Object Oriented Design with ABAP

Requirements Document for Associated Exercise Programs

This document describes both the functional and technical requirements for the exercise programs associated with the book Object Oriented Design with ABAP.

Table of Contents

1 Exercise programs

Completed sample exercise programs are provided for all chapters of the book Object Oriented Design with ABAP. These are to be created, edited and run in an SAP environment supporting ABAP programming development. The student is expected to copy the first sample exercise program to a student-specific copy and to use this as a starting point for subsequent exercises. Refer to the section **1.2 Exercise program naming convention** for more information on program naming conventions to be used for new copies of programs.

The sample exercise programs all exhibit the following characteristics:

1. The program property associated with Unicode compliance is set “on”. Students are advised to retain this setting in their own exercise programs.

2. The program produces no errors or warnings by the Extended Syntax Check even when selecting all options[1]. Students are advised to continue with this level of compliance in their own exercise programs.

3. Each sample exercise program has a comment block at the top of the code indicating how to define attributes associated with the program, such as title and selection texts. This can be used to determine whether all the components of the program are defined correctly.

4. Throughout the code, class attributes and methods are referenced using the class selector (=>) or self-reference selector (me->) qualifiers even when these qualifiers are not syntactically required. Students may opt to drop these qualifiers within their own exercise programs, but it is recommended to consider retaining them since they reinforce both the level (static or instance) of class members as well as clearly distinguishing class attributes from method variables and parameters. **Caution:** In some cases these class selector and self-reference selectors distinguish between an attribute and a method signature parameter with the same name. If the student were to choose to drop these optional qualifiers, it may require renaming the attribute or the signature parameter to avoid statements otherwise reduced to referencing the same identifier as both source and target of an operation.

1.1 Determining how to apply changes to exercise programs

In addition to the functional and technical requirements outlined in this document, the following additional aids may be used to determine the differences between two adjacent versions of exercise programs:

- A comment block appears at the top of the source code for each program in which there is a section preceded by a “Differences with preceding version:” banner. This will provide more information about how the new version differs from its predecessor version.

- Perform a comparison of the differences between two adjacent versions by following these simple steps in the SAP environment in which the exercises are being performed:
  1. Invoke transaction SE39.
  2. From menu, select: Utilities > Settings.
  3. On tab ABAP Editor, sub-tab Splitscreen, select check-mark to indicate "Ignore Indentations" and press enter.
  4. Specify the name of the object representing the preceding version in the Left Program slot, the newer version in the Right Program slot, and press Display.

1 Selecting the Programming Guidelines option with the Extended Syntax Check will produce warnings indicating the presence of global variables defined in the first few exercise programs. These global variables become completely eliminated by exercise 5.

5. Press the "Comparison On" button appearing on the button bar.
6. Alternate pressing the "Next Difference from Cursor" and "Next Identical Section from Cursor" (easiest to do this by holding CTRL+SHIFT and alternately pressing F9 and F11). Both the left and right sections are scrolled forward together as the next equal or different line is located.

Either of these methods may assist the student in deciding how to proceed with applying a change.

1.2 Exercise program naming convention

The exercise programs all are named using the following naming convention:

Character position	Value	Notes
1	Z	First character of all user programs must begin with Y or Z.
2 – 4	OOT	Acronym for Object Oriented Training.
5 – 7	Objects topic identification number	A 3-digit number corresponding to a topic presented in the book Object Oriented Design with ABAP. The numbers represent the following objects topics: Objects 101 – Encapsulation Objects 102 – Abstraction Objects 103 – Inheritance Objects 104 – Polymorphism Objects 105 – Interfaces Objects 301 – Design Patterns: Singleton pattern Objects 302 – Design Patterns: Strategy pattern Objects 303 – Design Patterns: Observer pattern Objects 304 – Design Patterns: Factory patterns Objects 305 – Design Patterns: Adapter pattern Objects 306 – Design Patterns: Decorator pattern Objects 307 – Design Patterns: Chain of Responsibility pattern Objects 308 – Design Patterns: Iterator pattern Objects 309 – Design Patterns: Template Method pattern Objects 310 – Design Patterns: Command pattern Objects 311 – Design Patterns: Null Object pattern Objects 312 – Design Patterns: State pattern Objects 313 – Design Patterns: Lazy Initialization Objects 314 – Design Patterns: Flyweight pattern Objects 315 – Design Patterns: Memento pattern Objects 316 – Design Patterns: Visitor pattern
8	Alphabetic character	Unique appendage distinguishing program names related to the same topic identification number, starting with A and proceeding through the alphabet as required.

The naming convention to be used by the student is modeled after the naming convention shown above, with the 3-character initials of the student to be placed between the first character “Z” and the “OOT” characters otherwise occupying the next three character positions, e.g.: ZLVBOOT101B for Ludwig Von Beethoven. All new exercise programs should be assigned package $TMP.

1.3 Exercise program intent

The exercise programs are intended to assist the student in understanding the new concepts associated with the object-oriented principles and design patterns. They are written specifically so that the same functionality provided by the first exercise program continues throughout the remaining exercises, with the expectation that the student will become familiar with the basic functionality with the first exercise program, and thereafter no longer will need to learn any new functionality other than what is being introduced with the new applicable concepts. Accordingly, these exercise programs focus on learning concepts and do not serve as suitable models for actual user programs, with the following among some of the reasons:

- To facilitate simplicity, these exercise programs are devoid of security checking and robust exception checking, aspects of programming normally present in production programs.

- To facilitate easy comparisons between two adjacent exercise programs via transaction SE39, all code associated with one exercise program is contained within a single object.

- To accommodate multiple students working through the exercise programs in the same SAP environment, the use of ABAP repositories has been avoided for defining new components. This eliminates the necessity to define and refer to user-specific component names within the exercise programs, enabling the use of naturally meaningful names within the code. It also means:

    - The structure used to produce the ALV report has been defined within the exercise program itself, and not as a structure defined to the ABAP DataDictionary. This requires some extra processing within the program to define ALV report column headers that otherwise would not be required had this structure been defined to the DataDictionary.

    - All object-oriented classes are defined as local classes. Often an object-oriented class will be of use to a multitude of programs, in which case it would be advantageous to define the class as a global class. Global classes are defined using the Class Builder transaction (SE24) similarly to the way function modules are defined using the Function Builder transaction (SE37). Learning object-oriented programming solely through the use of local classes enables the student to concentrate only on the object-oriented concepts, and not also on having to learn how to use a tool which simplifies the definition and organization of class components.

