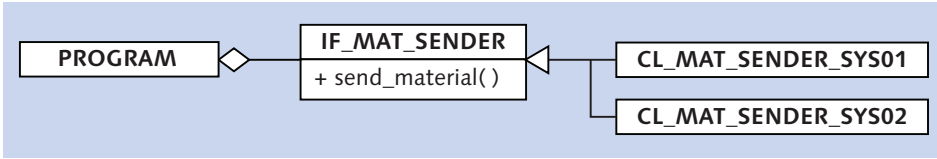


Figure 25.1 Architecture for the Strategy Design Pattern

In our solution, we will store common parameters inside of an interface and derive alternative implementations accordingly. For each target system, we will have a separate class implementation. That way, we can assign different classes to different developers who can work in parallel, and once finished, the main program won't require much work even if new requirements arise. If a new system is added, all we need to do is implement CL_MAT_SENDER_SYS03. Knowing that the rest of the system works already, you don't have to test existing components again.

Let's move forward and see what each element of the UML diagram will look like in ABAP code. Listing 25.2 demonstrates how the interface would be written.

```
INTERFACE zif_mat_sender
  PUBLIC .

  TYPES: BEGIN OF t_mara,
    matnr TYPE mara-matnr,
    matkl TYPE mara-matkl,
    meins TYPE mara-meins,
  END OF t_mara,
  tt_mara TYPE STANDARD TABLE OF t_mara WITH DEFAULT KEY.

  METHODS send_material
```

```
    IMPORTING
      !it_mara type tt_mara.

ENDINTERFACE.
```

Listing 25.2 Interface to Send Material Data

Moving forward, we can implement our first class targeting system 1, as seen in Listing 25.3.

```
CLASS zcl_mat_sender_sys01 DEFINITION
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.
    INTERFACES zif_mat_sender.
  PRIVATE SECTION.
  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_mat_sender_sys01 IMPLEMENTATION.

  METHOD zif_mat_sender~send_material.
    " Some code to create CSV file from it_mara
    " Some code to send CSV file
  ENDMETHOD.

ENDCLASS.
```

Listing 25.3 Class to Send Data to System 1

The class for system 2 doesn't look too different, as you can see in Listing 25.4.

```
CLASS zcl_mat_sender_sys02 DEFINITION
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.
    INTERFACES zif_mat_sender.
  PRIVATE SECTION.
  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_mat_sender_sys02 IMPLEMENTATION.

  METHOD zif_mat_sender~send_material.
    " Some code to create XML file from it_mara
    " Some code to send XML file
  ENDMETHOD.

ENDCLASS.
```

Listing 25.4 Class to Send Data to System 2

So far, so good. Now the key question is what our client program in Transaction SE38 going to look like, which you can see in Listing 25.5.

```
PARAMETERS:
   p_system TYPE seoclsname OBLIGATORY.

 PERFORM main.

 FORM main.

   DATA:
    lo_obj    TYPE REF TO object,
    lo_sender TYPE REF TO zif_mat_sender,
    lt_mara   TYPE zif_mat_sender=>tt_mara.

   SELECT matnr maktx matkl meins
     INTO CORRESPONDING FIELDS OF TABLE lt_mara
     FROM mara.

   CREATE OBJECT lo_obj TYPE (p_system).
   lo_sender ?= lo_obj.
   lo_sender->send_material( lt_mara ).

 ENDFORM.
```

Listing 25.5 Client Program Using Strategy

Let's go over the code in Listing 25.5 step-by-step and see what we are doing. At the selection screen, we asked the user for a class name using the following code:

```
PARAMETERS:
   p_system TYPE seoclsname OBLIGATORY.
```

Logically, the user is supposed to enter either `ZCL_MAT_SENDER_SYS01` or `ZCL_MAT_SENDER_SYS02` depending on the system he/she is targeting. In other words, the names of subclasses are determined on the selection screen.

For the sake of simplicity, we didn't include any defensive programming by checking class names nor did we provide a search help. In a real-world situation, we would store the names of the classes in a custom table and make the user select a more readable value instead of a technical class name, as seen in Table 25.1.

SYSTEM_ID	CLASS_NAME
SYSTEM_ONE	ZCL_MAT_SENDER_SYS01
SYSTEM_TWO	ZCL_MAT_SENDER_SYS02

Table 25.1 Class Names per Target System

Note

See Appendix B on subclass determination for alternative approaches to using the Z-table.

Next, when the program is executed, the first thing we do is to read the materials to be sent, as follows:

```
SELECT matnr maktx matkl meins
  INTO CORRESPONDING FIELDS OF TABLE lt_mara
  FROM mara.
```

Being object-oriented ABAP developers with design pattern knowledge, we actually wouldn't query the database directly from within the client program as we have done here. Instead, we would implement the MVC design pattern and create a model class to do the job. However, as stated before, we've kept the code simple to focus on the strategy design pattern.

What happens next is the heart of the entire example. First, we create an instance of the class whose name was provided by the user as follows:

```
CREATE OBJECT lo_obj TYPE (p_system).
```

Which class are we creating exactly? We don't know. Logically speaking, the variable `P_SYSTEM` would contain one of the following literals: `ZCL_MAT_SENDER_SYS01` or `ZCL_MAT_SENDER_SYS02`. However, if we later add support for new systems, we might be dealing with `ZCL_MAT_SENDER_SYS62`. The fact is that, as long as the class has implemented `IF_MAT_SENDER`, the class is compatible with our client program.

Next, we have the following code, in which the object in `LO_OBJ` is cast onto `LO_SENDER`:

```
lo_sender ?= lo_obj.
```

At this point, the following can happen:

- If the user entered `ZCL_MAT_SENDER_SYS01`, then `LO_SENDER` will act as an instance of the class `ZCL_MAT_SENDER_SYS01`.
- If the user entered `ZCL_MAT_SENDER_SYS02`, then `LO_SENDER` will act as an instance of the class `ZCL_MAT_SENDER_SYS02`.
- If a future user would enter `ZCL_MAT_SENDER_SYS62`, then `LO_SENDER` would act as an instance of `ZCL_MAT_SENDER_SYS62`.

Our final line goes as follows:

```
lo_sender->send_material( lt_mara ).
```

Unsurprisingly, this line of code would act as follows:

- If the user entered `ZCL_MAT_SENDER_SYS01`, then `LO_SENDER->SEND_MATERIAL` will execute `ZCL_MAT_SENDER_SYS01->SEND_MATERIAL`.
- If the user entered `ZCL_MAT_SENDER_SYS02`, then `LO_SENDER->SEND_MATERIAL` will execute `ZCL_MAT_SENDER_SYS02->SEND_MATERIAL`.
- If a future user would enter `ZCL_MAT_SENDER_SYS62`, then `LO_SENDER->SEND_MATERIAL` would execute `ZCL_MAT_SENDER_SYS62->SEND_MATERIAL`.

Using the same principle, you can juggle algorithms sharing the same interface in any way you like.

25.2 Advantages

The strategy design pattern provides an immense degree of flexibility to our applications. You can switch between alternative algorithms easily, and adding a new one is as simple as creating a new strategy class. You don't have to modify the central, time-tested application or model class at all. Even making a mistake in your new class will only affect the scenarios using your new class—not the entire application. In a sense, strategy automatically provides a certain degree of damage control.

One of the significant nontechnical advantages of the strategy design pattern is that once you agree on the interface, you can split the development of distinct concrete strategy classes among developers to meet a competitive deadline. In a typical case, the lead architect would create an interface and a couple of example strategy classes, and the rest of the team takes them as an example and develops further classes.

25.3 Passing Data to the Strategy Object

One of the core decisions you are going to make is how to pass data to the strategy object. There are two typical approaches:

- Passing data to the strategy interface in the form of internal tables, work areas, variables, objects, etc.
- Passing the context object itself (containing all the data and more) to the strategy interface.

Listing 25.6 demonstrates an example interface in which the data is passed to the strategy object.

```
INTERFACE zif_strategy
  PUBLIC .

  METHODS method1
    IMPORTING
      !it_itab TYPE ztt_sample
      !is_wa TYPE zs_sample
      !iv_var TYPE matnr
    CHANGING
      !cv_var TYPE clike.

ENDINTERFACE.
```

Listing 25.6 Strategy Interface Passing Data

Listing 25.7, on the other hand, demonstrates an example interface in which the object is passed to the strategy object.

```
INTERFACE zif_strategy
  PUBLIC .

  METHODS method1
    IMPORTING
      !io_obj TYPE REF TO zcl_model " Has all variables + more
    CHANGING
      !cv_var TYPE clike.

ENDINTERFACE.
```

Listing 25.7 Strategy Interface Passing Object

If you pass the data, you will make the application and the strategy less dependent; they would be loosely coupled. If you need to reuse the same strategy in an entirely different application, just convert and then pass the data in the new application to the strategy, and it will work like a charm. One disadvantage to this approach is that, if you realize that you need further variables in the strategy, you'll need to modify the interface and possibly drown in a huge run of the where-used list.

If you pass the context object itself, you make sure that you are providing everything the concrete strategy classes might need, and you will probably never need to modify the interface. The disadvantage is that you are hard wiring your strategy with your current application. Future applications may find it difficult to reuse your strategy, because they will have a different context object. The adapters for such a case can be complicated.

The decision between passing the data and passing the object depends on your strategy. If you foresee that the strategy might be reused by future applications, design the interface so that it imports data. If you are 100% sure that the strategy is specific to your current application and won't ever be reused, design the interface so that imports the context object.

25.4 Optional Strategies

At times, you might face a situation where having a strategy is optional. During runtime, your application will check if there is a strategy corresponding to the case in hand. Your application will use a concrete strategy class, if available. Otherwise, some default code is executed.

If you face such a case, we recommend wrapping the default code into a concrete strategy class because ideally it would use the same interface.

For example, you might need a user exit to calculate the sales price of your products. Assuming that we have special discount campaign on certain days of the week, we might come up with a customizing table like Table 25.2 containing our strategy classes for those special days.

DAY_OF_WEEK	PRICE_CLASS
3	ZCL_PRICE_WEDNESDAY
5	ZCL_PRICE_FRIDAY
7	ZCL_PRICE_SUNDAY

Table 25.2 Price Calculation Classes Corresponding to Days of the Week

Since our price classes represent a common price calculation strategy, they would be sharing a common interface like Listing 25.8.

```
INTERFACE zif_price_strategy
  PUBLIC .

  METHODS calculate_price
    IMPORTING
      !is_data TYPE zs_price_import
    EXPORTING
      !es_price TYPE zs_price_export.

ENDINTERFACE.
```

Listing 25.8 Sample Interface for Price Calculation

The following classes could implement this common interface:

- `CL_PRICE_WEDNESDAY` might contain some code where a discount of 15% is applied if multiple quantities of the same product are ordered.
- `CL_PRICE_FRIDAY` might contain some code where a discount of 20% is applied if the customer had ordered the same product last Friday as well.
- `CL_PRICE_SUNDAY` might contain some code where a discount of 10% is automatically applied to all orders.

A code snippet in the user exit would determine the day of the week and modify the sales price, as demonstrated in Listing 25.9.

```
DATA:
  lo_obj TYPE REF TO object,
  lo_str TYPE REF TO zif_price_strategy.

" Determine the day of the week and put into lv_day

SELECT SINGLE price_class
  INTO @DATA(lv_clsname)
  FROM zclass
  WHERE day_of_week EQ @lv_day.

IF sy-subrc EQ 0.
  CREATE OBJECT lo_obj TYPE lv_clsname.
  lo_str ?= lo_obj.

  lo_str->calculate_price(
    EXPORTING is_data = ls_data
    IMPORTING es_price = ls_price
  ).

  " Use data in LS_PRICE to modify user exit data
ENDIF.
```

Listing 25.9 Applying Discount Using Strategy Classes

At this point, our code looks for a strategy class corresponding to the day of the week and calls the corresponding class to set the special price for the given day.

However, our strategy customization in Table 25.2 contains data for Wednesday, Friday, and Sunday only. Therefore, the user exit demonstrated in Listing 25.9 will do absolutely nothing for other days.

If this is the desired situation, fine. However, we might have a requirement where we need to apply a default discount of 5% for the rest of the week.

First of all, we need to create a new strategy class `CL_PRICE_DEFAULT`, which will implement `IF_PRICE_STRATEGY` and apply the default discount of 5%.

The second step is to use this class in the user exit so we apply the default discount on the days missing in Table 25.2. To achieve this goal, two possible approaches are available.

The first approach is to hardcode the default strategy class into the user exit, as demonstrated in Listing 25.10.

```
DATA:
  lo_obj TYPE REF TO object,
  lo_str TYPE REF TO zif_price_strategy.

" Determine the day of the week and put into lv_day

SELECT SINGLE price_class
  INTO @DATA(lv_clsname)
  FROM zclass
  WHERE day_of_week EQ @lv_day.

IF sy-subrc NE 0.
  lv_clsname = 'ZCL_PRICE_DEFAULT'.
ENDIF.

CREATE OBJECT lo_obj TYPE lv_clsname.
lo_str ?= lo_obj.

lo_str->calculate_price(
  EXPORTING is_data = ls_data
  IMPORTING es_price = ls_price
).

" Use data in LS_PRICE to modify user exit data
```

Listing 25.10 Hardcoding the Default Strategy Class

If no ad-hoc strategy class is found in the database table ZCLASS, the default strategy ZCL_PRICE_DEFAULT would be used to apply the default discount of 5%.

The second approach would be to put the default strategy into the customizing table ZCLASS for the regular days, as seen in Table 25.3.

DAY_OF_WEEK	PRICE_CLASS
1	ZCL_PRICE_DEFAULT
2	ZCL_PRICE_DEFAULT
3	ZCL_PRICE_WEDNESDAY
4	ZCL_PRICE_DEFAULT
5	ZCL_PRICE_FRIDAY
6	ZCL_PRICE_DEFAULT
7	ZCL_PRICE_SUNDAY

Table 25.3 Default Strategy Class Fed into a Database Table

In this situation, we made sure that the user exit would definitely find a discount strategy class in ZCLASS.

However, we shouldn't ignore the risk of human errors. If a functional consultant mistakenly deletes one of the records in Table 25.3, we would end up not applying the default discount for a certain day. Therefore, although we are going to assume that the user exit will definitely find a class in Table 25.3, we should still raise an exception in case the user exit cannot find a class.

```
DATA:
  lo_obj TYPE REF TO object,
  lo_str TYPE REF TO zif_price_strategy.

" Determine the day of the week and put into lv_day

SELECT SINGLE price_class
  INTO @DATA(lv_clsname)
  FROM zclass
  WHERE day_of_week EQ @lv_day.

IF sy-subrc NE 0.
  " Raise an exception regarding the missing data
ENDIF.
```

```
CREATE OBJECT lo_obj TYPE lv_clsname.
lo_str ?= lo_obj.

lo_str->calculate_price(
  EXPORTING is_data = ls_data
  IMPORTING es_price = ls_price
).

" Use data in LS_PRICE to modify user exit data
```

Listing 25.11 Raising an Exception When Data Is Missing

As a programmer, validating your assumptions is usually a good idea. The code in Listing 25.11 is a perfect example of that. We have assumed that a class name will definitely be found in ZCLASS. If not, we are letting the user know that something is wrong—instead of tolerating an invalid price calculation.

25.5 Intermediate Abstract Classes

The strategy design pattern provides a good alternative to inheritance. However, if some of your strategy classes are very similar, don't be afraid to combine the strategy and template method design patterns (Chapter 26) and put some abstract classes in between, as sketched in Figure 25.2.

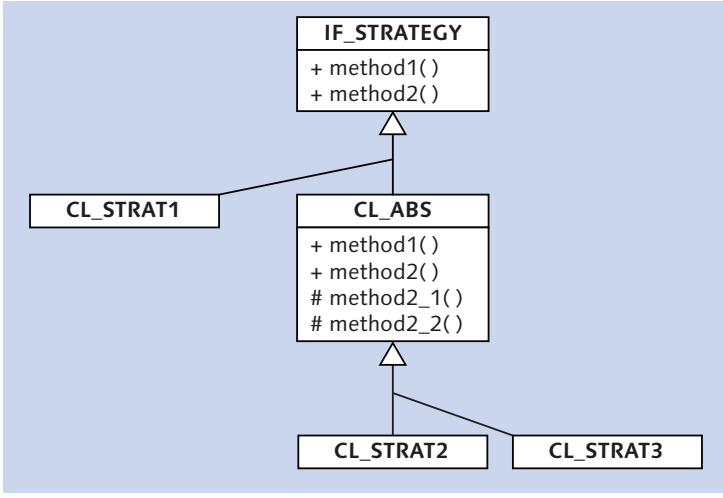


Figure 25.2 Intermediate Abstract Classes

In the sample diagram in Figure 25.2, our strategy is defined in the interface IF_STRATEGY. See Listing 25.12 for its implementation.

```
INTERFACE zif_strategy
  PUBLIC .

  METHODS method1.
  METHODS method2.

ENDINTERFACE.
```

Listing 25.12 Sample Strategy Interface with Two Methods

The first concrete strategy class, CL_STRAT1, implements the interface without doing anything special. CL_STRAT1~METHOD1 and CL_STRAT~METHOD2 would be filled as normal.

