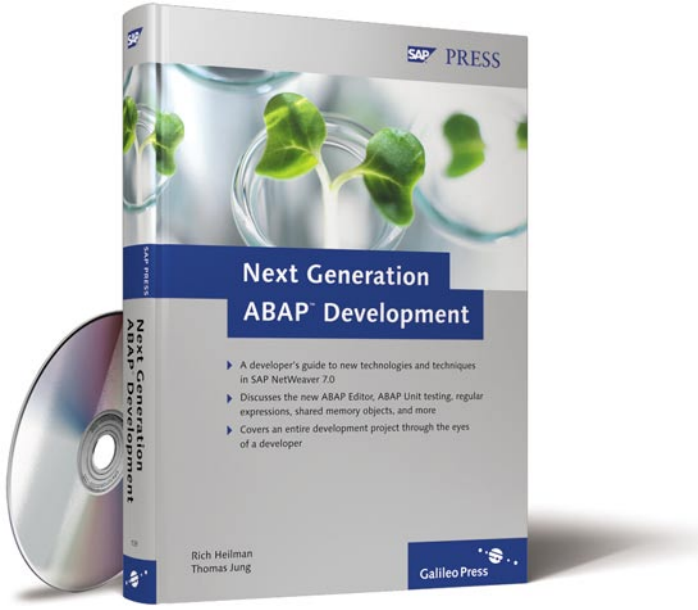
SAP PRESS

Rich Heilman, Thomas Jung

Next Generation ABAP™ Development



www.sap-press.com

Galileo Press

Bonn • Boston

Contents at a Glance

Contents

Introduction

This book represents 20 years of collective experience from real world ABAP Development projects. When setting out to write this book, our challenge was to share what it was like to be part of a cutting edge ABAP development project with our readers.

Ideally, our goal was for each reader to be able to sit down and observe an entire project from start to finish, and therefore learn the integral techniques of modern ABAP development. They would be able to see the latest ABAP technologies in action, in addition to examining the design and development processes used to maximize these technologies. Unfortunately, few developers ever get the opportunity to observe a project in this way. Too often they have to learn on the job, while dealing with unrealistic deadlines. Therefore, our objective was to allow you, the reader, to see and participate in the evolution of such a project in small incremental steps.

For that reason, this book is not your typical programming guide. Instead of focusing on just the technological aspects of developing in ABAP, we will study a fictional project so you can see how a project is developed. Each chapter will represent a phase or layer of this project's development, as well as one or two new key ABAP technologies. If you're interested in delving into these new technologies straightaway, you're welcome to skip to these respective chapters; however, we, the authors, encourage you to read this book in chronological order so that you'll have an opportunity to see the entire narrative of this project play out.

Fictional Project

Throughout the book, we'll be using a single fictional project for our practice scenario. This project takes place at a university, which is a long time SAP customer who runs their business systems on SAP R/3. For over four years, they've been running SAP R/3 4.6C and have used the Finance and Human Resources modules of SAP R/3 extensively, in addition to custom developing many modules of their own.

This university is in the middle of a typical upgrade cycle. They have begun the process of updating their SAP R/3 4.6C system to SAP ERP 6.0 (formerly named mySAP ERP 2005). SAP ERP 6.0 will run on top of SAP NetWeaver — specifically SAP NetWeaver 7.0 (formerly named SAP NetWeaver 2004s). They are also in the process of implementing the SAP NetWeaver Portal, as well as considering using SAP NetWeaver Process Integration (SAP NetWeaver PI — formerly known as SAP NetWeaver Exchange Infrastructure or SAP XI) and SAP NetWeaver Master Data Management (SAP NetWeaver MDM) in the near future.

This university has a small but strong IT team. Our story will focus on Russel, the lead developer of the IT team at this university. Russel has many years of experience in ABAP development to support the university's systems. Like many developers, he reads about the latest ABAP development technologies and techniques, but is somewhat constrained by the release level of the university's R/3 system. Consequently, he feels that his development skills are not up to date; for example, he has done very little ABAP Object-Oriented (ABAP OO) programming and has virtually no web-based development. Still, Russel is quite excited about the future upgrade to SAP ERP 6.0. He sees this as an opportunity to update his skills and learn about the newest ABAP development techniques.

Little does Russel know that he is about to get a crash course in ABAP development on SAP NetWeaver 7.0. In addition to the upgrade activities, the university is just beginning to offer a new distance learning curriculum. Like new offerings at many universities, this distance learning curriculum will offer online versions of many courses for people looking to complete their degrees, or take part in continuing education without disrupting their current career.

In support of this new curriculum, the university realizes that it will need significant new custom development. Their ERP system will house this development and ABAP will be the language in which the system is developed. This project will enable Russel to build the data access, business logic, and user interface aspects of this new system. This project will also be the first time that Russel will build something that entails *enterprise service-oriented architecture* (enterprise SOA).

Please note again that the context for the project that we're going to study throughout this book is fictional. It does not feature an actual university or SAP customer. The characters that we will meet, like Russel, are not real people; however Russel's experiences and reactions to events are based on our

(i.e., the authors) experiences, and hopefully will touch a familiar chord with many of you.

Structure of the Book

The structure of this book reflects the workflow of the development project. The first half of the book focuses on creating the data and application logic layers and then service-enabling them. The second half of the book focuses on creating the user interface layers.

- **Chapter 1: Workbench Tools and Package Hierarchy**
  Before we begin our project, we will review some of the changes and enhancements to the ABAP Workbench. In this chapter, we will look at the new ABAP Editor, the Refactoring Assistant, the new development tools perspectives in transaction SE80, and the new debugger. Lastly, we will create the packages and package hierarchies for the project.
- **Chapter 2: Data Dictionary Objects**
  In this chapter, we will model the data relationships and build the corresponding Data Dictionary objects. We'll study the tools for generating table maintenance, creating lock objects, and utilizing foreign keys. We'll also explore the new technology of strings and binary strings within transparent tables.
- **Chapter 3: Data Persistence Layer**
  In this chapter, we'll build the logic that controls the persistence of application data. We'll start by generating persistent object classes for the underlying data dictionary tables created in Chapter 2. Then, we'll build a set of business object classes to hide the inner technical details of the persistent objects. In addition to the new technology of Persistent Objects, we'll show you how to use ZIP compression on large strings.
- **Chapter 4: Consuming a Web Service**
  Not all project data will originate from one centralized system. For example, in the sample application, some data will be stored in a legacy system and accessed remotely via Web Services. In this chapter, we will examine the process for generating a Web Service proxy object and integrating this proxy into the data persistence layer.
- **Chapter 5: Shared Memory Objects**
  After some analysis, it will become apparent that the sample application has some static data that will be accessed repeatedly. In this chapter, we

will describe how you can provide the best performance by structuring the data access for this type of data into an ABAP Shared Memory Object.

