Reading Sample

This sample walks you through how to use the book before diving into the use of two example design patterns, model-view-controller (MVC) and abstract factory, with practical examples for each.

“Preface”
“MVC”
“Abstract Factory”

Contents
Index
The Author

Kerem Koseoglu
Design Patterns in ABAP Objects
387 Pages, 2016, $79.95
ISBN 978-1-4932-1464-8

www.sap-press.com/4277
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.
Preface

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.

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

**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.

- 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

- 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

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 lifesaver 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 expe-
rience 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 Appen-
dix 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.
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**.
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.

```abap
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.
```

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 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.

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).

```plaintext
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.

```
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.

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. 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.

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](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
Chapter 2

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.

Listing 2.1 demonstrates what the writer interface might look like.

```abap
INTERFACE zif_writer
  PUBLIC.
  METHODS write_file
     IMPORTING
     !iv_path TYPE clike.
ENDINTERFACE.
```

Listing 2.2 provides a simple portrait of the executer interface.

```abap
INTERFACE zif_executer
  PUBLIC.
  METHODS execute_app.
ENDINTERFACE.
```

Listing 2.1 Writer Interface

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.
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 List-
ing 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.

![Diagram](image.png)

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.

```java
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.

```java
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( ).
lo_win_executer->enter_admin_mode( ).
ro_exe ?= lo_win_executer.
ENDMETHOD.
```

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.