- The ABAP language accommodates specifying a local object-oriented class through both a *definition* component and a related *implementation* component. Both complementary components compose a complete class. In these exercise programs, the definition components for *all* of the local classes physically precede *any* implementation components. This is not the only way to organize these components, and there is considerable merit in keeping both definition and implementation components of a class adjacent to each other. These components have been separated from each other here only to eliminate extra considerations applying across dependent classes.[2]

- The final exercise program contains code to facilitate all of the design patterns presented in the book Object Oriented Design with ABAP. This should not be regarded as a suggestion to pack a program full of design patterns. Though tempting for programmers to use all programming concepts at their disposal, the decision whether to use, and if so, which, design patterns should be based on sound software engineering reasons.

2 This is not a concern we would encounter with ABAP global classes, but occurs because of the single-pass through the source code by the ABAP compiler.

1.4 Exercise program diagrams

The exercise programs are accompanied by various types of diagrams to assist the student in understanding the applicable changes to be applied. These diagrams fall into the following categories:

- Program structure diagrams, illustrating how the blocks of code are changing from one program to the next.
- Unified Modeling Language (UML) class diagrams
- State diagrams

Each exercise program will have either a program structure diagram or a UML class diagram, but not both. The book Object Oriented Design with ABAP first covers the basic principles of object-oriented programming, then covers an introduction to the Unified Modeling Language (UML), then covers Design Patterns. Accordingly, those exercise programs aligned with learning the basic principles of object-oriented programming will have associated program structure diagrams, and those aligned with learning Design Patterns, by which time the student has become familiar with the concepts of UML, will have associated UML class diagrams, obviating the need to continue with program structure diagrams.

State diagrams are provided only for some of the exercise programs associated with the State Design Pattern, specifically, those for which the functional specifications include changes requested by the user to introduce a new state.

All of the exercise program diagrams are provided in separate files, with one diagram per file. The name of a diagram file contains the name of the corresponding exercise program. For each exercise program. there will be one associated diagram file describing a program structure diagram and two associated diagram files describing a UML class diagram – one highlighting the differences with the UML class diagram of the preceding exercise program and another without highlighting. Those exercise programs for which a state diagram is applicable also will have one associated state diagram file.

2 Organization of requirements

This requirements document provides each exercise program with both a functional requirement and a technical requirement. The functional requirement describes what the user wants the program to do differently or what the user will see once a change is made. The technical requirement describes what the developer is to do to make the program work differently.

Not all aspects of the technical requirements are spelled out and it is assumed developers will find their own way to apply the necessary changes. For instance, a technical requirement might indicate to create a new local class with a new method, perhaps suggesting names for each one, but will not necessarily indicate the parameters to be defined for the method. It is expected that students will make their own decisions on such matters as class names, method names and associated parameters. In those cases where the student reaches a point where it becomes difficult to decide how to apply a new change, it is recommended to run the comparator between the previous and current versions of the two relevant sample exercise programs to see how changes were applied. A recommended process to follow for running the comparator is described in the comment block appearing at the top of every sample exercise program.

Due to the nature of the ABAP language, the definitions for all local interfaces and local classes must precede any references to them by subsequent local interfaces and local classes. Accordingly, the technical specifications will suggest the relative placement within the program for the definitions of new interfaces and classes. For consistency, it also is recommended to follow this same placement suggestion with the corresponding class implementations.

Sample exercise program ZOOT101A is the starting point. It is written almost entirely in classic ABAP[3]. By copying this program to your own student copy and applying changes described by the requirements, and doing this repeatedly for each new requirement, this simple program eventually is transformed into a complex program with many local classes making their respective contributions to the full set of functional requirements. If the comparative sizes of the first and last sample exercise programs are any indication to what students might expect with their own programs, the final exercise will result in a program nearly 20 times larger than the starting exercise program.

Though not used with the sample exercise programs, the student may opt to break the program into a series of components connected through INCLUDE statements.

It is recommended to make liberal use of TYPES statements to define various data types which might be required throughout the implementations for each new requirement, particularly in those cases where parameters are to be exchanged using a specific data type. The starting program gives a hint to this idea with the types associated with the parameters of the initial selection screen.

3 Indeed, the author of the program committed some breaches of what now are considered best ABAP programming practices, such as defining global variables, using references to screen variables throughout the subroutines and defining some subroutines with no signature, relying instead on passing values via global variables. This was done intentionally to provide a realistic example of the style of programming the student is likely to encounter in any one of thousands of standard SAP or customized ABAP programs. Refer to the book Official ABAP Programming Guidelines (Horst Keller, Wolf Hagen Thümmel; Galileo Press, 2010) for more information on avoiding such pitfalls.

3 Objects 101 – Encapsulation

This section describes the requirements for the exercise programs associated with the chapter covering the topic of Encapsulation in the book Object Oriented Design with ABAP.

3.1 Exercise 1

Program: ZOOT101A

Title: Objects 101: Car report using no local classes, step 1

Functional requirements

This program has a selection screen on which the user specifies the following information about a car:

- License plate
- Brand (aka Make)
- Model
- Year
- Color
- Starting location (city and state)
- Starting compass heading (N, E, S, W)
- Sequence of 3 turns (L=Left turn; R=Right turn; U=U-turn)
- Speed unit (e.g., MPH, KPH, etc.)
- Sequence of 3 speed increments (numeric, may include negative values)

After completing the selection screen, the user presses Execute and an ALV report is produced which presents a single row with columns showing the same information as provided on the selection screen, except:

- The Heading now represents the current heading after applying the 3 turns to the Starting compass heading

- The Speed shows the sum of the speed increments

Technical requirements

This is the starting program model. Other than the use of the ALV Object Model to produce the report, it is written using only the classic procedural ABAP capabilities – that is, it is devoid of any customized local or global classes. It merely obtains selection screen information and creates a corresponding single row in an ALV report using the information from the selection screen.

Become familiar with the code in this program. You will be using it as the starting model for applying changes corresponding to all of the concepts covered in the book Object Oriented Design with ABAP.

