Object-Oriented Design with ABAP

James E. McDonough

Object-Oriented Design with ABAP

A Practical Approach

James E. McDonough

Object-Oriented Design with ABAP: A Practical Approach

James E. McDonough Pennington, New Jersey, USA

ISBN-13 (pbk): 978-1-4842-2837-1 ISBN-13 (electronic): 978-1-4842-2838-8 DOI 10.1007/978-1-4842-2838-8

Library of Congress Control Number: 2017943501

Managing Director: Welmoed Spahr Editorial Director: Todd Green
Acquisitions Editor: Susan McDermott Development Editor: Laura
Berendson Technical Reviewer: Paul Hardy Coordinating Editor: Rita
Fernando Copy Editor: Mary Behr Compositor: SPi Global Indexer:
SPi Global

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit http://www.apress.com/ rights-permissions.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484228371. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

To my wife, Lorraine, who, at a time in my life when I needed a career change, suggested I consider the field of computer programming.

Contents at a Glance

Contents

About the Author

James E. McDonough received a degree in music education from Trenton State College. After teaching music for only two years in the New Jersey public school system, he changed careers. He's spent the past 35 years as a computer programmer while also maintaining an active presence as a freelance jazz bassist between New York and Philadelphia. Having switched from mainframe programming to ABAP 20 years ago, he now works as a contract ABAP programmer designing and writing ABAP programs on a daily basis. An advocate of using the object-oriented programming features available with ABAP, he has been teaching private ABAP education courses over the past few years, where his background in education

enables him to present and explain complicated concepts in a way that makes sense to beginners.

About the Technical Reviewer

Paul Hardy joined Heidelberg Cement in the UK in 1990. For the first seven years, he worked as an accountant. In 1997, a global SAP rollout came along; he jumped on board and has never looked back. He has worked on country-specific SAP implementations in the United Kingdom, Germany, Israel, and Australia.

After starting off as a business analyst configuring the good old IMG, Paul swiftly moved on to the wonderful world of ABAP programming. After the initial run of data conversion programs, ALV reports, interactive DYNPRO screens, and (urrggh) SAPscript forms, he yearned for something more and since then has been eagerly investigating each new technology as it comes out, which culminated in him writing the SAP Press book *ABAP To The Future*. Paul became an SAP mentor in March 2017 and can regularly be found blogging away on the SCN site. He wrote a series of blogs comparing the merits of object-orientated vs. procedural programming.

He often presents at SAP conferences in Australia (Mastering SAP Technology and the SAP Australian User Group annual conference) and at SAP TECHED Las Vegas. If you ever happen to be at one of these conferences, Paul invites you to have a drink with him at the networking event in the evening and to ask him the most difficult questions you can think of, preferably about SAP.

Acknowledgments

I could not have done this project without the help of others.

I extend my gratitude to Brian Brennan, who introduced me to the book *Head First Design Patterns*, which started me on the road to discovery of the power of object-oriented programming and eventually led to my decision to create the Object-Oriented Chalk Talks curriculum, lecture series, and accompanying exercises.

Thanks go to those of my colleagues on a project in southern New Jersey who dedicated their time and effort to attend the Object-Oriented Chalk Talks and perform the accompanying exercises, providing me with a laboratory setting in which I could receive the kind of feedback enabling me to refine the content in this book.

I am very grateful to Paul Hardy for agreeing to undertake the task of reviewing the content of the book and for doing such a magnificent job at it, offering many suggestions for improvement.

Susan McDermott, Rita Fernando, and Laura Berendson, my editors at Apress Media, LLC, were of enormous help in guiding me through the publication process and resolving the technical glitches we encountered along the way.

Finally, it would have been much more difficult to complete this project without the love and understanding I received from my family for tolerating my absences during those long hours on weekends and holidays while I was secluded in deep thought about how to organize and present this content.

Introduction

Although many books have been published on the subject of object-oriented design, this book caters specifically to programmers familiar primarily with ABAP, a language with an origin in procedural design, but after years of evolution, now accommodates both the procedural and object-oriented design paradigms. A significant aspect of this book is its accompaniment by a set of exercise programs, each one of which reinforces an object-oriented design concept presented in the book.

For Whom This Book Is Applicable

This book is applicable to virtually all experienced ABAP programmers in one way or another. It is subdivided into three parts, each part applicable to a specific level of familiarity and comfort by the reader with the content it presents.

The first part of the book is intended for ABAP programmers already skilled with the procedural aspects of writing code but who either know nothing about object-oriented programming or simply want to become more comfortable with the object-oriented paradigm. The basic principles of object-oriented programming and design are covered here.

The second and third parts of the book are intended for ABAP programmers already familiar with the basic principles of object-oriented programming but not yet familiar or comfortable with design patterns. The second part of the book presents an introduction to Unified Modeling Language and the third part of the book introduces many of the various design patterns typically associated with object-oriented design.

Those ABAP programmers already familiar with both the object-oriented basic principles as well as Unified Modeling Language and design patterns may find this book covers some design patterns that are altogether new to them or described and illustrated in a context applicable to ABAP programming.

How This Book Should Be Used

ABAP programmers unfamiliar with object-oriented programming should read the book sequentially from the beginning. Indeed, the book is organized primarily for the benefit of such programmers, with each subsequent chapter referring to concepts covered in previous chapters.

ABAP programmers already familiar with object-oriented concepts may skip the first part of the book and start with the second part, to become more familiar with the Unified Modeling Language and design patterns.

ABAP programmers already familiar with design patterns may find it most helpful simply to use this book as a reference.

Regardless of the level of comfort with object-oriented design, this book is modeled on the "learn by doing" premise. Accordingly, Appendix B contains information about retrieving the functional and technical requirements documentation for the accompanying comprehensive set of executable ABAP exercise programs, with each exercise program illustrating some new concept introduced in the book, from the most basic principles of object-oriented programming to the most advanced design patterns. This provides for a multitude of options for using the book and *doing* the corresponding exercise programs, among them:

- Writing each new exercise program based solely on the information provided by the requirements documentation and the diagrams accompanying the collection of executable example ABAP exercise program
- Writing each new exercise program after looking at how the new concepts were implemented in the corresponding executable example ABAP exercise program
- Dispensing entirely with writing any code and simply relying on the corresponding executable example ABAP exercise programs to illustrate the implementation

Refer to Appendix B for instructions for retrieving the accompanying collection of executable example ABAP exercise programs and their corresponding diagrams.

Why This Book Was Written

In March, 2013 I began presenting a series of weekly one-hour lunch-and-learn lectures on object- oriented programming concepts. These "Object-Oriented Chalk Talks," as I called them, were attended by my colleagues, all ABAP programmers and all highly skilled in the procedural style of coding, but, having learned their programming skills when ABAP was still a procedural language, they mostly were uncomfortable with the new object-oriented aspects recently introduced to the ABAP language. I pitched the class as one where I would cover general object-oriented concepts in a lecture format, mostly using nothing more than a white board, but that the class was specifically not about ABAP objects.[1] My goal was to introduce to my colleagues the *reasons why* there is an object-oriented paradigm for writing programs, and not merely to show *how* these concepts could be coded in ABAP, the syntax of which all of them already knew from having attended extensive training on that subject a few months earlier. I stated up front that students probably would be making a commitment of about 26 weeks (half a year) before we would complete all the material to be covered. Although the lectures were to focus on concepts and not a specific language, all the exercises accompanying the lectures were written in ABAP. The idea behind this arrangement was that students would attend the lecture, learn the concepts, and then go perform the associated exercises on their own before the next lecture. This first class began with about 16 students, fewer than the 20 students I considered to be the maximum number of students per

class considering the facilities at our disposal. Many others had heard about the Object-Oriented Chalk Talks and expressed interest in attending. I soon announced another section would begin in July of 2013, to which the response was so overwhelming that I needed to schedule two other concurrent sections of the Object-Oriented Chalk Talks to accommodate the unexpectedly high number of students wanting to attend. With still others expressing interest in the class, I started a fourth section in September of 2013, and other sections soon followed.

[1] *ABAP Objects* is the main title of two books by Horst Keller and Sacha Krűger (*ABAP Objects: An Introduction to Programming SAP Applications*, Addison Wesley, 2003) and its successor, *ABAP Objects: ABAP Programming in SAP NetWeaver,* 2nd edition, Galileo Press, 2007). Many developers now refer to *ABAP Objects* to mean the object-oriented aspects of the ABAP language.

From the feedback I received, the classes proved to be wildly popular and interesting to my colleagues. Considering that so many of my fellow ABAP programmers were finding so much value in the Object- Oriented Chalk Talks, I reasoned there probably are thousands of other ABAP programmers who find themselves in the same situation: being very capable programmers with the procedural aspects of ABAP but having difficulty making the leap to the object-oriented paradigm. I wrote this book to share the material covered in the Object-Oriented Chalk Talks with other programmers beyond my reach in a classroom format.

Credentials of the Author

My formal training in the data processing industry consists of one year at a community college learning mainframe languages (IBM assembler, COBOL, and PL/I) and, nearly 15 years later, a six-week seminar on ABAP programming. Compared with some of my colleagues over the years, I have very little formal training in computer programming. Indeed, I have absolutely no formal training in the concepts of object-oriented programming; everything I know on that subject I learned on my own. So, what makes me think I am qualified to teach anyone else about these concepts?

Prior to getting into the data processing industry over 30 years ago, I earned a college degree in Music Education and taught instrumental music for two years in two different public school districts in the state of New Jersey. During my college years I made an effort to learn and gain some modicum of proficiency with all of the band and orchestra instruments. My perception then was that I could be a better music educator by understanding more about the struggles students endure when they endeavor to learn to play a musical instrument. How, I thought, could I presume to teach a 7th grader how to play the trombone if I were not able to play it myself?

This philosophy on education served me well those two years I taught in the public schools, and I have continued with this approach ever since. Accordingly, although my credentials in data processing may not be as impressive as those of some of my colleagues, my background as an educator enables me to perceive the problems students are likely to encounter when learning any new skill. So I have learned all I could, on my own, about object-oriented programming, and over the past few years have been able to use this programming style with most of my ABAP development efforts. I believe that now, having gained a certain level of proficiency in this subject, I am ready to impart what I know to others who also wish to become familiar with this fascinating field of object-oriented programming.

Internationalization Considerations

I have made an attempt to consider the various backgrounds of potential readers, and consequently to avoid phrases and references that could be expected to be understood only by programmers who are familiar with daily life in the United States. However, having been exposed only to US culture all my life, the tone of the book exhibits a corresponding slant. This is particularly evident when describing examples where weights and measures are involved, since often the reader will be subjected to the U.S. Customary System of measurement, a system used by virtually no other nations of the world, instead of the much more logical International System of Units (a.k.a. the metric system) used virtually everywhere else. I trust this will not present too formidable a challenge to readers primarily steeped in other cultures.

PART I

Understanding the Concepts of Object-Oriented Design

CHAPTER 1

Preparing to Take the First Step

Object-oriented design offers many new concepts for us to explore, so we will want to be certain we've taken the necessary precautions to insure a successful expedition into this new realm. Accordingly, let's take a moment to prepare ourselves for the adventure we are about to undertake, to pause and give consideration to both the journey itself and the expectations we have about what we will encounter along the way.

Road Map to Object-Oriented Design

A road map is a useful metaphor to illustrate the path we will take from our familiar surroundings of procedural programming to the unfamiliar new territory known as object-oriented design. The road map shows the way. We know we will need to travel the road between these two locations, eventually reaching our destination, but that each step along the way is dependent upon having taken the previous steps. That is, we move continuously in one general direction from our point of origin to our destination, covering each mile as we encounter it, not beginning to cover the tenth mile until after we already have passed through the ninth mile to get there. Accordingly, we become familiar with those parts of the road closest to our point of origin before those parts farther along. As with most such journeys, we may find that the terrain associated with the first few steps is very similar to our starting location, but the terrain changes as we continue moving. This similarity of terrain between

adjacent steps enables us to adapt gradually to the changes awaiting us along the road.

So, before we start on our journey to object-oriented design, let's give some consideration toward preparing for a successful trip.

1. Where are we now?
2. Where are we going?
3. Why are we going there?
4. How are we going to get there?

Where We Are Now

If you are like many other programmers using SAP, you gained your experience with ABAP before SAP introduced object-oriented features into the ABAP language, or if these features had been introduced, your organization was not using a release where they were available to you. Some of you are former (perhaps even current) COBOL programmers. Regardless of how you found your way to ABAP, chances are you have been using only the "procedural" style of programming and have been very successful at it for years. SAP now refers to this as the "classical" ABAP programming style.

3 © James E. McDonough 2017 J. E. McDonough, *Object-Oriented Design with ABAP*, DOI 10.1007/978-1-4842-2838-8_1

With classical ABAP programming, we may have learned many bad habits, ones that are no longer tolerated with the object-oriented language features. However, because SAP has maintained backward compatibility with the ABAP compiler, there has never

been a requirement for ABAP programmers to begin using the object-oriented paradigm. As a consequence, we have been happily sticking with the classical ABAP programming style for years now, shunning the newfangled object-oriented features because 1) it would take some time and effort to learn them and 2) once we began using these features, the compiler would flag our use of those bad habits we had honed so carefully over the years.

SAP now provides a fork in the road regarding ABAP programming style. There are two different sets of ABAP language features available, one facilitating the classic procedural style we have been using for years, and another facilitating the object-oriented paradigm. Although they are mostly compatible, there are some restrictions. We could continue using classical ABAP and perhaps be successful with it for quite a while into the future. Conversely, we could bite the bullet and learn what we can about the object-oriented features and begin to use those.

One thing to consider when pondering this choice is that each new release of SAP contains more and more vendor ABAP code that makes use of the object-oriented features. Prior to using ABAP, we might have worked with an Enterprise Resource Planning (ERP) package where seldom, if ever, would we see vendor source code, regardless of the original language. But as ABAP programmers, it is commonplace to browse the source code of ABAP components supplied by SAP with the standard ERP release. The more this standard SAP code is written using the object-oriented style, the more reason we have to learn and understand it. However, becoming familiar with the syntax associated with the implementation of object-oriented features into the ABAP language is only one of the hurdles we must overcome. An even larger hurdle is understanding the fundamental concepts of object-oriented programming.