- **Chapter 6: Model Class**
  In this chapter, we will begin to implement the core application logic, which is implemented as a Model Class. This same class will later be used as the business logic layer of all the UI technology examples. This chapter focuses primarily on object-oriented design patterns while introducing techniques for sending email and manipulating XML.
- **Chapter 7: ABAP and SAP NetWeaver Master Data Management**
  This is the first of our "What-If" chapters. Here, we look at an alternative approach to the project where our master data is modeled and stored in SAP NetWeaver Master Data Management, instead of the local Data Dictionary. This chapter will focus on how we would alter the data persistence layer to read this data via the SAP NetWeaver MDM ABAP application programming interface instead of directly from the local database.
- **Chapter 8: ABAP Unit**
  Before building any additional objects on top of the existing application logic, this is a good point in the project to unit test what has been completed. In this chapter, we'll look at the built-in unit test tool, ABAP Unit, and examine how unit test classes can be integrated directly into the model class.
- **Chapter 9: Exposing a Model as a Web Service**
  Not all the logic from the sample model class will be exposed via a user interface. Instead, some of the data was designed to be exposed as a Web Service so that it can be accessible to external systems as well. In this chapter, we'll examine the Inside-Out approach for generating Web Services.
- **Chapter 10: Exposing a Model as a Web Service Using SAP NetWeaver Process Integration**
  This is the second of the two "What-If" chapters. In the previous chapter, we looked at the Inside-Out approach of generating Web Services via remote enabled function modules. In this chapter, we'll look at the world of Enterprise Service Modeling. We'll show you how the same logic could be modeled in SAP NetWeaver Process Integration and then implemented as a server proxy in ABAP using the Outside-In approach.
- **Chapter 11: Classic Dynpro UI/ALV Object Model**
  In this chapter, we turn our attention to user interface logic. In the sample application requirements, there are a group of internal users who are full time SAP GUI users and who need powerful reporting tools. Therefore,

we'll learn how to build a classic Dynpro screen on top of the Model View Controller, which uses the ALV Object Model for its reporting output.

- **Chapter 12: Web Dynpro ABAP**
  Since most of the sample application's users are not SAP GUI users, we'll look at how you can build a Web Dynpro user interface for these users. This chapter will focus on real world Web Dynpro applications that contain multiple component usages, ALV integration, and table popins.
- **Chapter 13: Business Server Pages**
  The next user interface use case is for an Internet-facing application. This user interface needs to be highly customized and stateless for scalability. Therefore, in this chapter, we will use Business Server Pages in order to show the flexibility they provide for highly customized style sheets and AJAX integration.
- **Chapter 14: Adobe Forms**
  Adobe Forms technology offers an interesting paper-like alternative user interface. In this chapter, we'll look at each of the major types of Adobe Forms — print forms, online interactive forms, and offline interactive forms.
- **Chapter 15: SAP NetWeaver Portal**
  Although we have focused on ABAP as the primary development environment until now, it is also important to see how some of the SAP NetWeaver Portal technologies can be used with the best aspects of ABAP. In this chapter, we'll explore how to wrap each of our user interface examples in iViews within the SAP NetWeaver Portal and how portal eventing can be used for cross iView communication. We'll also look at how we can use SAP NetWeaver Visual Composer to build code-free applications that consume ABAP services.
- **Chapter 16: RSS Feed Using an ICF Service Node**
  In this chapter, we'll examine how Internet Communication Framework Service Nodes can be combined with XML processing in ABAP to produce interesting Web 2.0 type projects. As the final example of the book, we'll implement an RSS Feed using these technologies.
- **Chapter 17: Closing**
  In the final chapter, we will look back on the completed project and review the most important points of what has been discussed.

Prerequisites

Whether you are relatively new to ABAP development or an experienced veteran, there is something in this text for everyone. We do, however, assume that the reader is already familiar with the ABAP Workbench and has some development experience in ABAP as of the 4.x release level. We will primarily focus on new techniques and tools that were introduced in the 6.x and higher releases.

The state of the ABAP development environment described in this book is SAP NetWeaver 7.0 SPS10. As SAP ERP 6.0 has been announced to be the primary release of ERP for customers through 2010, SAP expects this to become the "go-to" ERP release for many years to come. Therefore, capabilities of ABAP in SAP NetWeaver 7.0 will likely become the base-line technology level for most customer development as well.

If you don't already have access to a SAP NetWeaver 7.0 system, you can always download the free trial edition from the SAP Developer Network (*https://www.sdn.sap.com/irj/sdn/downloads*). This trial software has a full ABAP development environment, enabling you to recreate nearly all the examples contained within this book.

To help you follow along with the project as it unfolds in this book, we have also provided you with the source code for all examples in the book, as well as many supporting objects that are not discussed in detail on the accompanying CD. This should help to facilitate your skipping certain chapters if you want, without having to forego the prerequisite objects.

The source code on the CD is available in several different formats:

- First, there is a transport file. This is the simplest way to import all the development objects that are discussed in this book in their correct packages.
- Not all developers have the necessary security to import a transport file. For this reason, we have also included many of the development objects in SAPlink format (the open source XML based mechanism for exchanging ABAP development objects) and plain text files.

For complete instructions on how to work with each of these import formats, see the *ReadMe.pdf* file in the root directory of the CD or Appendix A.

In case you were wondering, please note that we won't forget about older releases just because our primary focus is on SAP NetWeaver 7.0. The technologies that we'll discuss were primarily released since SAP R/3 4.6C. As we

introduce each technology, we will try to indicate in which release it was first introduced, and, what differences, if any, there are between the releases.

As of SAP NetWeaver 7.0, SAP's ABAP foundation developers have not stopped innovating around the ABAP environment. As you read this book, dedicated teams are currently working on additional features and powerful new functions for the ABAP development environment. We will point out these anticipated features throughout the book; however they will simply be identified with the notation "Future Functionality."

With the direction of SAP ERP 6.0, SAP NetWeaver 7.0 will be an established release for many years to come. Therefore, some of this future functionality might find its way into SAP NetWeaver 7.0 via backports of the functionality delivered with support packages. Other new features may be too extensive to deliver in this way, and therefore be postponed until the next major release of SAP NetWeaver, or some other, as yet undetermined, delivery mechanism.