If you are not yet familiar with the ALV Object Model (ALV OM), this program will get you started. ALV OM is the successor to the ALV classes defined for the GUI front end (CL_GUI_ALV*), which are themselves successors to the function modules defined for presenting ALV (REUSE_ALV*). The subroutine present_report shows how to create and display an ALV object. One of the things ALV OM can do superior to its predecessor ALV classes is to present an ALV grid without the need for a field catalog. Notice the call to method cl_salv_table=>factory provides only the name of the internal table, and it receives a reference to an object. The ALV OM is capable of creating an object to present table content simply by providing the table itself – no field catalog is required. Indeed, the internal table used here is one for which the definition of the row structure is defined in the program itself (see definition of output_stack), and is not a global structure defined in the ABAP DataDictionary. This needs to be followed simply by a call to the display method of the object reference, which, again superior to its predecessor ALV classes[4], needs no local screen defined to accommodate the presentation of the ALV grid. In between these two method calls you will see a subroutine call to set column titles. This is necessary only because the resulting columns would otherwise have no titles. Confirm this for yourself by commenting out the call to the subroutine and you will see that all titles will appear blank.

3.2 Exercise 2

Program: ZOOT101B

Title: Objects 101: Car report using one local static class, step 2

Functional requirements

The requirement for this version is identical to the previous version – that is, the user sees no discernible difference. All changes are technical in nature.

Technical requirements

The changes implemented here are simple refactoring changes to prepare for defining local classes. Classic event blocks are to have as few statements as possible, delegating the bulk of processing to classic subroutines. In addition, screen variables are not to be referenced within any subroutines. Instead, the subroutines are to use only those parameters passed to them.

Refer to the associated program structure diagram. Copy ZOOT101A to create your student copy of ZxxxOOT101B, substituting “xxx” with your initials. It is recommended from this point forward to copy all associated objects (Documentation, Variants, User Interface, etc.) whenever making a copy of a program. Make the following changes:

- Define a new subroutine register_car_entry which is to accept “using” parameters for each field defined on the selection screen. Place this at the end of the program.

- Move all the perform statements in the “at selection-screen” block into the new subroutine register_car_entry, adjusting each one to reference the parameters passed into the subroutine instead of screen variables.

- Replace the perform statements removed from the “at selection-screen” block with a perform statement to the new subroutine register_car_entry, passing all of the screen variables as parameters.

- Change subroutine set_characteristics to accept “using” parameters for license plate, brand, model, year, color, location and speed unit. Use these parameters to set the values for the global variables.

This is the first program the student creates. It is recommended also to create at least 3 execution variants to go with it, using whatever values the student might find meaningful, such as the characteristics associated with the student's first car, current car and dream car.

4 These predecessor classes do provide some capabilities which ALV OM does not yet provide. Two notable examples of this are a) the fact that ALV OM does not enable direct user editing in the cells of the ALV grid and b) the absence in ALV OM of a method comparable to get_filtered_entries of class cl_gui_alv_grid.

Test the program to insure it still behaves as expected.

3.3 Exercise 3

Program: ZOOT101C

Title: Objects 101: Car report using one local static class

Functional requirements

The requirement for this version is identical to the previous version – that is, the user sees no discernible difference. All changes are technical in nature.

Technical requirements

Refactoring is applied to encapsulate all the information about the car, creating the first of many classes to be introduced into the program.

Refer to the associated program structure diagram. Copy ZxxxOOT101B (or ZOOT101B) to create your student copy of ZxxxOOT101C, substituting “xxx” with your initials. Make the following changes:

- Find in the program those types, constants, variables and subroutines related to the car and move these to a “car” local static class – types, constants and variables are to become class attributes and subroutines are to become class methods. Place the definition of the new “car” class following the report statement and preceding the screen definition, and its definition statement should include the following qualifiers[5]:
    - abstract – This qualifier indicates that instances cannot be created for this class. We will cover creating instances of classes in subsequent exercises.
    - final – This qualifier indicates that no inheritors may be defined for this class. We will cover inheritors in subsequent exercises.
    - create private – The “create” qualifier indicates the level at which this class is visible to entities for creating instances of it, and conforms to the same concept of visibility a class assigns each of its members. It is followed by a word indicating this level of visibility:
        - private – only this class may create an instance of this class
        - protected – only this class and its inheritors may create an instance of this class
        - public – any entity may create an instance of this class

      A create qualification always is applicable to a class, and defaults to public when not explicitly stated otherwise in the class definition.

5 Actually, none of these qualifiers are required at this point, however the combination of “abstract” and “final” on a class definition statement clearly indicates to the next programmer the definition of a static class. Without these, it would require the programmer to scan the entire class definition searching for the definition of a single instance member to know that it is *not* a static class, and upon finding none, concluding that it *is* a static class. These two statements in tandem save the programmer the trouble. The class later can be changed to a non-static class, but it will require the programmer to remove one or both of these qualifiers to do so, explicitly confirming the new intended use of the class is to be non-static. The “create private” qualifier has no bearing on the class at this point since it is not defined to accommodate the creation of any instances, however subsequent maintenance efforts to change this from a static class to a non-static class also will force the programmer to address this visibility level of creation, which suddenly would become applicable with this most restrictive visibility level upon the removal of the “abstract” and/or “final” qualifiers.

- In the remaining classic ABAP portions of the program, change those references to types and constants moved to the “car” class to now use the class name and class selector (=>) to qualify them, such as the following example to change field color of structure output_row:

```
Change   color        type color_type
To       color        type car=>color_type
```

This refactoring results in taking those aspects of the program that specifically relate to a “car” and encapsulating them into a class. While constructing the class, give consideration to the visibility of the class attributes – whether they should be defined publicly visible (to all external objects) or only privately visible (only to other members within the class itself). In accordance with the concept of Encapsulation, member visibility should be as restrictive as possible and only as visible as necessary.

Test the program to insure it still behaves as expected.

3.4 Exercise 4

Program: ZOOT101D

Title: Objects 101: Car report using two local static classes

Functional requirements

The requirement for this version is identical to the previous version – that is, the user sees no discernible difference. All changes are technical in nature.

Technical requirements

Some of the attributes related to the new “car” class can themselves be more appropriately encapsulated into a “navigator” class.

Refer to the associated program structure diagram. Copy ZxxxOOT101C (or ZOOT101C) to your student copy ZxxxOOT101D and make the following changes:

- Refactor the code accordingly, moving (see note below) those variables and methods related to navigation (compass, heading, left_turn, right_turn, etc.; change_heading, get_heading, set_heading) into a “navigator” local static class, using the same criteria for deciding the visibility of each class member as used for defining the “car” class. In accordance with the prescribed format that all local class definitions are to precede any local class implementations, place the definition portion for the new “navigator” class immediately following the report statement and preceding the definition portion of the “car” class, and place the implementation portion of the new “navigator” class immediately following the definition portion of the “car” class and preceding the implementation portion of the “car” class.