So here is where we find ourselves: capable classical ABAP developers, knowing very little about the new object-oriented ABAP statements and knowing even less about how to use those statements effectively. It should come as no surprise that many ABAP programmers contemplating this challenge will choose to continue on with their classical style and avoid the object-oriented features so long as the ABAP compiler enables them to do so. However, other ABAP programmers, who appreciate the significance of these new changes, who have become enlightened to the benefits of object-oriented programming, and who want to leverage these new capabilities, will undertake to embrace this new paradigm and use it to their full advantage.

Where We Are Going

Object-oriented programming languages have been in existence since the 1960s when Ole-Johan Dahl and Kristen Nygaard of the Norwegian Computing Center introduced the language named Simula I and its successor, Simula 67.[1] Since then, object-oriented concepts have provided the foundation for such languages as Smalltalk, C++, and Java. Other languages initially based on the procedural model have been extended to provide some object-oriented capabilities. ABAP falls into this latter category.

In our quest to reach this district known as object-oriented design, we are headed for a place that was founded over half a century ago and has since grown into a thriving metropolis within the data processing landscape, so it is hardly new. However, it is new to us. This is a place where we can use these object- oriented programming techniques in our ABAP programming efforts as freely and comfortably as the procedural style of coding ABAP has provided since its inception.

[1] http://campus.hesge.ch/daehne/2004-2005/langages/simula.htm

Why We Are Going There

Statistics show that the initial development effort of writing a computer program consumes only a small fraction of the total time spent during its life cycle, and that most of the time we devote to programming is in pursuit of maintenance efforts – change.[2] A significant reason offered by many experts for using the object-oriented programming paradigm is that it is much better than procedural programming at facilitating maintenance efforts. According to Scott Ambler, reusability is one of the great promises of object-oriented technology.[3] Simplifying program maintenance is a central theme with object-oriented design and one that is interwoven throughout this book. Also, as noted, each new release of SAP contains more and more object- oriented code, and it is in our best interest to become familiar with this new paradigm so we can more

easily understand how the system works, where and how we might place enhancements into the standard SAP code, and how we might begin to make effective use of the vast SAP-supplied global class repository. Although we could continue to ignore this new way of writing code for some time and still experience successes in our programming efforts, we do so at our peril.

How We Are Going to Get There

We are going to start where we are most comfortable and familiar, and then move slowly and methodically until we have mastered the fundamentals of object-oriented programming. This means we will start from the familiar surroundings in our home town of Procedureton and travel along the path of least resistance to our destination of Objectropolis.

Along the way from Procedureton to Objectropolis we will pass through the following districts:

- Encapsulation
- Abstraction
- Inheritance
- Polymorphism
- Interfaces

Each district will present its own unique landscape distinguishing it from the other districts. Although we will use this book primarily to provide the directions for navigating the new terrain, we will also take the opportunity to pause in each district long enough to become more familiar with the new concepts we will encounter by performing exercises designed to strengthen our grasp of the nuances and idiosyncrasies each district has to offer. In the same way that merely reading a book about swimming could not sufficiently prepare us for the experience of actually jumping into the water for the first time, merely reading this book without performing the accompanying exercises would leave us less than sufficiently prepared for the experience of actually using what we will be learning.

The first district we will encounter along the road to Objectropolis is Encapsulation, where the residents excel at organizing components in a way that reduces repetition and conceals those details we don't need to see. This is first because we already have some familiarity with this concept from procedural programming.

Farther down the road we will move through Abstraction, where the residents have mastered the art of describing the aspects of an entity and assigning a level of detail to components. They are also experts at *instantiation*, a technique used to make copies of things but where each copy has it own unique attribute values. We also will see how Abstraction and Encapsulation are related to each other.

[2]Software maintenance costs can be 75% of software total ownership costs ... http://galorath.com/wp-content/uploads/2014/08/software_total_ownership_costs-development_is_only_job_one.pdf

[3]www.drdobbs.com/a-realistic-look-at-object-oriented-reus/184415594

After Abstraction we will cross into Inheritance, a place where members of the same family tree exhibit similar attributes and behaviors. In some cases, the parents in this district perform many tasks on behalf of their children; in other cases, the children are offered

guidance for how to perform tasks themselves; and in still other cases, the children decide they know better than their parents how to perform some tasks and insist on doing things their own way.

Beyond Inheritance lies Polymorphism, where the residents exhibit multiple personalities, behaving differently from one moment to the next depending on how they are type-cast.

Finally, we will traverse through Interfaces, where the residents excel at developing standardized processes for communicating and exchanging information.

Upon passing through all these districts and reaching the end of the road, we will have arrived in Objectropolis, where we will stick our toe into the vast sea known as Unified Modeling Language (UML) from which we will learn some map-making skills, enabling us to orient ourselves as we continue to travel further into the inner sanctum of object-oriented design.

Eventually, we will leave the constraints inherent in the roads we had thus traveled and venture out beyond the stars, exploring the galaxy known as Design Patterns,[4] the center of which is an inferno providing the nuclear fusion for blending the concepts of Encapsulation, Abstraction, Inheritance, Polymorphism, and Interfaces into robust solutions to our programming challenges. So, buckle up as we embark on a celestial journey throughout the universe known as Object-Oriented Design.

Overcoming Psychological Barriers

I recall first hearing the term "object-oriented programming" in 1988 while I was working at Applied Data Research (ADR)[5] writing programs for mainframe computers predominately using proprietary languages. Some of the languages I used were invented by ADR and others by IBM, but I was writing code primarily in the primitive IBM 370-assembler language. I paid no attention to object-oriented concepts at that time. Years later I began hearing experts in the industry proclaiming that programmers who were familiar with procedural languages, by which many of them meant COBOL, had difficulty becoming familiar with and using object-oriented languages. The following statement from an article written by Michael Kölling of Monash University summarizes this claim:

> Learning to program in an object-oriented style seems to be very difficult after being used to a procedural style.[6]

I was one of these programmers already used to a procedural style, so naturally I assumed that I would encounter difficulty if ever I were to pursue learning an object-oriented language.

I learned the ABAP language in 1997 while it still was only a procedural language. Ten years later, undaunted by the Kölling claim, I took it upon myself to learn the object-oriented aspects of ABAP after already having spent years working in environments where the language included the object paradigm. Unlike some other languages, ABAP has evolved and now is considered amongst those languages that had their start as procedural languages, but have been extended with some object-oriented features. Having become comfortable with object-oriented concepts, I can now reflect on the Kölling statement above and attest that the difficulty lies not in the new programming style but in the fact that programming techniques

[4]"Roads? ... Where we're going, we don't need *roads*."; Dr. Emmett Brown in the 1985 movie *Back to the Future*. This is a fitting metaphor to illustrate the comparison between understanding the fundamentals of object-oriented design encountered along the road from Procedureton to Objectropolis and understanding the advanced level of power to which these fundamentals can be raised when incorporated into design patterns. [5]Later that same year ADR was acquired by Computer Associates. [6]Kölling, M., "The Problem of Teaching Object-Oriented Programming, Part 1: Languages," *Journal of Object-Oriented Programming,* 11(8): 8-15, 1999.

of any kind, procedural or otherwise, eventually become second nature, subconscious and automatic, and the programming techniques of any new and unfamiliar language and programming paradigm likewise will need to become second nature, subconscious and automatic.

Flip Turns

As a college undergraduate I was a member of the swimming team and specialized in the backstroke. For me, the primary event at a swim meet was the 200-yard backstroke, which required the swimmer to swim eight laps of a 25-yard pool. This entailed touching the wall of the pool and turning after seven of those eight laps. The most efficient way to make the turn is to use a technique known to swimmers as a flip-turn, whereupon, for the backstroke event, in touching the wall with the hand, the swimmer uses the leverage of the hand on the wall to flip backward, in a crouching position until upside down, twisting in mid-flip, throwing both feet against the wall, and then pushing off the wall in the other direction as quickly as possible. I finally mastered this early in my first year on the team.

During that first year I had reached a plateau with my event times. The coach took me aside and explained that he thought he could help. The conversation went something like this:

> "Jim, I think I know why your times are not improving. During your approach for a turn, you always reach for the wall with your right hand. Are you aware of that?"
>
> "Yes," I replied, "I always touch the wall with my right hand. It is the only way I am able to perform a flip turn."
>
> "You are losing precious time in those instances where your stroke cadence would not put your right hand on the wall when you arrive there, forcing you either to take a quick short stroke or to delay taking a stroke at all. You need to learn to perform a flip turn left-handed so you can turn with whichever hand is about to touch the wall. This will improve your times."

I worked on this for several weeks. At first it felt uncomfortable because of my proclivity to twist to the right upon touching the wall. I had to do this slowly so I could unlearn the automatic reflex to twist right and to get the feel of making the twist dependent on whichever hand was touching the wall.

Eventually I was able to perform the flip turn with either hand. My times began to improve noticeably because I was no longer handicapping myself to insure touching the wall right-handed. Indeed, much to my surprise, by the end of that first season I found that performing the turns left-handed had become my preference.

As with flip turns in swimming, we want to have the ability to *flip comfortably between the styles for both procedural code and object-oriented code* depending on the circumstances in which we find ourselves.

Baseball Bats

Most of us have played or watched a game of baseball, or at least are familiar with some athletic event where a long stick is held with both hands, such as cricket, ice and field hockey, lacrosse, golf, and pole vault. Most baseball players have a preference for holding the bat with the thumb of one hand touching or close to the pinky of other hand, a preference also applicable when holding the stick in those other sports mentioned. When held this way, whichever direction the bat is pointing in reference to the player holding both arms extended straight forward indicates whether the player is left-handed or right-handed. Once baseball players develop some skill with the game, they are most competitive when holding the bat their favored way, but can barely function when holding it opposite-handed. A few players are skillful when holding the bat either way; in baseball they are known as switch-hitters, able to "swing both ways," and are prized by their teams since they can stand in the batters box on whichever side of home plate gives them the most advantage at that point in the game, such as batting left-handed against a pitcher who is strongest against right-handed batters.

With baseball, swinging a bat left-handed is neither more difficult nor less difficult than doing so right- handed, but we will find one way to be more difficult than the other only after our preferred way of swinging the bat becomes subconscious and we no longer need to

think about it. As with baseball bats, we want to *develop our programing skills to be able to "code both ways,"* so that we are equally comfortable whether it is procedural or object-oriented code pitched at us.

Keyboards

Consider for a moment the keyboard you use to write code, which perhaps is the familiar QWERTY keyboard, so named because these are the first six letters of the first row of letters on the keyboard. Probably you have such facility with the keyboard by now that you do not even think where letters are located; typing has become second nature to you. The location of the letters on the QWERTY keyboard has an historical significance dating back to the time when typewriters were mechanical machines. These first generation typewriters had metal arms called type bars aligned in a semicircle within the machine, with each arm connected to one of the keys of the keyboard. At the other end of the metal arm was the "type," the print character that would strike a ribbon and leave the ink impression on the page. Typing characters in rapid succession would result in the type bar for one character retreating from the page and the type bar for another character advancing to strike the page. Some combinations of characters would cause adjacent type bars to move, with the respective type bars hitting each other as one was retreating and the other advancing, often causing jams that needed to be fixed manually. After some study, the characters were placed on the keys to avoid the most common combinations of characters from being adjacent type bars, and QWERTY was the result.

Now that typewriters have become relics of a past era, with their mechanical type bars having been replaced by electronics, there is no longer a need for the QWERTY keyboard. The constraint that caused this key arrangement to be invented in the first place no longer exists. Yet today we still use the QWERTY keyboard due to its use having become automatic for so many typists over the decades. Upon first seeing a QWERTY keyboard, many school children inquire why the letters are not arranged in the familiar alphabet series with which they recently have become familiar, and even some adults new to typing wonder why this keyboard is arranged the way it is as they struggle to master it. If we were to ask adults who have acquired some skill with the QWERTY keyboard to consider now using a keyboard where the letters are arranged in the alphabet series, we probably would find them resistant to the idea.[7] Here again, people respond this way because it becomes difficult to learn *anything* that deviates from what already has become established as second nature. Similarly, we want to *hone our facility with object-oriented programming concepts to the point where it becomes second nature to us* to the same degree as our facility with procedural programming concepts.

Programming from the Subconscious

The point here illustrated by flip turns, baseball bats, and QWERTY keyboards is that some aspects of our lives become automatic and subconscious, and this includes writing programs. We simply need to absorb the new programming paradigm into our subconscious, so that we can make the automatic response dependent on the programming paradigm in which we find ourselves writing code. It is not so much a case of difficulty with a new style as it is with what we are capable and incapable of doing subconsciously and automatically.

[7]One of my Object-Oriented Chalk Talks classes was attended by a woman who had immigrated to the United States from the former Soviet Union. When I raised this topic about switching from the familiar QWERTY keyboard, she related her story of already being familiar with a Cyrillic keyboard, and upon arriving in the US being confronted with the challenge of having to *switch to* QWERTY.

Getting Back on the Horse

Many readers may have already tried learning object-oriented programming on their own once before, abandoning the effort prior to completion when it became evident the task was more challenging than anticipated. This often leads to the perception that learning the concepts of object-oriented programming on one's own is an insurmountable obstacle.

After starting and abandoning an endeavour at the first sign of a problem, one usually is advised to "get back on the horse," a phrase meaning that one's confidence in overcoming a challenge should not be allowed to be defeated by the first fall, but simply to try again, perhaps using a different approach. Accordingly, this book provides a different approach for those who may be seeking one.

Ready to Take the First Step

Despite any previous attempts we might have made in trying to learn object-oriented programming and design, we have now prepared ourselves mentally and psychologically to meet the challenges we might encounter during the journey upon which we are about to embark. By now we should no longer subscribe to the notion that object-oriented design is so difficult that it cannot easily be understood by those who are familiar primarily with procedural design, a notion that only serves to shackle our efforts to absorb object-oriented concepts into our subconscious such that they become as familiar as procedural concepts. Although we may stumble at times along the way, we now possess the confidence not to let such minor setbacks impede steady progress toward our goal of reaching Objectropolis and beyond.