Acknowledgments

I would like to thank my wife, Shonna, for her love and support during the entire project, and for understanding how important this project was to me. Additionally, I would like to thank my kids, Kearston and Gavin, for their unconditional love and understanding while Daddy was working. Thanks for putting up with the laptop on the table during dinner and the shortened play time. Without my family's support, the past few months would have been much more difficult.

Thanks to my parents, for giving me the necessary foundation that enabled me to be successful in life. Thanks to my sister, Angie, for her inspiration and support, which gave me the "can-do" attitude that this project required.

I would also like to thank our editor, Stefan Proksch, for enabling me to share my knowledge with the rest of the world. Thanks to the SDN community for supplying great content that allowed me to learn directly from the experts. Last but certainly not least, I would like to thank my co-author, Thomas Jung, for the opportunity to work together on this project.

York (PA), May 2007
Rich Heilman

I also must start off by thanking my wife, Shari. Without her support, I certainly couldn't have completed the work required to create this book. What is even more amazing to me is that this time, she knew exactly what she was getting herself into and yet, she still agreed to let me work on the project. As with all accomplishments in my life, they simply would not have been possible without her love and support!

To my children, Megan and Madison, I owe my thanks as well. To them, it probably seemed like Daddy was hardly around for the last few months, since most nights and weekends he disappeared into his office.

The main character in the book, Russel, is named after my father. It is ironic that when we are teenagers, we want nothing to do with our parents, but as we grow older, we realize that the greatest compliment is hearing someone say how much we are like our parents. Mom and Dad gave me so much while I was growing up. I only wish they could be here today so that I could thank them.

To my friend, Brian McKellar, thanks for getting me started writing on SDN and giving me the opportunity to learn from you during our first book project together. I do and will carry those lessons with me in everything I do.

To the old gang at Kimball, there certainly is a little bit of each of you in this book, as I learned so much from everyone that I have had the opportunity to work with over the years.

To my new colleagues in SAP NetWeaver Product Management, for a virtual team who only sees one another a few times a year, we are an amazingly close-knit group. Everyone is absolutely wonderful to work with and so willing to share his or her knowledge.

To my friend and coworker, Peter McNulty, even before I decided to come to SAP, I figured if there were people this good in Product Management, then it was certainly an organization of which I wanted to be a part. Peter is always available to pitch ideas off of, and our discussions have influenced more than a few sections of this book.

To our editor, Stefan, thanks so much for giving us the opportunity to create this book. Your guidance and support have been instrumental in its completion.

Lastly, to my co-author, Rich, it has been a real pleasure working with you on this project, and to think it all started because you accepted an invitation to co-present with me at SDN Day at the SAP TechEd in Las Vegas in 2006.

Jasper (IN), May 2007
Thomas Jung

Russel has finished the programming for the database access layer. He now wants to optimize the read access to some of the data that is read frequently, but not updated often. For this, he will turn to the new shared memory objects technology that was introduced in SAP NetWeaver 2004.

5 Shared Memory Objects

Shared memory objects are ABAP Object Instances, which can be stored in the shared memory area on an application server. Instead of going to the database to retrieve the required data, the data is accessed through the shared memory, thereby providing faster data retrieval.

This shared memory area can be accessed by all of the ABAP programs running on that application server. Before the upgrade to SAP ERP 6.0, Russel used the `EXPORT/IMPORT` statements with the `SHARED BUFFER` or `SHARED MEMORY` extensions to access a similar memory buffer area. So what are the advantages of using this new functionality?

- First, it is read access to shared memory without the need to copy it into user session memory. Technically, an application does a remote attach to the memory segment within shared memory and directly interacts with it.
- Secondly, the new shared memory technique is implemented through ABAP Objects; therefore, you are provided with robust tools to interact with shared memory through code. Ultimately, you aren't just buffering raw sets of data; you're also providing a shared mechanism to access the business logic wrapped around this data.
- There are also dedicated tools for the monitoring and administration of these shared areas and the objects within them. Transaction SHMM, for example, provides tools to monitor the size and number of objects within a shared area, as well as enabling administrators to force objects out of memory if necessary.

5.1 Getting Started

Russel has spent a considerable amount of time developing the database access layer for this project and wants to ensure that performance is at an optimal level. He decides to leverage the shared memory objects functionality to increase performance when accessing some of the data in the database.

To use this feature of the ABAP runtime environment, Russel will have to create several new types of objects. Shared memory objects are implemented in two parts — the shared object *root* and *area* classes.

- The root class is the definition of the object that will be stored in shared memory. An instance (or multiple instances) of this class will reside in shared memory. Therefore this class's attributes should represent the data that you want cached and the methods of the class are the way that you access this data.
- The shared memory area class, on the other hand, will be a generated class. It abstracts a section of shared memory that is set aside for one or more instances of a particular root class. The methods of this area class provide the tools to attach to the shared memory area in order to read or manipulate it. The sole purpose of the area class is to return instances of the root class.

5.1.1 Area Root Class Creation

Russel decides that the `ZCS_COURSE` table would be a good candidate to create a shared memory object. Shared memory objects should primarily be used for objects that are read often, but updated infrequently. This is due to the locking mechanism that is used by shared objects. Although having multiple read locks across separate user sessions is possible and is the norm, any form of change lock is exclusive (i.e., it doesn't even allow parallel read locks on the same area instance).

This does make `ZCS_COURSE` a good fit. New courses are rarely created or changed during the school year. All updates are done all at once, before planning for the next semester begins. Technically, this means that this table will have frequent read accesses by students and teachers concurrently, but the data will rarely change.

Russel's first step in implementing a shared memory object to represent `ZCS_COURSE` is to create the area root class. This class implements the setter and getter methods, which are used to access the data to be stored in the shared

memory area. It could also include business logic that further manipulates the data during access operations. For instance, it might include calculations, the results of which could also be stored in shared memory. This is where the value of the shared memory object can extend well beyond the scope of just the buffering of data stored within the database.

Russel creates the class `ZCL_CS_COURSE_SHMO_ROOT` and assigns it to the `ZCS_DDIC` package using transaction code SE80 (see Figure 5.1).