However, assuming that CL_STRAT2~METHOD1 is exactly the same as CL_STRAT3~METHOD1 and that CL_STRAT2~METHOD2 is very similar to CL_STRAT3~METHOD2 (but not 100% the same), we can inherit them from an intermediate abstract class, which we would call CL_ABS. Although using abstract classes is discussed in greater detail in Chapter 26, we can inspect a sample structure anyway in Listing 25.13.

```
CLASS zcl_abstract DEFINITION
  PUBLIC ABSTRACT CREATE PUBLIC.

  PUBLIC SECTION.
    INTERFACES zif_strategy ALL METHODS FINAL.
  PRIVATE SECTION.
  PROTECTED SECTION.
    METHODS method2_1 ABSTRACT.
    METHODS method2_2 ABSTRACT.
ENDCLASS.

CLASS zcl_abstract IMPLEMENTATION.

  METHOD zif_strategy~method1.
    " Do stuff
    " Whatever you code here, will be inherited by ZCL_STRAT2 & 3
  ENDMETHOD.

  METHOD zif_strategy~method2.
    " Do stuff
    method2_1( ). " Abstract method, coded in ZCL_STRAT2 & 3
    " Do stuff
    method2_2( ). " Abstract method, coded in ZCL_STRAT2 & 3
```

```
    " Do stuff
  ENDMETHOD.

ENDCLASS.
```

Listing 25.13 Intermediate Abstract Class

Once we have this abstract class in our pocket, we can move forward and create `CL_STRAT2`, as seen in Listing 25.14.

```
CLASS zcl_strat2 DEFINITION
  PUBLIC
  INHERITING FROM zcl_abstract
  FINAL CREATE PUBLIC.

  PUBLIC SECTION.
  PRIVATE SECTION.
  PROTECTED SECTION.
    METHODS method2_1 REDEFINITION.
    METHODS method2_2 REDEFINITION.
ENDCLASS.

CLASS zcl_strat2 IMPLEMENTATION.

  METHOD method2_1.
    " Do stuff exclusive to strategy 2
  ENDMETHOD.

  METHOD method2_2.
    " Do stuff exclusive to strategy 2.
  ENDMETHOD.
ENDCLASS.
```

Listing 25.14 Strategy Class 2

`CL_STRAT3` is similar, as seen in Listing 25.15.

```
CLASS zcl_strat3 DEFINITION
  PUBLIC
  INHERITING FROM zcl_abstract
  FINAL CREATE PUBLIC.

  PUBLIC SECTION.
  PRIVATE SECTION.
  PROTECTED SECTION.
    METHODS method2_1 REDEFINITION.
    METHODS method2_2 REDEFINITION.
ENDCLASS.
```

```
CLASS zcl_strat3 IMPLEMENTATION.

  METHOD method2_1.
    " Do stuff exclusive to strategy 3
  ENDMETHOD.

  METHOD method2_2.
    " Do stuff exclusive to strategy 3.
  ENDMETHOD.
ENDCLASS.
```

Listing 25.15 Strategy Class 3

Object composition is recommended over inheritance because using composition results in a higher number of reusable independent classes. Therefore, we don't recommend using inheritance as the backbone of an architecture, but inheritance is still a better option than copying and pasting code among classes. Instead of inheriting `CL_STRAT2` and `CL_STRAT3` directly from the interface and duplicating their code in `METHOD1`, taking advantage of the template method is a better idea.

In this hierarchy, if we change anything in `CL_ABSTRACT~METHOD1` or `CL_ABSTRACT~METHOD2`, the changes are automatically inherited by `CL_STRAT2` and `CL_START3`. What we do in `CL_STRAT2~METHOD2_1` and `CL_STRAT2~METHOD2_2` will have no effect on `CL_STRAT3` (and vice versa).

To alter our original example, what if `CL_STRAT2~METHOD1` and `CL_STRAT3~METHOD2` were exactly the same, while `CL_STRAT2~METHOD2` and `CL_STRAT3~METHOD3` differed so much that they couldn't be hierarchized under a template method? In that case, the best solution would be to avoid inheritance and to separate the classes, making them implement the interface directly and taking advantage of toolkit methods. Under these circumstances, the code for strategy 2 would look like Listing 25.16.

```
CLASS zcl_strat2 DEFINITION
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.
    INTERFACES zif_strategy.
  PRIVATE SECTION.
  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_strat2 IMPLEMENTATION.

  METHOD zif_strategy~method1.
```

```
    zcl_toolkit=>method1_for_strat_2_and_3( ). " Imaginary method
  ENDMETHOD.

  METHOD zif_strategy~method2.
    " Do stuff excluse to strategy 2
  ENDMETHOD.

ENDCLASS.
```

Listing 25.16 Strategy Class 2 Using Toolkit Method Calls

Strategy 3 would look like Listing 25.17.

```
CLASS zcl_strat3 DEFINITION
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.
    INTERFACES zif_strategy.
  PRIVATE SECTION.
  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_strat3 IMPLEMENTATION.

  METHOD zif_strategy~method1.
    zcl_toolkit=>method1_for_strat_2_and_3( ). " Imaginary method
  ENDMETHOD.

  METHOD zif_strategy~method2.
    " Do stuff excluse to strategy 3
  ENDMETHOD.

ENDCLASS.
```

Listing 25.17 Strategy Class 3 Using Toolkit Method Calls

These two examples of code are clean and simple, plus there is no copy-paste programming involved. Using inheritance is generally considered poor practice and should only be used if it brings a distinct advantage and prevents a worse alternative. In our case, inheritance prevented code duplication between `CL_METHOD2` and `CL_METHOD3`, which would result in increased maintenance costs and error risks.

This brings us to one of the potential disadvantages of the strategy design pattern. As the population of strategy classes grows, some of the classes will inevitably have common functionality. Assuming that you have had class A running well for a long time, a new strategy requirement might arise where you need to develop

class B. The catch is that class B has some common functionality with the former class, so you want to centralize the common code in an abstract class for A and B—which is the clean architectural approach. However, as a result, you will be modifying the former class A, so its unit tests should be repeated. This situation might not be desirable for project management—especially if class A is performing a complex and critical task.

One thing should be emphasized: Whenever you feel the urge to create and call a toolkit method, always remember that using the servant design pattern (Chapter 23) might be an alternative as well. In this case, we could have used the servant design pattern instead of creating the method `METHOD1_FOR_STRAT_2_AND_3` inside `CL_TOOLKIT`. The decision between servants and toolkits really depends on the situation; making that decision is more art than science.

Nevertheless, you should have a good design pattern vocabulary and be able to pick a solution among multiple alternatives, build a Frankenstein pattern based on what you know, or create an entirely new pattern based on your overall insight on patterns. Just make sure that you are not reinventing the wheel.

25.6 Related Patterns

If the number of strategy objects is very high and they have partially shared states, you can consider applying the flyweight design pattern (Chapter 15) to reduce the memory footprint. However, don't create flyweights if you are going to pick a single strategy class and not use another until the end of your runtime. In such a case, flyweight would be overkill and an unnecessary optimization attempt.

25.7 Summary

Strategy enables us to develop alternative algorithms sharing the same interface and to pick them dynamically during runtime. This pattern lets us flexibly juggle classes, and adding a new algorithm is as easy as plugging a new USB device into a computer.

Two alternative ways of passing data to the strategy objects are available. Passing data will increase the memory footprint and independence. Passing the context object will decrease the memory foot and independence.

You should often consider the case where no suitable strategy can be determined. Having a default safety net strategy can save the day. Similar strategy classes are often derived from a common abstract class.

The template method design pattern is the showcase of abstract classes. You have an abstract class, which partially implements an algorithm but expects subclasses to fill its abstract methods. Each subclass overrides those methods to customize the algorithm as needed.

26 Template Method

The template method design pattern is the showcase of abstract classes. If you understand the concept of abstract classes, the template method design pattern will become clear. Basically, abstract classes are partially implemented parent classes. Some of their methods are fully implemented, while some of their methods are purposely left empty for child classes to implement. Those empty methods are usually marked as abstract.

When we derive child classes from abstract parent classes, we fill those abstract methods. By using a common parent class, we ensure that the common functionality of our child classes is centralized in the parent class and that code duplication is prevented. For more information on abstract classes, see Appendix A, Section A.4.

Template method can be considered *the* abstract class design pattern. In some cases, we may have a handful of algorithms that are partially similar to but also partially deviate from each other. In such a case, the pattern suggests creating a parent abstract class to implement the common part and leaving the distinctive part to child concrete classes.

This chapter will start off with a case study where we need to calculate the average transaction volume of completely different entities, such as purchase requisition items, sales order items, and purchase order items. We will ensure that the common part of the algorithm is consolidated into a common abstract class by implementing the template method design pattern. We will continue with our comprehensive discussion by covering when to use the pattern, its advantages and risks, accurate degrees of abstraction, and the "Hollywood principle."

26.1 Case Study: Average Transaction Volume

In this example, we need a program to calculate the average transaction volume of a given material, customer, or vendor. This program will need to include the following functionality:

- For a material, we need to calculate the average quantity over purchase requisition items (EBAN). However, if the material is deleted, we need to return zero.
- For a customer, we need to calculate the average quantity over sales order items (VBAP). However, if the customer is blocked, we need to return zero.
- For a vendor, we need to calculate the average quantity over purchase order items (EKPO). However, if the vendor is blocked, we need to return zero.

Without using a design pattern, our class would look like Listing 26.1.

```
CLASS zcl_model DEFINITION
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.

    " Some constants, etc.

    METHODS get_avg_material_quan
      IMPORTING
        !iv_matnr TYPE matnr
      RETURNING
        VALUE(rv_avg) TYPE menge_d.

    METHODS get_avg_customer_quan
      IMPORTING
        !iv_kunnr TYPE kunnr
      RETURNING
        VALUE(rv_avg) TYPE menge_d.

    METHODS get_avg_vendor_quan
      IMPORTING
        !iv_lifnr TYPE lifnr
      RETURNING
        VALUE(rv_avg) TYPE menge_d.

    " Some more methods, etc.

  PRIVATE SECTION.
    " Some more methods, etc.
  PROTECTED SECTION.
ENDCLASS.
```

```
CLASS zcl_model IMPLEMENTATION.

  METHOD get_avg_material_quan.
    CHECK is_material_deleted( iv_matnr ) EQ abap_false.

    DATA(lt_quan) = get_pr_items_of_material( iv_matnr ).
    CHECK lt_quan[] IS NOT INITIAL.

    LOOP AT lt_quan ASSIGNING FIELD-SYMBOL(<ls_quan>).
      ADD <ls_quan>-menge to rv_avg.
    ENDLOOP.

    DIVIDE rv_avg BY LINES( lt_quan ).
  ENDMETHOD.

  METHOD get_avg_customer_quan.
    CHECK is_customer_blocked( iv_kunnr ) EQ abap_false.

    DATA(lt_quan) = get_sales_items_of_customer( iv_kunnr ).
    CHECK lt_quan[] IS NOT INITIAL.

    LOOP AT lt_quan ASSIGNING FIELD-SYMBOL(<ls_quan>).
      ADD <ls_quan>-menge to rv_avg.
    ENDLOOP.

    DIVIDE rv_avg BY LINES( lt_quan ).
  ENDMETHOD.

  METHOD get_avg_vendor_quan.
    CHECK is_vendor_blocked( iv_lifnr ) EQ abap_false.

    DATA(lt_quan) = get_po_items_of_vendor( iv_lifnr ).
    CHECK lt_quan[] IS NOT INITIAL.

    LOOP AT lt_quan ASSIGNING FIELD-SYMBOL(<ls_quan>).
      ADD <ls_quan>-menge to rv_avg.
    ENDLOOP.

    DIVIDE rv_avg BY LINES( lt_quan ).
  ENDMETHOD.

ENDCLASS.
```

Listing 26.1 Average Calculation Class

As usual, we have used imaginary method calls (like `GET_PO_ITEMS_OF_VENDOR`) to simplify the code and make it more readable. Also note that we assumed that all the units of measure are the same and ignored possible deletion flags of the documents. We could have also improved the performance using the aggregate SQL

command AVG. However, we have left these considerations out for the sake of simplicity.

Although the code will probably run without too much of a hassle, there is a problem: We have three distinct methods (material, customer, vendor) that share the exact same algorithm. This algorithm performs the following tasks:

1. Check if an entity is available (data source varies).
2. Get a list of quantities of the given entity (data source varies).
3. Ensure that the list is not empty (fixed code).
4. Calculate the sum of quantities in the list (fixed code).
5. Calculate the average by dividing the sum by the item count (fixed code).

The algorithm partially contains varied code (the first two steps) and partially contains fixed code (the final three steps). Overall, all entities (material, customer, vendor) share the same algorithm.

If we leave the code in this form and don't use any design pattern, we will face the following disadvantages:

- We need to write very similar code repeatedly in multiple methods, which is a waste of time.
- To add support for a further entity, such as average volume per requisitioner (field AFNAM in table EBAN), you need rewrite the entire algorithm—both variable and fixed parts. Copy and paste helps, but it is an anti-pattern.
- If the major algorithm contains an error, we need to modify three distinct methods to correct it, which only gets worse the more methods you have.
- If a new feature is requested like "Return zero if there are less than 10 transactions," you need to modify three distinct methods. Again, imagine having thirty distinct methods.
- If you make a mistake on the "fixed" part of only one method, your users might be confused because customer and vendor calculations would look OK, but material calculations would look weird.

The template method can come to the rescue, as well as other object-oriented tools. For the calculation, we will take advantage of an abstract class where the fixed part of the algorithm is coded directly and where the variable part expects data from subclasses.

See Figure 26.1 for the Unified Modeling Language (UML) diagram corresponding to our new architecture.

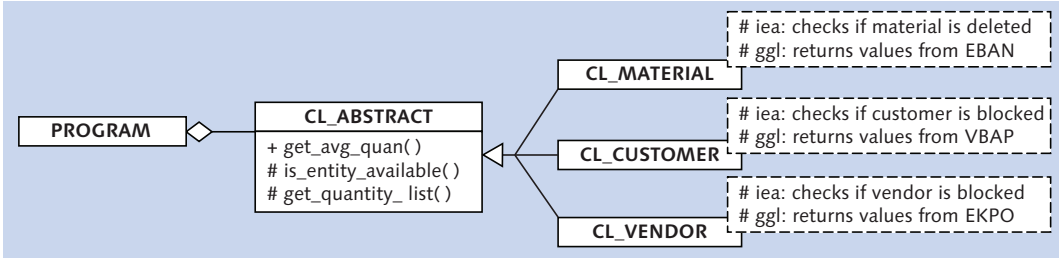


Figure 26.1 Template Method Architecture

To start, let's take a look at `CL_ABSTRACT` in Listing 26.2 to get the big picture.

```
CLASS zcl_abstract DEFINITION
  PUBLIC ABSTRACT CREATE PUBLIC.

  PUBLIC SECTION.

    TYPES: BEGIN OF t_quan,
      doc_key TYPE string,
      menge TYPE menge_d,
      meins TYPE meins,
    END OF t_quan,
    tt_quan TYPE STANDARD TABLE OF t_quan WITH DEFAULT KEY.

    METHODS get_avg_quan FINAL
      IMPORTING
        !iv_key TYPE clike
      RETURNING
        VALUE(rv_avg) TYPE menge_d.

  PRIVATE SECTION.
  PROTECTED SECTION.

    METHODS is_entity_available ABSTRACT
      IMPORTING
        !iv_key TYPE clike
      RETURNING
        VALUE(rv_available) TYPE abap_bool.

    METHODS get_quantity_list ABSTRACT
      IMPORTING
        !iv_key TYPE clike
      RETURNING
```

```
        VALUE(rt_list) TYPE tt_quan.

ENDCLASS.

CLASS zcl_abstract IMPLEMENTATION.

  METHOD get_avg_quan.
    CHECK is_entity_available( iv_key ) EQ abap_true.

    DATA(lt_quan) = get_quantity_list( iv_key ).
    CHECK lt_quan[] IS NOT INITIAL.

    LOOP AT lt_quan ASSIGNING FIELD-SYMBOL(<ls_quan>).
      ADD <ls_quan>-menge to rv_avg.
    ENDLOOP.

    DIVIDE rv_avg BY LINES( lt_quan ).
  ENDMETHOD.

ENDCLASS.
```

Listing 26.2 Template Class for Generic Average Calculation

Basically, the algorithm is coded into `CL_ABSTRACT->GET_AVG_QUAN`. This method contains the core algorithm, which performs the following:

1. Check if the entity is available via `IS_ENTITY_AVAILABLE`. The method call is present in `GET_AVG_QUAN`, but the method itself is abstract and will be implemented by the subclasses.
2. Get a list of quantities of the given entity via `GET_QUANTITY_LIST`. The method call is present in `GET_AVG_QUAN`, but the method itself is abstract and will be implemented by the subclasses.
3. Ensure that the list is not empty (fixed code).
4. Calculate the sum of quantities in the list (fixed code).
5. Calculate the average by dividing the sum by the item count (fixed code).

This core algorithm is written in the abstract class only once and includes method calls for variable parts and direct code for fixed parts. We expect the subclasses to implement the variable parts only.

In a real-world situation, we would make `CL_ABSTRACT~IS_ENTITY_AVAILABLE` and `CL_ABSTRACT~GET_QUANTITY_LIST` raise exceptions, in case the ABAP programmer forgets to override (redefine) them in his/her subclass.

What do the subclasses look like? Let's start with the material, which is demonstrated in Listing 26.3.

```
CLASS zcl_material DEFINITION
  PUBLIC
  INHERITING FROM zcl_abstract
  FINAL CREATE PUBLIC.

  PUBLIC SECTION.
  PRIVATE SECTION.
  PROTECTED SECTION.
    METHODS is_entitiy_available REDEFINITION.
    METHODS get_quantity_list REDEFINITION.
ENDCLASS.

CLASS zcl_material IMPLEMENTATION.

  METHOD is_entity_available.
    SELECT SINGLE mandt INTO sy-mandt ##write_ok
      FROM mara
      WHERE matnr EQ iv_key AND lvorm EQ abap_false.
    CHECK sy-subrc eq 0.
    rv_available = abap_true.
  ENDMETHOD.

  METHOD get_quantity_list.
    SELECT banfn AS doc_key menge meins
      INTO CORRESPONDING FIELDS OF TABLE rt_list
      FROM eban
      WHERE matnr EQ iv_key.
  ENDMETHOD.
ENDCLASS.
```

Listing 26.3 Implementation for Material

The key logic of the template method algorithm consists of two steps. First, we derived CL_MATERIAL from CL_ABSTRACT, as follows:

```
CLASS zcl_material DEFINITION
  PUBLIC
  INHERITING FROM zcl_abstract
  FINAL CREATE PUBLIC.
```

This means that CL_MATERIAL is a semiclone of CL_ABSTRACT. In this logic, child class CL_MATERIAL inherits all the variables and methods of the parent class CL_ABSTRACT. If, in the future, we change anything in CL_ABSTRACT~GET_AVG_QUAN, the same change is inherited by CL_MATERIAL~GET_AVG_QUAN instantly.