Note: Regarding the refactoring of methods, all that “moves” to the new “navigator” class are the implementations for “change_heading”, “get_heading” and “set_heading”. The definitions for these methods, though copied to the “navigator” class, are to remain unchanged in the “car” class, as are the calls to those “car” methods from the classic procedural subroutine. The “car” class still requires its own implementations for these methods, however now these methods simply delegate the call to the counterpart method in the “navigator” class. Accordingly, the “change_heading”, “get_heading” and “set_heading” methods of the “car” class are relegated to “pass-through” methods.

The classic procedural subroutines making the calls to set, change and get headings are to remain oblivious to the fact that the “car” class does not actually do these things itself anymore, but instead delegates these behaviors to methods of the “navigator” class. In short, simply because we extracted some functionality out of the “car” class and moved it into the “navigator” class does not suddenly compel all callers of these “car” methods to change.

Test the program to insure it still behaves as expected.

3.5 Exercise 5

Program: ZOOT101E

Title: Objects 101: Car report using maximum local static classes

Functional requirements

The requirement for this version is identical to the previous version – that is, the user sees no discernible difference. All changes are technical in nature.

Technical requirements

Most of the remaining classic variables and subroutines relate to the production of the ALV report.

Refer to the associated program structure diagram. Copy one of the previous versions (ZxxxOOT101D or ZOOT101D) forward to your student copy of this program and make the following changes:

- Refactor the program to encapsulate the remaining aspects of the program into a local static “report” class. Place the definition portion of the new “report” class following the definition portion of the “car” class and preceding the implementation portion of the “navigator” class, and place the implementation portion of the new “report” class following the implementation portion of the “car” class and preceding the screen definition.

  Note: From this point forward, the suggestion for where within the code to locate a new local class will imply that the definition and implementation portions are to be separated from each other, each portion to be placed relative to the respective definition and implementation portions of an existing local class.

When you have completed this refactoring, the classic procedural sections of the program will contain only the selection screen definition and the classic event blocks, each of which simply invokes a method of the local “report” class. The bulk of the processing has been encapsulated into local classes. Also, there are no global variables remaining.

Note: At this point you may not agree that everything now residing in class "report" belongs there. If so, you already have a good sense for defining classes properly. Yes, there are some things defined for "report" that are inappropriate, but we will defer adjusting this until later with subsequent exercise programs which will introduce the associated object-oriented principles and concepts.

Test the program to insure it still behaves as expected.

4 Objects 102 – Abstraction

This section describes the requirements for the exercise programs associated with the chapter covering the topic of Abstraction in the book Object Oriented Design with ABAP.

4.1 Exercise 6

Program: ZOOT102A

Title: Objects 102: Car report using local class object instantiation

Functional requirements

The users no longer like the fact that the report is capable of showing the registration for only one car at a time. They want to be able to enter a series of car registrations and see all of them appear in the report. Enable the user to register multiple cars using a new button to be provided on the initial selection screen:

- Add new car

The following status message is to be shown at the bottom of the screen with each newly registered car:

Entry registered for <license plate>

Once the user has completed registering all cars, the Execute button is to be pressed to produce the ALV report to show all cars registered.

Technical requirements

We now need to accommodate multiple rows in the ALV report. To do this we will be adding a status and an additional classic event block to the program as well as changing the definitions of some of the local classes.

Refer to the associated program structure diagram. Copy one of the previous versions forward to your student copy of this program and make the following changes:

- Copy GUI status %_00 from SAP-delivered program RSSYSTDB to your new version as GUI status SELECTION_SCREEN[6]. Change the status to include a new command “NEWCAR”, with description “Add new car” and icon ICON_CREATE, using Function Keys slot F5, Application Toolbar item 12 and no Menu Bar entry.

6 To do this via the ABAP Editor (SE38):

1. Select for edit the exercise program to receive the copied status.
2. Select from menu: Program > Other Object...
3. From the popup screen, select the tab for Program.
4. Specify program RSSYSTDB, select the radio button for GUI status, specify status name %_00 and press Copy.
5. In the Copy Status popup box, change the destination program by specifying your exercise program name, specify destination status SELECTION_SCREEN and press Copy.
6. On the next popup screen, providing the opportunity to change texts, simply press Copy.
7. Upon return to the editor, select from menu: Utilities > Display Object List
8. Expand the node for GUI Status
9. Select the new status SELECTION_SCREEN and apply changes as described above
10. Activate this status.

- Include a new classic event block “initialization” which will set the pf-status to SELECTION_SCREEN prior to presenting the initial selection screen.

- Change the logic in classic event block “at selection-screen” to register a car entry only when the “Add new car” button is pressed.

- Change the “car” and “navigator” classes so that each one no longer is a static class – each one now is to handle multiple instances of its own class. This means that all class members (attributes and methods) previously defined as static (CLASS-DATA; CLASS-METHODS) will need to be defined as instance members. It also means that to enable creating instances of these classes from any external entity, the qualifiers “abstract” and “create private” need to be removed from their class definition statements (or, a variation of removing the “create private” qualifier is to change it to “create public”).

- Change the “car” class to define an attribute which is a reference to its corresponding “navigator” object. In method “set_characteristics” of class “car”, create a new “navigator” object for the car into its new “navigator” reference attribute. In this way, a “navigator” object is embedded within the car object – effectively, the car “has a” navigator. The “car” class delegates all its navigational behaviors to the “navigator” object it has.

- Change the “report” class to define a new internal table of references to “car” objects along with constants for the new selection screen GUI status name and function to add a new car, both of which need to match their respective counterpart values defined for the new GUI status.

- Change the “report” class to CREATE a new car object in the car registration method, retain the reference to the new object in the new internal table of references to “car” objects, and issue a status message when a new car is registered. The status message is to indicate “Entry registered for xxx”, where “xxx” is replaced with the license plate of the car registered.

- Change the “report” class to loop through the new internal table of references to “car” objects when creating rows for the ALV report. Now one row in the report is created for each car object in the table.

This refactoring transforms the program from one using only *static* class concepts to one also using *instance* class concepts, where the “car” and “navigator” classes become abstractions for the multiple instances of cars and navigators created during execution.