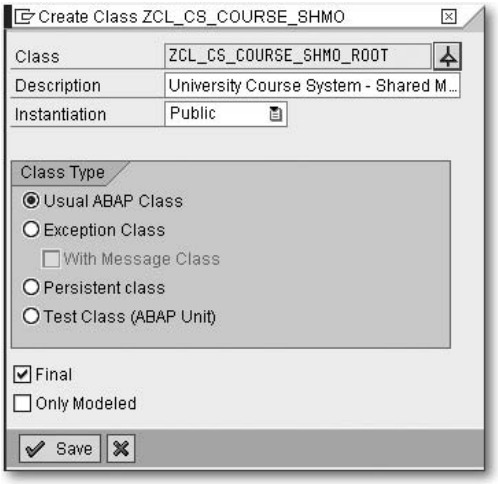


Figure 5.1 Root Class Creation

Russel then sets the **Shared Memory-Enabled** checkbox on the **Properties** tab (see Figure 5.2). This tells the system that the class is eligible to be used as a root class for a shared memory object.

The idea of using shared memory objects is to store data in memory, which can be used at runtime. Therefore, Russel needs to add an attribute to this class that will hold the data retrieved from the `ZCS_COURSE` database table.

Although it is technically possible to create public attributes of the root class that can be accessed directly from an instance of the class, Russel wants to follow good object-oriented designs and encapsulate all of his attribute accesses within methods. This gives him more control in case he wants to embed other operations within an access to this attribute. Therefore he defines the attribute as a **Private Instance** attribute (see Figure 5.3).

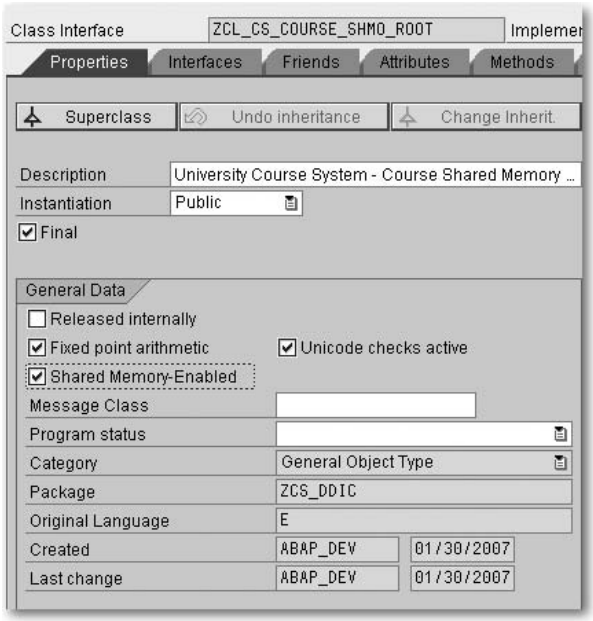


Figure 5.2 Root Class Properties

Figure 5.3 Define Attribute

The class now requires methods that can be used to populate or read this attribute. To start, Russel needs a `SET` method, which will be used to fill the `COURSE_LIST` attribute with all records in the database table. This method should be defined as a **Public Instance** method (see Figure 5.4).

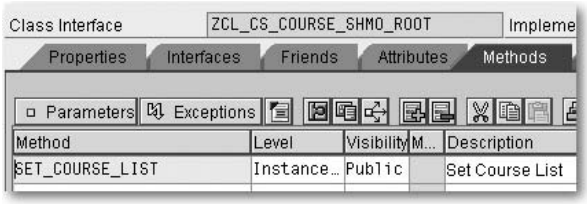


Figure 5.4 Define SET_COURSE_LIST Method

In the implementation of the SET_COURSE_LIST method, Russel leverages the persistent object for database table ZCS_COURSE to fill the instance attribute COURSE_LIST. As shown in Listing 5.1, Russel is simply borrowing some of the persistent object code from the method READ_ALL_COURSES of the class ZCL_CS_COURSE that he wrote in Chapter 3. He then loops through the objects and populates the returning parameter with the values.

```
METHOD set_course_list.
  DATA:  l_agent    TYPE REF TO zca_cs_course_pers,
         l_pers_obj TYPE REF TO zcl_cs_course_pers,
         l_objects  TYPE        osreftab.
  FIELD-SYMBOLS: <wa_object>  LIKE LINE OF l_objects,
                 <wa_course>  LIKE LINE OF course_list.
  DATA: query_manager TYPE REF TO if_os_query_manager,
        query         TYPE REF TO if_os_query.

  TRY.
    l_agent    = zca_cs_course_pers=>agent.
    query_manager = cl_os_system=>get_query_manager( ).
    query = query_manager->create_query( ).

    l_objects =
       l_agent->if_os_ca_persistency~get_persistent_by_query(
            i_query   = query ).
    IF LINES( l_objects ) = 0.
      RAISE EXCEPTION TYPE zcx_cs_course
         EXPORTING
           textid = zcx_cs_course=>bad_query.
    ENDIF.

    LOOP AT l_objects ASSIGNING <wa_object>.
      l_pers_obj ?= <wa_object>.
      APPEND INITIAL LINE TO course_list
        ASSIGNING <wa_course>.
      <wa_course>-syllabi = l_pers_obj->get_syllabi( ).
      <wa_course>-cost = l_pers_obj->get_cost( ).
      <wa_course>-course_id = l_pers_obj->get_course_id( ).
      <wa_course>-course_schedule =
         l_pers_obj->get_course_schedule( ).
      <wa_course>-course_sdesc =
         l_pers_obj->get_course_sdesc( ).
      <wa_course>-course_year =
         l_pers_obj->get_course_year( ).
      <wa_course>-credit_hrs = l_pers_obj->get_credit_hrs( ).
```

```
      <wa_course>-currency = l_pers_obj->get_currency( ).
      <wa_course>-deletion_flag =
          l_pers_obj->get_deletion_flag( ).
      <wa_course>-description =
          l_pers_obj->get_description( ).
      <wa_course>-end_time = l_pers_obj->get_end_time( ).
      <wa_course>-faculty_id = l_pers_obj->get_faculty_id( ).
      <wa_course>-major = l_pers_obj->get_major( ).
      <wa_course>-semester = l_pers_obj->get_semester( ).
      <wa_course>-start_time = l_pers_obj->get_start_time( ).
      <wa_course>-student_limit =
          l_pers_obj->get_student_limit( ).
    ENDLOOP.
  ENDTRY.
ENDMETHOD.
```

Listing 5.1 SET_COURSE_LIST Method Implementation

Russel also needs to define the `GET` methods, which will be used to retrieve the data. First, Russel needs a `GET` method to retrieve all the courses. The signature of this method will contain a `RETURNING` parameter, which is defined as the table type `ZCS_COURSES_TT` (see Figure 5.5 and Figure 5.6).