The second step is to implement the abstract methods that should be modified to run for materials. We don't want CL_MATERIAL to be an exact clone of CL_ABSTRACT. Instead, we want a partial clone. IS_ENTITY_AVAILABLE and GET_QUANTITY_LIST won't be cloned from CL_ABSTRACT (which are abstract methods anyway), but everything else will. Therefore, we have implemented those methods targeting material logic in Listing 26.4.

```
METHOD is_entity_available.
  SELECT SINGLE mandt INTO sy-mandt ##write_ok
    FROM mara
    WHERE matnr EQ iv_key AND lvorm EQ abap_false.
  CHECK sy-subrc eq 0.
  rv_available = abap_true.
ENDMETHOD.

METHOD get_quantity_list.
  SELECT banfn AS doc_key menge meins
    INTO CORRESPONDING FIELDS OF TABLE rt_list
    FROM eban
    WHERE matnr EQ iv_key.
ENDMETHOD.
```

Listing 26.4 Abstract Methods Implemented in Material Class

As you can see, the core algorithm remains in CL_ABSTRACT. We didn't duplicate any code at all; the core algorithm is "cloned" from CL_ABSTRACT to CL_MATERIAL via inheritance, and the two algorithms are linked: changes on CL_ABSTRACT~GET_AVG_QUAN are automatically applied to CL_MATERIAL~GET_AVG_QUAN. We have simply redefined and implemented the variable part—IS_ENTITY_AVAILABLE and GET_QUANTITY_LIST—and coded them to cover material functionality.

Following the same logic, the customer class will look very similar but will fulfill a different task, as demonstrated in Listing 26.5.

```
CLASS zcl_customer DEFINITION
  PUBLIC
  INHERITING FROM zcl_abstract
  FINAL CREATE PUBLIC.

  PUBLIC SECTION.
  PRIVATE SECTION.
  PROTECTED SECTION.
    METHODS is_entity_available REDEFINITION.
    METHODS get_quantity_list REDEFINITION.
ENDCLASS.
```

```
CLASS zcl_customer IMPLEMENTATION.

  METHOD is_entity_available.
    SELECT SINGLE mandt INTO sy-mandt ##write_ok
      FROM kna1
      WHERE kunnr EQ iv_key AND sperr EQ abap_false.
    CHECK sy-subrc eq 0.
    rv_available = abap_true.
  ENDMETHOD.

  METHOD get_quantity_list.
    SELECT vbap~vbeln AS doc_key vbap~kwmeng AS menge vbap~meins
      INTO CORRESPONDING FIELDS OF TABLE rt_list
      FROM vbap
      INNER JOIN vbak ON vbak~vbeln EQ vbap~vbeln
      WHERE vbak~kunnr EQ iv_key.
  ENDMETHOD.
ENDCLASS.
```

Listing 26.5 Implementation for Customer

As you can see, the methods are doing the same thing, but for customers. As you might expect, the same will apply to the vendor logic, as demonstrated in Listing 26.6.

```
CLASS zcl_vendor DEFINITION
  PUBLIC
  INHERITING FROM zcl_abstract
  FINAL CREATE PUBLIC.

  PUBLIC SECTION.
  PRIVATE SECTION.
  PROTECTED SECTION.
    METHODS is_entity_available REDEFINITION.
    METHODS get_quantity_list REDEFINITION.
ENDCLASS.

CLASS zcl_vendor IMPLEMENTATION.

  METHOD is_entity_available.
    SELECT SINGLE mandt INTO sy-mandt ##write_ok
      FROM lfa1
      WHERE lifnr EQ iv_key AND sperr EQ abap_false.
    CHECK sy-subrc eq 0.
    rv_available = abap_true.
  ENDMETHOD.

  METHOD get_quantity_list.
```

```
    SELECT ekpo~ebeln AS doc_key ekpo~menge ekpo~meins
      INTO CORRESPONDING FIELDS OF TABLE rt_list
      FROM ekpo
      INNER JOIN ekko ON ekko~ebeln EQ ekpo~ebeln
      WHERE ekko~lifnr EQ iv_key.
  ENDMETHOD.
ENDCLASS.
```

Listing 26.6 Implementation for Vendor

So far, so good. How are we going to consolidate and make our program perform calculations per material, customer, or vendor? With very little fuss! Listing 26.7 demonstrates what such a code would look like.

```
DATA:
  lo_obj TYPE REF TO object,
  lo_abs TYPE REF TO zcl_abstract.

* Assuming that we store the class name in GV_CLASS_NAME,
* which could be 'ZCL_MATERIAL' , 'ZCL_CUSTOMER' or 'ZCL_VENDOR';
* create the object
  CREATE OBJECT lo_obj TYPE (gv_class_name).
  lo_abs ?= lo_obj.

* Assuming that we store the entity key in GV_KEY,
* which could be a MATNR, KUNNR or LIFNR;
* calculate the average quantity
  DATA(lt_avg_quan) = lo_abs->get_avg_quan( gv_key ).
```

Listing 26.7 Client Program Code Snippet

Let us inspect and evaluate this code line by line to demonstrate the logic and power of the template methods. First, we have the following line, which is not too complicated:

```
  CREATE OBJECT lo_obj TYPE (gv_class_name).
```

`GV_CLASS_NAME` is an imaginary variable holding the class name as a literal. It is up to you how you choose to fill the class name. In our case, `GV_CLASS_NAME` will contain one of the following literals:

- `'ZCL_MATERIAL'->LO_OBJ` will become an object of type `ZCL_MATERIAL`.
- `'ZCL_CUSTOMER'->LO_OBJ` will become an object of type `ZCL_CUSTOMER`.
- `'ZCL_VENDOR'->LO_OBJ` will become an object of type `ZCL_VENDOR`.

Note

See Appendix B on subclass determination for alternative approaches to determining subclasses.

At this point, LO_OBJ is transformed into the object we want it to be. However, since LO_OBJ is defined as a general object (TYPE REF TO OBJECT), we can't make direct method calls. For that reason, we need to cast it to a concrete object with a statically defined class type, as follows:

```
lo_abs ?= lo_obj.
```

Now we are getting somewhere! LO_ABS will contain the following:

- If GV_CLASS_NAME was 'ZCL_MATERIAL', then LO_ABS is an object of type ZCL_MATERIAL.
- If GV_CLASS_NAME was 'ZCL_CUSTOMER', then LO_ABS is an object of type ZCL_CUSTOMER.
- If GV_CLASS_NAME was 'ZCL_VENDOR', then LO_ABS is an object of type ZCL_VENDOR.

The last executable line is the execution itself:

```
DATA(lt_avg_quan) = lo_abs->get_avg_quan( gv_key ).
```

What does this line do? If GV_CLASS_NAME was 'ZCL_MATERIAL', then ZCL_MATERIAL~GET_AVG_QUAN will be executed, as seen in Listing 26.8.

```
METHOD get_avg_quan. " INHERITED FROM ZCL_ABSTRACT

  " THIS WILL EXECUTE ZCL_MATERIAL~IS_ENTITY_AVAILABLE
  CHECK is_entity_available( iv_key ) EQ abap_true.

  " THIS WILL EXECUTE ZCL_MATERIAL~GET_QUANTITY_LIST
  DATA(lt_quan) = get_quantity_list( iv_key ).
  CHECK lt_quan[] IS NOT INITIAL.

  LOOP AT lt_quan ASSIGNING FIELD-SYMBOL(<ls_quan>).
    ADD <ls_quan>-menge to rv_avg.
  ENDLOOP.

  DIVIDE rv_avg BY LINES( lt_quan ).
ENDMETHOD.
```

Listing 26.8 Flow for Material

If GV_CLASS_NAME was 'ZCL_CUSTOMER', then ZCL_CUSTOMER~GET_AVG_QUAN will be executed, as seen in Listing 26.9.

```
METHOD get_avg_quan. " INHERITED FROM ZCL_ABSTRACT

  " THIS WILL EXECUTE ZCL_CUSTOMER~IS_ENTITY_AVAILABLE
  CHECK is_entity_available( iv_key ) EQ abap_true.

  " THIS WILL EXECUTE ZCL_CUSTOMER~GET_QUANTITY_LIST
  DATA(lt_quan) = get_quantity_list( iv_key ).
  CHECK lt_quan[] IS NOT INITIAL.

  LOOP AT lt_quan ASSIGNING FIELD-SYMBOL(<ls_quan>).
    ADD <ls_quan>-menge to rv_avg.
  ENDLOOP.

  DIVIDE rv_avg BY LINES( lt_quan ).
ENDMETHOD.
```

Listing 26.9 Flow for Customer

If GV_CLASS_NAME was 'ZCL_VENDOR', then ZCL_VENDOR~GET_AVG_QUAN will be executed, as seen in Listing 26.10.

```
METHOD get_avg_quan. " INHERITED FROM ZCL_ABSTRACT

  " THIS WILL EXECUTE ZCL_VENDOR~IS_ENTITY_AVAILABLE
  CHECK is_entity_available( iv_key ) EQ abap_true.

  " THIS WILL EXECUTE ZCL_VENDOR~GET_QUANTITY_LIST
  DATA(lt_quan) = get_quantity_list( iv_key ).
  CHECK lt_quan[] IS NOT INITIAL.

  LOOP AT lt_quan ASSIGNING FIELD-SYMBOL(<ls_quan>).
    ADD <ls_quan>-menge to rv_avg.
  ENDLOOP.

  DIVIDE rv_avg BY LINES( lt_quan ).
ENDMETHOD.
```

Listing 26.10 Flow for Vendor

In this example, we have a rather simple algorithm. However, imagine having an algorithm with 300 lines containing 10 variable method calls. The value of using the template method design pattern has a positive correlation with the complexity of the algorithm in question. Imagine that after going live, a new requirement pops up: "If the entity has less than ten transactions, return zero." Because we used the template method, we only need to modify a single method, as seen in Listing 26.11.

```
CLASS zcl_abstract IMPLEMENTATION.

  METHOD get_avg_quan.
    CHECK is_entity_available( iv_key ) EQ abap_true.

    DATA(lt_quan) = get_quantity_list( iv_key ).
    CHECK lt_quan[] IS NOT INITIAL.
    CHECK LINES( lt_quan ) GE 10.

    LOOP AT lt_quan ASSIGNING FIELD-SYMBOL(<ls_quan>).
      ADD <ls_quan>-menge to rv_avg.
    ENDLOOP.

    DIVIDE rv_avg BY LINES( lt_quan ).
  ENDMETHOD.

  " Further code

ENDCLASS.
```

Listing 26.11 Modifying the Abstract Class

This change to the abstract class will automatically be inherited by CL_MATERIAL, CL_CUSTOMER, and CL_VENDOR.

Here is another imaginary scenario: After going live, we are told that the same functionality is required for purchase requisitioners. In other words, we need the ability to calculate the average volume for field AFNAM in table EBAN. Using the traditional method, we would need to follow the dreadful copy–paste–replace methodology. However, using the template method design pattern, all we need is a new class—we don't repeat any code at all. You can see this new class in Figure 26.2.

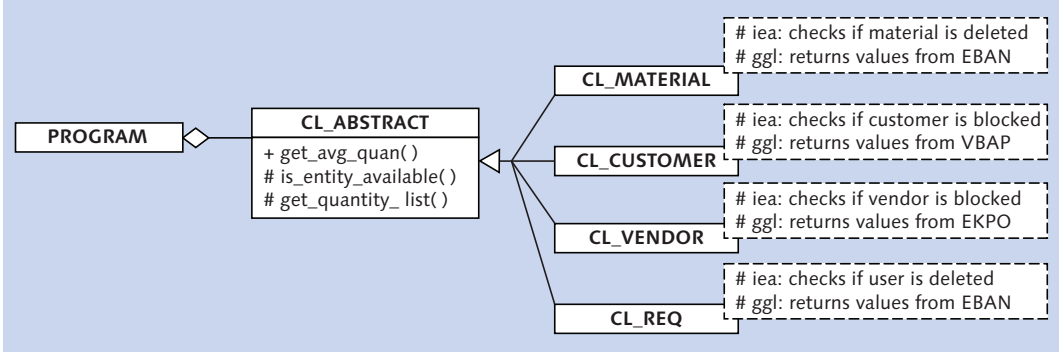


Figure 26.2 New Concrete Class Derived from the Abstract Class

Listing 26.12 showcases what the new class CL_REQ would look like.

```
CLASS zcl_req DEFINITION
  PUBLIC
  INHERITING FROM zcl_abstract
  FINAL CREATE PUBLIC.

  PUBLIC SECTION.
  PRIVATE SECTION.
  PROTECTED SECTION.
    METHODS is_entity_available REDEFINITION.
    METHODS get_quantity_list REDEFINITION.
ENDCLASS.

CLASS zcl_req IMPLEMENTATION.

  METHOD is_entity_available.
    SELECT SINGLE mandt INTO sy-mandt ##write_ok
      FROM usr02
      WHERE bname EQ iv_key
        AND ( gltgb IS INITIAL OR gltgb GT sy-datum ).
    CHECK sy-subrc eq 0.
    rv_available = abap_true.
  ENDMETHOD.

  METHOD get_quantity_list.
    SELECT banfn AS doc_key menge meins
      INTO CORRESPONDING FIELDS OF TABLE rt_list
      FROM eban
      WHERE afnam EQ iv_key.
  ENDMETHOD.
ENDCLASS.
```

Listing 26.12 Implementing a New Average Class

That's it! No copying and pasting, no code repetition, nothing! There is no risk of ruining CL_MATERIAL, CL_CUSTOMER, or CL_VENDOR while adding the new requisitioner functionality because everything is safely contained in discrete classes.

26.2 When to Use

A typical signal that points to the template method pattern is when you find out that you are designing or developing (at least) two classes that work mostly on the same logic but have small deviations. You can't make the classes identical due to the deviations, but they very much look alike, and you either need code duplication or need to write many toolkit methods in order to prevent it.

If that's your feeling about your classes, it might be the right time to move forward to the template method design pattern.

26.3 Advantages and Risks

Template method enables subclasses to implement alternating behavior and does so while avoiding duplication in the code. The general flow and the varying part are isolated, and what subclasses are supposed to override is also controlled.

This design pattern promotes inheritance, for agreeable reasons. However, if you are not careful, you might experience the downsides of inheritance. One of the most significant risks is that once you change something in the abstract class, you might be affecting any of its subclasses, so test scenarios of all subclasses might need to be rechecked.

Another risk is that any subclass can override nonabstract public or protected methods to alter the flow of your abstract class. Sometimes, we want to provide this flexibility intentionally. But, if not, this might pose a risk to your application logic. We can semantically and technically determine what should and shouldn't be overridden with a combination of abstract/final/private methods. A subclass can't override private and final methods, so make use of it. What must be overridden should be marked abstract. What can be overridden should be public or protected. What should not be overridden should be private or final.

Performance is another potential risk. Occasionally, you will run into the dilemma of delegation. In one approach, you can fulfill a task inside the abstract class by using dynamic queries and data assignments. In this case, you will require less coding in subclasses, but the performance won't be at its peak—that's the natural cost of dynamic programming. In the alternative approach, you can delegate the task to the subclasses completely, which will require more coding in subclasses. However, performance will be better because you can optimize your queries and data operations by using indexes, hashed tables, etc.

Which approach is better? Well, there is no straightforward answer to that—approaches should be evaluated on a case-by-case basis. Assuming that you are the developer of all the classes in question, if the performance gain is going to be significant, we suggest delegating the task to subclasses. If the data volume is relatively small and you don't see any performance risks, fulfilling the task inside

the abstract class can save you quite some time. If you are not the only developer and expect a huge number of upcoming subclasses, you can't foresee everything. In this case, you can do the dynamic coding in a protected method of the abstract class—remembering that any subclass can override the method if they need to. As a result, you would be providing a default method that can be substituted easily.

26.4 Degree of Abstraction

In our example, an abstract class was the most abstract object in the hierarchy, which is fine for educational purposes. However, in a real-world situation, having an interface as well is preferable. The abstract class implements the interface, and only after that are concrete classes derived from the abstract class. The client program deals with the interface and not the abstract class directly.

Why?

In the case of the template method design pattern, you assume that you can foresee part of an algorithm; so you code it into an abstract class. However, in the future, a completely different requirement may arise, which will require you to develop an alternative set consisting of an abstract class with concrete subclasses. Your former abstract class won't help you much because its structure and abstract methods are corresponding to the former logic.

Having an interface to provide a single entry point (covering both abstract classes) is a nice and clean situation. Doing so makes the template method look like a subbranch of the strategy design pattern; they really make a nice combination.

26.5 The "Hollywood Principle"

Most of the time, we see subclasses calling methods of their superclasses. In this pattern, we see the exact opposite: The superclass template method calls methods of its subclasses, an approach is known as the "Hollywood Principle": "Don't call us, we'll call you."

Occasionally, subclasses override the nonabstract methods of their superclass. In this pattern, if you feel like you need to override a lot of nonabstract methods, you should re-evaluate your design. Chances are you need a distinct concrete class

that is directly derived from a common interface. Alternatively, you might need a distinct abstract class (derived from a common interface) with a few subclasses—one of which is your suspicious subclass.

The more you violate the "Hollywood Principle," the more suspicious you should get.

26.6 Summary

Template method is a typical application of abstract classes. You would code the common part of subclasses into the parent abstract class and the variable part into the concrete subclasses. This design pattern prevents code duplication and, unusual for design patterns, promotes inheritance. Following the "Hollywood Principle," the parent class calls methods of the subclasses.

An error in the abstract class will affect all subclasses, so extra care should be taken there. To help subclasses behave exactly as they should, methods can be marked as abstract, final, or private.

To increase the degree of abstraction, you can derive the abstract class from an interface.

The visitor design pattern enables us to extend the functionality of a class without ever modifying a single line of its code. Visitor is typically used when you have an untouchable VIP class or when you simply want to add various utility functionalities to a model class without making it bloated.

27 Visitor

Visitor is a design pattern that is typically used when you are dealing with a VIP class. A VIP class is critical and untouchable, but you may still need to extend its functionality. In such cases, visitor provides the required flexibility and lets you extend the class without altering it.

Another case when you might use visitor is when you release a class that is part of a product. Your clients might need to add new functionality to the class, but you don't want them to edit/enhance the source code—otherwise, you might overwrite their modifications on future releases. In such a case, the visitor design pattern provides an excellent solution by keeping your class unmodified and enabling the client to extend its functionality at the same time.

We will start off with the case study where we have a model class that needs to be extended in the future. Instead of modifying (and potentially bloating) the model class every time we need to add a new satellite functionality, we will take advantage of the visitor design pattern, which will enable us to extend the central class with help of satellite classes. The case study will be followed by discussions on when to use the pattern and other related patterns.

27.1 Case Study: Incoming Invoice Processing

In this example, we will create a software product that manages incoming invoices. We want the client to be able to create a new document, attach a scanned PDF file, send the file through the workflow for approval, and post an actual SAP ERP Materials Management (MM) invoice using our product.

At the center of the product, we would have a model class covering the core capabilities (creating documents, attaching PDFs, approvals, and posting invoices), as sketched in Figure 27.1.