CHAPTER 2

The Elements of Object-Oriented

Programming

In this chapter, we will cover the basic elements of object-oriented programming and how they differ from the elements found in procedural programming. The associated concepts are applicable to virtually all object-oriented languages, so we will not see anything specific to ABAP in this chapter. Some new vocabulary will be introduced to describe the corresponding concepts and some suggestions will be presented for how to approach the task of extracting design information from the paragraphs found in associated requirements documentation.

So, how, exactly, is object-oriented programming different from procedural programming? Whereas the focus with procedural programming is on the processes to be applied to data, the focus with object- oriented programming is on the data upon which those processes are applied. The architecture of a procedural programming environment lends itself to components containing steps in a process, whereas the architecture of an object-oriented programming environment lends itself to components containing real-world representations of entities that can contribute to a process. While procedural programming is achieved through writing components such as programs and utility functions, object-oriented programming is achieved through writing components known as classes. Indeed, the *class* is a fundamental concept of object-oriented design. Whereas procedural programs are composed of main routines, subroutines, and the data fields on which they operate, object-oriented classes are composed of *attributes* and the *behaviors* defined for those attributes. Attributes are the data fields of a class (constants, variables, etc.), while the behaviors, also known as *methods*, are the actions applicable upon those attributes. Methods may be defined with a parameter interface, also known as a *signature*, by which information can be exchanged between the method and its caller. Both the attributes and the behaviors of a class are known collectively as its *members*. Although there are other aspects involved with object-oriented programming, classes, attributes, and behaviors constitute the basic elements of this programming paradigm.

Because it can have multiple attributes and various behaviors associated with those attributes, a class is known as a *complex data object*. Compare this with a simple data object, which represents a value primarily through a data type and a length, with occasional additional information. For example, a material number might be defined as 18 (its length) left-justified characters (its data type), while a sale price might be defined as 13 (its length) right-justified digits (its data type) where 3 of those digits constitute decimal positions (its additional information). In contrast, a class might be defined as a *sales order item*, and, amongst others, it may contain the two attributes (material number and sale price) just described, as well as behaviors to get and set the value of the material attribute and to calculate its sale price. Indeed, a complex data object is not restricted to containing only simple data objects as attributes; it may have other complex data objects as its own attributes. An example of this might be a class defined as *sales order*, which might contain an attribute defining an internal table of its associated sales order items, each of which, as just described, defined as a class containing the material and sale price of each item.

11 © James E. McDonough 2017 J. E. McDonough, *Object-Oriented Design with ABAP*, DOI 10.1007/978-1-4842-2838-8_2

Just as a program may have multiple data fields defined to hold the values of material numbers (simple data objects), so too can it have multiple data fields defined as classes (complex data objects), although the correct way to describe this is to say that the data fields are

references to class objects. Populating a field defined as a material number is a simple matter of moving a value into the field. Populating a field defined as a class reference is a bit different: in this case we *create* a class *object.*[1] Creating an object of a class is performed by the program at execution time and is known as creating an *instance* of the class, a process also described by the term *object instantiation*. For example, a beer *class* describes the aspects of beer, whereas a beer *instance* represents a specific glass of beer. We cannot drink the aspects of beer, but we can drink a glass of beer, and this is the reason why the beer has to be instantiated – that is, poured.[2]

So, in just the first few steps along the road from Procedureton to Objectropolis we have already learned some important new words in the vocabulary of object-oriented programming:

Class The definition of a complex data object composed mainly of data fields and actions applicable to those data fields

Attribute The term used to describe a data field defined for a class Behavior The term used to describe an action applicable to an attribute

Method Another name for behavior Signature The term used to describe the parameter interface facilitating the exchange of information between a method and its caller

Member The term used to refer generically to any component of a class Object A term used to describe the result of creating a class entity during program execution

Instance Another name for an object

Instantiation The term used to describe creation of an instance of a class

A Simple Approach to Object-Oriented Design

Often in object-oriented design it is helpful to conceive of these elements using words associated with sentence construction and grammar:

- Classes are described using *nouns*.
- Attributes are described using *adjectives*.
- Behaviors are described using *verbs*.

Let's see how this works using an example: we can define a class called *car* and attributes for it describing such things as relative age, appearance, aerodynamic quality, and color. Accordingly, we can define within our program two data fields which are references to objects of class *car*, and then, through object instantiation, place into them references to the instances of this class and describe each one using its corresponding adjectives:

1. The old, dirty, boxy, blue car.
2. The new, clean, sleek, red car.

[1]We shall see that we can also move values from one class reference field to another, but creating an object into one of those class reference fields is what enables moving a value into another class reference field. [2]A brilliant metaphor contributed by Paul Hardy.

Here we see that *car*, the description of the class, is a noun, and the values for its attributes *relative age* (old; new)*, appearance* (dirty; clean)*, aerodynamic quality* (boxy; sleek)*,* and *color* (blue; red) are all adjectives. Furthermore, we can define behaviors for the *car* class enabling us to set and get these attributes: *set_color, get_color, set_relative_age, get_relative_age,* etc. For behaviors, the first word represents the verb associated with each behavior, in this case, *set* and *get.*[3]

Taking this example to the next step, let's define a program, using the pseudocode shown in Listing 2-1, which contains two fields to hold the references to the two car objects and show how the program interacts with those objects.

Listing 2-1. Pseudocode for a Program Referencing Two Car Objects

```
relative_age type string appearance type string aerodynamic_quality type string
color type string car_01 type class of car car_02 type class of car

create new instance of car object into car_01 invoke behavior set_relative_age of car_01 with "old" invoke
behavior set_appearance of car_01 with "dirty" invoke behavior set_aerodynamic_quality of car_01 with
"boxy" invoke behavior set_color of car_01 with "blue"

create new instance of car object into car_02 invoke behavior set_relative_age of car_02 with "new"
invoke behavior set_appearance of car_02 with "clean" invoke behavior set_aerodynamic_quality of
car_02 with "sleek" invoke behavior set_color of car_02 with "red"

(more processing occurs here ...)

invoke behavior get_relative_age of car_01 into relative_age invoke behavior get_appearance of car_01 into appearance
invoke behavior get_aerodynamic_quality of car_01 into aerodynamic_quality invoke behavior get_color of car_01 into
color display "car 01:", relative_age, appearance, aerodynamic_quality, color

invoke behavior get_relative_age of car_02 into relative_age invoke behavior get_appearance of car_02 into appearance
invoke behavior get_aerodynamic_quality of car_02 into aerodynamic_quality invoke behavior get_color of car_02 into
color display "car 02:", relative_age, appearance, aerodynamic_quality, color
```

[3]*Set* and *get* probably are the two most common verbs you will see associated with methods of classes, and for good reason: set behaviors enable external callers to apply changes to the attributes of class objects, while get behaviors enable external callers to retrieve the values of those attributes. Methods enabling changing and retrieving attribute values are known as *accessor* methods. Using *setter* and *getter* methods provides a distinct advantage over accessing the attributes directly, as we will see later.

Indeed, much of object-oriented design can be gleaned from simple sentences. Here is a famous sentence:

The quick brown fox jumps over the lazy dog.[4]

Simply by identifying the nouns, verbs, and adjectives in this sentence leads us to a rough object- oriented design. The nouns are *fox* and *dog*. This means we will have two classes: a fox class and a dog class. The verb is *jumps* and it describes a behavior of the fox. Accordingly, our fox class will have a behavior (method) called jump. The adjectives describing the fox are *quick* and *brown,* so we might define two attributes for the fox class: one called *alacrity* which can hold the value "quick," and another called *color* which can hold the value "brown." Similarly, the adjective *lazy* describes the dog and is comparable to the adjective *quick* describing the fox, so we might define one attribute for the dog class, also called alacrity, which can hold the value "lazy."[5]

Upon completing this simple exercise of identifying sentence words, we find we have the beginning of an object-oriented design, as shown in Table 2-1.

Table 2-1. *The Beginning of an Object-Oriented Design*

Class	fox	dog

Attributes alacrity
alacrity color
Behaviors jump

At this point, we can take the liberty of adding two behaviors to the fox class, set_color and set_alacrity, and one behavior to the dog class, set_alacrity, to enable us to change the values of their corresponding attributes.

Let's provide some pseudo-code to fulfill the intent of the original statement from which we extracted these classes, attributes, and behaviors. The syntax of the pseudocode shown in Listing 2-2 follows the "object.action" model for classes. That is, to invoke a behavior of a class we first specify the class separated from its behavior by a dot; any parameters that follow are enclosed in parenthesis.

Listing 2-2. Pseudocode to Fulfill the Intent of the Original Statement

```
create object fox fox.set_color("brown") fox.set_alacrity("quick")
create object dog dog.set_alacrity("lazy")
fox.jump(dog)
```

The signatures for the behaviors (to set the color of the fox, to set the alacrity of the fox and dog, and for the fox to jump) implied in the statements above exemplify a technique using what are known as *positional parameters.* With positional parameters, the corresponding method accepts the values in the sequence specified on the statement into the method signature parameters in the order they appear. In the example above, we see only single values being passed to each method in its signature. Let's change this a bit by

[4]This is an example of a *pangram*, a statement which contains every letter of the English language. [5]We might also consider that since both fox and dog have an attribute for alacrity, it might be appropriate that both also have an attribute for color; however, this famous sentence does not provide any information about the color of the dog.

defining the signature for the set_color method to accept a color specified using the RGB[6] color model, where three integer parameters denote the light intensity values to be used for the colors red, green, and blue. We would change our example statement above for setting the color of the fox to the following statement:

```
fox.set_color(165,42,42)[7]
```

Some object-oriented environments support positional parameters in method signatures while others provide support for a technique that uses what are known as *keyword parameters,* where each value is associated with a keyword, as illustrated in the following modified example of the statement to invoke the method to set the color of the fox:

```
fox.set_color(red=165,green=42,blue=42)
```

Here, each parameter is associated with a keyword identifying its corresponding value, such as the keyword "red" associating the RGB setting for red with the value 165. Keyword parameters provide programmers the advantage of being able to specify multiple parameters in any order, since it is its keyword, not its sequence, that associates a value with a method signature parameter.

Indeed, this concept of positional and keyword format for parameters is not unique to object-oriented environments, each format having existed in procedural languages for decades.

Rarely do we ever encounter foxes and dogs in our programming efforts, so let's take this to the next level and inspect a sentence with more applicability to data processing:

> A hazardous-material sales order must be assigned a placard value prior to transmitting as an outbound XML document through the exchange portal.

Here, again, to get a rough object-oriented design, we simply identify the nouns, verbs, and adjectives in the sentence. Through identifying the nouns we define three classes: *sales order, XML document, exchange portal.*[8] Through identifying the adjectives, we assign 1) to the sales order an attribute we'll call *material type*, which indicates whether or not it is for a *hazardous material*, 2) to the XML document an attribute we'll call *direction*, which indicates whether it is inbound or *outbound*, and 3) to the exchange portal no attributes at all. Through identifying the verbs we assign *transmit* as a behavior for an XML document and *assign placard* as a behavior for a sales order. The result is shown in Table 2-2.

Table 2-2. *Result of Extracting Information from the Statement with More Applicability to Data processing*

Classes sales order XML document exchange portal

Attributes material type direction

Behaviors assign placard transmit

Actual object-oriented design is not nearly so neat and simple as depicted here, but this illustrates how you might parse the sentences representing requirements outlined in a design document to arrive at a general set of object-oriented classes as a starting point to satisfy the design requirements.

[6]RGB is the acronym for each of the additive primary colors (red, green, blue) used to set the colors of light for, among other things, pixels on computer monitors. See www.rapidtables.com/web/color/RGB_Color.htm. [7]This RGB color setting (red at 165, green at 42, and blue at 42) represents the color brown. [8]A case can be made that "sales" is an adjective for "order", and that "XML" similarly qualifies "document." Here we will regard "sales order" as a distinct entity (a noun) and likewise "XML document" and "exchange portal" as distinct entities.

Pillars of Object-Oriented Design

There are many different object-oriented languages and software development environments in popular use today, each with its own unique characteristics. In general, there are four aspects of object-oriented design, regarded by many experts as *principles* shared by virtually all of these languages and environments:

- Encapsulation
- Abstraction
- Inheritance
- Polymorphism

These can be regarded as the pillars of object-oriented design, each contributing its relevant concepts in support of the overall architecture of an object-oriented system. As illustrated in Figure 2-1, object-oriented design rests upon these four pillars. Remove one of the pillars and the architecture becomes out of balance, no longer capable of fully supporting an object-oriented environment.

A fifth aspect of object-oriented design, Interfaces, is found in some but not all such environments. Its purpose is to provide external access to object members through established communication and data exchange formats, so it is illustrated in our colonnade graphic in Figure 2-1 as a doorway.

Figure 2-1. *Object-oriented architecture*

Summary

In this chapter, we learned how object-oriented programming primarily differs from procedural programming, specifically that procedural design focuses on the process to be performed whereas the focus with object-oriented design is on managing the data used. We also learned how to begin the process of identifying the various classes we might need for a software design by extracting information from the sentences we find in the corresponding requirements. Some new vocabulary words applicable to object- oriented design were presented:

- Class

- Attribute
- Behavior
- Method
- Signature
- Member
- Object
- Instance
- Instantiation

In addition, we learned that there are certain principles shared by virtually all object-oriented environments:

- Encapsulation
- Abstraction
- Inheritance
- Polymorphism

We saw how all of these terms contribute to supporting the architecture of object-oriented design and learned that interfaces, while also a principle of object-oriented design, is not applicable to all object- oriented environments.

Exercise Preparation

Refer to Chapters 1 and 2 of the functional and technical requirements documentation (see Appendix B) describing the accompanying ABAP exercise programs. Take a break from reading the book at this point to become familiar with the concepts behind the exercises and to prepare your ABAP training environment for writing, changing, and executing the corresponding exercise programs.

CHAPTER 3

Encapsulation

We start our journey to Objectropolis from the perspective of Encapsulation, largely because this is the object-oriented design concept with which most procedural programmers already are familiar. The word *encapsulate* means "to encase in or as if in a capsule."[1] Procedural programmers have been doing this for years via subroutines and locally-defined variables.

To illustrate this concept, let's establish a familiar frame of reference for encapsulation. Suppose we write a procedural program like any one of thousands you might find in your own programming environment. It might be constructed similar to the outline shown in Listing 3-1.