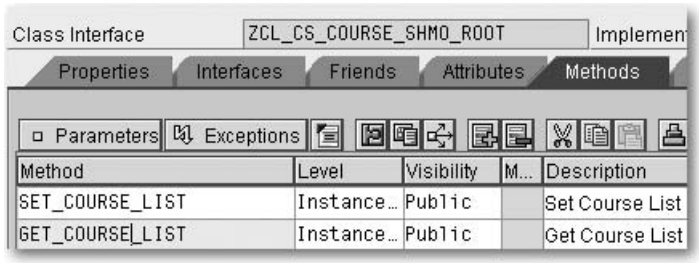


Figure 5.5 Define GET_COURSE_LIST Method

Figure 5.6 Define GET_COURSE_LIST Method Signature

Of course as soon as Russel uses a returning parameter, he negates one of the advantages of the shared memory object, namely, the copy free read. Imagine if you had a very large table that could either be exported to shared memory or placed in a shared memory object. In this example, you want to sort the internal table and then read a subset of the records.

With an internal table that was simply exported to shared memory, the entire table would have to be imported before any operations could be performed on it. This entails making a copy of the entire internal table and placing it into the internal session of the running application.

With a shared memory object, however, all of this logic could be placed within the shared object root class and only the resulting few records would be returned. This prevents you from having to copy anything, but the result set, out of shared memory and into the internal session.

In the shared object root class that Russel is building, he needs to support both kinds of accesses. He will eventually build a method that returns a single record, but some applications also need access to the entire course listing. For these applications, it doesn't make sense to keep a constant read attachment to the shared object instance, therefore, he decides to return a copy of the entire internal table attribute. Returning parameters are always marked as **Pass by Value** (see third column in Figure 5.6), making this copy operation happen automatically.

The `GET_COURSE_LIST` has a very simple implementation. Russel only needs to pass the instance attribute `COURSE_LIST` to the `RETURNING` parameter `RE_COURSE_LIST` (see Listing 5.2).

```
METHOD get_course_list.
  re_course_list = course_list.
ENDMETHOD.
```

Listing 5.2 GET_COURSE_LIST Method Implementation

Additionally Russel needs a `GET` method, which will be used to get a single course record. By clicking on the **Parameters** button, the signature of the method is displayed. The signature of this method contains an `IMPORTING` parameter for the `COURSE_ID`, which will be used to select the specific course. The second parameter is a `RETURNING` parameter, which will be used to return the course data. This `RETURNING` parameter is typed like `ZCS_COURSE` (see Figure 5.7 and Figure 5.8).

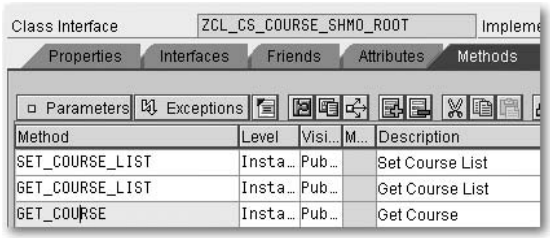


Figure 5.7 Define GET_COURSE Method

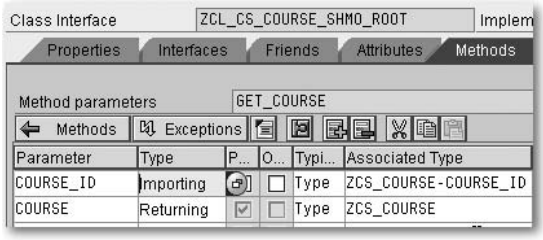


Figure 5.8 Define GET_COURSE Method Signature

Again, the implementation for the `GET_COURSE` method is fairly simple. A simple read statement will read the `COURSE_LIST` attribute and return the corresponding row based on the `IMPORTING` parameter `COURSE_ID` (see Listing 5.3).

```
METHOD get_course.
  READ TABLE course_list INTO course
                WITH KEY course_id = course_id.
ENDMETHOD.
```

Listing 5.3 GET_COURSE Method Implementation

5.1.2 Defining the Shared Memory Area

Russel now needs to create the shared memory area. The transaction code SHMA allows you to create the area and define its properties. When the shared memory area is created, a global class with the same name as the area is created automatically. Therefore, we recommend that you use the standard naming convention for classes, `CL_*` or `ZCL_*`, to name the memory area. This shared memory area class inherits from the class `CL_SHM_AREA`, which is a sub-class of `CL_ABAP_MEMORY_AREA`, giving it all the necessary methods for accessing area root class.

Russel uses transaction SHMA to create the shared memory area. The subsequent screen allows him to specify the properties of the area as well as the

root class that this area will be defined for (see Figure 5.9). For now Russel will leave the default properties that were suggested, no limits on the area size, lifetime, or number of versions. Later you will see how he can use some of these properties to set up automatic initialization of his shared object on the first read request.

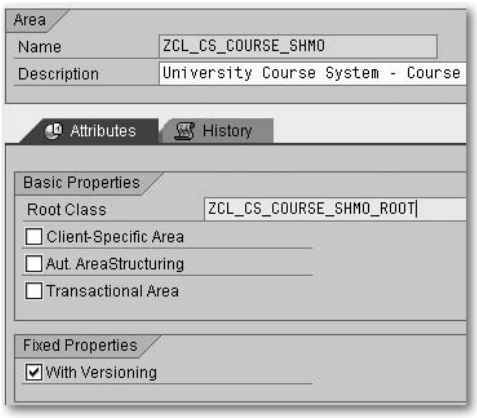


Figure 5.9 Area Properties

Now that the area class has been generated, Russel can look at the public methods that he will use to access the shared memory object (see Figure 5.10). The `ATTACH` methods will return *area handles*, which are instances of the area class.