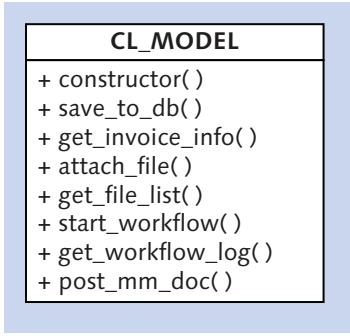


Figure 27.1 Model Class for Invoice Processing

The central pattern being used here is MVC (model–view–controller). However, this central class does much more than a single class is supposed to do—it's starting to look like the blob anti-pattern discussed in Appendix C. While you wouldn't do this in a real-world application, we wanted to keep things simple and focus on our subject, the visitor design pattern.

To improve our imaginary design, you can assume that `CL_MODEL` doesn't perform all of those tasks alone—instead, `CL_MODEL` contains other private objects (composition) that actually perform the tasks, and the methods in Figure 27.1 are only wrappers.

In the scope of the product, client programs would use the oversimplified model class seen in Listing 27.1.

```
CLASS zcl_model DEFINITION
  PUBLIC
  FINAL CREATE PUBLIC.

  PUBLIC SECTION.

    " Some public definitions

    METHODS constructor
      IMPORTING !iv_docid TYPE zd_docid.

    METHODS save_to_db.

    METHODS get_invoice_info
```

```
      RETURNING VALUE(rs_info) TYPE zs_inv_info.

    METHODS attach_file
      IMPORTING !is_file TYPE zs_file.

    METHODS get_file_list
      RETURNING VALUE(rt_list) TYPE ztt_file_list.

    METHODS start_workflow.

    METHODS get_workflow_log
      RETURNING VALUE(rt_log) TYPE ztt_wf_log.

    METHODS post_mm_doc
      RETURNING VALUE(rt_ret) TYPE bapiret2_tab.

  PRIVATE SECTION.
    " Some helpful private definitions
    " Some helpful private methods
  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_model IMPLEMENTATION.

  METHOD constructor.
    " Some code to initialize object
  ENDMETHOD.

  METHOD save_to_db.
    " Do some checks
    " Write data to DB
  ENDMETHOD.

  METHOD get_invoice_info.
    " Return details of the invoice
  ENDMETHOD.

  METHOD attach_file.
    " Add file to GOS
    " Add file info to some private ITAB
  ENDMETHOD.

  METHOD get_file_list.
    " Return file list from some private ITAB
  ENDMETHOD.

  METHOD start_workflow.
    save_to_db( ).
    " Raise an event, targeting the workflow
  ENDMETHOD.

  METHOD get_workflow_log.
```

```
    " Read workflow log using BAPI calls and return
  ENDMETHOD.

  METHOD post_mm_doc.
    " Call BAPI and return messages
    " Update status of the invoice
  ENDMETHOD.

ENDCLASS.
```

Listing 27.1 Model Class

Let's inspect the components of this class for a better overview on what is happening:

- `CONSTRUCTOR` is the method to initialize the class when creating a new instance.
- `SAVE_TO_DB` will save the current state to the database.
- `GET_INVOICE_INFO` will return the details of the current invoice.
- `ATTACH_FILE` will attach an uploaded PDF file and associate it with the current invoice, for example, uploading a scanned PDF of an incoming invoice.
- `GET_FILE_LIST` will return a list of attached PDF files.
- `START_WORKFLOW` would be called after all files are attached and the invoice is saved. It will initiate the approval process of the current invoice.
- `GET_WORKFLOW_LOG` will return the approval history of the current invoice.
- `POST_MM_DOC` would be called after the approval process is complete. It will post a new MM invoice document regarding the incoming invoice.

Next, we have to consider that, before publishing our product, we want to ensure that clients can extend this class and add new functionality to it. For example, a client might want this class to delete all stored files. Another one might need to post an SAP ERP Financial Accounting (FI) document. Yet another might need to save the workflow log into a special Z-table.

Each of these sample tasks can be performed by adding a new method to the model class. However, since we don't want the client to modify our precious model class at all, we will take advantage of the visitor design pattern. We will simply add one central method called `ACCEPT` and let the client use it to visit our class through a supplementary class of their own.

The basic idea is that, for each additional task (like deleting files, posting FI documents, and saving the workflow log), the client will create a new class imple-

menting a common interface that we provide. Figure 27.2 contains the overview of the model class and visitor interface.

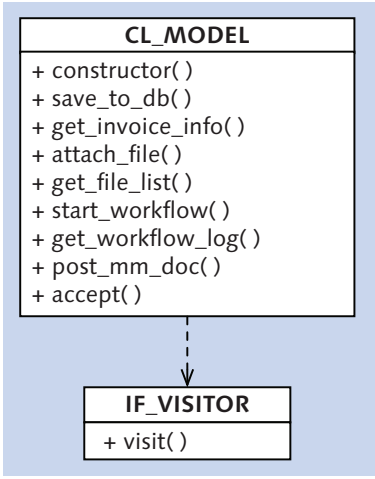


Figure 27.2 Adding the Visitor Interface

To make things more clear, Listing 27.2 showcases what the interface looks like.

```
INTERFACE zif_visitor
  PUBLIC .

  METHODS visit
    IMPORTING
      !io_model TYPE REF TO zcl_model.

ENDINTERFACE.
```

Listing 27.2 Visitor Interface

The `ACCEPT` method in `ZCL_MODEL` will import a class implementing `IF_VISITOR` and call its `VISIT` method, as seen in Listing 27.3.

```
CLASS zcl_model DEFINITION
  PUBLIC
  FINAL CREATE PUBLIC.

  PUBLIC SECTION.

    " Some public definitions

    METHODS accept
      IMPORTING !io_visitor TYPE REF TO zif_visitor.
```

```
    METHODS constructor
      IMPORTING !iv_docid TYPE zd_docid.

    METHODS save_to_db.

    METHODS get_invoice_info
      RETURNING VALUE(rs_info) TYPE zs_inv_info.

    METHODS attach_file
      IMPORTING !is_file TYPE zs_file.

    METHODS get_file_list
      RETURNING VALUE(rt_list) TYPE ztt_file_list.

    METHODS start_workflow.

    METHODS get_workflow_log
      RETURNING VALUE(rt_log) TYPE ztt_wf_log.

    METHODS post_mm_doc
      RETURNING VALUE(rt_ret) TYPE bapiret2_tab.

  PRIVATE SECTION.
    " Some helpful private definitions
    " Some helpful private methods
  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_model IMPLEMENTATION.

  METHOD accept.
    io_visitor->visit( me ).
  ENDMETHOD.

  METHOD constructor.
    " Some code to initialize object
  ENDMETHOD.

  METHOD save_to_db.
    " Do some checks
    " Write data to DB
  ENDMETHOD.

  METHOD get_invoice_info.
    " Return details of the invoice
  ENDMETHOD.

  METHOD attach_file.
    " Add file to GOS
    " Add file info to some private ITAB
  ENDMETHOD.
```

```
  METHOD get_file_list.
    " Return file list from some private ITAB
  ENDMETHOD.

  METHOD start_workflow.
    save_to_db( ).
    " Raise an event, targeting the workflow
  ENDMETHOD.

  METHOD get_workflow_log.
    " Read workflow log using BAPI calls and return
  ENDMETHOD.

  METHOD post_mm_doc.
    " Call BAPI and return messages
    " Update status of the invoice
  ENDMETHOD.

ENDCLASS.
```

Listing 27.3 Model Class Accepting Visitors

How do things work when the client wants to extend the functionality of this class? First of all, their developers need to define a new class that, for example, deletes files. You can see a sketch of this file-deleting class in Figure 27.3.

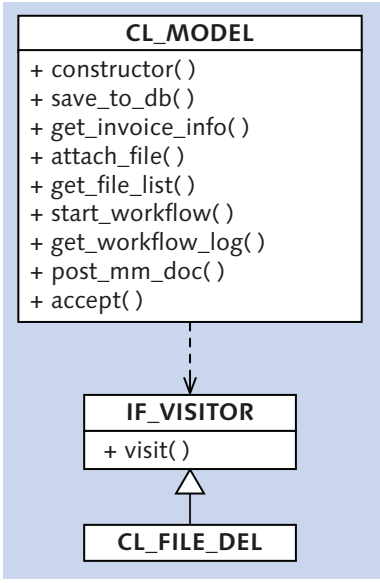


Figure 27.3 Visitor Class to Delete Files

Listing 27.4 demonstrates what CL_FILE_DEL would look like.

```
CLASS zcl_file_del DEFINITION
  PUBLIC
  FINAL CREATE PUBLIC.

  PUBLIC SECTION.

    INTERFACES zif_visitor.

  PRIVATE SECTION.
  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_file_del IMPLEMENTATION.

  METHOD zif_visitor~visit.
    DATA(lt_file) = io_model->get_file_list( ).

    LOOP AT lt_file ASSIGNING FIELD-SYMBOL(<ls_file>).
      " Some code to delete <ls_file> from GOS
    ENDLOOP.
  ENDMETHOD.

ENDCLASS.
```

Listing 27.4 Visitor Class to Delete a File

Now, whenever a client developer needs CL_MODEL to delete invoice files, all he/she needs to do is to pass an instance of CL_FILE_DEL to CL_MODEL, as demonstrated in Listing 27.5.

```
DATA:
  lo_deleter TYPE REF TO zcl_file_del,
  lo_model TYPE REF TO zcl_model,
  lo_visitor TYPE REF TO zif_visitor.

lo_model = NEW #( 15 ). " Imaginary invoice number

lo_deleter = NEW #( ).
lo_visitor ?= lo_deleter.

lo_model->accept( lo_visitor ). " Will delete stored files

ENDCLASS.
```

Listing 27.5 Making the Visit Happen

Using the visitor design pattern, we can extend CL_MODEL as much as we need to; see Figure 27.4 for further possible classes, such as CL_POST_FI to post FI documents and CL_SAVELOG to save messages to the application log.

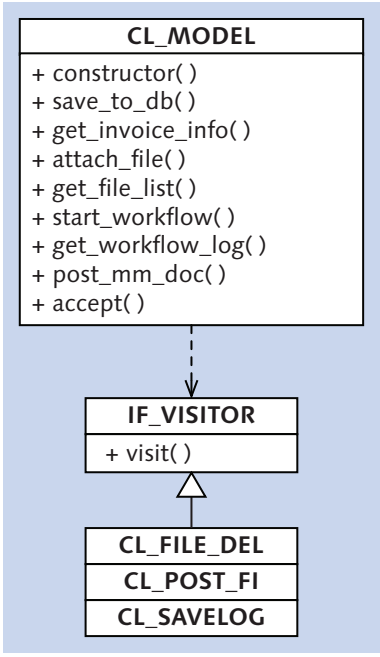


Figure 27.4 Further Visitor Classes

For example, the class CL_POST_FI in Listing 27.6 will have the same logic as CL_FILE_DEL had in Listing 27.4.

```
CLASS zcl_post_fi DEFINITION
  PUBLIC
  FINAL CREATE PUBLIC.

  PUBLIC SECTION.

    INTERFACES zif_visitor.

  PRIVATE SECTION.
  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_post_fi IMPLEMENTATION.
```

```
  METHOD zif_visitor~visit.
    io_model->save( ).
    DATA(ls_info) = io_model->get_invoice_info( ).
    " Call some BAPI to create the FI document using ls_info
  ENDMETHOD.

ENDCLASS.
```

Listing 27.6 Visitor Class to Post an FI Document

`CL_SAVELOG` won't look too different either, as can be seen in Listing 27.7.

```
CLASS zcl_savelog DEFINITION
  PUBLIC
  FINAL CREATE PUBLIC.

  PUBLIC SECTION.

    INTERFACES zif_visitor.

  PRIVATE SECTION.
  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_savelog IMPLEMENTATION.

  METHOD zif_visitor~visit.
    DATA(lt_log) = io_model->get_workflow_log( ).
    " Fill some Z table from lt_log
  ENDMETHOD.

ENDCLASS.
```

Listing 27.7 Visitor Class to Save Logs

To see a demonstration of `CL_POST_FI` and `CL_SAVELOG` in action, take a look at Listing 27.8.

```
DATA:
  lo_post_fi TYPE REF TO zcl_post_fi,
  lo_savelog TYPE REF TO zcl_savelog,
  lo_model TYPE REF TO zcl_model,
  lo_visitor TYPE REF TO zif_visitor.

lo_model = NEW #( 15 ). " Imaginary invoice number

lo_savelog = NEW #( ).
lo_visitor ?= lo_savelog.
lo_model->accept( lo_visitor ). " Will save WF log into Z-table
```

```
lo_post_fi = NEW #( ).
lo_visitor ?= lo_post_fi.
lo_model->accept( lo_visitor ). " Will post an FI document

ENDCLASS.
```

Listing 27.8 Multiple Visits in Action

Note

The names of the subclasses (`ZCL_FILE_DEL`, `ZCL_SAVELOG`, and `ZCL_PPOST_FI`) are hardcoded in this example. See Appendix B on subclass determination for alternative approaches.

At times, you might find yourself trying to decide whether a certain method needs to be placed into the model class or into a visitor class. The answer will depend on what the method does. If the method is part of the core purpose of the model class, place it into the class itself. If it provides an optional trivial functionality, or does something out of the class's core scope, place it into a visitor class.

Our case study has demonstrated that visitor is a useful design pattern when you want to add satellite functionalities to a central class without modifying/bloating it.

27.2 When to Use

We have already seen that visitor can be used with a VIP class. Another case is when similar operations need to be performed on objects of disparate types. For example, you may have distinct customer, vendor, and employee classes, each of which need to have their contact information formatted in a certain way. Instead of coding the format logic multiple times, you can have a distinct address formatting class that "visits" those classes to read and return the nicely formatted address. Note that servant design pattern (Chapter 23) might be the answer for this case as well.

Another case would be when many disparate operations need to be performed on objects of similar types. One example may be having an employee class and five employee reports. Instead of filling the employee class with methods related to reporting, for each report, you can have a distinct class that visits the employee class to get the required data and that calculates and formats the data as needed.

Similar to the VIP case, yet another example is when you expect many future operations to be added to an object. Imagine an MVC architecture where your model class runs through a complex calculation and returns an internal table, possibly for an ALV display. In the future, you may need to call four different Business Application Programming Interfaces (BAPIs), depending on what the user clicked. Instead of pumping up the model class with BAPI code, you can implement the visitor design pattern and have a visitor class for each BAPI in question. As a result, responsibilities are nicely distributed among distinct classes, and the central class remains unmodified.

When using the visitor design pattern, you should ensure that the visited class is fairly stable and won't change much over time. Every time you change something in the visited class, you might need to modify visitors as well, which could result in a huge test load.

27.3 Related Patterns

Occasionally, you may find yourself in a situation where you have to decide between the visitor and servant design patterns. Although they have different purposes, their functionalities may seem to overlap at first. In both patterns, you have a central class and multiple client classes that make use of the central class. So, what is the distinction?

If you have one master (VIP) class and want the flexibility to add new functionalities without modifying it, visitor would be the way to go. Remember: Visitor provides a tightly coupled relationship between the host class and the visitor class. In other words, you can't easily reuse the logic in the visitor class—unless you put extra effort into it.

If you have one master (VIP) functionality and want the flexibility to add it to multiple classes without modifying it, servant would be the way to go. Remember: Servant often forces client classes to implement a certain interface, which makes the classes and the interface coupled as well.

In either case, your choice of pattern should reflect your architectural intention, which will make the life of future programmers much easier. If your intention is to dynamically extend the functionality of a class, pick visitor. If your intention is to share a certain functionality among disparate classes in a web service fashion, pick servant.

27.4 Summary

The visitor design pattern lets us add new functionality to a model class without writing any code into it. Instead, we have little visitor classes sharing a common interface, each containing extension code.

Visitor is typically used when you have an untouchable VIP class or when you need to add satellite functionalities to a central class without making it bloated.

Appendices

A **Object-Oriented Programming** 339

B **Subclass Determination** 361

C **Principles** 371

D **The Author** 383

Design patterns provide time-proven templates for class hierarchies. Without basic knowledge of object-oriented programming (OOP), understanding and making use of any pattern is impossible. This appendix is a basic walkthrough for object-oriented ABAP and contains the minimum knowledge required to follow the book.

A Object-Oriented Programming

In this book, our main goal was to gain an understanding of design patterns, which requires an intermediate understanding of object-oriented programming. However, for readers without any prior OOP experience, this appendix will provide you with a quick look at the basics.

By end of this appendix, you should be familiar with the terminology and concepts associated with the ABAP development environment, including classes, superclasses, abstract classes, interfaces, and Unified Modeling Language (UML).

A.1 Object-Oriented ABAP Development Environment

When we talk about object-oriented programming, we will mostly be talking about classes. In a typical ABAP development environment, you can define a new class in two ways, as follows:

1. You can define a class as part of a program, as you would within Transaction SE38. In that case, you would typically define a local class that is only available to the main program.
2. You can define a class in Transaction SE24. In that case, the class would be available to all programs, functions, classes, etc. in the system.

Unless you are writing an event handler class for a local GUI object, defining global classes within Transaction SE24 is generally the best practice. If you won't be sharing your class for reusability, inheritance, etc., why create a class at all? The examples used in this book will all assume that your classes are being created globally in Transaction SE24.

You have two view options in Transaction SE24, configurable in your ABAP Workbench settings:

- FORM BASED VIEW
  This will display your types, attributes, and methods in a table control, and you can double-click them to access the code.
- SOURCE CODE BASED VIEW
  This will display the entire class in a text editor, much like ABAP programs in Transaction SE38.

Both views are useful in distinct ways, as follows—feel free to experience and decide which you like most:

- The form-based view is useful when you need an overview of all methods and edit them individually.
- The source code–based view is useful when you need to do bulk edit/copy/paste operations—simply because this view gives you access to the entire code at once. You may also prefer the source code–based view if you have a background in a popular non-SAP development environment, such as Java, .NET, etc. Most of those environments provide a text-based editor, which is similar to the source code–based view.

The latest generation of ABAP code is no longer written in SAP GUI, though SAP has provided us with the ABAP toolkit, which runs on Eclipse. Many developers prefer and recommend using Eclipse for ABAP coding due to the advantages brought by Eclipse. Some significant advantages are:

- Eclipse lets you open and arrange multiple ABAP objects in a single window.
- Even if you get disconnected, Eclipse keeps the unsaved code. You can keep working offline.
- You don't need to worry about disconnections when you are in the middle of a complex where-used list. Eclipse will keep your work until you reconnect.
- Eclipse tasks remember which ABAP objects you are working on so jumping between tasks is easy. You don't need to reopen the ABAP objects; Eclipse remembers them.
- You can query your SAP database directly from within Eclipse by writing any query with joins, aggregate functions, etc.
- Eclipse has great support for refactoring and provides powerful tools for text editing, such as IntelliSense and pattern insertion.