Listing 3-1. Representative Procedural Program

```
program bnx0037 global_variable_1
global_variable_2 global_variable_3
o o o global_variable_n main_routine
    do subroutine_a do subroutine_b subroutine_a
local_variable_1 local_variable_2 statement 1
statement 2 subroutine_b
  local_variable_3 statement 3 statement 4
```

With this example we see a program containing a set of global variables, a main driving routine, and two subroutines. The two subroutines illustrate encapsulation of code, procedures to be performed from some other point in the program, in this case from the main routine. These subroutines also illustrate

[1] *American Heritage Dictionary of the English Language*, 4th edition, Houghton Mifflin Company, 2000, p. 588.

19 © James E. McDonough 2017 J. E. McDonough, *Object-Oriented Design with ABAP*, DOI 10.1007/978-1-4842-2838-8_3

20 encapsulation of data fields, where each subroutine has its own local variables. For this program, the subroutines are visible and available only within the program unit bnx0037 and are not visible beyond the constraints of this program unit[2].
The global variables also are visible and available from anywhere within he program unit bnx0037, but are not accessible beyond this program unit. Similarly, access to local variables defined within a subroutine are visible and available from anywhere with the subroutine[3], but are restricted to the scope of the subroutine, and are not visible beyond the constraints of the subroutine in which the local variable is defined.

Separation of Concerns

Encapsulation is a technique used to facilitate modularizing code in pursuit of the design principle known as *separation of concerns*. This concept, credited as first being presented by Edsger W. Dijkstra in a 1974 paper he wrote titled "On the Role of Scientific Thought"[4], addresses the maintenance benefits to be gained by arranging software components into modules such that the processing performed by a single module is limited to a specific concern and does not cross the distinct boundaries separating areas of processing. This often is illustrated using the Model-View-Controller design pattern, which segregates the programming logic for the application (model), presentation (view), and manipulation (controller) of information into separate components.

Visibility

Encapsulation is often associated with the concept of *information hiding*[5], whereby elements and implementations defined in software components can be segregated from each other and from outside entities under the expectation that subsequent changes would not require a proliferation of modifications across many components. The degree to which information within a class can be hidden is facilitated in many object-oriented languages through the assignment of a *visibility* level to that information.

The term *visibility* describes a fundamental concept associated with the object-oriented principle of encapsulation. Each member of each class has a visibility level, controlled by the programmer, similar to the global and local visibility just described in the procedural program example. The visibility of a member is a designation of how that member is visible to other entities. With object-oriented programming, visibility is described using neither global nor local, but using other words that denote gradations between most visible and least visible.

To illustrate this, we'll use the tapered shape diagram shown in Figure 3-1, which illustrates maximum visibility at the widest part and minimum visibility at the narrowest part, and places the visibility descriptors in their natural sequence from least visibility to most visibility.

[2]In some programming languages, notably ABAP, such subroutines *are* accessible from outside the program unit, but also require the name of the program as a qualifier for access to them, as in
perform subroutine_b in program bnx0037 ...

This is now considered an obsolete ABAP programming technique and is discouraged for any new development efforts. [3]In some languages, the placement of the definition for a local variable has an effect on its availability within the subroutine, making it available only beyond the point at which it is defined. This generally is the case with ABAP, although some dynamic techniques enable access to fields defined subsequently. [4]See www.cs.utexas.edu/users/EWD/transcriptions/EWD04xx/EWD447.html. [5]See www.defit.org/information-hiding/.

The bottom of the taper depicts the most restrictive visibility to class members. This lowest level of visibility is reserved for *method variables*, which are the functional equivalent of local variables in procedural programming.[6] Technically, these method variables are not themselves members of classes, but are variables defined within methods that are members of classes.

The next widest level of visibility is called *private*. It describes the visibility level of members whose visibility is restricted to this

class only, and might be considered the functional equivalent to the visibility of both subroutines and global variables in procedural programming.

The next widest level is called *package*[7]. It describes the visibility level of members whose visibility is restricted to this class and any other components within the same package. This visibility level has no equivalent in procedural programming. In Java, this is known as *package-private* visibility.

The next widest level is called *protected*. It describes the visibility level of members whose visibility is restricted to this class, any other components within the same package and any classes inheriting[8] from this class. This visibility level has also no equivalent in procedural programming.

The next widest level is called *public*. It describes the visibility level of members whose visibility is not restricted at all; that is, a member with public visibility is visible to any other component within the programming environment. This visibility level is similar in nature to the way ABAP reports are available to any other component through the SUBMIT statement, as well as to the way ABAP function modules are available to any other component through the CALL FUNCTION statement, to the way global class and interface definitions in ABAP are available to any other component simply by referring to the global class or interface, and to the way domains, data elements, tables, etc. defined in the ABAP DataDictionary are available to virtually any other objects defined within the ABAP repository.

Visibility levels also are known as *access modifiers*. Table 3-1 summarizes the visibility settings for class members.

Figure 3-1. *Visibility gradations*

[6]Here we are using the term *variable* to refer to these entities; however, this also would include constants, type definitions, and any other variation of defining or assisting in defining a data field. [7]This visibility level is not available in all object-oriented environments. Java provides support for this visibility level. It was not available to ABAP with the first release containing object-oriented support, but evidence that it is becoming available can be found at https://help.sap.com/saphelp_nwpi71/helpdata/en/45/ c2b44f23b352f5e10000000a1553f7/content.htm. [8]This will be explained in more detail in the section on Inheritance.

Figure 3-2 shows the same information using a different format[9], where the rows of the visibility level column show increasingly wider levels of visibility from bottom to top, and where the columns to the right of the visibility level column show increasingly greater levels

of access from right to left. Notice how the “Yes” and “No” accessibility values fall evenly distributed on both sides of the diagonal line.

Table 3-1. *Visibility Levels*

Visibility level	Accessibility
Public	Accessible to any other entity within the same environment.
Protected	Accessible within this class, within any other entities assigned the same package, and within any inheritors of this class.
Package	Accessible within this class and within any other entities assigned the same package.
Private	Accessible within this class only.
Method variable	A variable is accessible only within the method in which it is defined.

Figure 3-2. *Visibility levels with indications of access to same class, same package, subclass, and other entities*

Let’s explore this concept through an example. Suppose we have groves of oak trees in various settings around town, offering the benefits of shade in summer and spectacular changes of color in autumn. We will define a visibility level to each grove of oak trees based upon the particular setting in which we find it.

First we will consider a grove of oak trees we find in the public park in the center of town. This grove of oak trees is a member of the park, and being a public park, the grove is accessible to all residents of the town as well as to all non-residents of the town. The benefits offered by this grove are available to everyone who strolls through the park. Accordingly, we assign the grove of oak trees *public* visibility.

Next we will consider a grove of oak trees we find in a wildlife sanctuary located on the outskirts of town. This grove of oak trees is a member of the wildlife sanctuary, and being a protected place, the grove is not accessible to all town residents and non-residents. While there are many ways these folks can become associated with the wildlife sanctuary, the benefits offered by this grove of oak trees are available only to those who have a reason for strolling through this protected wilderness. Accordingly, we assign the grove of oak trees *protected* visibility, meaning only those folks with a specific type of association to the wildlife sanctuary, and their friends,[10] have access to the grove of oak trees.

[9]The format is borrowed from the chart available at http://docs.oracle.com/javase/tutorial/java/javaOO/ accesscontrol.html. [10]Friendship is a concept related to encapsulation and will be covered later in this chapter.

Next we will consider a grove of oak trees we find in a common area of a gated community also located on the outskirts of town. This grove of oak trees is a member of the common area of the gated community. The maintenance for this grove of oak trees comes from the homeowners association fees paid by the residents who live in the gated community. The homeowners

association fees are part of the package of regulations accepted by all residents of the gated community for the privilege of living within its walls. The grove is accessible only to the residents of the gated community. Accordingly, we assign the grove of oak trees *package* visibility, meaning only those folks who have accepted the same package of regulations for being residents of the gated community and their friends have access to the grove of oak trees.

Next we will consider a grove of oak trees we find in the backyard of a house located near the center of town. This grove of oak trees is a member of the property on which the house sits, and since it is located on private property, the grove is accessible only to the residents of this house. Accordingly, we assign the grove of oak trees *private* visibility, meaning only the residents of the house on this property and their friends have access to the grove of oak trees.

Finally we will consider a grove of oak trees appearing in a picture hanging on the wall of the den we find in that same house near the center of town. This grove of oak trees is a member of only one room of the house. Residents of the house have access to this grove of oak trees only when they enter the den, and it immediately becomes unavailable to them when they leave the den. Accordingly, we assign the grove of oak trees *method variable* visibility, meaning only those residents engaged in the behavior of using the den have access to the grove of oak trees.

Regarding visibility, the conventional wisdom within the object-oriented community is to define these variables, attributes, and methods with the most restrictive visibility possible while still enabling access to them as necessary. This means

- Variables should be defined as local method variables unless there is a compelling reason for them to be defined as attributes of the class.

- Class members should be assigned private visibility unless there is a compelling reason for the visibility to be set to a wider visibility level.

My advice with new object-oriented development is to define all variables as method variables and all methods with private visibility, and then to widen the visibility level as necessary, but then only to the extent of the least visibility that will still accommodate the required access.

Why Visibility Matters

My experience with existing procedural programs has been to find a proliferation of global variables that have no reason to be defined with global visibility. This occurs for a variety of reasons:

- The author developed a bad habit of defining all variables with global visibility rather than giving any thought towards which of them should be defined locally.

- The author recognized there were multiple subroutines that required using the same set of local variables, and so defined them once globally rather than locally for each subroutine.

- The author might have been enticed by the presence of program segments, generated manually or automatically, the primary purpose of which is to contain global components of programs.[11]

- Some aspects of the programming language or environment work only when interacting with variables defined globally.[12]

[11]In ABAP, the so-called TOP include of function groups is a good example of this type of program segment. [12]In ABAP, the interaction with screen definitions is based implicitly upon defining global variables in the program with the same names as those defined for the screens.

• A precedent had been set by the original author, who defined variables unnecessarily with a global visibility, and this style was perpetuated by subsequent maintenance programmers to retain stylistic consistency.

Regardless of the reason why, the indiscriminate use of global variables is one of the biggest impediments to subsequent maintenance efforts. Too often it requires the maintenance programmer to perform an exhaustive search through the program for each global variable to determine whether or not it is safe to use or change its value at some critical point in the code. Frequently it is expediency in writing the first release of a program that lures developers into taking shortcuts with the design, trading ease of initial design for subsequent ease of maintenance. Since it is likely that much more time will be spent in the maintenance of a program than in its initial design, such cavalier inattention to variable visibility eventually causes problems, and usually it is only during subsequent maintenance efforts where the consequences become evident.

Visibility assigned at the most restrictive level to permit the necessary processing becomes a benefit during maintenance efforts by reducing the time it takes the developer to identify the necessary changes required as well as enabling the refactoring of code to be performed without any concern for rendering other components syntactically invalid. This is why visibility matters. Accordingly, developers who take the time to apply the appropriate visibility levels to program variables during design enable subsequent maintenance programmers to reap the benefit of shorter maintenance efforts.

With most of my object-oriented endeavors, I have found that attributes usually have a more restrictive visibility assigned to them than behaviors. Indeed, I usually tell my students that if they follow the advice offered by object-oriented scholars they will probably arrive at a class design that incorporates "public behaviors and private attributes." This arrangement stems from the necessity to offer external entities access to the values of the attributes of a class but not to the attributes themselves. Accordingly, many times there will be definitions for methods known as *getter* and *setter* methods, behaviors that offer public access by external entities to values of attributes of a class that are defined with a visibility level other than public. External entities use *getter* methods to make a request to *get* the value of an attribute of the class, and *setter* methods to make a request to *set* the value of an attribute of the class. In both cases, the class is in complete control over whether the value of the attribute will be allowed to be retrieved by the getter method or altered by the setter method.

Realm

In some object-oriented environments, members of classes belong to one of two different realms:

- Those members that *are* associated with a specific instance of a class
- Those members that *are not* associated with a specific instance of a class

Virtually every object-oriented environment supports the definition within a class of members that are associated with a specific instance of the class. Those that also offer support for class members not associated with a specific instance of the class include C++, C#, Visual Basic, Smalltalk, Java, and, of course, ABAP. In most of these languages, those members not associated with a specific instance of a class are known as *static* members.[13] Accordingly, we can refer to which one of these two realms a member is associated by the following terms:

- Instance member
- Static member

[13]In Smalltalk, these are known as *class* variables and methods. In Visual Basic, these are known as *shared* members because the term *static* has a wholly different meaning in this language.

In many object-oriented languages, including C++, C#, Java, and ABAP, a static member is marked so by including a qualifier with the definition of the member indicating it is a static member, with the absence of a such a qualifier indicating it is an instance member. Indeed, in the languages C++, C#, and Java, the qualifier used with a member definition to denote that it is a static member is the word *static*.[14]

Whereas instance members are bound with a specific instance of the class, static members are available and accessible to all instances of the same class. Indeed, static members of a class are available during execution even when there have been *no*

instantiations of the class, meaning that these members are available immediately once a program containing the class definition begins to execute. It is not necessary to create any instances of the class to use its static members. The same rules of visibility apply equally to static members and instance members.

An instance attribute will exist once for each instance of the class with which it is associated. In contrast, a static attribute of a class exists only once for the entire execution of the program and is shared across all instances of its class. Instance methods have access to both the instance members and static members of its class. In contrast, static methods have access only to the static members of its class. Instance members exist only upon instantiation of the class, and only for as long as the instance remains active. In contrast, static members exist during the entire execution of the program, irrespective of the presence or absence of instances of its class.