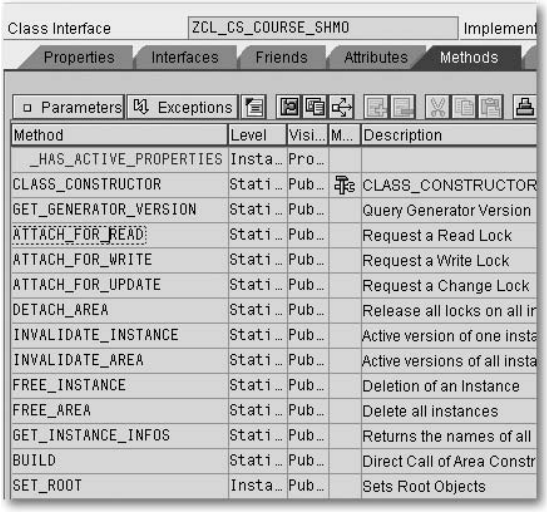


Figure 5.10 Methods of the Area Class

For example, the ATTACH_FOR_READ method will return an area handle, which can then be used to read the shared memory area. Similarly, the ATTACH_FOR_WRITE method will return an area handle, which will allow you to write to the shared memory area. The DETACH_AREA method removes the binding between the area class and the area handle.

5.1.3 Testing the Shared Memory Object

Russel wants to see the shared memory object in action before trying to use it directly in the rest of the course system. He decides to develop several short test programs to get a feel for how it all works. The first program will be a test write program, which will create the area instance of the area root class and place it into the shared memory area (see Listing 5.4).

```
REPORT  zcs_course_shmo_write.

DATA: course_handle TYPE REF TO zcl_cs_course_shmo,
      course_root   TYPE REF TO zcl_cs_course_shmo_root.

TRY.
    course_handle = zcl_cs_course_shmo=>attach_for_write( ).
    CREATE OBJECT course_root AREA HANDLE course_handle.
    course_handle->set_root( course_root ).
    course_root->set_course_list( ).
    course_handle->detach_commit( ).
  CATCH cx_shm_attach_error.
    ...
ENDTRY.
```

Listing 5.4 Write Test Program

Notice that fairly normal conventions are used for creating the COURSE_ROOT instance. Russel still uses the CREATE OBJECT syntax, but now with the new addition AREA HANDLE. These extra statements direct the ABAP runtime to instantiate the root class within shared memory instead of the internal session memory.

Russel writes a second program to test the reading of the data from the shared memory object (see Listing 5.5). This test program will allow Russel to ensure that the GET_COURSE_LIST method and the GET_COURSE method work properly. Before Russel runs this program, he must run the write program to load the memory area. Otherwise, he'll get an ABAP short dump when trying to access an unloaded memory area.

```
REPORT  zcs_course_shmo_read.

DATA: course_handle TYPE REF TO zcl_cs_course_shmo.
DATA: gt_courses TYPE zcs_courses_tt.
DATA: gs_courses TYPE zcs_course.

PARAMETERS: p_rad1 RADIOBUTTON GROUP grp1 DEFAULT 'X'.
PARAMETERS: p_rad2 RADIOBUTTON GROUP grp1.
PARAMETERS: p_csid TYPE zcs_course-course_id.

AT SELECTION-SCREEN.
  IF p_rad2 = 'X'
      AND p_csid IS INITIAL.
    MESSAGE e001(00) WITH 'Enter a course id'.
  ENDIF.

START-OF-SELECTION.

  TRY.
      course_handle = zcl_cs_course_shmo=>attach_for_read( ).
    CATCH cx_shm_attach_error.
  ENDTRY.

  CASE p_rad1.
    WHEN 'X'.
      gt_courses = course_handle->root->get_course_list( ).
    WHEN OTHERS.
      gs_courses = course_handle->root->get_course( p_csid ).
      APPEND gs_courses TO gt_courses.
  ENDCASE.

  course_handle->detach( ).

  LOOP AT gt_courses INTO gs_courses.
    WRITE:/ gs_courses-course_id,
            gs_courses-course_sdesc+0(20),
            gs_courses-faculty_id,
            gs_courses-semester,
            gs_courses-course_year,
            gs_courses-major,
            gs_courses-credit_hrs,
            gs_courses-student_limit,
            gs_courses-deletion_flag,
            gs_courses-start_time,
            gs_courses-end_time,
```

```
            gs_courses-course_schedule,
            gs_courses-cost,
            gs_courses-currency.

  ENDLOOP.
```

Listing 5.5 Read Test Program

5.1.4 Shared Memory Monitor

The shared memory monitor provides an interface in which you can monitor the area instances in the shared objects memory. The monitor allows you to view areas, area instances, versions, and locks. Drill-down functionality allows you to drill into these overviews via double-clicking on them.

Russel goes to transaction SHMM to check that the data has been written to the shared memory area by his test applications. He can see that there is one instance of the area class stored in the shared memory area `ZCL_CS_COURSE_SHMO` (see Figure 5.11). He can also see memory usage, number of instances, number of versions, and the status breakdown of the versions.

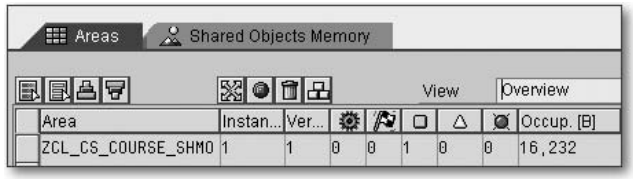


Figure 5.11 Shared Memory Monitor — Areas

If the `COURSE_LIST` attribute of the area root class was defined as **Public**, Russel could also see the data that is currently stored in the shared memory object. **Private** attributes, however, are not visible. This is also where he can delete shared memory areas.

For developers, the ability to delete a shared memory area within this transaction is probably one of the shared memory monitor's most useful functions. If you make any changes to the coding of the root class and reactivate it, the class will be given a new generation timestamp. The generation timestamp of the root class definition in the database is checked by the area class, whenever an access is made.

Therefore, if you make any changes to the root class after it has been stored within a shared area, this will cause an invalid version exception to be thrown every time you try to access the area. After each change to the root

class, you will have to delete any and all versions of the shared memory area before you can test your changes.

5.2 Automatic Preloading

Russel has reviewed what he has learned so far about shared memory objects and realizes that there are some weaknesses in his test applications. For example, it could be problematic if the shared memory area was read before it had been instantiated via a write operation. In other words, reading an unloaded shared memory area will only result in a short dump.

This can occur after the application server has been shut down and restarted. The shared memory areas are all cleared at this time. For the best reliability of his applications, Russel needs to find a way to preload the memory area at the time of the first read. Fortunately, the shared memory object implementation that SAP supplies has just the optional functionality he needs.

5.2.1 Adding the Interface

In order to take advantage of this functionality, he first must add the interface, `IF_SHM_BUILD_INSTANCE`, to the area root class `ZCL_CS_COURSE_SHMO_ROOT`. Once the interface is added, the `BUILD` method appears in the **Methods** tab (see Figure 5.12). This static method is automatically fired if any of the `ATTACH` methods of the area class are called and the shared memory area has not been loaded.