Test the program to insure it still behaves as expected and now produces multiple rows in the ALV report.

4.2 Exercise 7

Program: ZOOT102B

Title: Objects 102: Car report using local class object constructor

Functional requirements

The requirement for this version is identical to the previous version – that is, the user sees no discernible difference. All changes are technical in nature.

Technical requirements

Refactoring is applied to introduce the use of the instance constructor method which can set the values of instance attributes to their initial values, eliminating the necessity for the corresponding methods performing this activity along with the need of the caller to invoke those methods.

Refer to the associated program structure diagram. Copy one of the previous versions forward to your student copy of this program and make the following changes:

- Replace the “set_heading” method of the “navigator” class and the “set_heading” and “set_characteristics” methods of the “car” class with instance constructors which accept this information and assign it during the creation of the corresponding object. Now that the constructor of class “car” will be creating a “navigator” object, its constructor must have a parameter defined for it to accept the initial heading used to set the “navigator” instance it creates.

In general, it is better to have a new instance of a class already prepared for its use upon its creation rather than requiring it to be set to some initial state by external entities through the subsequent invocation of its methods.

Test the program to insure it still behaves as expected.

4.3 Exercise 8

Program: ZOOT102C

Title: Objects 102: Car report using mix of instance and static attribs/meths

Functional requirements

The users want a new column added to the ALV report to hold the registration number assigned to the car. A registration number is to be assigned internally and is simply to be the next incremental number beyond the registration number for the previous car. Registration numbers begin with 1001.

Technical requirements

The “car” class is changed to contain a combination of both static and instance members. A new static attribute and corresponding static method for this class facilitates generating the next serial number to be used with each new car instance. Also introduced here is the use of the static constructor method which can set the value of static attribute to its initial value.

Refer to the associated program structure diagram. Copy one of the previous versions forward to your student copy of this program and make the following changes:

- Create a new static attribute for the “car” class to hold the serial number assigned to the last “car” object created.

- Create a new static constructor for the “car” class (class_constructor) to initialize the last assigned serial number to 1000.

- Create a new static method of the “car” class which can be invoked to get the next serial number to assign to a new “car” object being created.

- Change to “car” class to define a new private instance attribute to hold the serial number associated with the “car” instance.

- Change the “get_characteristics” method of the “car” class to pass the serial number of the “car” instance back to the caller.

- Change the instance constructor of the “car” class to invoke the new static method for providing the next serial number and place this into the new private instance attribute to hold the serial number.

- Change the “report” class to include in the ALV report a new column “Serial” which is to contain the serial number assigned to the car when it was registered. Change the “build_report” method of the “report” class to retrieve the serial number along with the other characteristics of the “car” instance, and include this value in the ALV report.

Test the program to insure it still behaves as expected and now includes a Serial column in the report with values different for each row.

5 Objects 103 – Inheritance

This section describes the requirements for the exercise programs associated with the chapter covering the topic of Inheritance in the book Object Oriented Design with ABAP.

5.1 Exercise 9

Program: ZOOT103A

Title: Objects 103: Vehicle report using local classes with inheritance

Functional requirements

The users now want to be able register multiple trucks along with the multiple cars. A new button is to be provided on the initial selection screen:

- Add new truck

The status message shown at the bottom of the screen with each newly registered vehicle is to indicate whether it is a car or truck:

<Car|Truck> entry registered for <license plate>

Once the user has completed registering all cars and trucks, the Execute button is to be pressed to produce the ALV report to show all cars and trucks registered. The registration number for each vehicle is to remain unique – that is, registration number 1001 is to be used only once, regardless whether it is assigned to a car or a truck. Also, the registration number assigned to a vehicle is to be higher than the registration number assigned to a previously registered vehicle, regardless whether the previous vehicle was a car or truck.

Technical requirements

We now need to accommodate registering both cars and trucks. To do this we will be renaming the “car” class as the “vehicle” class so that its attributes and methods can be used for both cars and trucks.

Refer to the associated program structure diagram. Copy one of the previous versions forward to your student copy of this program and make the following changes:

- Change GUI status SELECTION_SCREEN to include a new command “NEWTRUCK”, with description “Add new truck” and icon ICON_CREATE, using Function Keys slot F6, Application Toolbar item 13 and no Menu Bar entry.
- Change the logic in classic event block “at selection-screen” to register a truck entry only when the “Add new truck” button is pressed.
- Rename “car” class as “vehicle” class so that its attributes and methods can be used for both cars and trucks. Since “vehicle” now will be defined as a class from which other classes can inherit, remove its “final” qualifier.
- Create new definitions for classes “car” and “truck”, each of which inherits from class “vehicle”. Place the definition of the new “car” and “truck” classes one after the other between the renamed “vehicle” class and the "report" class. These require no additional attributes or methods defined for them other than what is inherited from the superclass.

- Include an internal table in the “report” class to keep track of new “truck” objects created. Model this after the one which exists for tracking “car” objects. Also create a new constant for the new function to add a new truck, the value for which needs to match the counterpart value defined in the GUI status SELECTION_SCREEN.

- Add a new method to the “report” class for registering trucks. It is to be functionally equivalent to the method used to register cars.

- Change the “report” class to CREATE a new “truck” object in the truck registration method, to hold the reference to the new object in the new internal table of “truck” objects, and to issue a message when a new truck is registered. Make certain the status messages issued when registering a car and truck specifically indicates which type of vehicle has been registered.

- Change the “report” class to include looping through the new internal table of references to “truck” objects when creating rows for the ALV report.

This refactoring transforms the program from one which uses only final classes to one where classes inherit attributes and behaviors from other classes.

At this point the 3 execution variants carried forward by copying from previous versions no longer suffice for the new capabilities added to the program. It is recommended to create 3 additional execution variants to correspond to trucks.

Test the program to insure it still behaves as expected and now produces multiple rows of both cars and trucks in the ALV report.

5.2 Exercise 10

Program: ZOOT103B

Title: Objects 103: Vehicle report using abstract method, step 1

Functional requirements

The users want the initial selection screen to be changed to include two new selection criteria:

- Weight unit
- Empty vehicle weight

Empty weight also is known by the term *tare weight*. They also want the ALV report to include these two columns in the report. The column for Weight unit is to be titled WUoM and the title for column speed unit, previously just “Unit”, is now to be changed to “SUoM” to distinguish it from the weight unit column.