```java
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( ).
```

Listing 2.9 Unix Factory Class
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.
## 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
# Contents

## Part III Structural Design Patterns

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Lazy Initialization</td>
</tr>
<tr>
<td>6</td>
<td>Multiton</td>
</tr>
<tr>
<td>7</td>
<td>Prototype</td>
</tr>
<tr>
<td>8</td>
<td>Singleton</td>
</tr>
<tr>
<td>9</td>
<td>Adapter</td>
</tr>
<tr>
<td>10</td>
<td>Bridge</td>
</tr>
<tr>
<td>11</td>
<td>Composite</td>
</tr>
<tr>
<td>12</td>
<td>Data Access Object</td>
</tr>
<tr>
<td>13</td>
<td>Decorator</td>
</tr>
<tr>
<td>14</td>
<td>Façade</td>
</tr>
<tr>
<td>15</td>
<td>Flyweight</td>
</tr>
</tbody>
</table>

## Part III Structural Design Patterns

<table>
<thead>
<tr>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lazy Initialization</td>
<td>77</td>
</tr>
<tr>
<td>5.1 Case Study: Logging Errors</td>
<td>77</td>
</tr>
<tr>
<td>5.2 Advantages</td>
<td>81</td>
</tr>
<tr>
<td>5.3 Related Patterns</td>
<td>81</td>
</tr>
<tr>
<td>5.4 Summary</td>
<td>82</td>
</tr>
<tr>
<td>Multiton</td>
<td>83</td>
</tr>
<tr>
<td>6.1 Case Study: Vendor Balance</td>
<td>84</td>
</tr>
<tr>
<td>6.2 When to Use</td>
<td>88</td>
</tr>
<tr>
<td>6.3 When to Avoid</td>
<td>88</td>
</tr>
<tr>
<td>6.4 State Modification</td>
<td>89</td>
</tr>
<tr>
<td>6.5 Summary</td>
<td>89</td>
</tr>
<tr>
<td>Prototype</td>
<td>91</td>
</tr>
<tr>
<td>7.1 Case Study: Item Clone</td>
<td>91</td>
</tr>
<tr>
<td>7.2 Changing Class Properties</td>
<td>95</td>
</tr>
<tr>
<td>7.3 Clone Operations</td>
<td>96</td>
</tr>
<tr>
<td>7.4 Related Patterns</td>
<td>96</td>
</tr>
<tr>
<td>7.5 Summary</td>
<td>97</td>
</tr>
<tr>
<td>Singleton</td>
<td>99</td>
</tr>
<tr>
<td>8.1 Case Study: Subcomponents</td>
<td>99</td>
</tr>
<tr>
<td>8.2 Advantages and Disadvantages</td>
<td>106</td>
</tr>
<tr>
<td>8.3 Related Patterns</td>
<td>106</td>
</tr>
<tr>
<td>8.4 Summary</td>
<td>106</td>
</tr>
<tr>
<td>Adapter</td>
<td>109</td>
</tr>
<tr>
<td>9.1 Case Study: Project Management Tools</td>
<td>109</td>
</tr>
<tr>
<td>9.2 Glue Code</td>
<td>118</td>
</tr>
<tr>
<td>9.3 Two-Way Adapters</td>
<td>118</td>
</tr>
<tr>
<td>9.4 Legacy Classes</td>
<td>122</td>
</tr>
<tr>
<td>9.5 Summary</td>
<td>122</td>
</tr>
<tr>
<td>Bridge</td>
<td>123</td>
</tr>
<tr>
<td>10.1 Case Study: Messaging Framework</td>
<td>123</td>
</tr>
<tr>
<td>10.2 Advantages</td>
<td>131</td>
</tr>
<tr>
<td>10.3 Summary</td>
<td>131</td>
</tr>
<tr>
<td>Composite</td>
<td>133</td>
</tr>
<tr>
<td>11.1 Recursive Programming: A Refresher</td>
<td>133</td>
</tr>
<tr>
<td>11.2 Case Study: HR Positions</td>
<td>137</td>
</tr>
<tr>
<td>11.3 Advantages</td>
<td>144</td>
</tr>
<tr>
<td>11.4 Disadvantages</td>
<td>144</td>
</tr>
<tr>
<td>11.5 When to Use</td>
<td>145</td>
</tr>
<tr>
<td>11.6 Related Patterns</td>
<td>145</td>
</tr>
<tr>
<td>11.7 Summary</td>
<td>146</td>
</tr>
<tr>
<td>Data Access Object</td>
<td>147</td>
</tr>
<tr>
<td>12.1 Case Study: Potential Customers</td>
<td>147</td>
</tr>
<tr>
<td>12.2 Redundant Code Prevention</td>
<td>152</td>
</tr>
<tr>
<td>12.3 Related Patterns</td>
<td>153</td>
</tr>
<tr>
<td>12.4 Summary</td>
<td>153</td>
</tr>
<tr>
<td>Decorator</td>
<td>155</td>
</tr>
<tr>
<td>13.1 Case Study: User Exit</td>
<td>155</td>
</tr>
<tr>
<td>13.2 Advantages and Challenges</td>
<td>161</td>
</tr>
<tr>
<td>13.3 Related Patterns</td>
<td>162</td>
</tr>
<tr>
<td>13.4 Summary</td>
<td>162</td>
</tr>
<tr>
<td>Façade</td>
<td>165</td>
</tr>
<tr>
<td>14.1 Case Study: Bonus Calculation</td>
<td>166</td>
</tr>
<tr>
<td>14.2 When and Where to Use</td>
<td>168</td>
</tr>
<tr>
<td>14.3 Related Patterns</td>
<td>170</td>
</tr>
<tr>
<td>14.4 Summary</td>
<td>170</td>
</tr>
<tr>
<td>Flyweight</td>
<td>171</td>
</tr>
<tr>
<td>15.1 Case Study: Negative Stock Forecast</td>
<td>171</td>
</tr>
<tr>
<td>15.2 Disadvantages</td>
<td>181</td>
</tr>
</tbody>
</table>
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
Index

A
ABAP toolkit, 340
ABAP Workbench, 340
Abstract class, 17, 54, 139, 210, 217, 224,
229, 241, 258, 263, 297, 302, 305, 308,
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
invar 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, 360–373, 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.)

open-closed, 23, 75, 81, 161, 222, 241, 258, 261, 269, 278, 283, 286, 291, 323, 326, 334, 374
single responsibility, 29, 34, 118, 148, 152, 158, 162, 222, 241–242, 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
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
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
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

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
Source class, 262
push model, 262
pull model, 262
Source code-based view, 340
State, 19, 21–22, 207, 277, 372
State index, 254
Static container, 194
Static data type, 345
Static method, 258
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

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
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.