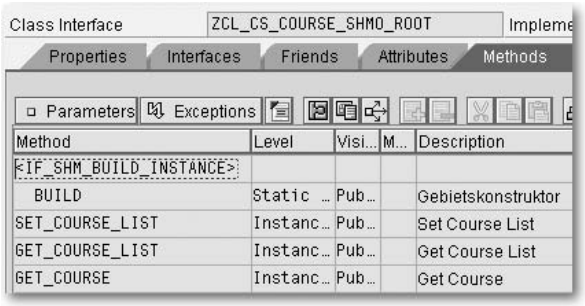


Figure 5.12 Build Method

Now Russel can copy and paste the code, which he wrote in the write test program `ZCS_COURSE_SHMO_WRITE` into the `BUILD` method (see Listing 5.6). This not only sets up the technical initialization of the root instance within

the area, but also provides an opportunity to preload all the data from the database via the call to the SET_COURSE_LIST method.

```
METHOD if_shm_build_instance~build.

  DATA: course_handle TYPE REF TO zcl_cs_course_shmo,
        course_root   TYPE REF TO zcl_cs_course_shmo_root,
        excep         TYPE REF TO cx_root.

  TRY.
      course_handle = zcl_cs_course_shmo=>attach_for_write( ).
    CATCH cx_shm_error INTO excep.
      RAISE EXCEPTION TYPE cx_shm_build_failed
                      EXPORTING previous = excep.
  ENDTRY.

  TRY.
      CREATE OBJECT course_root AREA HANDLE course_handle.
      course_handle->set_root( course_root ).
      course_root->set_course_list( ).
      course_handle->detach_commit( ).
    CATCH cx_shm_error INTO excep.
      RAISE EXCEPTION TYPE cx_shm_build_failed
                      EXPORTING previous = excep.
  ENDTRY.

  IF invocation_mode = cl_shm_area=>invocation_mode_auto_build.
    CALL FUNCTION 'DB_COMMIT'.
  ENDIF.

ENDMETHOD.
```

Listing 5.6 Build Method Implementation

Simply adding the BUILD method is not enough to have it triggered by the area class. Russel must return to transaction SHMM and adjust the properties on his area. He needs to set the flag for **Automatic Area Structuring** and the **Autostart** value for **Area Structure**.

Also he has to define the **Constructor Class**. This is the class where he implemented the BUILD method. Notice that no assumption is made that the BUILD method will be part of the root class. That is a common approach, but the BUILD method can actually belong to any global class.

5.2.2 Modifying the Read Program

Finally Russel needs to modify the read test program ZCS_COURSE_SHMO_READ. Although the static BUILD method of the root class will be called automatically now, it does so asynchronously.

Instead of simply calling the method ATTACH_FOR_READ, Russel needs to take into account the asynchronous BUILD method and modify the program so that it waits for the shared memory area to be loaded by the BUILD method. Then, he needs to call the ATTACH_FOR_READ method again. The BUILD method is actually fired in a different work process, which accounts for needing the WAIT statement (see Listing 5.7).

```
START-OF-SELECTION.
  TRY.
      course_handle = zcl_cs_course_shmo=>attach_for_read( ).
    CATCH cx_shm_no_active_version.
      WAIT UP TO 1 SECONDS.
      course_handle = zcl_cs_course_shmo=>attach_for_read( ).
  ENDTRY.
```

Listing 5.7 Read Program Modification

Russel can now use transaction SHMM to delete any shared memory areas that may still exist. Since Russel has modified the area root class ZCL_CS_COURSE_SHMO_ROOT, he must delete any existing shared memory areas for this root. If this isn't done, Russel will get an ABAP runtime exception stating an inconsistency is present.

Russel can now run the read program directly instead of having to first run the write program. The output proves that the preloading of the shared memory object is working correctly.

5.3 Implementing into the Business Object Layer

Russel has finally completed the programming required for the shared memory object and has tested that it works correctly. The next step is to implement this shared memory object in the business object layer of the course system.

The main goal is to swap out the persistent object code and replace it with the shared memory object code. When the exchange is complete, the changes should have no affect on any developments that use the business

object. This allows us to hide any complexities of using the shared memory object from the application logic. Activities like having to wait for the asynchronous BUILD method to complete will all be handled within the business object class now.

Example Source Code

It is important for you to understand that normally Russel (i.e., the developer) would be directly modifying the business object class ZCL_CS_COURSE to implement the shared memory object. In order to illustrate how the business objects change as we delve further into the development of the examples that accompany this book, we will show you how to implement the shared memory object in a copy of the ZCL_CS_COURSE business object class.

For a complete example of all of the changes that you need to make to the business object class ZCL_CS_COURSE, see the class ZCL_CS_COURSE_SHM_ACCESS.

5.3.1 Developing a Test Program

Russel wants to develop a simple program to test data retrieval using the business object class. This simple report program will retrieve all of the courses and write the data out to a standard list display (see Listing 5.8). Later he will use this same program to test the implementation of the shared memory object for the course database.

```
REPORT  zcs_course_obj_read.

DATA: gt_courses  TYPE STANDARD TABLE OF zcs_course_att.
FIELD-SYMBOLS: <gs_courses> LIKE LINE OF gt_courses.
DATA: gt_courses_obj TYPE zcs_courses_tbl.
FIELD-SYMBOLS: <gs_courses_obj> LIKE LINE OF gt_courses_obj.

START-OF-SELECTION.

  gt_courses_obj = zcl_cs_course=>read_all_courses( ).
  LOOP AT gt_courses_obj
          ASSIGNING <gs_courses_obj>.
    APPEND INITIAL LINE TO gt_courses
                   ASSIGNING <gs_courses>.
    MOVE-CORRESPONDING
           <gs_courses_obj>-course->course
                       TO <gs_courses>.
  ENDLOOP.

  LOOP AT gt_courses ASSIGNING <gs_courses>.
```

```
    WRITE:/ <gs_courses>-course_id,
            <gs_courses>-course_sdesc+0(20),
            <gs_courses>-faculty_id,
            <gs_courses>-semester,
            <gs_courses>-course_year,
            <gs_courses>-major,
            <gs_courses>-credit_hrs,
            <gs_courses>-student_limit,
            <gs_courses>-deletion_flag,
            <gs_courses>-start_time,
            <gs_courses>-end_time,
            <gs_courses>-course_schedule,
            <gs_courses>-cost,
            <gs_courses>-currency.
  ENDLOOP.
```