The following example expands upon our fox and dog classes from the preceding chapter. Here we have included some new members for the dog class, shown highlighted in Table 3-2:

• A new instance attribute called registration_number

• A new instance behavior called set_registration_number, which will use the value it receives to set the new instance attribute registration_number

• A new static attribute called last_used_registration_number

• A new static behavior called get_next_registration_number, which will add 1 to the last_used_registration_number and return the new value

Table 3-2. *Attributes and Behaviors for the Fox and Dog Classes*

Class Fox Dog
Attributes alacrity
color
25 alacrity **registration_number static last_used_registration_number**
Behaviors jump **set_registration_number**
static get_next_registration_number

Notice that all members of the fox and dog classes that are not explicitly described as static are, by default, instance members. This includes all of the attributes and behaviors of the fox, the alacrity and registration_ number attributes of the dog, and the set_registration_number behavior of the dog. The dog class also has a static attribute called last_used_registration_number and a static behavior called get_next_registration_ number. Notice also that the dog class now has a combination of both instance and static members.

Now, each time we create a new instance of class dog, we can assign it a registration number unique from all other dog instances simply by invoking the static method get_next_registration_number of the dog class to get the next value and then calling the instance method set_registration_number to apply it to our dog instance attribute registration_number. This works because the static attribute last_used_registration_ number is available to all dog instances. Upon creating the first dog instance and invoking the static behavior

[14]As we shall see, the prefix "class-" is used to denote a static member in ABAP.

26get_next_registration_number, the static attribute last_used_registration_number would be incremented and the new value returned to us, which we would place into the instance attribute registration_number of our new dog instance via the set_registration_number behavior. Then when we create a second dog instance, we would go through the same series of actions but we would get a different registration number. This is because the static members are shared amongst all instances of the dog class, and retrieving a registration number for our second dog instance uses the same static attribute that had been used to register our first dog instance. Accordingly, if we were to continue using this sequence we would get a unique registration number with each new instantiation of a dog class, as each new dog instance leaves behind its registration number in the static attribute last_used_registration_number for the next dog instance to see. This concept of instance members versus static members may be difficult to grasp, so let's use a metaphor that illustrates it more clearly. Suppose we learn that our management has made arrangements for us to attend one of the annual technology conventions in Las Vegas, Nevada this year. Upon arriving at our hotel we find that we have a reservation for one of the hotel guest rooms, each of which is virtually identical to each of the other guest rooms. Accordingly, our reservation permits us to occupy one instance of a guest room. Others who are attending the same

technology convention also are staying at the same hotel, but each person has a unique hotel reservation, permitting each person to occupy some other instance of a guest room. Our room has a unique room number on the door and we have been issued a card key for entrance to the room.

This is the only instance of *hotel room* to which we have sole access; it is our specific instance. Upon entering the room, we can turn on one of its many lights, tune the television to our favorite channel, and adjust the climate controls of our guest room without these changes having any effect upon the lights, television, or climate of any other guest room in the same hotel. Indeed, the other guests could be making similar changes to lights, television, and climate in their guest rooms, but their changes do not affect our guest room. We could sit in one of the chairs in our guest room, and then even proceed to rearrange the furniture in our room, with no effect upon the furniture in any of the other guest rooms. In this case, the lights, television, climate controls, and furniture constitute instance attributes of the hotel room, accessible only to the guest occupying the room, and the actions turn_on_guest_room_light, tune_guest_room_ television, adjust_guest_room_climate_controls, and rearrange_guest_room_furniture constitute instance methods applicable to these instance attributes.

Later, we decide to leave our guest room and visit the hotel lobby, a room in the hotel shared by all of the hotel guests. The lobby also has lights, a television, climate controls, and furniture, things that are not associated with a specific hotel guest but are available to all hotel guests, and constitute static members of hotel room. Now if we find an empty chair in the lobby and sit in it, the chair cannot be used by any of the other hotel guests until we unseat ourselves from it. Similarly, if we were to reset the lobby lights, change the channel on the lobby television, alter the lobby climate controls, and rearrange the lobby furniture, these changes would immediately affect every hotel guest who also is visiting the lobby. In addition, when one of the other hotel guests makes changes to these things while we are in the lobby, these changes immediately affect us. In this case, the lights, television, climate controls, and furniture constitute static members of hotel room, shared by and accessible to all hotel guests, and the actions turn_on_lobby_light, tune_lobby_ television, adjust_lobby_climate_controls, and rearrange_lobby_furniture constitute static methods applicable to these static attributes. As guests of the hotel, we have access to the instance members of our own guest room as well as those static members located in the hotel lobby.

Furthermore, we could depart our own hotel and walk down the street to the next hotel, where none of its guests have yet arrived and registered. Accordingly, it has no instances of guest room occupants. Despite no instances of guests, and even though we have no intention of registering as a guest at this hotel, we still are able to walk into its lobby and sit in a seat, which now cannot be used by anyone else visiting the lobby of this hotel. Indeed, there is nothing to stop us from resetting the lobby lights, changing the channel on the lobby television, altering the lobby climate controls, and rearranging the lobby furniture,[15] all of which would have an immediate effect upon every other person also visiting the lobby. In this case, the lights, television,

[15]Except, perhaps, hotel security guards, who might not tolerate our adventures in lobby redecorating.

climate controls, and furniture still constitute static members of this hotel room, but now we see that even though they are shared by and accessible to all hotel guests, they also are available to those who are not guests of the hotel, even at a time when the hotel does not yet have any guest room occupants. Because we are not guests of this hotel, we have access only to the static members of the hotel room, and not to any instance members of a guest hotel room.

In summary, an instance attribute exists once per instance of the class, and each one is unique and separate from the instance attributes of other instantiations of the class, whereas static attributes exist once per program execution, are available to all instances of the class, and are available to external entities even when there are no corresponding instances of the class. Instance methods have access to both the instance attributes and static attributes of its class, whereas static methods have access only to the static attributes of its class.

The Encapsulation Unit

Encapsulation unit is a term referring to the boundaries of encapsulation exhibited by an entity. In object- oriented design, encapsulation units exist at two different levels.

At one level, the entire class definition serves as the encapsulation unit. It can be regarded as a container with a lid, as shown in Figure 3-3. The walls of the container define the boundaries of the encapsulation unit. External access to the class members is available only through the public interface, which is represented by the neck of the container and its open lid. Only those attributes and behaviors that have public visibility are accessible through the public interface. Class members having any other visibility are not publicly accessible, but they are contained within the class encapsulation unit and are accessible to the other members within the same class.

Figure 3-3. *Encapsulation unit*

CHAPTER 3 ■ ENCAPSULATION

A new class-level encapsulation unit is established when a class is instantiated. This encapsulation unit remains in effect for the life of the class instance.

At another level, each method of the class is itself an encapsulation unit, allowing for the definition of local variables that are not available outside the method. A new method-level encapsulation unit is established with each invocation of a method, and is destroyed upon exiting the method. Any local variables defined within the method are set with their initial or default values upon entering the method.[16] This means that the values of these local variables from a previous invocation *are not retained* for any subsequent invocations.[17]

Instance Encapsulation Units

We can define a class such that every one of its members belongs to the instance realm; that is, none of its members are marked as static. This constitutes an *instance class*. It is necessary to create instances of an instance class in order to access its members. Instance classes are used in those cases where we need multiple instances of the class to facilitate the necessary processing, when each instance would contain the same set of attributes but when each instance has different values assigned to those attributes. For example, we might define a class *sales order item* for which we might expect there to be multiple instances of its objects to handle the processing for a single sales order, and we would expect that no two instances of sales order item would have identical attribute values.[18]

Static Encapsulation Units

We can define a class such that every one of its members belongs to the static realm; that is, each member is marked as static. This constitutes a *static class*. It is not necessary to create any instances of a static class in order to access its members. Indeed, while it may be technically possible to create an instance of a static class, it would have no instance members that could be accessed. On the other hand, because a static class is not instantiated, there can be only one copy of its members available for access. This means that a static class should be used only in those cases where we need only one entity of the class to facilitate the necessary processing. For instance, we would expect to have multiple instances of the sales order item class, noted above, to handle the processing for a single sales order. Meanwhile, we might define a class called screen manager for which we certainly would not want multiple instances; a single screen manager is all we would need to manage the presentation of information to the user, and having more than one screen manager would present a conflict.

[16]The concept that subroutine variables are initialized with every invocation is not unique to object-oriented languages. This also applies to local variables defined between form and endform in classical ABAP. [17]This even applies to recursive invocations of methods, where the values of local variables in an outer level of method invocation are not available to each subsequent inner level unless these values explicitly have been passed between these levels via the method signature. [18]Although it is not mandatory that each instance of a sales order item, or for that matter a pair of instances for any type of class, contain a unique combination of attribute values, it stands to reason that if any two of them had the same values, they could simply be consolidated into a single instance, which could indicate the aggregate quantity of the two instead of having separate identical instances.

Some object-oriented scholars dismiss the idea of static classes as a technique that falls short of the necessary requirements to be truly considered object-oriented, discouraging their use and pointing out that there is no concept of *object* associated with a static class[19]. Indeed, in object-oriented environments, supporting multi-threading static classes is not considered "thread safe."[20] While it is true that static classes will not be able to take full advantage of all the principles of object-oriented programming (for instance, polymorphism is not possible within static classes), they do offer a stepping stone for procedural programmers to become more comfortable with programming in object-oriented environments. Of note, in ABAP environments, static classes are very similar to simple

function groups.[21] Since function groups are an area of ABAP familiar to most procedural programmers, perhaps it would be helpful to compare the two to see how they are similar and how they differ:

- Both static classes and function groups encapsulate their components.
- Neither static classes nor function groups need to be instantiated to be used.
- The function modules of a function group are equivalent to the public methods of a static class.
- The data definitions usually defined in the TOP include of a function group are equivalent to the private attributes defined for a static class.
- The subroutines defined within function groups are equivalent to the private behaviors defined for a static class.

Static classes offer some other advantages not available to function groups:

- A static class can define public attributes, such as constants, field formats, and structures to be used with parameters of public method signatures

  For example, a class alarm_clock can define a public type alarm_setting, defined as type single character, along with public constants alarm_on and alarm_off as type alarm_clock.alarm_setting, having values X and space, respectively, and public method set_alarm whose signature contains parameter alarm defined as type alarm_ clock.alarm_setting. An external entity using class alarm_clock can now define its own field using the public type alarm_clock.alarm_setting, and then move the public constant alarm_clock.alarm_on into this field before using it with the alarm parameter on a call to the set_alarm method. In this way, the caller can refer to the class in terms the class provides.

- The ABAP compiler enforces a stricter compliance with syntax for static classes than it can for function groups

Accordingly, a static class can be defined as a substitute for a simple function group, offering all the same features and capabilities available to simple function groups, and more.

[19]The design pattern known as Singleton, covered in a subsequent chapter, is a better alternative to a static class, offering a way to define a class for which we expect one and only one instance to be available. [20]See https://msdn.microsoft.com/en-us/library/a8544e2s.aspx. [21]By simple function groups, I mean those that do not have function modules defined with the extra clauses for special operation, such as “destination,” “starting new task,” “in background task,” and “in update task,” as well as those that are written to facilitate special features such as BAPI and IDOC processing.

Hybrid Encapsulation Units

We can define a class such that some of its members belong to the instance realm while other members belong to the static realm. This constitutes a *hybrid class*. It is not necessary to create any instances of a hybrid class in order to access its static members; however, its instance members become accessible only upon creating an instance of a hybrid class. This is a common scenario in object-oriented design.

A good example of how this can be used effectively is to take the example of the sales order item class we have been using so far and embellish it. To turn this otherwise instance class into a hybrid class, we need to include some static members. We could include a static attribute to represent the next available sales order item number as well as a static behavior to get the next sales order item number, which when invoked will increment the static attribute holding the next available sales order item number and

then send this value back to the caller. Since the new attribute and new method are both static, they both are available to all instances of sales order item, with the value of the attribute holding the next sales order item number being shared across all instances of sales order item. Upon creating each new instance of a sales order item, this static method can be invoked to insure that the item number assigned to the new instance is unique amongst the set of all instances of sales order items being created.

Constructors and Destructors

A special type of method, known as a *constructor*, can be defined for a class to specify the activities to be undertaken during the creation of the class encapsulation unit, such as initializing the values of attributes. Another special type of method, known as a *destructor*, also can be defined for a class to specify the activities to be undertaken during the destruction of the class encapsulation unit, such as releasing those resources an object might have acquired.

A class encapsulation unit can be created regardless of whether or not the class has a constructor method defined for it, and similarly can be destroyed regardless of whether or not there is a destructor method defined for it. These constructor and destructor methods are defined by the programmer to instruct the runtime environment what to do *in addition* to creating and destroying the class encapsulation unit.

Although they are known as methods, constructors and destructors are not regarded as members of the class since they can be used only to create and destroy the class encapsulation unit and are not otherwise available during the lifetime of the class. Furthermore, constructors and destructors are defined such that each one applies either to the static realm or to the instance realm.

Instance Constructors and Destructors

Instance constructors are invoked automatically by the runtime environment when a new object is being created. The instance constructor facilitates setting the new object to an initial state, and will be executed once and only once for an object. Because new instances of objects are requested explicitly, an instance constructor can include a method signature, the parameters of which are available for use during the process of creating the instance. Instance constructors have access to both static and instance members of the class.

Instance destructors are invoked automatically by the runtime environment when an existing object is to be destroyed. The instance destructor facilitates any cleanup activities required during object destruction.

Whereas virtually every object-oriented environment supports instance constructors, only some provide support for instance destructors. Object-oriented environments providing support for the concept of Resource Acquisition Is Initialization (RAII),[22] such as C++, necessarily provide support for instance destructors.[23] In contrast, object-oriented environments providing support for what is known as automatic garbage collection[24] employ the dispose pattern[25] for resource cleanup, and some of these environments, such as Java and ABAP, provide no support for the explicit definition of instance destructors.[26]

Static Constructors and Destructors

Static constructors are invoked automatically by the runtime environment when a class is first accessed, which may be due to an external reference to a static member of the class or to the creation of the first instance of the class. The static constructor facilitates setting the static attributes of the class to an initial state, and will execute once and only once for the class. When the static constructor is triggered due to the creation of the first instance of the class,[27] it will run to completion before creation of the first instance begins. This means that the static constructor for a class will always start and finish prior to the start of the instance constructor when creating the first instance of that class. Because there is no specific statement that will cause a static constructor to be invoked, a static constructor cannot include a method signature. Static constructors have access to the static members of the class, but cannot access instance members.

Static destructors are invoked automatically by the runtime environment when the static members of a class are to be destroyed. The static destructor facilitates any cleanup activities required during class destruction.