Any class you create using Eclipse will also appear in Transaction SE24. Both the Transaction SE24 and Eclipse development environments are intuitive, and many excellent resources on how to use them are available. An in-depth look at these development environments is outside the scope of this book, but if you are completely unfamiliar with Transaction SE24 or Eclipse, you can find many resources online to help.

Further Resources

SAP Community Network (SCN) provides helpful documents on ABAP in Eclipse, which is available at *http://scn.sap.com/community/abap/eclipse*.

Like many other tools, ABAP on Eclipse is available on SAP's on-demand site at *https://tools.hana.ondemand.com*.

A.2 Class

When talking about object-oriented programming, the first concept to understand is *class*. Many of you are probably familiar with function modules; basically, a class can be imagined as a collection of function modules. However, compared to a function group, a class has many more capabilities, which we'll see shortly. Listing A.1 demonstrates what a class looks like.

```
CLASS zcl_material DEFINITION
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.
    METHODS constructor IMPORTING !iv_matnr TYPE matnr.
    METHODS get_text RETURNING VALUE(rv_maktx) TYPE maktx.
  PRIVATE SECTION.
    DATA gv_matnr TYPE matnr.
  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_material IMPLEMENTATION.
  METHOD constructor.
    gv_matnr = iv_matnr.
  ENDMETHOD.

  METHOD get_text.
    SELECT SINGLE maktx INTO rv_maktx
      FROM makt
      WHERE matnr EQ gv_matnr
        AND spras EQ sy-langu.
```

```
  ENDMETHOD.

ENDCLASS.
```

Listing A.1 A Simple Class

Let's inspect the structure of the class now. We start with the command `CLASS`, where the name and basic properties of the class are defined. The important keywords are as follows:

- `FINAL` prevents the class from having any further subclasses. You can remove `FINAL` completely or substitute `ABSTRACT` if you are creating an abstract class. Abstract classes will be discussed in Section A.4.
- `CREATE PUBLIC` means any program, class, etc. can create an instance of this class freely. If we need to limit the creation of this class to a single method, we can change it to `CREATE PRIVATE`. Don't let this approach intimidate you; we'll explore it in greater detail shortly.

After the initial definition, we move forward to public, private, and protected sections. These sections will contain definitions of variables and methods only. You won't find any executable code here. In short, each section has the following properties:

- `PUBLIC SECTION`
  Any variable or method defined here can freely be accessed from any external program, class, etc. If you take a look at our sample program in Listing A.1, the method `GET_TEXT` is called. We are able to call `GET_TEXT` because the definition of `GET_TEXT` is placed under the `PUBLIC SECTION`.
- `PROTECTED SECTION`
  Any variable or method defined here can be accessed by the subclasses only; they can't be accessed from external programs. The concept of subclasses/inheritance will be discussed in Section A.3.
- `PRIVATE SECTION`
  Any variable or method defined here can't be accessed by external code at all; they can be accessed by the methods of `ZCL_MATERIAL` only. If you take a closer look at Listing A.1, `GV_MATNR` is marked as private. `GV_MATNR` is invisible to the report `ZSAMPLE`; however, all the methods of `ZCL_MATERIAL` can read and write `GV_MATNR`.

The distinction of public/protected/private sections is traditionally called *encapsulation*, one of the basic principles of object-oriented programming. Some other basic principles are *inheritance* (covered in Section A.3) and *polymorphism* (covered in Section A.4).

Moving forward, we get to the implementation section of the class. Here, we code each and every method. The implementation section is where you'll find the executable code, and this section is where you should place break points as well.

Now, how would we use this class in our regular ABAP program? Easily, actually—check out Listing A.2.

```
REPORT zsample.

  DATA(lo_mat) = NEW zcl_material( 'M12345' ).
  DATA(lv_maktx) = lo_mat->get_text( ).
  WRITE lv_maktx.
```

Listing A.2 How to Use a Class

This listing shows the basic structure of an ABAP class. If you are new to object-oriented ABAP, you can go to Transaction SE24, create the same sample class, and call it from a sample program.

A.3 Superclass

When we talk about a superclass, we almost inevitably must also talk about inheritance. Although class-to-class inheritance is not the preferred approach in design patterns, it is important to know that it is possible.

To understand superclasses and inheritance, imagine that you have a class that reads certain SAP ERP Sales and Distribution (SD) orders from the system and calls a Business Application Programming Interface (BAPI) to post delivery documents. Listing A.3 outlines what such a class would look like.

```
CLASS zcl_order DEFINITION
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.
    " Some data definitions
    METHODS read_orders.
    METHODS call_bapi.
  PRIVATE SECTION.
```

```
  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_order IMPLEMENTATION.
  METHOD read_orders.
    " Some complex code to read VBAK, VBAP, etc.
  ENDMETHOD.

  METHOD call_bapi.
    " Prepare ITABs expected by the delivery BAPI
    " Call delivery BAPI
  ENDMETHOD.

ENDCLASS.
```

Listing A.3 A Simple Class to Post a Delivery Document

Now, leaving all the bells and whistles aside, an extremely simple program using this class is demonstrated in Listing A.4.

```
DATA(lo_order) = NEW zcl_order( ).
lo_order->read_orders( ).
lo_order->call_bapi( ).
```

Listing A.4 Code Snippet to Consume the Class

So far, so good. Next, imagine that a new requirement has emerged. You now need another program that will read the exact same SD documents and call a BAPI to post invoice documents (but not delivery documents). You could achieve this goal in a dozen ways, and while inheritance is not the best approach here, we will move forward with it anyway, for illustrative purposes.

Listing A.5 demonstrates the code for this new program, using inheritance.

```
CLASS zcl_order2 DEFINITION
  INHERITING FROM zcl_order
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.
    METHODS call_bapi REDEFINITION.
  PRIVATE SECTION.
  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_order2 IMPLEMENTATION.

  METHOD call_bapi.
    " Prepare ITABs expected by the invoice BAPI
```

```
    " Call invoice BAPI
  ENDMETHOD.

ENDCLASS.
```

Listing A.5 A Basic Subclass

The sections of interest are marked as bold, and we perform the following actions with each:

- `INHERITING FROM zcl_order`
  With this statement, we are telling SAP that `ZCL_ORDER2` will be a clone of `ZCL_ORDER` and can be thought of a mirror of `ZCL_ORDER`. Whatever we change in `ZCL_ORDER` will also be changed in `ZCL_ORDER2`. As a result, if you change the method `ZCL_ORDER~READ_ORDERS`, then `ZCL_ORDER2~READ_ORDERS` will also be changed.
- `METHODS call_bapi REDEFINITION`
  With this statement, we are telling SAP that we don't want `ZCL_ORDER2` to be an exact clone of `ZCL_ORDER`. Instead, we want the method `CALL_BAPI` to behave differently, which makes sense—we want to call the BAPI to create invoices instead of deliveries. We didn't redefine the method `READ_ORDERS`; therefore, it stays the same.
- `METHOD call_bapi`
  With this statement, we code the method of the new class.

The program to create invoices will look similar to the program to create deliveries. The only thing to change is the name of the class, as can be seen in Listing A.6.

```
DATA(lo_order2) = NEW zcl_order2( ).
lo_order2->read_orders( ).
lo_order2->call_bapi( ).
```

Listing A.6 Using the Subclass

A.4 Abstract Class

The concept of abstract classes is similar to the concept of superclasses, with one major difference. For a superclass, inheritance is optional. You can use the superclass in your application directly or let it have children and use one of the subclasses in your application. In an abstract class, you can't create an object that refers to the abstract class directly. Your object may refer to one of the subclasses.

Creating an abstract class is actually a declaration of the original developer's intent. Whenever you see an abstract class, you can safely assume that the purpose of the abstract class is to make a partial implementation and to let the subclasses complete whatever is missing.

Let's take our earlier example and manipulate it to provide an example of an abstract class, as can be seen in Listing A.7.

```
CLASS zcl_order DEFINITION
  PUBLIC ABSTRACT CREATE PUBLIC.

  PUBLIC SECTION.
    " Some data definitions
    METHODS read_orders.
    METHODS call_bapi ABSTRACT.
  PRIVATE SECTION.
  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_order IMPLEMENTATION.
  METHOD read_orders.
    " Some complex code to read VBAK, VBAP, etc.
  ENDMETHOD.

ENDCLASS.
```

Listing A.7 A Simple Abstract Class

The logic behind this abstract class is pretty intuitive. By marking the class as abstract in the definition part, we tell SAP that this class can't be instantiated directly. The method `CALL_BAPI` is also marked as abstract, which means we can't code this method in `ZCL_ORDER`. Instead, we leave this task to its subclasses.

Semantically, one of the subclasses would call the delivery BAPI, and the other one would call the invoice BAPI. Listing A.8 demonstrates how this works.

```
CLASS zcl_order_dlv DEFINITION
  INHERITING FROM zcl_order
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.
    METHODS call_bapi REDEFINITION.
  PRIVATE SECTION.
  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_order_dlv IMPLEMENTATION.
```

```
  METHOD call_bapi.
    " Prepare ITABs expected by the delivery BAPI
    " Call delivery BAPI
  ENDMETHOD.

ENDCLASS.
```

Listing A.8 A Subclass of the Abstract Class

Not too different from what we have seen in superclasses, right? To complete the example, let's take a look at the second subclass in Listing A.9 as well.

```
CLASS zcl_order_inv DEFINITION
  INHERITING FROM zcl_order
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.
    METHODS call_bapi REDEFINITION.
  PRIVATE SECTION.
  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_order_inv IMPLEMENTATION.

  METHOD call_bapi.
    " Prepare ITABs expected by the invoice BAPI
    " Call invoice BAPI
  ENDMETHOD.

ENDCLASS.
```

Listing A.9 Another Subclass of the Abstract Class

In its simplest form, using one of the subclasses in our program is not too different from our previous examples in Section A.3. Listing A.10 shows this in action.

```
DATA(lo_order_dlv) = NEW zcl_order_dlv( ).
lo_order_dlv->read_orders( ).
lo_order_dlv->call_bapi( ).

DATA(lo_order_inv) = NEW zcl_order_inv( ).
lo_order_inv->read_orders( ).
lo_order_inv->call_bapi( ).
```

Listing A.10 Using Both Subclasses

At this point, we can introduce the concept of polymorphism. Polymorphism is the ability to present the same interface for different underlying classes. You may

have different classes serving different purposes. However, if they share the same interface, you can easily use them interchangeably. You can even dynamically determine which class to use during runtime.

Since both ZCL_ORDER_DLV and ZCL_ORDER_INV are subclasses of ZCL_ORDER, they can be cast onto an object of type ZCL_ORDER. It's simpler than it sounds, actually—just check out the code snippet in Listing A.11.

```
DATA lo_order TYPE REF TO zcl_order.

DATA(lo_order_dlv) = NEW zcl_order_dlv( ).
lo_order ?= lo_order_dlv.
lo_order->read_orders( ).
lo_order->call_bapi( ).
```

Listing A.11 Basic Polymorphism

This code looks cool, but how is it useful? Well, imagine that your clients want a radio button on the selection screen where he/she can decide between creating a delivery or an invoice. Without polymorphism, your code would look like Listing A.12.

```
IF p_dlv EQ abap_true.
  DATA(lo_order_dlv) = NEW zcl_order_dlv( ).
  lo_order_dlv->read_orders( ).
  lo_order_dlv->call_bapi( ).
ENDIF.

IF p_inv EQ abap_true.
  DATA(lo_order_inv) = NEW zcl_order_inv( ).
  lo_order_inv->read_orders( ).
  lo_order_inv->call_bapi( ).
ENDIF.
```

Listing A.12 Lack of Polymorphism

In this code, the method call sequence is duplicated. This duplication might look OK in this simple example, but if you had a complex case where you deal with twenty methods among various business rules, copying and pasting the code for different BAPI types is not the best idea—and it will make future maintenance a nightmare.

Instead, we can take advantage of polymorphism and make the code look like Listing A.13.

```
DATA lo_order TYPE REF TO zcl_order.

IF p_dlv EQ abap_true.
  DATA(lo_order_dlv) = NEW zcl_order_dlv( ).
  lo_order ?= lo_order_dlv.
ENDIF.

IF p_inv EQ abap_true.
  DATA(lo_order_inv) = NEW zcl_order_inv( ).
  lo_order ?= lo_order_inv.
ENDIF.

lo_order->read_orders( ).
lo_order->call_bapi( ).
```

Listing A.13 Polymorphism in Action

As you can see, after creating the appropriate object and casting onto `LO_ORDER`, the only object we need to deal with is `LO_ORDER`, and we can avoid code repetition altogether. User preference will determine how the program will behave, as follows:

- If `P_DLV` is marked, `LO_ORDER` will behave like `ZCL_ORDER_DLV`. Therefore, `LO_ORDER->CALL_BAPI` will execute `ZCL_ORDER_DLV~CALL_BAPI`.
- If `P_INV` is marked, `LO_ORDER` will behave like `ZCL_ORDER_INV`. Therefore, `LO_ORDER->CALL_BAPI` will execute `ZCL_ORDER_INV~CALL_BAPI`.

Here is another cool trick: You can even determine the name of the class during runtime. Listing A.14 shows how this can be done.

```
DATA:
  lo_obj TYPE REF TO object,
  lo_order TYPE REF TO zcl_order,
  lv_clsname TYPE seoclsname.

* Determine name of the subclass we need
  lv_clsname = COND #(
    WHEN p_dlv EQ abap_true THEN 'ZCL_ORDER_DLV'
    ELSE 'ZCL_ORDER_INV' ).

* Create object
  CREATE OBJECT lo_obj TYPE (lv_clsname).
  lo_order ?= lo_obj.

* Do stuff
  lo_order->read_orders( ).
  lo_order->call_bapi( ).
```

Listing A.14 Polymorphism with Dynamic Class Names

This approach comes with a risk though: If you give a nonexistent class name, a nice short dump in Transaction ST22 might be waiting for you. Despite that, you can move a step forward and even keep the class names in a Z-table, as seen in Listing A.15.

```
DATA:
  lo_obj TYPE REF TO object,
  lo_order TYPE REF TO zcl_order.

* Determine name of the subclass we need
  DATA(lv_ctype) = COND zctype(
    WHEN p_dlv EQ abap_true THEN 'ORDER'
    ELSE 'INVOICE' ).

  SELECT SINGLE clsname INTO @DATA(lv_clsname)
    FROM zt_classes
    WHERE clstype EQ @lv_ctype.

* Create object
  CREATE OBJECT lo_obj TYPE (lv_clsname).
  lo_order ?= lo_obj.

* Do stuff
  lo_order->read_orders( ).
  lo_order->call_bapi( ).
```

Listing A.15 Polymorphism with Dynamic Class Names from a Custom Table

The concept of polymorphism works perfectly on superclasses as well but is not typically preferred because using polymorphism on concrete classes encourages inheritance. The principles of design patterns state that object composition should be preferred over inheritance. You can read more about polymorphism and object composition in Appendix C.

Here is a pro tip: In your abstract class, the methods that must be redefined are marked as `ABSTRACT`. The methods that can optionally be redefined are not marked at all. What if you have methods that should not be redefined? Well, you can mark them as `FINAL`. You can see both `ABSTRACT` and `FINAL` at work in Listing A.16.

```
CLASS zcl_order DEFINITION
  PUBLIC ABSTRACT CREATE PUBLIC.

  PUBLIC SECTION.
    " Some data definitions
    METHODS read_orders FINAL.
    METHODS call_bapi ABSTRACT.
```

```
  PRIVATE SECTION.
  PROTECTED SECTION.
ENDCLASS.

CLASS zcl_order IMPLEMENTATION.
  METHOD read_orders.
    " Some complex code to read VBAK, VBAP, etc.
  ENDMETHOD.

ENDCLASS.
```

Listing A.16 Marking Methods as FINAL

In this case, the method `READ_ORDERS` can never, ever be redefined by subclasses. Doing so not only makes your abstract class more robust but also makes your intention clearer to other programmers.

A.5 Interface

Moving from superclasses to abstract classes, we have gained one degree of abstraction. To gain another degree of abstraction, we will move from abstract classes to interfaces.

Basically, an interface is a class that contains no code and can't be instantiated. If it contains no code at all, then how on earth is an interface useful? There's no short answer to that question, but this entire book demonstrates how useful interfaces are, and by the end of it, you should be able to answer this question for yourself. For now, let's move forward to the example.

Defining a new interface is extremely simple; think of it as a stripped-down version of a class. A basic interface can be seen in Listing A.17.

```
INTERFACE zif_company.
  PUBLIC .

  TYPES:
    BEGIN OF t_info,
      name1 TYPE name1_gp,
      land1 TYPE land1_gp,
    END OF t_info.

 METHODS get_info
    IMPORTING
      !iv_code TYPE char10
```

```
    RETURNING
      VALUE(rs_info) TYPE t_info.

ENDINTERFACE.
```

Listing A.17 A Simple Interface

This interface sets the standard for each future class of companies. For example, we may need to implement a class that returns the name of a given customer and another one that returns the name of a given vendor. If both classes comply with the standards provided in `IF_COMPANY`, we will have an easier time using those classes. Let's take a look at the customer class in Listing A.18.

```
CLASS zcl_customer DEFINITION
  PUBLIC FINAL CREATE PUBLIC.

  PUBLIC SECTION.
    INTERFACES zif_company.
  PRIVATE SECTION.
PROTECTED SECTION.
ENDCLASS.

CLASS zcl_customer IMPLEMENTATION.

  METHOD zif_company~get_info.
    SELECT SINGLE name1 land1
      INTO CORRESPONDING FIELDS OF rs_info
      FROM kna1
      WHERE kunnr eq iv_code.
  ENDMETHOD.

ENDCLASS.
```

Listing A.18 Sample Class Implementing the Interface

In the structure of the class, you will notice the snippet `INTERFACES zif_company`. This command tells SAP that this class will behave nicely and comply with the standards set by `ZIF_COMPANY`. This ensures that the class will have a method called `GET_INFO` to return the basic info of the given company; in our example, this is a customer. As a result, the class is predictable and interchangeable, which is a concept we will see in a moment. But first, let's take a look at our vendor class in Listing A.19.

```
CLASS zcl_vendor DEFINITION
  PUBLIC FINAL CREATE PUBLIC.
```

```
  PUBLIC SECTION.
    INTERFACES zif_company.
  PRIVATE SECTION.
PROTECTED SECTION.
ENDCLASS.

CLASS zcl_vendor IMPLEMENTATION.

  METHOD zif_company~get_info.
    SELECT SINGLE name1 land1
      INTO CORRESPONDING FIELDS OF rs_info
      FROM lfa1
      WHERE lifnr eq iv_code.
  ENDMETHOD.

ENDCLASS.
```

Listing A.19 Another Sample Class Implementing the Interface

Listing A.19 is similar to the customer class, but that's the point. Since we set the standards via the interface, each concrete class implementing this interface will have the exact same public structure.