Listing 5.8 Course Object Test Program

5.3.2 Modifying the Business Object Class

Russel has proven that the current business object class `ZCL_CS_COURSE` works well using the persistent object for the course database. To keep things simple, we'll focus now on only those changes required for the `READ_ALL_COURSES` method. Listing 5.9 shows that the code for the persistent object has been removed, and the new code to retrieve the data from the shared memory object has been inserted.

```
METHOD read_all_courses.

  DATA: course_handle TYPE REF TO zcl_cs_course_shmo.
  DATA: lt_courses TYPE zcs_courses_tt.
  FIELD-SYMBOLS: <ls_courses> LIKE LINE OF lt_courses,
                 <wa_course>  LIKE LINE OF r_courses.
  TRY.
      course_handle = zcl_cs_course_shmo=>attach_for_read( ).
    CATCH cx_shm_no_active_version.
      WAIT UP TO 1 SECONDS.
      course_handle = zcl_cs_course_shmo=>attach_for_read( ).
  ENDTRY.
  lt_courses = course_handle->root->get_course_list( ).
  course_handle->detach( ).

  LOOP AT lt_courses ASSIGNING <ls_courses>.
    APPEND INITIAL LINE TO r_courses ASSIGNING <wa_course>.
    <wa_course>-course_id = <ls_courses>-course_id.
```

```
    CREATE OBJECT <wa_course>-course
      EXPORTING
        i_course = <ls_courses>.
  ENDLOOP.

ENDMETHOD.
```

Listing 5.9 READ_ALL_COURSES Method

Also notice that the variable being passed to the CREATE OBJECT statement has changed. Instead of passing the persistent object, Russel is now passing a flat structure, which contains the course data. This means that the signature of the CONSTRUCTOR method of the business object class must also be modified. The I_COURSE parameter must be typed like ZCS_COURSE (see Figure 5.13). Because the CONSTRUCTOR is private and only called via static factory methods, this sort of change has no effect on the applications that are using the business object class.

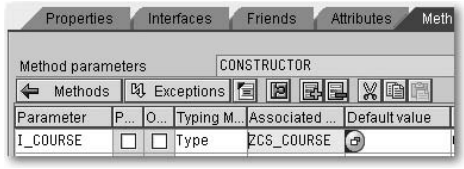


Figure 5.13 Constructor Signature

The CONSTRUCTOR implementation has changed a bit as well. Since Russel is now passing a flat structure to the I_COURSE parameter, the CONSTRUCTOR must do something with this data. Russel has added a new private instance attribute called SHMA_DATA. This attribute will hold the data that is passed from the I_COURSE parameter (see Listing 5.10).

```
METHOD constructor.
  me->shma_data = i_course.
  me->course_preq_pers =
     me->load_course_preqs( i_course-course_id ).
  me->map_shared_to_struc( ).
  me->load_supporting_details( ).
ENDMETHOD.
```

Listing 5.10 Constructor Modifications

Russel has also added a new method called MAP_SHARED_TO_STRUC that replaces the mapping from the persistent object, and will be used to map the data from the SHMA_DATA attribute to the COURSE attribute of the business object (see Listing 5.11).

```
METHOD map_shared_to_struc.

  course-course_id       = shma_data-course_id.
  course-course_sdesc    = shma_data-course_sdesc.
  course-faculty_id      = shma_data-faculty_id.
  course-semester        = shma_data-semester.
  course-course_year     = shma_data-course_year.
  course-major           = shma_data-major.
  course-credit_hrs      = shma_data-credit_hrs.
  course-student_limit   = shma_data-student_limit.
  course-deletion_flag   = shma_data-deletion_flag.
  course-start_time      = shma_data-start_time.
  course-end_time        = shma_data-end_time.
  course-course_schedule = shma_data-course_schedule.
  course-cost            = shma_data-cost.
  course-currency        = shma_data-currency.
  course-description     = shma_data-description.

* Load faculty using business object class
  TRY.
      course-faculty =
        zcl_cs_faculty=>read_faculty( course-faculty_id ).
    CATCH zcx_cs_faculty.
  ENDTRY.

  DATA l_syllabi TYPE xstring.
  l_syllabi = shma_data-syllabi.
  DATA izip TYPE REF TO cl_abap_gzip.
  IF l_syllabi IS NOT INITIAL.
    CREATE OBJECT izip.
    izip->decompress_text( EXPORTING gzip_in = l_syllabi
                   IMPORTING text_out = course-syllabi ).
  ENDIF.

  FIELD-SYMBOLS: <wa_pers> LIKE LINE OF course_preq_pers,
                 <wa_preq> TYPE zcs_course_preq_att.
  LOOP AT course_preq_pers ASSIGNING <wa_pers>.
    APPEND INITIAL LINE TO course-pre_req ASSIGNING <wa_preq>.
    <wa_preq>-preq_id = <wa_pers>-course_preq->get_preq_id( ).
  ENDLOOP.

ENDMETHOD.
```

Listing 5.11 MAP_SHARED_TO_STRUC Method Implementation

5.3.3 Testing the Changes

Now that Russel has completed the changes required to the business object class, he can use the test program, which he created earlier to see whether the data is being retrieved correctly. Again Russel would have directly modified the business object class ZCL_CS_COURSE, so there would be no changes required to the test program to make it work. For our purposes, we have implemented the changes in a copy of the business object class. Therefore, the test program ZCS_COURSE_OBJ_READ must be slightly modified to use the new business object class ZCL_COURSE_SHM_ACCESS.

Listing 5.12 shows that Russel is simply swapping out the table type used to receive the objects from the business object class, and the static call to method READ_ALL_COURSES.

```
REPORT  zcs_course_obj_read.
DATA: gt_courses  TYPE STANDARD TABLE OF zcs_course_att.
FIELD-SYMBOLS: <gs_courses> LIKE LINE OF gt_courses.
*DATA: gt_courses_obj TYPE zcs_courses_tbl.
DATA: gt_courses_obj TYPE zcs_courses_tbl_sma.
FIELD-SYMBOLS: <gs_courses_obj> LIKE LINE OF gt_courses_obj.

START-OF-SELECTION.

* gt_courses_obj = zcl_cs_course=>read_all_courses( ).
  gt_courses_obj =
         zcl_cs_course_shm_access=>read_all_courses().
```

Listing 5.12 Test Program Modifications

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z