Only some object-oriented environments provide support for static constructors. C# and ABAP are two that do. Neither C++ nor Java supports this concept, although Java does support what are known as static initialization blocks.

Meanwhile, it is rare to find any object-oriented environment supporting static destructors. C++, C#, Java, and ABAP do not offer support for this.[28]

Friendship

Some object-oriented languages, among them C++ and ABAP, offer the capability for a class to enable other specific classes to have unrestricted access to its members. This effectively renders all members of the class publicly visible to those other classes. This capability is known as *friendship*.[29] A class offers friendship to other classes by specifying the names of those classes that it considers its friends.

Friendship only can be offered; it is not reciprocated. That is, a class offering friendship to another class does not itself suddenly become a friend of that other class; the other class also needs to offer its own explicit friendship.

[22]See http://en.cppreference.com/w/cpp/language/raii. [23]See http://en.cppreference.com/w/cpp/language/destructor. [24]See http://basen.oru.se/kurser/koi/2008-2009-p1/texter/gc/index.html. [25]See https://msdn.microsoft.com/en-us/library/b1yfkh5e(v=vs.110).aspx. [26]Some languages running in environments with garbage collection provide a clean-up method, such as Java's *finalize,* which is invoked during the garbage collection phase, but it is indeterminate exactly when the corresponding object will be garbage collected. Meanwhile, destructors are invoked immediately as the object is being destroyed. The difference typically stems from the capability in the language explicitly to delete an object. Objects in C++ can be explicitly deleted, so a destructor method is applicable. Objects in Java and ABAP are not explicitly deleted. [27]A static constructor can be triggered well before the creation of the first instance of the class, but if not, then it is certainly triggered upon creation of the first instance of the class. [28]One language that does support static destructors is D, in which the static destructor for a class is invoked upon thread termination. See http://dlang.org/class.html#StaticDestructor. [29]See Friend classes at www.cplusplus.com/doc/tutorial/inheritance/.

Since a friend class has what amounts to public visibility to all of the members of the class offering the friendship, it has the undesirable effect of *breaking encapsulation*, because the class no longer is in complete control over its own members. Other classes can

change the values of non-public attributes of the class offering the friendship as well as invoke its non-public methods.

Also, with friendship in effect, it is now more difficult to perform subsequent maintenance and refactoring on the class offering the friendship because now it requires more careful consideration when making changes to protected and private members. For instance, we may change the definition of a private attribute and change all of the references to it within the class and think we have completed the task, but now we may have rendered a friend class syntactically incorrect for those statements that access the private attribute in statements that relied on a specific type of definition. This also applies to the signature of private methods, which also could render friend classes invoking those methods syntactically incorrect for a variety of reasons, such as the wrong type used with a parameter, or is now missing a parameter that has not been marked optional.

Some in the object-oriented community repudiate the use of friendship while others advise using friendship with caution.

Considerations for Using Encapsulation Effectively

So, how do we go about deciding what to include in an encapsulation unit? Here are some guidelines:

- **Encapsulate repetition.**

  This is a technique also used in procedural programming: write the code once and call it from multiple locations as necessary. This can be applied at the method encapsulation unit level, where a method of a class is invoked by other methods of the same class. It also can be applied at the class encapsulation unit level, where one class will provide the repetitive activities to be performed at the request of other classes.

- **Encapsulate complexity.**

  This is another technique used in procedural programming: write particularly long and complex algorithms or processing sequences in their own subroutines.[30] As with repetition, this also can be applied at the method encapsulation unit level, where a method of a class is invoked by another method of the same class, as well as at the class encapsulation unit level, where one class will provide the complex processing to be made available to other classes.

- **Encapsulate what is likely to vary.**

  When it is known the code contains processes likely to change in the future, separate these processes into their own encapsulation units. By *vary* we do not mean applying changes to code for the purpose of fixing bugs, but changes to code because the user is asking for new features and capabilities. For instance, a pizza parlor might have software to handle online customer orders. The boilerplate code handling such things as estimating time to deliver the order, collecting payment, and printing receipts probably is unlikely to change often, but the various menu entries, toppings available, item prices, and promotional campaigns are likely to change to keep up with customer demands.

[30] I've lost count of the number of times I have encountered ABAP subroutines containing hundreds of lines of code where the execution of two or three long and complex algorithms are made mutually exclusive based on some simple logical condition. Instead of moving the algorithms to their own subroutines, the authors chose to leave them embedded in the conditional logic found in the subroutine. In cases like these, the conditional logic becomes lost amongst the hundreds of lines of algorithms.

- **Encapsulation units should have only a single responsibility.**

  Restricting encapsulation units to only a single responsibility is to comply with what is known as the *Single Responsibility Principle,*[31] a term coined by Robert C. Martin, who regards a *responsibility* as a *reason to make a modification*. This can be applied at the method encapsulation unit level, where, for instance, a class handling printed output will have one method to facilitate setting print parameters and another method to facilitate issuing the request for printing. If a single method were to facilitate both of these requirements, then a change to the way the method accommodates print parameters and a change to the way a request is made for printing would constitute two different reasons for changing the same method. This can also be applied at the class encapsulation unit level, where, for

instance, there is one class to facilitate printing content and another class to facilitate displaying content, despite that both classes handle output of content.

ABAP Language Support for Encapsulation

The object-oriented extensions to the ABAP language accommodate encapsulation in the following ways:

- By supporting the concept of a class with attributes and behaviors (methods)
- By providing a way to assign a visibility level to class members
- By facilitating the definition and use of method variables
- By supporting both instance and static members for classes
- By supporting instance classes, static classes, and hybrid classes
- By supporting both instance and static constructors for classes
- By enabling a class to offer friendship to other classes

Unlike most other object-oriented languages, the syntax for defining a class in ABAP separates its definition from its implementation. The definition component must precede the implementation component. This is achieved through these complementary class constructs:

class *class_name* definition [*options*].
o o o endclass. class *class_name* implementation.
o o o endclass.

Attributes and method signatures are specified within *visibility sections* appearing in the definition component. Each visibility section begins with the section name and ends with the definition of the next visibility section name or with the endclass scope terminator. Those members defined within a visibility section are assigned that corresponding visibility.

[31]See http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod.

Listing 3-2 shows an example of the ABAP syntax for a class describing a car.

Listing 3-2. ABAP Code for the Car Class

```
class car definition.
  public section.
  methods : get_year exporting year type string , set_year importing year type string . private
  section.
    data : model_year type string
                                    . endclass. class car
implementation.
  method get_year.
          year = model_year. endmethod. method set_year.
          model_year = year. endmethod. endclass.
```

Here we see both a public and a private visibility section in the definition component. The public section defines two methods and their signatures: get_year and set_year. The private section defines one attribute: model_year. The implementation component contains the implementation for each of the two public methods defined in the definition component.

In this example, the only statements with which an experienced procedural ABAP programmer would be expected to be familiar are the data statement defining the attribute model_year and the assignment statements in the implementations for each of the methods. Everything else is new with object-oriented ABAP.

New, perhaps, but not entirely unfamiliar. The method-endmethod construct appearing in the implementation component is functionally identical to the subroutines defined in procedural ABAP using the form-endform construct. They differ only in syntax; the procedural form-endform construct accommodates a signature on the form statement itself, whereas the object-oriented method-endmethod construct has its signature provided by its counterpart *methods* statement in the class definition component. Indeed, the concepts associated with defining and using local variables apply equally to both procedural subroutines and object-oriented methods.

The example in Listing 3-2 shows a class where all of its members belong to the instance realm. Accordingly, we would need to create an instance of this class in order to access these members. This is done via the *create object* statement:

```
report.
  o o o
          data : rental_car type ref to car. o o o
    create object rental_car.
```

Here we have defined a data field, rental_car, which defines an object reference variable to an object whose type is class car. Upon encountering the create object statement, an object of class car is created, somewhere in storage, by the runtime environment, and a reference to this storage is placed into the object reference variable rental_car. Once the object is created, it is through the object reference variable that we can access the object:

```
call method rental_car->set_year
  exporting year = '2014'.
```

The call method statement initiates access to a method of a class object through the object reference variable. In the preceding example statement, rental_car is the reference variable providing access to the car object, set_year is the name of a public method defined for the car object, and year is a parameter specified in the signature for the public method set_year. The -> symbol separating the name of the object reference variable from the name of the method is known as the *object component selector* and is used to access instance members of an object. There are variations on the statement to invoke methods of objects which are not covered here.[32]

An alternative to defining the car class, as shown in Listing 3-2, is to define the class such that all of its members belong to the static realm. In ABAP, the qualifier *class-,* prefixed to the data and methods statements in the definition component of the class, as shown highlighted in the Listing 3-3, consigns these members to the static realm.

Listing 3-3. Static Definition for the Car Class

```
class car definition.
  public section.
  class-methods: get_year exporting year type string , set_year importing year type string .
  private section.
    class-data : model_year type string
                              . endclass. class car
implementation.
  method get_year.
      year = model_year. endmethod. method set_year.
      model_year = year. endmethod. endclass.
```

Accordingly, the definition component of a class accommodates assigning both member visibility and member realm. Note that this now relegates the car class as a static encapsulation unit, one for which multiple instances are not possible and for which the create object statement is not applicable. A class defined this way, a static class, is one that is immediately available for use in a program (it does not require instantiation) and its behaviors also can be accessed by a call method statement, but the syntax of the call method statement differs in two significant aspects:

- Members of the class are accessed simply via the name of the class instead of through an object reference variable.

[32]Refer to Horst Keller and Sascha Krűger, *ABAP Objects: ABAP Programming in SAP Netweaver*, 2nd edition, Galileo Press, 2007.

• In place of the object component selector, a *class component selector* (=>) is used to access the static members of the class.

Here is the same call method statement we saw before, altered accordingly for accessing a member of a static class, with differences highlighted:

```
call method car=>set_year
  exporting year = '2014'.
```

It is easy for us to see that the example in Listing 3-3 defines a static class; every one of its members is relegated to the static realm, with its single attribute defined using class-data and each of its two behaviors defined using class-methods. If we were to have even one member defined to the instance realm, then the class could no longer be considered a static class. Once a class has a significant number of members defined for it, determining whether or not it is a static class becomes more difficult through the process of checking whether or not there is at least one instance member. Imagine a class with over 100 attributes and more than that many methods[33]. It would be easy to miss a single instance member buried among such a large class definition, and a few years later, when we find ourselves maintaining this class, we long ago would have forgotten whether or not we had defined it as a static class. A way to determine this instantly is to apply some optional qualifiers on the class definition statement:

```
class class_name definition abstract final.
```

Here we see the additional qualifiers called abstract and final. Abstract indicates that the class cannot be instantiated. Final indicates that the class cannot have any subclasses[34]. Together they insure that there can be no instantiations of the class. Applying these two qualifiers to a class intended to function as a static class is an easy way both to guarantee as well as to document that the class is indeed a static class. It also alleviates the programmer from the tedious and potentially inaccurate process of visually applying a full body scan of the class definition only to arrive at the same conclusion.

The ABAP language provides for both instance and static constructors, as shown highlighted in Listing 3-4 of the car class, which has both instance and static members.

Listing 3-4. Car Class with Both Static and Instance Members

```
class car definition.
  public section.
    class-methods: class_constructor
                 , get_next_serial_number
                   exporting next_serial_number type i . methods : constructor
  , get_year exporting year type string , set_year importing year type string . private section.
    class-data : last_used_serial_number
                         type i . data : serial_number type i
                 , model_year type string .
```

[33]Although such a class can exist, it represents a class that should be divided into multiple smaller classes. [34]I will cover subclasses later in the book.


```
endclass. class car implementation.
  method class_constructor.
    last_used_serial_number = 1000. endmethod. method constructor.
  call method get_next_serial_number
    importing
      next_serial_number = serial_number. endmethod. method
  get_next_serial_number.
  add 01 to last_used_serial_number. serial_number = last_used_serial_number. endmethod. method
```

```
get_year.
    year = model_year. endmethod. method set_year.
    model_year = year. endmethod. endclass.
```

As illustrated in Listing 3-4, in addition to being defined to contain only instance members or only static members, a class can also be defined as a hybrid, containing a mix of both static and instance members.

Local method variables are defined within the bounding method and endmethod scope terminators, similar to how they would be defined between the procedural form and endform scope terminators:

```
method validate_registration.
  data : is_registered type abap_bool.
                                    o o o if licence_plate is not initial.
    is_registered = abap_true. endif.
                        o o o endmethod.
```

When specified for a class, ABAP requires the programmer to use the names constructor and class_ constructor for the instance and static constructors, respectively. The static constructor appears on a class- methods statement in the public visibility section and must have no signature. The instance constructor appears on a methods statement in an appropriate visibility section[35] and may have a signature.

In the example, the presence of a static constructor will cause the corresponding method implementation to be invoked when the class is first accessed. As shown in Listing 3-4, the method class_ constructor will set the static attribute last_used_serial_number to the value 1000. This simply initializes this field to a starting value before any instances of the class are created. The static constructor should be used for such setup tasks associated with its static attributes.

[35]There was a time when ABAP required the instance constructor to be defined only in the public visibility section. This no longer is the case. Refer to the ABAP language documentation for more detail.

Meanwhile, the presence of an instance constructor will cause the corresponding method implementation to be invoked each time a new instance of the class is created. As shown in Listing 3-4, the constructor method will invoke method get_next_serial_number of class car to receive the serial number for the new car instance. Notice here that method get_next_serial_number is defined as a static method, and it references the static attribute last_used_serial_number. After incrementing the value of this static attribute, method get_next_serial_number returns the updated value to the caller via the method signature. Accordingly, because static attributes are shared across all instances of a class, each newly instantiated car object will get a unique serial number since the value used for the previous car instance will be left behind to be incremented for the next one.

Since ABAP provides no explicit support for destructors, the way to destroy an object that no longer is required, and as a consequence release the storage it occupies, is simply to clear the value in the corresponding object reference variable. The ABAP runtime environment automatically keeps track of all the active references to an object, and when it detects that an object no longer has any active references, it marks the object for *garbage collection*, a storage optimization feature of the runtime environment.