Technical requirements

An abstract method is defined for the “vehicle” class and its implementation is provided by both the “car” and “truck” classes.

Refer to the associated program structure diagram. Copy one of the previous versions forward to your student copy of this program and make the following changes:

- A new public method is to be defined for the “vehicle” class for getting the gross weight. This method is to be defined as abstract, which means it is not implemented for the “vehicle” class, and it MUST BE implemented by all classes inheriting from “vehicle”. Also, now that the “vehicle” class will contain the definition for an abstract method, the entire class must be marked as abstract (the “abstract” qualifier is now required on the class definition statement).

- The constructor method of the “vehicle” class is to be adjusted to accept the two new parameters representing tare weight and weight unit.

- New attributes for tare weight and weight unit are to be defined for the “vehicle” class. Attribute tare weight is to be defined in the protected section so it can be visible to all inheriting classes. Attribute weight unit is to be defined in the private section since it does not need to be visible beyond the “vehicle” class.

- Both the “car” and “truck” class definitions are to indicate that each redefines the method to get the gross weight.

- The methods defined in the “report” class for registering cars and trucks are to be adjusted to accept the two new parameters for tare weight and weight unit. Callers to these methods similarly are to be changed to include the two new parameters.

- The ALV report layout is to be changed to include the values for gross weight and weight unit.

- Implementations are now to be defined in both the “car” and “truck” classes to get the gross weight, which, for now, simply are to return the tare weight protected attribute.

- The method in the “report” class which builds the report is to be changed to include getting the weight unit along with all the other characteristics of the “vehicle” instance as well as invoking the new method to get the gross weight. It needs to do this for both internal tables holding references to “car” and “truck” objects.

- The initial selection screen is to be changed to include new parameters for weight unit and empty vehicle weight.

This refactoring transforms the program into one which uses abstract methods, methods where the corresponding implementation is supplied by the inheriting class and not by the defining class.

It is recommended to update all execution variants to accommodate the new selection criteria.

Test the program to insure it still behaves as expected and now includes the new columns.

5.3 Exercise 11

Program: ZOOT103C

Title: Objects 103: Vehicle report using super constructor

Functional requirements

The users want the initial selection screen to be changed to include two new selection criteria:

- Cargo weight (truck only)
- Passengers (car only)

They also want the the ALV report to be changed to include in the gross weight column 1) for cars, the additional weight contributed by the number of passengers, using an average weight of 180 pounds per passenger, and 2) for trucks, the additional weight contributed by the cargo.

Technical requirements

The instance constructor of a subclass – a class defined with the “inheriting from ...” clause – needs to invoke the instance constructor of its superclass. Instance constructors are defined for the “car” and “truck” subclasses to facilitate the new information applicable to them.

Refer to the associated program structure diagram. Copy one of the previous versions forward to your student copy of this program and make the following changes:

- Both classes “car” and “truck” are to be changed so that each one now has an instance constructor defined for it. The instance constructor for the “car” class is to retain the number of passengers in a corresponding new attribute. Similarly, the instance constructor for the “truck” class is to retain the cargo weight in a corresponding new attribute. The parameters accepted by the constructors are the same parameters as accepted for the constructor of class “vehicle” with the additional parameters 1) the number of passengers for the “car” constructor and 2) the cargo weight for the “truck” constructor.

- The “car” and “truck” constructors first call their super constructor, passing the parameters it requires, then retain in their corresponding attributes their respective 1) passenger count, for “car”, and 2) cargo weight, for “truck”.

- The method to calculate the gross weight for “car” objects is to be changed to include the additional weight contributed by the number of passengers using an average weight of 180 pounds per passenger. This weight needs to be converted into the weight unit specified for the car, so include unit conversion as necessary (use function module UNIT_CONVERSION_SIMPLE).

- The method to calculate the gross weight for “truck” objects is to be changed to return the sum of the tare weight and cargo weight.

- Methods in class “report” are to be changed as necessary to accommodate accepting parameters specified on the screen and passing them through to other method calls.

- The initial selection screen is to be changed to include new parameters for cargo weight and passengers.

It is recommended to update all execution variants to accommodate the new selection criteria.

Test the program to insure it still behaves as expected and now includes the new columns.

5.4 Exercise 12

Program: ZOOT103D

Title: Objects 103: Vehicle report using abstract method, step 2

Functional requirements

The users have complained that it is too easy to mistakenly press the “Add new car” when they intended “Add new truck”, and vice versa, so users would now like to see a descriptor column added to the report indicating how the gross weight was calculated (for car or for truck).

Technical requirements

A new abstract method, defined by class “vehicle” and implemented by classes “car” and “truck”, facilitates the new descriptor requested by the users.

Refer to the associated program structure diagram. Copy one of the previous versions forward to your student copy of this program and make the following changes:

- Class “vehicle” is to be changed to include a new abstract method for getting the description of the vehicle.

- The “car” and “truck” classes are to redefine the new abstract method defined by class “vehicle”. Each class now is to include a private attribute – descriptor – to hold the string “Car” or “Truck”, respectively, and to define an implementation for the method to return the value of this attribute.

- The “report” class is now to invoke this new method of the “car” and “truck” class to get the descriptor of the corresponding row as it processes through the two internal tables holding references to cars and trucks. The ALV report layout is to be changed to include a new column “Descriptor”, a field to hold up to 15 characters.

Test the program to insure it still behaves as expected and includes the new column in the report.

6 Objects 104 – Polymorphism

This section describes the requirements for the exercise programs associated with the chapter covering the topic of Polymorphism in the book Object Oriented Design with ABAP.

6.1 Exercise 13

Program: ZOOT104A

Title: Objects 104: Vehicle report using polymorphism, step 1

Functional requirements

The requirement for this version is identical to the previous version – that is, the user sees no discernible difference. All changes are technical in nature.

Technical requirements

Refactoring is applied, taking the first step toward using polymorphism by consolidating reference variables to two different types of objects into one reference variable that can handle both types.

Refer to the associated program structure diagram. Copy one of the previous versions forward to your student copy of this program. The only changes to this version are in the “report” class.

- There are two loops in the method which builds the report: one through the internal table holding references to “car” objects and another through the internal table holding references to “truck” objects. Remove the definition of the 2 fields which receive the respective table entries, one a reference to “car”, the other a reference to “truck”, and replace them with a single field defined as a reference to “vehicle”. Both loops now are to place the corresponding “car” or “truck” object into the “vehicle” reference field. Invocations within the loop to instance methods to get characteristics, heading, speed, gross weight and description are to be changed accordingly to refer to the field holding the reference to the “vehicle”. You should find that both loops are now identical except for the name of the corresponding internal table to be processed.