So, how is that useful? First of all, we can juggle the classes and change/determine them during runtime, just like what we did with abstract classes. Listing A.20 demonstrates this technique.

```
DATA lo_company TYPE REF TO zif_company.

* Based on the selection screen,
* create the target object & cast to interface
  IF p_cust EQ abap_true.
    DATA(lo_cust) = NEW zcl_customer( ).
    lo_company ?= lo_cust.
  ENDIF.

  IF p_vend EQ abap_true.
    DATA(lo_vend) = NEW zcl_vendor( ).
    lo_company ?= lo_vend.
  ENDIF.

* Get details & write
  DATA(ls_info) = lo_company->get_info( p_code ).
  WRITE: ls_info-name1, ls_info-land1.
```

Listing A.20 Casting Objects onto Interface References

As we did in Listing A.14, we could have created the classes dynamically, as demonstrated in Listing A.21.

```
DATA:
  lo_obj TYPE REF TO object,
  lo_company TYPE REF TO zif_company.

* Based on the selection screen, determine class name
  DATA(lv_clsname) = COND seoclsname(
    WHEN p_cust EQ abap_true THEN 'ZCL_CUSTOMER'
    WHEN p_vend EQ abap_true THEN 'ZCL_VENDOR' ).

* Create the target object & cast to interface
  CREATE OBJECT lo_obj TYPE (lv_clsname).
  lo_company ?= lo_obj.

* Get details & write
  DATA(ls_info) = lo_company->get_info( p_code ).
  WRITE: ls_info-name1, ls_info-land1.
```

Listing A.21 Dynamic Class Names

Note

Storing class names in a Z-table is usually a good practice.

In the world of design patterns, you should start from the most abstract level and move down to the concrete level. In practice, if you are going to have interchangeable classes, you should almost always start with an interface and create concrete classes accordingly. Your client program should only know about the interface but know nothing about the concrete classes because new classes might be created in the future and you want to avoid modifying an existing, tested client program.

In other words, object composition via interfaces is generally preferred over inheritance via superclasses. Check out Appendix C, Section C.1.2 and Section C.1.3, for more information.

However, after creating an interface, but before moving forward to the concrete classes, you may realize that some of your concrete classes are going to behave similarly to each other. In this case, don't hesitate to create an intermediate abstract class (implementing your interface) and derive your concrete classes from it. If this sounds too complicated at this point, don't worry—you will see detailed examples of this concept within the samples of individual design patterns.

Knowing that a class can implement multiple interfaces is useful. However, a class may have only one parent superclass or abstract class (which is optional).

Here is a pro tip: If you are implementing an interface into an abstract class, you can define which methods are abstract or final. Let's see some syntax examples. In Listing A.22, all methods of IF_SAMPLE are marked as abstract and must be implemented by subclasses.

```
CLASS zcl_order DEFINITION
  PUBLIC ABSTRACT CREATE PUBLIC.

  PUBLIC SECTION.
    INTERFACES if_sample ALL METHODS ABSTRACT.
```

Listing A.22 Marking All Methods as Abstract

In Listing A.23, all methods of IF_SAMPLE are marked as final and must be implemented by the abstract class.

```
CLASS zcl_order DEFINITION
  PUBLIC ABSTRACT CREATE PUBLIC.

  PUBLIC SECTION.
    INTERFACES if_sample ALL METHODS FINAL.
```

Listing A.23 Marking All Methods as Final

In Listing A.24, METH1 and METH2 are marked as abstract and must be implemented by subclasses. METH3 and METH4 are marked as final and must be implemented by the abstract class. The rest of the methods are left in their default state.

```
CLASS zcl_order DEFINITION
  PUBLIC ABSTRACT CREATE PUBLIC.

  PUBLIC SECTION.
    INTERFACES if_sample
      ABSTRACT METHODS meth1 meth2
      FINAL METHODS meth3 meth4.
```

Listing A.24 Mixed Marking

A.6 UML

UML, which stands for *Unified Modeling Language*, is a simple, yet effective, standard to signify class hierarchies. All of examples in this book are accompanied by corresponding UML diagrams, so it is important for you to understand UML before you move on the rest of the book. If you feel comfortable with this subject, you can skip this section.

Rather than start off with theoretical explanations, we'll take a shortcut and jump directly into an exploration of UML shapes. Some shapes may look like those in your daily flowcharts, but those shapes probably mean something else, so don't be fooled!

In their most basic forms, classes and interfaces are represented using a rectangle. In Figure A.1, we see a class and an interface that are completely unrelated to each other.

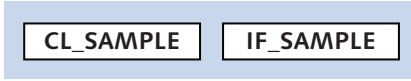


Figure A.1 Basic UML Shapes

Attributes of a class are represented by signs after its name, as seen in Figure A.2. Public attributes start with a + sign, while private attributes start with a – sign. Protected attributes start with a # sign.

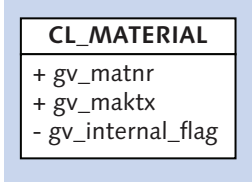


Figure A.2 Attributes in UML

The methods of a class are represented following its attributes. The signs have the exact same meaning as in attributes, as seen in Figure A.3.

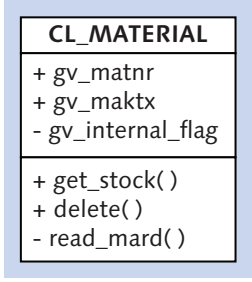


Figure A.3 Methods in UML

So far, we've seen instance attributes and methods. What about static ones? In UML, they are simply underlined. In Figure A.4, GT_CACHE and GET_INSTANCE are static.

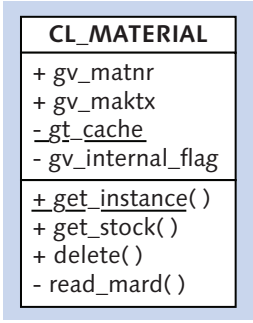


Figure A.4 Static Elements in UML

In Figure A.5, we see the representation of an inheritance where CL_CHILD has been derived from CL_PARENT. Meanwhile, you may have noticed that METHOD2 is in italics, meaning that the method is abstract and will be implemented by CL_CHILD.

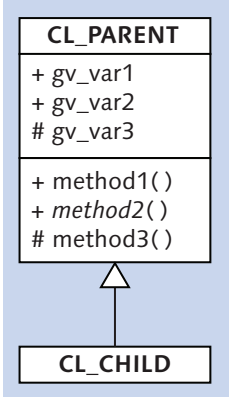


Figure A.5 Class Inheritance in UML

The same shape can be used for interface implementations as well. In Figure A.6, both classes have implemented the interface IF_DATA.

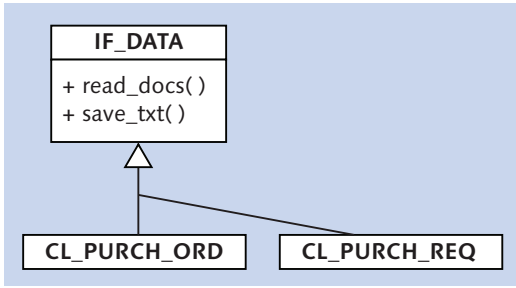


Figure A.6 Interface Implementation in UML

In Figure A.7, we see the representation of containment. A purchase order contains instances of `CL_LINE_ITEM` inside `GT_LINE_ITEM`. On the other hand, `CL_LINE_ITEM` contains an instance of `CL_MATERIAL` in `GO_MATERIAL`.

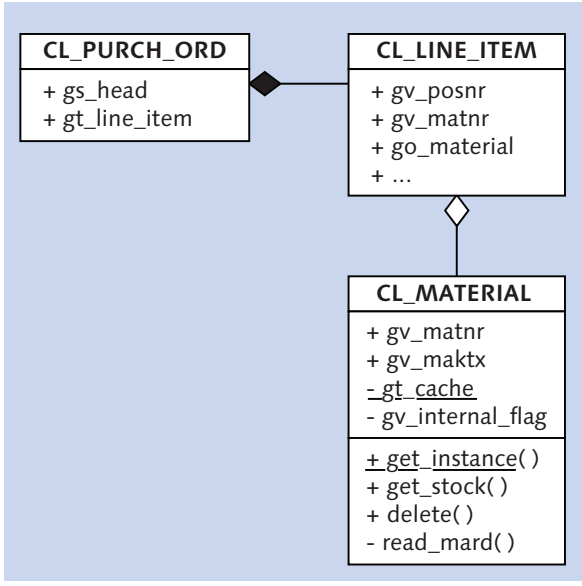


Figure A.7 Containment

You have probably noticed that, next to `CL_PURCH_ORD`, a filled diamond appears, while next to `CL_LINE_ITEM`, an empty diamond appears. The filled shape is used when the component can't exist without the container. In our example, a purchase order line item can't exist without the purchase order itself. Therefore, this relation is represented with a filled shape.

The empty shape is used when the component can exist without the container. In our example, the existence of a material is completely independent from the existence of a purchase order line item. Therefore, this relation is represented with an empty shape.

In Figure A.8, we see the representation of usage. We have a class called `CL_POST_PO`, which is used to create purchase orders. In order to do so, `CL_POST_PO` takes advantage of the utility classes `CL_BATCH_INPUT` and `CL_APP_LOG`.

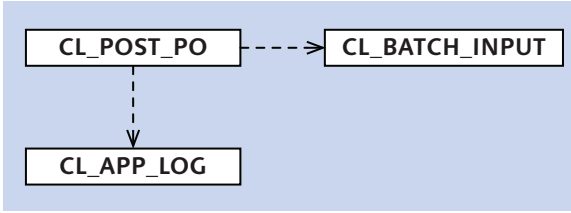


Figure A.8 Usage

Remember that UML diagrams are visual representations for your application. A good idea to start is to start with the UML first and start coding only after you are satisfied with the diagram. If your UML diagram is too complicated to be drawn, your software design is probably too complicated to be implemented.

There is more to UML diagrams, of course. We've provided you with some basic knowledge that will be enough to follow the examples in this book.

Further Reading

If you want to go deeper, *http://www.uml.org* contains everything you need to know about UML. This resource can be considered the "official" UML homepage.

A.7 Summary

In this appendix, we have inspected the basic elements of object-oriented programming. For some, we merely refreshed your knowledge of the known concepts. For others, we provided the necessary preparation to ensure that you know the concepts and terms required to follow this book.

Class is the base concept of object-oriented programming and can be described as a collection of attributes and methods. We can mark the elements as `PUBLIC`, `PROTECTED`, or `PRIVATE`, and define their scope as `STATIC` or `INSTANCE`.

A class can be created from the scratch or can be derived from another class, where all the attributes and methods are inherited. In such a case, the parent class is called the superclass. We can modify the methods in the new class as needed. Inheritance can be forced or limited with help of the keywords `ABSTRACT` and `FINAL`.

If we create a partially implemented superclass with the intention of letting other developers derive new subclasses from it, this specific superclass is called an abstract class. You can't create objects from an abstract class; it merely serves as a template for other classes.

If we increase the level of abstraction even more, we arrive at the concept of interfaces. An interface can be defined as a basic class definition that only contains the method signatures for upcoming classes.

We have also discussed the concept of casting, in which the following is true:

- Any class derived from a superclass can be cast onto the superclass.
- Any class derived from an abstract class can be cast onto the abstract class.
- Any class implementing an interface can be cast onto the interface.

Finally, we made ourselves familiar with basic UML shapes to represent class relations.

B Subclass Determination

The various examples in this book share a common task to be fulfilled. The developer needs to determine which classes implement a certain interface or which classes are derived from a certain abstract or parent class. The examples contain some scattered solutions for this requirement. However, this appendix consolidates the possible solutions as a guideline to inspire you.

The most significant approaches to determine subclasses are hardcoding their names, applying a certain convention rule, using SAP's standard class tables, or creating a custom table to store class names. Let's inspect each approach individually.

B.1 Hardcoding

Hardcoding class names is neither the most flexible nor the most elegant way to determine subclasses. However, this appendix would be incomplete without mentioning this notorious approach. If the subclass your client application needs to use is fixed and you are relatively certain that it won't ever change, hardcoding the class name can be used as a quick and dirty approach.

Listing B.1 demonstrates an example.

```
DATA:
  lo_int TYPE REF TO zif_my_interface,
  lo_cls TYPE REF TO zcl_my_class.

  lo_cls = NEW #( ).
  lo_int ?= lo_cls.

  lo_int->do_something( ).
```

Listing B.1 Hardcoded Class Names

B.2 Convention over Configuration

This approach requires that your class names follow a standard convention or pattern. Imagine that you have four scenarios: A, B, C, and D. Assuming that your classes must implement the interface `ZIF_MY_INTERFACE`, the name of each implementing class would contain the name of the scenario:

- ZCL_MY_CLASS_A
- ZCL_MY_CLASS_B
- ZCL_MY_CLASS_C
- ZCL_MY_CLASS_D

If it is possible to use a pattern like this one, you can build the class name by a simple concatenate operation and dynamically create the corresponding object instance. An example of this can be seen in Listing B.2.

```
DATA:
  lo_int TYPE REF TO zif_my_interface,
  lo_obj TYPE REF TO object,
  lv_clsname TYPE seoclsname.

  DATA(lv_scenario) = get_scenario( ). " Contains A, B, C or D
  lv_clsname = |ZCL_MY_CLASS_{ lv_scenario }|.

  CREATE OBJECT lo_obj TYPE (lv_clsname).
  lo_int ?= lo_obj.

  lo_int->do_something( ).
```

Listing B.2 Class Determination via Convention

A naming convention is useful for patterns in which you need to determine which unique class can fulfill the required task during runtime. The strategy (Chapter 25), data access object (DAO, Chapter 12), and adapter (Chapter 9) design patterns are some example patterns.

The risk of this approach is that you depend entirely on the naming convention of the classes. Think about the chain of responsibility design pattern (Chapter 18), for instance, where you daisy-chain classes, hoping that one of them would handle the request at hand. In this approach, you will probably have multiple classes corresponding to the same scenario. If your convention is based on differentiating class names by the scenario name, you will end up in a dead end because you can't have multiple classes with the same name.

If one of your fellow developers fails to follow the convention, the approach will fail as well.

Therefore, this approach also requires some defensive programming. If the class doesn't exist, you should raise an exception. You can check to see if a class exists by looking into `SEOCLASS` first or simply using `TRY…CATCH` for the `CREATE OBJECT` command.

B.3 SAP Class Tables

Table SEOMETAREL is one of the magical tables of SAP. It contains the parent–child relations among classes within the system. If ZCL_MY_CLASS has implemented the interface ZIF_MY_INTERFACE, then SEOMETAREL will contain an entry where:

- CLSNAME = ZCL_MY_CLASS
- REFCLSNAME = ZIF_MY_INTERFACE

Therefore, if you need to get a list of all the classes implementing ZIF_MY_INTERFACE, you can simply get that information from SEOMETAREL, as seen in Listing B.3.

```
DATA:
  lo_int TYPE REF TO zif_my_interface,
  lo_obj TYPE REF TO object.

  SELECT clsname INTO TABLE @DATA(lt_smr)
    FROM seometarel
    WHERE refclsname EQ 'ZIF_MY_INTERFACE'.

  LOOP AT lt_smr ASSIGNING FIELD-SYMBOL(<ls_smr>).
    CREATE OBJECT lo_obj TYPE (<ls_smr>-clsname).
    lo_int ?= lo_obj.
    lo_int->do_something( ).
  ENDLOOP.
```

Listing B.3 Determining Interface Implementations Using Table SEOMETAREL

This approach is useful for patterns where you need to loop through all classes implementing a certain interface. The decorator (Chapter 13), chain of responsibility (Chapter 18), and observer (Chapter 22) design patterns are some examples.

The same approach can be used with parent classes as well. If ZCL_MY_CLASS is derived from ZCL_MY_ABSTRACT, then SEOMETAREL will contain an entry with the following:

- CLSNAME = ZCL_MY_CLASS
- REFCLSNAME = ZCL_MY_ABSTRACT

The code to access the list of child classes is identical to Listing B.3, as can be seen in Listing B.4.

```
DATA:
  lo_abs TYPE REF TO zcl_my_abstract,
  lo_obj TYPE REF TO object.
```

```
SELECT clsname INTO TABLE @DATA(lt_smr)
  FROM seometarel
  WHERE refclsname EQ 'ZCL_MY_ABSTRACT'.

LOOP AT lt_smr ASSIGNING FIELD-SYMBOL(<ls_smr>).
  CREATE OBJECT lo_obj TYPE (<ls_smr>-clsname).
  lo_abs ?= lo_obj.
  lo_abs->do_something( ).
ENDLOOP.
```

Listing B.4 Determining Subclasses Using Table SEOMETAREL

There is a catch, however: If you have used intermediate abstract classes, which can't be instantiated, your application will produce a short dump. For example, let's say that we have the hierarchy in Figure B.1.

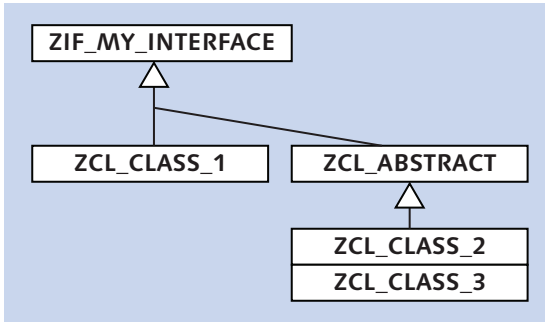


Figure B.1 Class Hierarchy Containing an Intermediate Abstract Class

The hierarchy consists of an interface, `ZIF_MY_INTERFACE`, which is implemented by the concrete class `ZCL_CLASS_1` and the abstract class `ZCL_ABSTRACT`. Furthermore, the abstract class has two concrete child classes: `ZCL_CLASS_2` and `ZCL_CLASS_3`.