Finally, the ABAP language provides for a class to offer friendship to other classes by naming them on the class definition statement in a friends clause:

```
class car definition [global | local] friends <friend1 [friend2 ...] >.
```

Encapsulation Units in ABAP

Since the concept of the encapsulation unit is not new to ABAP, perhaps it would be helpful to distinguish between those

encapsulation units typically associated with procedural (classic) ABAP and those that are new with object-oriented ABAP. Table 3-3 shows on the left those encapsulation units available to procedural programming and on the right those new with object-oriented programming, along with the ABAP statements to access the encapsulation unit.

Table 3-3. *Comparison of Procedural and Object-Oriented Encapsulation Units*

Procedural (classic) encapsulation units	Object-oriented encapsulation units
Report/program (SUBMIT)	Class
Subroutines within report (PERFORM)	
Method (CALL METHOD)	Function group (CALL FUNCTION)
Transaction/dialog (CALL TRANSACTION)	
Screens (CALL SCREEN) (No direct screen processing capability)[36]	

It should be noted that most of the encapsulation units associated with procedural ABAP are available for use by the encapsulation units associated with object-oriented ABAP, and vice versa. That is, a method of a class may contain a PERFORM (local class only) or CALL FUNCTION statement to a procedural encapsulation unit, and a classic ABAP subroutine or function module may contain a CALL METHOD statement to an object-oriented encapsulation unit.

[36]Screen processing within object-oriented ABAP is provided through WebDynpro and its successors or through the use of function groups that encapsulate screen definitions and their processing.

Managing Class Encapsulation Units in the ABAP Repository

The ABAP source code repository facilitates retaining objects containing the ABAP code describing classes using two different designations. One designation makes the described class available to all other objects contained within the ABAP source code repository; the other designation limits the availability of the described class exclusively to the compilation unit with which it is used. The ABAP developer chooses which designation to use based on whether the class has the potential to be used in multiple settings. Accordingly, each class definition will be designated either a *global* class or a *local* class.

A global class is created and maintained via the Class Builder (transaction SE24). Once defined and activated, the class becomes available for use by any other object in the ABAP repository. Using the Class Builder to build a class is analogous to using the Function Builder (transaction SE37) to build a function of a function group. Both the Class Builder and Function Builder guide the developer to define the class or function module using a form-based approach, through which the developer is presented a set of tabs enabling the assignment of some aspect of the class or function module. The Class Builder also has the option of a source code-based approach, enabling the developer to create and maintain the entire class definition on a single editor screen, and provides the capability to toggle between the form-based and source code-based approaches as desired.

A local class is created and maintained via one of the standard ABAP source code editors (e.g., SE38, SE80). Local classes can coexist in an object with other non-class code. A typical scenario is one where local classes are embedded in a classic procedural report program, either in the same object as the procedural code or in an INCLUDE object that is included along with the other components to compose a complete compilation unit. Once defined and activated, the local class becomes available to all other components in the same compilation unit. That is, when included with a report, the local class can be used by all the procedural components defined in the report, but it is unavailable to entities defined outside the boundaries of the report. Indeed, local classes may be defined within function groups and global classes, and when they are, their availability is restricted to the components within the function group or global class compilation unit. A common use of local classes is to define classes to facilitate ABAP unit testing, the automated unit testing feature provided as part of the ABAP workbench, since the ABAP unit feature is initiated only through the methods of local classes.

Whether using the global class source code-based editor or one of the standard ABAP editors to define a local class, in both cases the definition portion of the class must physically precede its implementation portion. When multiple local classes are defined in a single component, it is common for these two complementary portions of the class to be adjacent to each other. There are situations, however, where the definition and implementation portions defining a single class will be separated from each other by the definition portions of one or more other local classes. This usually becomes necessary when there is an interdependency between multiple local classes, and is a consequence of the ABAP compiler being a single- pass compiler[37]. For instance, a local physician class holds a reference to a local patient class, and the local patient class holds a reference to the local physician class; the local classes are mutually dependent. In cases such as this, it is necessary for the definition portion of both local classes to precede the implementation portion of either class.[38] This anomaly does not apply to global classes since only a single global class may be contained within a global class compilation unit.

Orienting Ourselves After Having Traversed Encapsulation

So, we have traveled some distance along the path from Procedureton to Objectropolis, and now, having completed our traversal through the object-oriented district known as Encapsulation, we are familiar with its principles and can now speak the language spoken by the residents in this district. Since this is such a new

[37]See www.uobabylon.edu.iq/eprints/publication_10_847_344.pdf. [38]We will see an example of where this becomes necessary in the chapter on the State design pattern (Chapter 22).

40place for us, it would be helpful to orient ourselves and determine where we are in relation to some other place with which we are more familiar. Accordingly, let's determine how far away we are from a place we know so well in ABAP procedural programming: Function Groups. We have learned that static classes have many similarities with function groups, but we also know that classes offer other capabilities beyond just static classes.

Refer to the chart in Appendix A, illustrating the comparison between function groups and classes on how each one facilitates the capabilities of the principles of object-oriented programming. The first 10 rows show how these two programming formats support the principles of Encapsulation. Although function groups, a place more familiar to us than classes, do not support all of these principles, there is enough common ground for us to conclude we are not that far away from the district where we would find function groups to be prominent; the terrain is similar, but we find geographical features in the object-oriented landscape of encapsulation that we do not find in function groups.

Summary

In this chapter, we became more familiar with the object-oriented concept of Encapsulation, and that it facilitates adherence to the design principle known as Separation of Concerns. We now know about visibility, how it is applied to components of classes using words representing a scale of gradations from least to most visibility, and why this is such an important concept to understand. We also learned that members of classes can exist in one of two realms:

- Static realm
- Instance realm

We also learned about encapsulation units and that classes can be defined as purely static encapsulation units, as purely instance encapsulation units, or as a hybrid combination of both static and instance encapsulation units. The concepts of constructors and destructors were introduced, which also have an aspect of applying either to the static realm or the instance realm. We learned about friendship, a concept applicable only to some object-oriented environments, and that using it breaks encapsulation. We also learned some considerations for using the principle of encapsulation effectively:

- Encapsulate repetition.
- Encapsulate complexity.
- Encapsulate what is likely to vary.
- Encapsulation units should have only a single responsibility.

Encapsulation Exercises

Refer to Chapter 3 of the functional and technical requirements documentation (see Appendix B) for the accompanying ABAP exercise programs associated with this chapter. Take a break from reading the book at this point to reinforce what you have read by changing and executing the corresponding exercise programs. The exercise programs associated with this chapter are those in the 101 series: ZOOT101A through ZOOT101E.

CHAPTER 4

Abstraction

The next stop on our journey to Objectropolis takes us from Encapsulation to a place called Abstraction. The word abstract means

> "Considered apart from concrete existence ... thought of or stated without reference to a specific instance."[1]

It is at this point where we depart familiar procedural territory. Some definitions for abstraction in object-oriented programming describe it as the representation of real-world entities.

> "The objects in an object-oriented system are often intended to correspond directly to entities in the 'real world.'"[2]

This certainly is an important aspect of abstraction, but, as we shall see, it is not the only aspect. Erich Gamma and his colleagues offer the following advisory:

> "[O]bject-oriented designs often end up with classes that have no counterparts in the real world. ... Strict modeling of the real world leads to a system that reflects today's realities but not necessarily tomorrow's. The abstractions that emerge during design are key to making a design flexible."[3]

Abstract Art

One good way to envision abstraction is to consider how a cartoon image conveys information about an entity. If I were to try to describe a car to you, I might draw a cartoon picture of a car, applying to it only those details relevant to the discussion. The cartoon would be an abstraction of a car. It would not represent any particular car, but merely serve as a generalization for any car simply to facilitate the discussion I wanted to have about cars. The discussion might be about car windshields. If at some point in the discussion I were to tell everyone to go outside into the parking lot to find a car with the type of windshield I was discussing, it is unlikely that everyone would gravitate to the same car. This is because my cartoon image of the car would not exhibit enough detail for everyone to identify the same car; it is simply an abstraction for any car possessing the level of detail I provided for it.

[1] *American Heritage Dictionary of the English Language*, 4th edition, Houghton Mifflin Company, 2000, p. 8.

[2] www.prenhall.com/divisions/esm/app/kafura/secure/chapter1/html/1.2_abstraction.htm [3] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software,* Addison-Wesley, 1994, p. 11.

41 © James E. McDonough 2017 J. E. McDonough, *Object-Oriented Design with ABAP*, DOI 10.1007/978-1-4842-2838-8_4

Another way to envision abstraction is to consider how authors of romance novels develop their characters. A typical trio of characters is hero, villain, and damsel in distress. Each of these characters is developed as the novel progresses, but the author provides only

enough information about them to support the story, and we use our imagination to fill in the missing details. Accordingly, each reader of the novel creates their own unique images of each character, perhaps drawing upon similarities with themselves or people they know, enjoying the novel even though each character is an abstraction for a person and not intended to represent any particular person.

Representing Real-World Entities

Car, bank account, and *weather forecast* are three examples of real-world entities. We can represent these entities in an object-oriented program by defining each one as a class: a *car* class, a *bank account* class, and a *weather forecast* class. These classes represent abstractions for their real-world counterparts.

Each of these classes can include definitions for its attributes. The *car* class can include attributes for make, model, year, serial number, average miles per gallon, speed, direction, remaining fuel quantity, etc. The *bank account* class can include attributes for account number, type of account, current balance, minimum balance, interest rate, overdraft penalty, etc. The *weather forecast* can include attributes for expected high temperature for today; expected overnight low temperature; chance of precipitation for today, tomorrow, the next day; etc.

Each of these classes can also include definitions for behaviors that act upon its attributes. The car class can include behaviors for start, stop, turn and accelerate, each having an effect upon the attributes retaining information about fuel consumption, movement, and direction of travel. Likewise, the bank account class can include behaviors for deposit, withdraw, and calculate interest, each having an effect upon the attributes retaining information about current balance and overdraft penalty. Similarly, the weather forecast class can include behaviors for both updating and disclosing its own attribute values for tracking the weather.

So, how is this different from the procedural programming style with which we are so familiar? While procedural programs are primarily designed to address a specific processing requirement, classes are primarily designed to manage information. The car class manages information about a car, but has no capacity to manage any other type of information, such as how many other cars have ever been owned by the driver, information that, while it might be important to us, has nothing to do with the car class.

Let’s see this through an example. A manufacturer of replacement vehicle mufflers needs to track the quantity and location of finished goods in transit between its manufacturing plant in Cleveland, Ohio and its three regional warehouses in Linden, New Jersey; Kansas City, Missouri; and Hayward, California, each of which receive their inventory of mufflers via shipping containers moving by both rail and truck. A report is required to show the various shipping containers in use, indicating the container id, which of them are being loaded, which are being unloaded, and for those in transit, which are on trucks and the name of the trucking company, which are on rail cars and the name of the railroad company, where each one is located, the warehouse to receive the shipping container, and its estimated time of arrival. The format of the report is similar to the output shown in Table 4-1.

Table 4-1. *Example Output for Report*

Container id Destination warehouse Current location Status Carrier ETA

HGF107588 Linden, NJ Linden, NJ Unloading YKZ503401 Linden, NJ Altoona, PA On rail car Norfolk Southern Tomorrow

BNX969443 Linden, NJ Newark, NJ On truck RNX Logistics Today BNX969447 Kansas City, MO Indianapolis, IN On rail car BNSF Tomorrow

BNX963850 Kansas City, MO Kansas City, MO Unloading

YKZ500067 Kansas City, MO Kansas City, MO On truck Harris Transport Today HGF109905 Hayward, CA Salt Lake City, UT On rail car BNSF Two days

YKZ501991 Hayward, CA Chicago, IL On rail car BNSF Four days HGF103556 Hayward, CA Oakland, CA On truck Western Express Today

HGF102002 Hayward, CA Cleveland, OH Loading Five days

When writing this program using the procedural style we might define a global variable as a table to keep track of all the information required for shipping containers as well as other variables to record information about the three warehouses, the trucking companies, the railroad companies, and the manufacturing plant. It may be composed of various subroutines for resolving the information about each shipping

container and presenting this information in a report. Such a program is designed like many other procedural programs: it focuses on the processing requirement.

When writing this program using the object-oriented style we might define classes for shipping container, warehouse, manufacturing plant, truck, trucking company, rail car, railroad company, and report. Each of these classes is an abstraction of its real-world counterpart and has its respective attributes and behaviors. The report class produces the report while the shipping container class keeps track of information about a single shipping container. Each class retains information relevant to the abstraction it represents *and no other information*. Such a program is designed like many other object-oriented programs; it focuses on data and how the data is organized and managed.

Comparing the two programming styles illustrated above, we see that the procedural program contains information about every contributing entity all combined into one single program. There is no discernible separation between a shipping container and the truck that might be carrying it, since information for both is retained in the same global table. By contrast, the object-oriented program manages the information by segregating it into distinct classes, where each class manages only the information relevant to it. The shipping container retains the container id and its quantity, but knows nothing about the truck or rail car transporting it.

Class Cohesion

The degree to which members defined for a class are relevant to each other is a measure of the *cohesion* of a class.[4] A class containing only those members relevant to its corresponding abstraction reflects high cohesion among its members, while a class containing members that have little relevancy to one another reflects low cohesion. Classes should be defined in a way that offers high cohesion among its members.

[4]See www.aivosto.com/project/help/pm-oo-cohesion.html.

In the previous example for the manufacturer of replacement vehicle mufflers, a variety of classes were proposed to facilitate the object-oriented programming style. The class proposed to represent an abstraction for a shipping container offers high cohesion when its

attributes and behaviors all relate to each other. For example, its attributes might be container id, color, height, width, depth, storage capacity, tare weight, maximum gross weight, net weight, quantity of loaded items, and current security tag number; and its behaviors might include, load, unload, get_gross_weight, and apply_security_tag. All of them offer the class high cohesion since they all relate to a shipping container. If we were to include for this class an attribute to hold the name of the trucking company moving the container, we would cause the class to offer lower cohesion since this attribute has little to do with a shipping container, but instead is relevant only when a truck is transporting it. It would represent an example of an attribute that would not be applicable when the shipping container is being transported by train. Worse, to include in the shipping container class an attribute to represent the number of employees working for the manufacturer of mufflers the previous year would further diminish the cohesiveness of the class since this information is completely irrelevant to shipping containers.