This illustrates the power of polymorphism. The same field defined as a reference to “vehicle” is now holding both “car” and “truck” objects as the loops are processed. The specific methods invoked will be for the “car” or “truck” instance depending on which type of object occupies the “vehicle” reference field at the time of processing.

This concept is known as dynamic dispatch, where the method to be executed is determined at run-time based on the “dynamic type” of the object to which the “vehicle” reference field is pointing. The “vehicle” reference field itself has a “static type” of “vehicle”, meaning it can hold a reference to an object of type “vehicle” or any of the classes inheriting, directly or indirectly, from “vehicle”. Static type refers to the most general type of class the object reference variable can accommodate (in this case, “vehicle”). Dynamic type refers to the type of object actually residing in the object reference variable at the time of processing (in this case, it will be either a “car” or a “truck” object, both of which inherit directly from “vehicle”).

The variable defined as type reference to “vehicle” may be the recipient of objects of type “vehicle”, “car” or “truck”. Since the “car” and “truck” classes both inherit from “vehicle”, a reference to a “car” object or “truck” object can be moved into the “vehicle” reference field with no type checking. This follows the concept of “All dogs are animals, but not all animals are dogs” (in this case all cars are vehicles, but not all vehicles are cars).

Note: Moving values between variables defined for dissimilar abstraction levels within the same inheritance structure, where one variable is of a type of superclass to the other, such as between a reference to “vehicle” and a reference to “car”, is known as generalized casting and specialized casting. These also are known commonly by the terms widen-casting, narrow-casting , up-casting and down-casting, terms which are avoided here because there is no industry-wide agreement on which of these apply to generalized casting and which apply to specialized casting.

No type checking is necessary with generalized casting, that is, moving a reference value from the more specific variable type to the more general one, since it always will be a valid move.

With specialized casting it cannot be known until run-time whether or not the move is valid. If a variable of type reference to “vehicle” is holding a reference to a “car” object, and the value is moved to a variable of type reference to “car”, then the move will be valid, however if the “vehicle” reference variable is holding a reference to a “truck” object, then moving this to the “car” reference variable is invalid and triggers a run-time exception.

In ABAP, the validity of a specialized-casting move can be checked during execution by using the specializing cast assignment operator (MOVE … ?TO …, or … ?= …) for the move to the target reference variable. The specializing-cast move is covered later with the exercises related to the Iterator design pattern.

Test the program to insure it still behaves as expected.

6.2 Exercise 14

Program: ZOOT104B

Title: Objects 104: Vehicle report using polymorphism, step 2

Functional requirements

The requirement for this version is identical to the previous version – that is, the user sees no discernible difference. All changes are technical in nature.

Technical requirements

In this version the “report” class is changed from managing one list of “car” objects and another list of “truck” objects to managing a single list of “vehicle” objects.

Refer to the associated program structure diagram. Copy one of the previous versions forward to your student copy of this program. The only changes to this version are in the “report” class.

- Change the “report” class removing the two internal tables, one for references to “car” objects and the other for references to “truck” objects, and replace these with a single internal table of references to “vehicle” objects.

- In the “build_report” method of the “report” class, remove the loop construct through the “truck” reference table. Change the loop construct through the “car” reference table to loop through the “vehicle” reference table. In effect, this consolidates the two loop constructs through the two internal tables into a single loop construct through the single internal table.

- In the “car” and “truck” registration methods of the “report” class, remove explicit reference to the “car” and “truck” classes, replacing them with references to the “vehicle” class. In these same methods, change the CREATE OBJECT statement so that the receiving variable is of type reference to “vehicle”, and on the CREATE OBJECT statement include the TYPE clause on which is specified the actual type of object to create – “car” or “truck”.

Test the program to insure it still behaves as expected.

7 Objects 105 – Interfaces

This section describes the requirements for the exercise programs associated with the chapter covering the topic of Interfaces in the book Object Oriented Design with ABAP.

7.1 Exercise 15

Program: ZOOT105A

Title: Objects 105: Vehicle report using interfaces, step 1

Functional requirements

The requirement for this version is identical to the previous version – that is, the user sees no discernible difference. All changes are technical in nature.

Technical requirements

Refactoring is applied so that the navigator class can be accessed through an interface reference variable instead of through a class reference variable. The decision was made to apply this refactoring now because we have heard about a different type of navigation unit that could be made available to vehicles of the users, and we want to be able to handle this new unit with minimal changes to the program.

Refer to the associated program structure diagram. Copy one of the previous versions forward to your student copy of this program and make the following changes:

- Move the public members of the “navigator” class, except for its constructor method, to an interface definition for “simple_navigation”. Also move the private constant attribute holding the compass points of the “navigator” class to the interface “simple_navigation” (this now becomes a public attribute because all interface members implicitly are public). Place the definition of the new “simple_navigation” interface between the report statement and the "navigator" class (unlike a local class, which has both a definition portion and an implementation portion, a local interface consists only of a definition).

- Change the high compass offset limit variable of the “navigator” class from a constant to a static variable.

- Change the “navigator” class definition to indicate it now implements the “simple_navigation” interface. Include with this a series of aliases for each of the behaviors of the “simple_navigation” interface enabling the developer to refer to those interface names within the “navigator” class without the necessity to prefix each one of them with the interface component selector (“simple_navigation~”).

- Define a static constructor for the “navigator” class which will set the high compass offset limit static variable to the length minus 1 of the compass variable of the “simple_navigation” interface. We need to do this because the compass variable is now no longer defined within the “navigator” class (we could merely leave it set to its hard-coded value of 3, but now that it is describing the high offset of a variable no longer defined within the same class, it is safer to calculate it at run time).

Test the program to insure it still behaves as expected.

7.2 Exercise 16

Program: ZOOT105B

Title: Objects 105: Vehicle report using interfaces, step 2

Functional requirements

The requirement for this version is identical to the previous version – that is, the user sees no discernible difference. All changes are technical in nature.

Technical requirements

The new navigation unit about which we had heard finally became available to the vehicles of users. The program is changed to facilitate its use.