In this case, `SEOMETAREL` will look like Table B.1.

CLSNAME	REFCLSNAME
ZCL_CLASS_1	ZIF_MY_INTERFACE
ZCL_ABSTRACT	ZIF_MY_INTERFACE
ZCL_CLASS_2	ZCL_ABSTRACT
ZCL_CLASS_3	ZCL_ABSTRACT

Table B.1 Class Hierarchy in the Database SEOMETAREL

If you selected all the entries corresponding to ZIF_MY_INTERFACE, you would select ZCL_ABSTRACT as well. Since abstract classes can't be instantiated directly, you can't create an instance of ZCL_ABSTRACT. Additionally, you will miss ZCL_CLASS_2 and ZCL_CLASS_3. To overcome this difficulty, you would need to write some recursive code to browse through SEOMETAREL entries and preserve instanceable classes only.

Listing B.5 demonstrates a quick and dirty sample code for this requirement.

```
CLASS zcl_bc_ooa_toolkit DEFINITION
  PUBLIC
  FINAL
  CREATE PUBLIC .

  PUBLIC SECTION.

    TYPES:
      BEGIN OF t_clsname,
         clsname TYPE seoclsname,
      END OF t_clsname,

      tt_clsname TYPE STANDARD TABLE OF t_clsname
        WITH DEFAULT KEY,
      rt_clsname_rng TYPE RANGE OF seoclsname.

  METHODS get_instanceable_subclasses
      IMPORTING
        !iv_refclsname TYPE seometarel-refclsname
      RETURNING
        VALUE(rt_clsname) TYPE tt_clsname.

  METHODS get_subclasses
      IMPORTING
        !iv_refclsname TYPE seometarel-refclsname
      RETURNING
        VALUE(rt_clsname) TYPE tt_clsname.

  PRIVATE SECTION.

    CONSTANTS: c_option_eq TYPE ddoption VALUE 'EQ',
               c_sign_i    TYPE ddsign VALUE 'I'.

    DATA gt_clsname TYPE tt_clsname.

    METHODS get_subclasses_recursive
        IMPORTING
          !iv_refclsname TYPE seometarel-refclsname
          !iv_rec TYPE abap_bool DEFAULT abap_false
```

```
        RETURNING
          VALUE(rt_clsname) TYPE tt_clsname.

  PROTECTED SECTION.

ENDCLASS.

CLASS zcl_bc_ooa_toolkit IMPLEMENTATION.

  METHOD get_instanceable_subclasses.

    CLEAR gt_clsname[].
    rt_clsname = get_subclasses_recursive( iv_refclsname ).
    CHECK rt_clsname[] IS NOT INITIAL.

    DATA(lt_clsname_rng) = VALUE tt_clsname_rng(
      FOR ls_cn IN rt_clsname (
        option = c_option_eq
        sign = c_sign_i
        low = ls_cn-clsname ) ).

    SELECT clsname INTO TABLE @DATA(lt_abstract)
      FROM seoclassdf AS sd1
      WHERE clsname IN @lt_clsname_rng
        AND version EQ (
          SELECT MAX( version )
          FROM seoclassdf AS sd2
          WHERE clsname EQ sd1~clsname )
        AND clsabstrct EQ @abap_true
        ORDER BY clsname.

    LOOP AT rt_clsname ASSIGNING FIELD-SYMBOL(<ls_clsname>).

      READ TABLE lt_abstract TRANSPORTING NO FIELDS
        WITH KEY clsname = <ls_clsname>-clsname
        BINARY SEARCH.

      CHECK sy-subrc EQ 0.

      DELETE rt_clsname.
      CONTINUE.

    ENDLOOP.

  ENDMETHOD.

  METHOD get_subclasses_recursive.

    DATA(lt_clsname_local) = get_subclasses( iv_refclsname ).
```

```
    LOOP AT lt_clsname_local ASSIGNING FIELD-SYMBOL(<ls_cl>).
      APPEND:
        <ls_cl> TO gt_clsname,
        LINES OF get_subclasses_recursive(
          iv_refclsname = <ls_cl>-clsname
          iv_rec = abap_true ) TO gt_clsname.
    ENDLOOP.

    CHECK iv_rec EQ abap_false.

    SORT gt_clsname BY clsname.
    DELETE ADJACENT DUPLICATES FROM gt_clsname COMPARING clsname.

    rt_clsname[] = gt_clsname[].

  ENDMETHOD.

  METHOD get_subclasses.
    SELECT clsname INTO CORRESPONDING FIELDS OF TABLE rt_clsname
      FROM seometarel
      WHERE refclsname EQ iv_refclsname.
  ENDMETHOD.

ENDCLASS.
```

Listing B.5 Live Code for Recursive Instanceable Class Determination

Let's inspect the methods of this class:

- `GET_SUBCLASSES` is a dull public method that simply reads the database table `SEOMETAREL` on a singular level. If we call the method for `ZIF_MY_INTERFACE`, it would return `ZCL_CLASS_1` and `ZCL_ABSTRACT`, which is not desirable if we need to get all instanceable subclasses. However, `GET_SUBCLASSES` is a necessary utility method.
- `GET_INSTANCEABLE_SUBCLASSES` is the magic public method. This method will read `SEOMETAREL` recursively via the private methods of the class and return instanceable subclasses only. If we call this method for `ZIF_MY_INTERFACE`, it would return `ZCL_CLASS_1`, `ZCL_CLASS_2`, and `ZCL_CLASS_3`.
- `GET_SUBCLASSES_RECURSIVE` is the hidden recursive method under the hood. This method recursively reads `SEOMETAREL` for the given class name and returns a list of classes. While `GET_INSTANCEABLE_SUBCLASSES` gets the spotlight, `GET_SUBCLASSES_RECURSIVE` is the hero behind the scenes, like Alfred to Batman!

B.4 Custom Table

In this approach, you create a Z-table and simply store the class names. A custom table is useful for scenarios where you need to process the classes in a certain order or where you want to be selective about which classes to process.

Imagine a Z-table like the one in Table B.2, where each class has implemented `ZIF_MY_INTERFACE`.

CLSNAME	PROCESS_ORDER
ZCL_DELIVERY	2
ZCL_INVOICE	3
ZCL_ORDER	1

Table B.2 Database Containing Class Names and Their Process Orders

If you need to process those classes in the given order, your code will take `PROCESS_ORDER` into consideration, as seen in Listing B.6.

```
DATA:
  lo_int TYPE REF TO zcl_my_interface,
  lo_obj TYPE REF TO object.

  SELECT clsname INTO TABLE @DATA(lt_z)
    FROM ztable
    ORDER BY process_order.

  LOOP AT lt_z ASSIGNING FIELD-SYMBOL(<ls_z>).
    CREATE OBJECT lo_obj TYPE (<ls_z>-clsname).
    lo_int ?= lo_obj.
    lo_int->do_something( ).
  ENDLOOP.
```

Listing B.6 Class Determination Using a Custom Table

Another approach is to use selective classes. For example, if you are implementing the decorator design pattern (Chapter 13), the initialization of some decorator objects may take a long time, so you want to leave irrelevant decorators out during runtime. This goal can be achieved by using a Z-table where each decorator must be registered and marked for their relevant scenarios, as seen in Table B.3.

CLSNAME	RELEVANT_SCENARIO
ZCL_DECORATOR_1	A
ZCL_DECORATOR_1	B
ZCL_DECORATOR_1	C
ZCL_DECORATOR_2	A
ZCL_DECORATOR_2	D
ZCL_DECORATOR_3	D

Table B.3 Database Containing Class Names and Their Relevant Scenarios

The code making use of this table would look like Listing B.7.

```
DATA:
  lo_int TYPE REF TO zif_my_interface,
  lo_obj TYPE REF TO object,
  lv_clsname TYPE seoclsname.

  DATA(lv_scenario) = get_scenario( ). " Contains A, B, C or D

  SELECT clsname INTO TABLE @DATA(lt_z)
    FROM ztable
    WHERE relevant_scenario EQ @lv_scenario.

  LOOP AT lt_z ASSIGNING FIELD-SYMBOL(<ls_z>).
    CREATE OBJECT lo_obj TYPE (<ls_z>-clsname).
    lo_int ?= lo_obj.
    lo_int->do_something( ).
  ENDLOOP.
```

Listing B.7 Scenario-Based Class Determination Using a Custom Table

How will this code run?

- If we have scenario A, `ZCL_DECORATOR_1` and `ZCL_DECORATOR_2` will be processed.
- If we have scenario B, only `ZCL_DECORATOR_1` will be processed.
- If we have scenario C, only `ZCL_DECORATOR_1` will be processed.
- If we have scenario D, `ZCL_DECORATOR_2` and `ZCL_DECORATOR_3` will be processed.

As you see, instead of looping through all decorators in all scenarios, we have left the irrelevant ones out. One disadvantage of this approach is that each class must be registered here—or they will be invisible to the system.

C Principles

The general principles of object-oriented programming (OOP) and design patterns are scattered throughout the sections of this book. Without going into too much detail, we will consolidate some of the most significant principles in this appendix for reference.

C.1 Object-Oriented Principles

This section will cover the following object-oriented programming principles: abstraction, composition, inheritance, encapsulation, polymorphism, and decoupling.

C.1.1 Abstraction

The principle of abstraction states that any desired functionality can be represented by an abstract interface—instead of being represented by a concrete class directly. As a result, you can have multiple classes implementing the same interface, and the client application doesn't need to know anything about the internal details of those concrete classes. All subclasses comply with the standards of the interface, and if the client application knows how to deal with the interface, it can easily use any concrete class implementing the interface—even interchangeably!

Abstract classes are less abstract than interfaces but more abstract than concrete classes.

C.1.2 Composition

The principle of composition states that objects can hold other objects just like variables can. Objects can choose to expose or hide those internal objects, depending on the case. A typical example of composition can be found in the composite design pattern (Chapter 11); another one can be found in flyweight (Chapter 15).

C.1.3 Inheritance

The principle of inheritance states that an object can be derived from another object. The child object inherits all of the properties and methods of the parent method and can override them if allowed by the parent.

Generally speaking, inheritance is not the most preferred and advised approach in the world of design patterns. Composition is generally preferred to inheritance—with a few exceptions, such as the template method design pattern (Chapter 26). Even in this case, the parent class is an abstract class (not a concrete class).

In conclusion, you can roughly assume that you should avoid a concrete class → concrete class inheritance if possible. An abstract class → concrete class inheritance is tolerable. The best scenario would be an interface → abstract class (optional) → concrete class type of inheritance.

C.1.4 Encapsulation

The principle of encapsulation states that objects are free to expose or hide their members as much as necessary. You can decide which type, attribute, or method should be marked as public, protected, or private.

Public members are globally accessible by all other classes or client programs. Protected members can only be accessed by subclasses or friend classes. Private members are hidden and can only be accessed by the class itself.

In abstract classes, you can go one step further and mark members as final or abstract. Final members can't be overridden (redefined) by subclasses. Abstract members must be overridden by subclasses. Subclasses may redefine public and protected members.

C.1.5 Polymorphism

The principle of polymorphism states that objects sharing the same interface can be used interchangeably. The casting operation is a typical case of polymorphism. If you have a variable referring to an interface, any object can be casted onto this variable. In the rest of the flow of the program, the interface variable will refer to the casted object.

For a typical example of polymorphism, see Chapter 24 on the state design pattern.

C.1.6 Decoupling

The principle of decoupling states that you can hide the complex structure of a class by marking elements as private. Instead of exposing class components to client applications and expecting everyone to know your variables and the relations between them, you would provide a simple and intuitive public interface instead, which is both safer and easier to use.

A typical example is when you mark the attributes of a class as private. Instead of letting clients modify the variables directly, you provide `GET_*` and `SET_*` methods. Within those methods (especially `SET_*`), you can ensure that the consistency between variables is preserved.

Assume that you have two internal tables: `GT_SUM` and `GT_DETAIL`. If the client application needs to add a new line, it would need to add it to both `GT_SUM` and `GT_DETAIL` and possibly go through some currency/unit conversions. Instead of exposing them, you would mark them as private and provide a public method called `ADD_LINE`. This method would do all the calculations and store the data in a consistent matter.

To expose the data, you can write additional getters such as `GET_SUM` (which returns a read-only copy of `GT_SUM`) and `GET_DETAIL` (which returns a read-only copy of `GT_DETAIL`). An alternative to using getters might be making the variables public but marking them as `READ-ONLY`.

C.2 Design Principles

This section will discuss principles used in design patterns collectively, SOLID: single responsibility, open–closed, Liskov substitution, interface segregation, and dependency inversion.

C.2.1 Single Responsibility

This principle states that a class should have a single responsibility, not more.

Often, the examples in this book highlight that a bloated class taking the lion's share of the overall functionality is the last thing we want. Instead, we want a collection of loosely coupled classes that client applications can flexibly combine as needed.

C.2.2 Open–Closed

This principle states that a class should be open for extension but closed for modification.

When writing your code, you should keep in mind that, if you ever need to extend a class's behavior, you shouldn't be in a position where you must change the class; instead, you should need to merely extend it, which is a good practice in terms of backwards compatibility and reducing test times.

A typical example is the usage of abstract classes—the template method design pattern (Chapter 26) in particular. Any subclass can extend the functionality of an abstract class without changing its interface.

Another example is the visitor design pattern (Chapter 27) where the functionality of the VIP class can be extended, almost without touching a single line of code.

Hardcoding business rules with `IF`/`CASE` statements into a central bloated class is a typical violation of this principle. Instead, having an array of business rule classes sharing the same interface is a better idea. As a result, you'll keep the central class thin, and it can juggle business rules as needed.

C.2.3 Liskov Substitution

This principle states that objects should be replaceable with instances of their subtypes without altering the client application at all.

The structure of most design patterns is based on the idea of similar classes implementing a common interface or being derived from a common (possibly abstract) parent class. Therefore, it is a natural result that classes implementing a common interface/parent class can be used interchangeably.

A typical example of the violation of this principle is adding a semantically obligatory initialization method to a subclass. If your (parent) abstract class doesn't have this method and you are dynamically creating subobjects, this violation will ruin your client application.

C.2.4 Interface Segregation

This principle states that having multiple interfaces for distinct functionalities is better than having one general purpose interface.

Interface segregation means that you should be careful about what you put inside an interface. Just as having one bloated class trying to do everything is a bad idea, so is having one bloated interface trying to cover everything. Instead, multiple interfaces covering distinct functionalities, letting classes flexibly implement what they should, is a more flexible approach.

Imagine an interface called `IF_SHAPE` with two methods: `CALCULATE_AREA` and `GET_SIDE_COUNT`. While this interface works well for squares and rectangles, what about a circle? Are you going to force the circle to implement the `GET_SIDE_COUNT` method only to raise an exception and thus confuse client applications that are programmed to only use `IF_SHAPE`?

The solution would be to have two distinct interfaces: `IF_SHAPE` (with `CALCULATE_AREA`) and `IF_EDGY` (with `GET_SIDE_COUNT`). In this case, `CL_SQUARE` and `CL_RECTANGLE` can implement both, while `CL_CIRCLE` would only implement `IF_SHAPE`.

C.2.5 Dependency Inversion

This principle states that one should depend on abstractions (typically interfaces) not concrete classes.

The implementation of this principle is seen among most, if not all, of the examples in this book. When you look at client programs (in Transaction SE38), you can see that their structure never depends on concrete classes. Instead, these applications rely on abstract interfaces, which is an approach that gives us immense flexibility. The object behind the abstract interface can be changed whenever needed, without touching any client applications.

C.3 Anti-Patterns

So far, we have discussed what should be done, but we should also keep in mind what to avoid. Although many anti-patterns have been defined, we will emphasize the most significant ones for ABAP developers. This section should be entertaining and useful at the same time.

Further Reading

If you like the anti-patterns here and would like to know more, you can check out the Wikipedia page *https://en.wikipedia.org/wiki/Anti-pattern*, which not only lists the

most common anti-patterns but also provides links to detailed examples and discussions. SourceMaking also has a great page on software development anti-patterns at *https://sourcemaking.com/antipatterns/software-development-antipatterns*.

C.3.1 Blob

The blob occurs when a central class takes the lion's share of the responsibilities. As a result, the blob is a huge class with countless attributes and methods that tries to achieve everything under the sun.

Such a class is complicated and inflexible and may be more of an asset or a liability. The blob violates most, if not all, SOLID design principles. Such an architecture should be avoided at all costs—even if it means slicing and dicing an existing class into interfaces and subclasses.

C.3.2 Copy–Paste Programming

Always a frustrating sight, this anti-pattern is the process of copying an existing code, pasting it elsewhere, and making a small modification to make it work in a slightly different way.

This method may look like a good investment: The work gets done faster because existing (and hopefully already-tested) code is copied. However, the real cost comes up in the long term. When you need to modify source code, you would need to modify every single place where it was pasted. Likely, you, or your successor, will probably miss some duplicate code, and part of the system will produce ugly bugs.

The correct approach is to turn the code into a parametric subroutine, possibly as a subroutine or an entirely different class. If the code is too big, you can take advantage of abstraction via an appropriate design pattern such as the template method or strategy.

C.3.3 Functional Decomposition

Functional decomposition is the process of breaking a complex process into smaller, simpler parts. If you think about structural languages, such as Pascal or FORTRAN, a typical program will have a main method that makes calls to other methods in a certain order to fulfill its purpose.

The same applies to classic ABAP as well. If you think about a typical old-fashioned report program, `START-OF-SELECTION` will contain the main flow of the program, which will make calls to forms or functions to fulfill its purpose. This is all well and good until you step into the object-oriented realm—where the functional habits may turn into the functional decomposition anti-pattern. This anti-pattern usually emerges as a result of a functional programmer working in an object-oriented environment. Old habits die hard, and some programmers with a strong background in functional programming manipulate class structures to match their former practices.

Instead of creating a class structure based on design principles, the resulting structure looks like an old-fashioned classic ABAP code in disguise. Some typical structures are:

- The class has a "main" entrance method, which calls other private methods in a certain order to fulfill a certain task, where a possible class hierarchy is completely ignored.
- A handful of classes have singular methods, where each class is considered a function module. A central class with a main method calls other classes as if they are function modules.
- A central toolkit class containing dozens of static methods acts like a function group, ignoring a possible class hierarchy. Various classes access those methods as if they are accessing function modules.

Such practices explicitly violate the principles of object-oriented programming, discussed in Section C.2.