Reusable Components

The report on shipping containers in use illustrated in Table 4-1 might cause us to consider whether its construction using the object-oriented programming model is worth the effort. An argument could be made that we might be able to complete it faster using the procedural programming model. This is a good point to ponder, but we should consider that rarely do we ever encounter business organizations where only one type of report provides sufficient information about business entities such as shipping containers. Instead, we are likely to find ourselves given requirements later to write a report on those shipping containers that are out of circulation due to having sustained damage or in the process of being repaired and refurbished, on the current age and expected lifespan of active shipping containers, and a host of other reports related to shipping containers.

With this in mind, it makes sense to use the object-oriented programming model so that the shipping container class we define for the in-use report can also be reused with these other reports. Not only that, but during the maintenance of each report it becomes easier to identify the discrete component having responsibility for managing the information we seek to convey in the report. Eventually we will have compiled a comprehensive library of reusable components, each component representing a ready-to-use abstraction for an entity that needs to participate in the next new report program that becomes necessary.

Establishing a Level of Abstraction

An abstraction level provides aspects for both detail and scope of an entity and is a measure of the precision we choose to assign it. This provides us with the ability to manage the complexity of an entity through a more practical perspective, allowing us to ignore details that may not be relevant to the way in which we need to use the entity. We see more detail at lower levels of abstraction and fewer details at higher levels. Similarly, we are afforded wider scope at higher levels of abstraction and narrower scope at lower levels. The two aspects of detail and scope have an inverse relationship with each other; as one increases, the other decreases.

For example, we might find ourselves standing on a patio adjoining the back of a house. The patio is composed of common red bricks measuring about 20cm long x 10cm wide x 5cm deep and they arranged in a herringbone pattern, such as presented in Figure 4-1.

When we are standing on the patio, our eyes are about five feet above it. At this level of abstraction, when we look down at the patio, we can see distinct red bricks in a herringbone pattern. If we were to raise our abstraction level to 50 feet above the patio, we now might see red blocks arranged in a herringbone pattern; however, we can no longer identify that they are bricks. Raise the abstraction level to 500 feet and we might see that the patio is red, but we can no longer tell there is a herringbone pattern. At 5,000 feet, we can no longer tell that the patio is red. At 50,000 feet, we can no longer see the patio.

Similarly, at the five-foot abstraction level, our scope is limited to only the portion of the patio on which we are standing. At the 50-foot abstraction level, our scope widens and we can see the entire patio, the roof of the adjoining house, and some of the yard surrounding it. At 500 feet, our scope widens still more and we also can see some of the neighbors' houses and their patios. At 5,000 feet, we can see the entire town. At 50,000 feet, we can see the entire county.

Table 4-2 organizes these different vertical perceptions of patios, enabling us to compare the corresponding scope and level of detail as we change our level of abstraction.

Figure 4-1. *Herringbone pattern*

Table 4-2. *Relationship between Scope and Level of Detail*

Altitude (in feet)	Scope	Level of Detail
50,000	We can see the entire county.	We cannot discern any patios.
5,000	We can see the entire town.	We can discern patios but

cannot resolve
their color, pattern, or material.
500 We can see a few houses in the
neighborhood.

45 We can discern patios and their color but cannot resolve their pattern or material. 50 We can see this entire patio, the roof of the adjoining house, and some of its yard.
We can discern the entire patio color and pattern but cannot resolve its material.
5 We can see a portion of this patio. We can discern for a section of the patio its
color, pattern, and material.

Let's put this into terms of classes and their members. Suppose we have three classes representing three different abstraction levels:

1. Enclosed shape
2. Enclosed shape with at least one angle
3. Rectangle

We have arranged these classes in Figure 4-2 to illustrate the vertical nature of the relationship between these classes, with each class providing less scope and more detail as we descend from the top of the chart to the bottom.

Now we will ask three people selected at random (Boris, Natasha, and Sherman) to provide two-dimensional drawings for each class, instructing them to "use your imagination" but stay within the constraints as described by the name of the class. As shown in Figure 4-3, none of these people provided the same drawing, but all their drawings conform to the criteria described by the name of each class.

Figure 4-2. *Classes arranged where scope is greatest at top and level of detail is greatest at bottom*

Let's define some members for these classes. Area and perimeter are two attributes we should be able to apply to any two-dimensional enclosed shape, so for the *Enclosed shape* class we'll define these two attributes and two corresponding methods, getArea and getPerimeter.

The *Enclosed shape with at least one angle* class can accommodate all of the members defined for *Enclosed shape*, but this class is at a lower level of abstraction, providing more detail and less scope. Its scope is limited to those drawings with at least one angle, and so cannot accommodate the drawings for the *Enclosed shape* class provided by Boris and Natasha. However, because its level of abstraction includes the detail "at least one angle," we can define for it additional members to reflect this level of detail. Let's define for it an attribute called smallest angle and a corresponding method getSmallestAngle.

Similarly, the *Rectangle* class can accommodate all of the members defined both for *Enclosed shape* and for *Enclosed shape with at least one angle*. Since it has a lower level of abstraction than *Enclosed shape with at least one angle*, let's capitalize on this and provide it with an attribute called hypotenuse and a corresponding method named getHypotenuse. Here we specify for *Rectangle* more detail, but we also know that its scope does not permit it to accommodate any of the drawings for the *Enclosed shape* or *Enclosed shape with at least one angle* classes provided by both Boris and Natasha.

Sherman provided the same drawing for all three classes; however, we need to recognize that even though a rectangle was provided for the *Enclosed shape* class, this class has neither a hypotenuse attribute nor a getHypotenuse method. The *Enclosed shape* class can determine only the area and the perimeter of a rectangle since this is the extent of information its level of abstraction allows it to see for the drawing it holds.

Figure 4-3. *Drawings provided by Boris, Natasha, and Sherman*

CHAPTER 4 ■ ABSTRACTION

Table 4-3 summarizes our three classes and their respective members.

Table 4-3. *Summary of the Members of the Three Classes*

Classes	Enclosed shape	Enclosed shape with at least one angle	Rectangle
Attributes	area	area perimeter	perimeter smallest angle
	Notice that, by virtue of its additional members, each successive class represents a specialization of the one preceding it. Another way of looking at this is to consider that each preceding class represents a generalization of the one following it. As we move from a more general to more specific class, we can accommodate a smaller set of applicable shapes but we gain detail about each one (additional members of the class). Similarly, as we move from a more specific to more general class, we can accommodate a larger set of applicable shapes but we lose detail about each one (fewer members of the class). The set of shapes that can be considered a *Rectangle* is a subset of the larger set of shapes that can be considered an *Enclosed shape with at least one angle*. Likewise, the set of shapes that can be considered an *Enclosed shape with at least one angle* is a subset of the larger set of shapes that can be considered an *Enclosed shape*. We can correlate this set of classes and their respective members with the perspectives illustrated by the patio metaphor noted above. Let's imagine the *Rectangle* class offers us a perspective on shapes from 50 feet away, the *Enclosed shape with at least one angle* offers a perspective from 500 feet, and *Enclosed shape* offers a perspective from 5,000 feet. At 50 feet, we can see via *Rectangle* that a shape has an area, perimeter, smallest angle, and hypotenuse. At 500 feet, we can see via *Enclosed shape with at least one angle* that a shape has an area, perimeter, and smallest angle, but we can no longer see whether it has a hypotenuse; this perspective offers us fewer details. However, the shapes we can see are now no longer limited to rectangles; it offers us a wider scope of applicable shapes. Similarly, at 5,000 feet, we can see via *Enclosed shape* that a shape has an area and a perimeter, but we can no longer see whether it has either a smallest angle or a hypotenuse, offering us even fewer details. However, the shapes we can see are now no longer limited to those with at least one angle, offering us a wider scope of shapes. Suppose we find ourselves with the requirement to create a report that shows the area and perimeter of a set of shapes. Which of the classes shown in Table 4-3 should we choose to facilitate this? The most appropriate class would be the *Enclosed shape*. It provides the necessary level of detail to reflect the area and perimeter of any shape we might encounter. Whereas one of the shapes our report program may encounter could be a rectangle, and there is a *Rectangle* class among the choices above, the report does not need to use any attribute or behavior applicable specifically to rectangles. The smallest angle and hypotenuse attributes offered by the *Rectangle* class represent details beyond those required for a report showing simple area and perimeters of shapes. Accordingly, our report does not need to know that it is working with a rectangle to produce the information required; the *Enclosed shape* class is a sufficient level of abstraction to handle any rectangle shapes that would participate in the report.	area perimeter smallest angle hypotenuse
Behaviors	getArea	getPerimeter	getArea getPerimeter getSmallestAngle
	getArea getPerimeter getSmallestAngle getHypotenuse		

Multiple Instantiations

Once we have established its level of abstraction, we now have a class that can act as a proxy for its real-world counterpart, offering only the level of detail applicable to the entity it represents. We also have a pattern for an entity from which multiple instances of the entity may be created. There is a word used to describe each instance of an abstraction: *object*.

This is the very foundation for the term "object-oriented" in describing both the language and the design concepts associated with object-oriented programming.

Classes define these entities. Objects are their instantiation. In this context, object means the same thing as instance of a class. In a single program there is virtually no limit to the number of objects that can be instantiated for a class. Once an object is instantiated, it becomes a concrete representation of the abstraction, because although objects of the same class all have the same attributes and behaviors according to the class definition, each object has its own unique attribute values.

While abstraction represents a conceptualization for some entity, a concretion denotes a tangible instance. Table 4-4 gives some other examples of the relationship between abstraction and concretion.

Table 4-4. *Examples of the Relationship Between Abstraction and Concretion*

Abstraction	Concretion
General	Specific
Potential	Actual
Conceptual	Tangible
Product mold	Product Cookie cutter Cookie
Car	My first car
Book	This copy of this book
Building	Carnegie Hall
Road	Pennsylvania Turnpike
River	Rhine
Mountain range	Andes
Continent	Antarctica
Planet	Jupiter
Star	Rigel
Galaxy	Andromeda

Relationship Between Abstraction and Encapsulation

The principles of abstraction and encapsulation have a complementary relationship. A class is an abstraction of an entity and the class encapsulates the attributes and behaviors relevant to the corresponding level of abstraction represented by the class.

Steve McConnell, author of *Code Complete*, offers an interesting perspective on this. After summarizing the concept of abstraction with the following paragraph:

> Abstraction is the ability to engage with a concept while safely ignoring some of its details – handling different details at different levels. ... If you refer to an object as a "house" rather than a combination of glass, wood and nails, you're making an abstraction. If you refer to a collection of houses as a "town," you're making another abstraction.[5]

he describes the relationship between abstraction and encapsulation with this exquisite passage:

> Encapsulation picks up where abstraction leaves off. Abstraction says "You're allowed to look at an object at a high level of detail." Encapsulation says, "Furthermore, you aren't allowed to look at an object at any other level of detail."[6]

driving the concept home with this explanation:

> Encapsulation says that, not only are you allowed to take a simpler view of a complex concept, you are *not* allowed to look at any of the details of the complex concept. What you see is what you get – it's all you get![7]

where the simple view of a complex concept is represented by an abstraction.

ABAP Language Support for Abstraction

The object-oriented extensions to the ABAP language accommodate abstraction in the following ways:

- By providing the ability to define representations for real-world entities
- By providing the capability to establish a clear level of abstraction
- By supporting the concept of creating multiple instances of a class

The same statements we saw for encapsulation also enable us to accommodate abstraction, with the construct block

class *class_name* definition [*options*].

o o o endclass.

providing the ability to define representations for real-world entities and establishing clear levels of abstraction through judicious selection of a class name, and also facilitating effective class cohesion by establishing the boundaries for containing the members to be included in the class.

Among the options available on the class definition are the qualifiers *abstract* and *final*. A class may have either or both of these qualifiers, and, as discussed in the section on encapsulation, when both are present it indicates a static class. As described in that section on encapsulation, the qualifier *abstract* insures that the class cannot be instantiated. It is the absence of the *abstract* qualifier that enables instances of the class to be created.

[5]Steve McConnell, *Code Complete*, 2nd edition, Microsoft Press, 2004, p. 89. [6]Ibid, p. 90. [7]Ibid, p. 91.

Another of the class definition qualifiers, the *create* qualifier, controls the creation of instances of the class. The *create* qualifier is followed by a word ascribing an instantiability level to the class and restricts the relationship an entity must have with the class for it to be able to create an instance of the class. The syntax is

... create [public | protected | private] ...

where *create private* means that only the class itself may create instances of the class; *create protected* means that only the class itself and any inheriting subclasses may create instances of the class; and *create public* means that any entity may create instances of the

class. When not explicitly indicated on a class definition statement, *create public* is the default.

Notice the similarities between the words used to assign visibility levels for class members discussed in the section on encapsulation and the instantiability level available with the create qualifier of the class definition statement noted above, where object creation levels also indicate gradations from most visible to least visible. The visibility of the members of a class and the instantiability level assigned to the class are completely independent of each other. However, the same implications class friendship has upon visibility levels of class members also apply to the ability of a class to be instantiated by a friend class; specifically, the instantiability level of a class also is elevated to *create public* for its friends regardless of the actual create level assigned on its class definition statement.

In addition, the ability to create multiple instances of a class is facilitated by the use of the same class name on multiple reference variables and then creating instances of the same class into each one of these, as illustrated in this example:

```
report.
  o o o
    data : rental_car type ref to car
                    , classic_show_car
                        type ref to car , limousine type ref to car , hearse type ref to car .
  o o o
    create object: rental_car
                    , classic_show_car , limousine , hearse .
```

Orienting Ourselves After Having Traversed Abstraction

So, we have made more progress on our journey from Procedureton to Objectropolis, and now, having completed our traversal through the object-oriented district known as Abstraction, we are familiar with its principles and can now converse in the dialect characteristic of the local population. The landscape in Abstraction may have some similarity with that of our home town of Procedureton, but it also offers some new exotic features. Even so, we have become familiar with the peculiarities commonplace in this district and feel confident we can orient ourselves to maneuver effectively over the terrain of Abstraction.