A typical solution to this problem is to let an architect build the overall class hierarchy and use functional programmers to fill the methods within the hierarchy only. Periodically validating the code of notorious programmers may be a good idea as well.

C.3.4 Golden Hammer

The golden hammer is an illusionary anti-pattern. A developer mastering a new technique feels like he/she has a golden hammer, and every problem starts to look like a nail. In other words, the developer implements the technique everywhere, lacking an objective evaluation of the correctness of the decision. As a result, the favored technique is misapplied.

One possible solution to this problem is making sure that the technical leaders/architects are present within the team to play the role of the devil's advocate during the planning stage.

C.3.5 Grand Old Duke of York

This anti-pattern is based upon the idea that not all developers are good architects. While one can be a good coder, being an architect is a distinct skill that requires theoretical knowledge of architectural practices and an architecture instinct. In other words, being a good architect is a mixture of knowledge and talent.

Letting nonarchitectural developers create software architecture often leads to excess complexity and high maintenance costs.

The theoretical solution is simple: Make sure that your project has at least one architect. If it is a big project, you may invite multiple architects. In practice, hiring multiple architects can be challenge, though: Architects are rare, and even if you find some, who has the required knowledge to evaluate their abstract architectural skills in an interview?

Well, this book is one source that might help you become an architect and evaluate architects too!

C.3.6 Input Kludge

This anti-pattern emerges in programs accepting free user input that will be handled by ad-hoc algorithms. Because you have no control of what the end users can type or click, they can break the program with unexpected inputs.

The catch is that input kludges are difficult to catch during unit tests but the end user may find (and maybe exploit) them easily. Imagine watching an end user bypass your complex user exit code to disable the price field of the purchase order screen simply by deleting the material number. Not fun.

C.3.7 Jumble

We all know *n*-tiered systems. The operating system is low level; the programs we use are high level. A Java virtual machine is low level; our Java applications are high level. An SAP kernel is low level; ABAP programs are high level.

In an ideal case, the various components of the system should be independent. Low-level code should not depend on high-level code and, actually, shouldn't really even be aware the high-level code exists. However, when you mix horizontal and vertical design elements, you often end up in the jumble anti-pattern where elements of the system are strongly interdependent. You can't remove, add, or modify components easily, and the entire structure becomes unstable.

The solution? Don't mix horizontal and vertical elements! Abstraction is your friend, and many patterns in this book will help you with that.

C.3.8 Lava Flow

This anti-pattern usually occurs when changes occur in the development team. If the knowledge transfer to successor developers is not complete, the team may end up having to maintain some foreign code that's not totally understood. Instead of taking the risk of doing significant modifications, they accept the former code as it is and invent workarounds.

One work around after the other, and guess what? The system becomes a complex mess.

The solution is to make sure that the team fully understands how things work and provide solid and detailed test cases. Thus, even if the successor breaks something, he/she would see the problem via the inherited tests and probably fix it rather quickly.

C.3.9 Object Orgy

This seemingly R-rated anti-pattern describes a case when classes have all their attributes and methods marked as `PUBLIC`. Any external code can do whatever it pleases with these classes, which will likely cause inconsistencies.

Creating such a class is usually the work of an inexperienced programmer. He/she either is new to object-oriented programming and doesn't know what `PRIVATE`/`PROTECTED` means, or he/she can't foresee what could be needed where and simply marks everything as `PUBLIC`.

This anti-pattern clearly violates the principle of encapsulation discussed in Section C.1.4. A good class should expose only what is necessary and hide the rest to protect consistency and make the intention of the class clear.

C.3.10 Poltergeist

A poltergeist is a class that briefly initiates an action in another concrete class. Just as a poltergeist occasionally moves objects around the house, the poltergeist class does things to another class that is explicitly not its responsibility.

The solution is simple: delete the poltergeist class and insert its functionality into the invoked class.

C.3.11 Reinvent the Wheel

This anti-pattern describes a lack of reusability. If your developers are unaware of each other's work, they might unintentionally find themselves in a position where they write redundant code. The same method might be written multiple times by different developers, or in dramatic cases, similar sets of class hierarchies might have been developed with only minor differences.

Reusability is a significant part of design patterns, and the lack of reusability can lead to many serious problems. Communication among developers, the presence of leading architects, and the habit of running where-used lists before starting development are simple and effective factors to preventing this anti-pattern.

C.3.12 Spaghetti Code

Many of you have probably heard this term before. Spaghetti code is a code snippet that looks like, well, spaghetti. If you are looking at 4,000 lines of code with endless `LOOP`s, `SELECT`s and whatnot, congratulations! You have encountered spaghetti code.

Spaghetti code is like a long tunnel with no exits. It is inflexible, can't be partially reused, and is hard to test. Just when you think that things can't get worse than that, they usually do: You find out that you are not dealing with just some spaghetti code. You might find out that you are dealing with an entire bowl of spaghetti!

Instead of producing long pieces of spaghetti, one should aim at producing small pieces of penne. If you have multiple, loosely coupled pieces of reusable code, the flexibility and testability of your application will increase dramatically.

C.3.13 Swiss Army Knife

Let us start by stating that we don't have anything against Switzerland! Although a Swiss Army knife is a useful tool outdoors, using one is a terrible practice indoors—and if you are programming. This anti-pattern implies a development practice where the code attempts to provide an interface to cover all possible uses for today and the future. This practice leads to an excessively complex class interface. Some methods of many subclasses will probably be empty, and a client application based on that interface will possibly fail due to the complexity.

With this anti-pattern, performance is also at risk because many unnecessary operations may be executed even in the simplest method call.

This anti-pattern clearly violates the SOLID principle of interface segregation. Refer back to Section C.2.4 for suggested solutions.

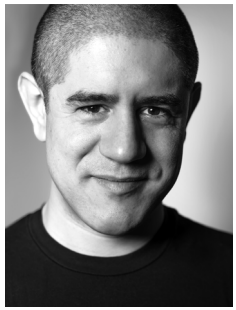
C.3.14 Vendor Lock-In

This anti-pattern points to an architecture that is completely dependent on a single product from a single vendor. A typical non-SAP example would be if you developed an entire application assuming the continued existence of a certain database brand. The same idea can be extended to cloud services, document management systems, external web service protocols, etc.

The problem with this approach is that vendor lock-in makes the design completely dependent on a certain vendor. If the vendor goes out of business, or if you have a license disagreement and can't continue using the product, the development cost of moving to an alternative product would be too high. We all have seen vendors pumping out their support prices because of their awareness of such a lock-in.

The basic solution of this anti-pattern is to build an architecture based on the idea within data access object (DAO, Chapter 12) or strategy (Chapter 25) design pattern. As a result, switching between alternative products will be much easier.

D The Author



Dr. Kerem Koseoglu is a freelance SAP software architect, working professionally since 2000. He has specialized in the development of comprehensive applications using design patterns and conducts technical training.

He has participated many projects in various countries, taking diverse roles such as lead architect, team lead, developer, technical advisor, instructor, and project manager. His former publications include four published books and numerous articles for technical magazines.

He has a PhD in organizational behavior, is bilingual (English/Turkish), and speaks German fluently. When he is not coding or writing, you are likely to find him on stage playing his bass guitar or stretching on his yoga mat.

For more information and contact, you can visit his website at *http://kerem.koseoglu.info* or send an email to *kerem@koseoglu.info*.

Index

A

ABAP toolkit, 340
ABAP Workbench, 340
Abstract class, 17, 54, 139, 210, 217, 224, 229, 241, 258, 263, 297, 302, 305, 308, 319, 345, 360–361, 363–364, 371–372, 374
Abstract factory, 19, 21–22, 39, 75, 106
Adapter, 19, 21–22, 109, 170, 207, 293, 362
  glue code, 118
  two-way adapter, 118
Anti-pattern, 375
  blob, 34, 113, 169, 209, 241, 265, 269, 271, 281, 283, 323, 333–334, 376
  copy-paste programming, 23, 29, 32, 49, 52, 75, 162, 167, 222, 265, 271, 275, 278, 300–301, 376
  functional decomposition, 30, 221, 257, 376
  golden hammer, 18, 377
  grand old duke of York, 113, 221, 378
  input kludge, 71, 202, 206, 296, 378
  jumble, 118, 196, 378
  lava flow, 122–123, 155, 379
  object orgy, 379
  poltergeist, 380
  reinvent the wheel, 17, 75, 302, 380
  spaghetti code, 33, 114, 155, 186, 209, 380
  Swiss army knife, 209, 221, 265, 269, 324, 333–334, 381
  vendor lock-in, 148, 381

B

Bridge, 19, 21–22, 123, 197, 284
Builder, 19–20, 22–23, 39, 47, 49, 75, 81, 106, 197

C

Cache, 103
Caretaker class, 249
Casting, 349, 360
Central class, 255
Central management class, 237
Chain of responsibility, 19, 21–22, 197, 209, 362–363
Class, 341, 360
  ABSTRACT, 319, 342, 350, 360, 372
  ABSTRACT METHODS, 355
  CREATE PRIVATE, 342
  CREATE PUBLIC, 342
  FINAL, 319, 342, 350, 360, 372
  FINAL METHODS, 355
  hierarchy, 124
  INHERITING FROM, 345
  INSTANCE, 360
  METHOD, 356
  PRIVATE, 319, 360, 372–373, 379
  PRIVATE SECTION, 342
  PROTECTED, 319, 360, 372, 379
  PROTECTED SECTION, 342
  PUBLIC, 360, 372, 379
  PUBLIC SECTION, 342
  READ-ONLY, 373
  REDEFINITION, 345
  STATIC, 357, 360
Clone, 91
Command, 19, 21, 23, 221, 243
Composite, 19, 21–22, 133, 217, 371
Composite object, 133
CSV file, 49

D

Data access object, 19, 21–22, 147, 362, 381
  SCRUD (select/create/read/update/delete), 152
Data references, 193
Decorator, 19, 21–23, 155, 185, 196, 363, 368
Default handler, 217
Design principles, 373
  dependency inversion, 39, 54, 151, 196, 231, 258, 290, 320, 375
  interface segregation, 54, 283, 374

Design principles (Cont.)
Liskov substitution, 23, 54, *69*, 115–116, 128, 144, 151, 203, 224, 260, 275, 285, 290–291, 374
open-closed, 23, *75*, 81, 161, 222, 241, 258, 261, 269, 278, 283, 286, 291, 323, 326, 334, 374
single responsibility, 29, 34, 118, 126, 148, 152, 158, 162, 224, 236, 240–241, 261, 269, 287, 326, 333, 373
Development tools, 339
ABAP toolkit, 340
ABAP Workbench, 340
Eclipse, 340
form-based view, 340
SAP Business Workflow, 185, 209, 323
SAPUI5, 167
source code-based view, 340
Web Dynpro ABAP, 167
Document creation, 221
Dynamic method call, 202

E

Eclipse, 340
Enhancement point, 186
Exception, 362

F

Façade, 19, 21–23, 165, 169, 222
Factory, 19–20, 22–23, 39, 44, 47, 66, *69*, 75, 81, 86, 105–106, 145, 176, 182, 238
Flux, 34
Flyweight, 19–20, 22, *96*, 106, 145, 171, 284, 302, 371
extrinsic state, 180
interdependency, 181
intrinsic state, 180
Form-based view, 340
Function module, 29, 99–100, 341

G

Global flag, 277
GUI, 33, 167, 232, 252, 256

H

Handler class, 217

I

Instance container, 194
Instance method, 258
Interface, 17, 40, 53, 109, 124, 126, 149, 157, 173, 203, 224, 232, 257, 262–263, 271, 275, 278, 285, 287, 291, 293, 320, 327, 334, 351, 357, 360–361, 371–372, 374–376, 381
Invoker, 227

L

Lazy initialization, 19–20, 22–23, 77, 106, 207
Leaf object, 133
Legacy class, 122

M

Mediator, 19, 21–22, 162, 197, 231, 264
use cases, 240
Memento, 19, 21–22, 229, 243, 246
Multiple inheritance, 118
Multiton, 19–20, 22–23, 81, 83, *96*, 106, 171, 176, 181–182
Mutually-dependent objects, 236
MVC, 19–21, 23, 27, 72, *99*, 167, 200, 244, 290, 324, 334

O

Object oriented principles, 371
abstraction, 70, 371
composition, 17, 34, *96*, 122, 124, 128, 169, 178, 199, 245, 248, 300, 371
decoupling, 66, 102, 137, 144, 165, 167, 180, 373
encapsulation, 17, 30, 74, 88, 102, 105, 284, 343, 372

Object oriented principles (Cont.)
inheritance, 17, 207, 258, 300–301, 311, 319, 343–344, 350, 357, 361, 372
inheritence, 30, 96, 124, 144, 284
polymorphism, 17, 30, 46, 69, 116, 140, 275, 284, 343, 347, 372
Object reference, 233
Observer, 19, 21–23, 34, 197, 241, 255, 258, 363
performance, 261
Originator class, 250

P

Parametric subroutine, 376
Priority system, 209
Property container, 19, 21–22, 162, 185, 189, 229, 263
Protection proxies, 206
Prototype, 19–20, 22, 75, 81, 91, 229
Proxy, 19, 21–22, 170, 199

R

Receiver class, 222
Recursive method, 133
Redo operation, 253
Remote proxies, 207
RFC function, 29, 167

S

SAP Business Workflow, 185, 209, 323
SAP Process Integration, 231
SAPUI5, 167
Servant, 19, 21, 23, 265, 270, 302, 333–334
extensions, 274
Singleton, 19–20, 22–23, 47, 81, 83, 86, 99, 170–171, 182, 217, 238, 284
Source class, 262
pull model, 262
push model, 262
Source code-based view, 340
State, 19, 21–22, 207, 277, 372
State index, 254
Static container, 194
Static data type, 55
Static method, 258
Strategy, 19, 21–23, 59, 66, 70, 148, 153, 197, 218, 284–285, 320, 362, 376, 381
Subject → Central class
Superclass, 343
Surrogate → Proxy

T

Table control, 244
Template method, 19, 21–23, 162, 229, 241, 263, 275, 284, 297, 305, 372, 374, 376
Toolkit class, 269
Tree structure, 134

U

UML, 18, 23, 30, 44, 70, 79, 93, 110, 168, 186, 191, 200, 202, 257, 271, 279, 287, 309, 355, 360
Undo operation, 229, 243
User exit, 156

V

Virtual proxies, 207
Visitor, 19, 21, 23, 275, 323, 374

W

Web Dynpro ABAP, 32, 167
Workflow rules, 210

Service Pages

The following sections contain notes on how you can contact us.

Praise and Criticism

We hope that you enjoyed reading this book. If it met your expectations, please do recommend it. If you think there is room for improvement, please get in touch with the editor of the book: meaganw@rheinwerk-publishing.com. We welcome every suggestion for improvement but, of course, also any praise!

You can also share your reading experience via Twitter, Facebook, or email.

Supplements

Supplements (sample code, exercise materials, lists, and so on) are provided in your online library and on the web catalog page for this book. You can directly navigate to this page using the following link: http://www.sap-press.com/4277. Should we learn about typos that alter the meaning or content errors, we will provide a list with corrections there, too.

Technical Issues

If you experience technical issues with your e-book or e-book account at SAP PRESS, please feel free to contact our reader service: support@rheinwerk-publishing.com.

About Us and Our Program

The website http://www.sap-press.com provides detailed and first-hand information on our current publishing program. Here, you can also easily order all of our books and e-books. Information on Rheinwerk Publishing Inc. and additional contact options can also be found at http://www.sap-press.com.

Legal Notes

This section contains the detailed and legally binding usage conditions for this e-book.

Copyright Note

This publication is protected by copyright in its entirety. All usage and exploitation rights are reserved by the author and Rheinwerk Publishing; in particular the right of reproduction and the right of distribution, be it in printed or electronic form.

© 2017 by Rheinwerk Publishing, Inc., Boston (MA)

Your Rights as a User

You are entitled to use this e-book for personal purposes only. In particular, you may print the e-book for personal use or copy it as long as you store this copy on a device that is solely and personally used by yourself. You are not entitled to any other usage or exploitation.

In particular, it is not permitted to forward electronic or printed copies to third parties. Furthermore, it is not permitted to distribute the e-book on the Internet, in intranets, or in any other way or make it available to third parties. Any public exhibition, other publication, or any reproduction of the e-book beyond personal use are expressly prohibited. The aforementioned does not only apply to the e-book in its entirety but also to parts thereof (e.g., charts, pictures, tables, sections of text).

Copyright notes, brands, and other legal reservations as well as the digital watermark may not be removed from the e-book.

Digital Watermark

This e-book copy contains a **digital watermark**, a signature that indicates which person may use this copy. If you, dear reader, are not this person, you are violating the copyright. So please refrain from using this e-book and inform us about this violation. A brief email to info@rheinwerk-publishing.com is sufficient. Thank you!

Trademarks

The common names, trade names, descriptions of goods, and so on used in this publication may be trademarks without special identification and subject to legal regulations as such.

All of the screenshots and graphics reproduced in this book are subject to copyright © SAP SE, Dietmar-Hopp-Allee 16, 69190 Walldorf, Germany. SAP, the SAP logo, ABAP, Ariba, ASAP, Duet, hybris, SAP Adaptive Server Enterprise, SAP Advantage Database Server, SAP Afaria, SAP ArchiveLink, SAP Business ByDesign, SAP Business Explorer (SAP BEx), SAP BusinessObjects, SAP BusinessObjects Web Intelligence, SAP Business One, SAP BusinessObjects Explorer, SAP Business Workflow, SAP Crystal Reports, SAP d-code, SAP EarlyWatch, SAP Fiori, SAP Ganges, SAP Global Trade Services (SAP GTS), SAP GoingLive, SAP HANA, SAP Jam, SAP Lumira, SAP MaxAttention, SAP MaxDB, SAP NetWeaver, SAP PartnerEdge, SAPPHIRE NOW, SAP PowerBuilder, SAP PowerDesigner, SAP R/2, SAP R/3, SAP Replication Server, SAP SI, SAP SQL Anywhere, SAP Strategic Enterprise Management (SAP SEM), SAP StreamWork, SuccessFactors, Sybase, TwoGo by SAP, and The Best-Run Businesses Run SAP are registered or unregistered trademarks of SAP SE, Walldorf, Germany.

Limitation of Liability

Regardless of the care that has been taken in creating texts, figures, and programs, neither the publisher nor the author, editor, or translator assume any legal responsibility or any liability for possible errors and their consequences.