Refer to the associated program structure diagram. Copy one of the previous versions forward to your student copy of this program and make the following changes:

- Copy the “navigator” class to a new class named “gps”. Place the definition of the new “gps” class between the “navigator” class and the "vehicle" class. In the new “gps” class, discard the static constructor and compass offset variables; change the definition of the variable heading to type integer, rename it from heading to bearing, and include 3 new integer constants to denote 90 degrees, 180 degrees and 360 degrees.

- In the “vehicle” class, change the definition of the “navigation” unit to reference class “gps”, but retain the reference to class “navigator” as a comment.

- Implement the “gps” class methods constructor, change heading and get heading to use degrees rather than compass direction (perhaps it would be easiest to copy this from the corresponding exercise program, or at least use that as a model for implementing manually). This causes the new “gps” class to be based on the concept of *bearing* rather than *heading*. A bearing is measured using degrees of a circle, with the current bearing indicated as the number of degrees in the angle between our present direction and the north-south meridian (for instance, a heading of South is indicated by a bearing of 180 degrees).

Upon completion, you have defined a new “gps” alternative to the “navigator” class and changed the program to use it rather than the original “navigator” class.

Test the program to insure it still behaves as expected.

After testing reveals the same behavior, take the opportunity to set breakpoints in the program and watch how the methods of class “gps” operate. Then change the “vehicle” class back to using the original “navigator” class and repeat. In each case, the “simple_navigation” interface is providing the public definition used by both the “navigator” and “gps” classes. The logic within the program does not need to change to swap between them; simply changing the definition of the navigation unit variable will determine which of them is used.

7.3 Exercise 17

Program: ZOOT105C

Title: Objects 105: Vehicle report using interfaces, step 3

Functional requirements

The users have become aware that the program now is capable of handling two different types of navigation units, and they want to be able to assign a vehicle a specific navigation unit.
The initial selection screen is to be changed to provide a pair of radio buttons by which the user may select the navigation unit to apply to the vehicle:

- Equipped with basic navigation
- Equipped with GPS navigation

The ALV report is to be changed to include a new column at the end in which the description of the selected navigation unit is to appear.

Technical requirements

The program is changed to enable the user to select which of two different types of navigation devices is to be associated with a vehicle.

Refer to the associated program structure diagram. Copy one of the previous versions forward to your student copy of this program and make the following changes:

- Change the “navigator” and “gps” classes so that each one now has a publicly visible constant holding the upper-case value of the class name (type seoclsname), similar to what was done for the “car” and “truck” classes.

- Change the “vehicle” class to define a variable to hold the type of navigation unit associated with the vehicle.

- Change the get characteristics method of the “vehicle” class so that it returns the type of navigation unit.

- Change the constructor of the “vehicle” class so that it accepts parameter flags corresponding to the radio buttons denoting the selection of navigation unit. Accordingly, use these flags to determine whether to create a navigation unit of type “navigator” or of type “gps”. Save this navigation unit type in the new “vehicle” variable holding the unit type, using the new class name defined in the “navigator” and “gps” classes to indicate the type of unit. Use this identifier when creating the navigation unit object [CREATE OBJECT navigation_unit TYPE (navigation_unit_type) …]. For more robust programming, indicate that the type “navigator” is to be used by default – that is, the type of unit to be created is to be type “navigator” even when all parameter flags are blank.

- Change the definition of the navigation unit of the “vehicle” class so that now it is a reference to interface “simple_navigation”.

- Change the constructors of the “car” and “truck” classes to accept parameter flags corresponding to the radio buttons denoting the selection of navigation unit.

- Change the registration methods for car and truck in the “report” class to accept and pass through parameter flags corresponding to the radio buttons denoting the selection of navigation unit.

- Change the ALV layout to include a new column “Navigation” indicating the navigation type associated with the vehicle for that row.

- Change the initial selection screen to include two radio buttons of the same button group: one to indicate selection of basic navigator unit; the other to indicate selection of gps unit.

Upon completion, you have enabled the user to choose the type of navigation unit to assign to the registered vehicle.

It is recommended to update all execution variants to accommodate the new selection criteria.

Test the program to insure it still behaves as expected and includes the new column in the report.

7.4 Exercise 18

Program: ZOOT105D

Title: Objects 105: Vehicle report using interfaces, step 4

Functional requirements

Users are happy with the new ability to select the navigation unit for the vehicle, but they have raised the point that some vehicles do not have navigation units. They want to be able to indicate that a vehicle has no navigation unit and the direction the vehicle is heading is indeterminate. The initial selection screen is to be changed to include an additional radio button by which the user indicates no associated navigation unit applies to the vehicle:

- Equipped with no navigation

This will denote using a technique of navigation known as “dead reckoning”.

Technical requirements

The program is changed to handle a new type of navigation unit. The refactoring applied to previous versions of the program enables this new capability to be implemented easily.

Refer to the associated program structure diagram. Copy one of the previous versions forward to your student copy of this program and make the following changes:

- Copy the “navigator” class to a new class named “dead_reckoning”. Place the definition of the new “dead_reckoning” class between the “gps” class and the "vehicle" class. In the new “dead_reckoning” class, discard the static constructor and compass offset variables, define a new constant unknown_direction with a value of question mark and change the value of its class id variable accordingly.

- For new class “dead_reckoning”, define the constructor method to set its heading to the value of the unknown_direction; define the get heading method to return its heading; and define the change heading method to be empty of any statements.

- In the constructor method of the “vehicle” class, change the default navigation type from “navigator” to “dead_reckoning” and accommodate the flag indicating no navigation unit to create a navigation unit instance of class “dead_reckoning”.

- In classes “vehicle”, “car” and “truck”, change those methods which facilitate parameters for the “navigator” and “gps” flags to now accommodate the parameter for no navigation unit.

- Change the initial selection screen to include a new radio button in the existing button group to indicate selection of no navigation unit.

The design of the “dead_reckoning” class closely approximates the concept of a null object, which is covered more thoroughly in a subsequent exercise. A null object is an actual object that does nothing. Null objects are useful in object-oriented programming to simplify the implementation of other classes, so they do not need to check whether they have a reference to an object before using it, which, if that were necessary, would complicate the design. Instead, the using class of the null object simply continues to regard the null object reference like any other object, and the null object itself facilitates doing nothing. Our “dead_reckoning” class is not exactly a null object because it does more than nothing – specifically, it will set and get the unknown_direction, but this is about as close to doing nothing as we might expect.

It is recommended to update all execution variants to accommodate the new selection criteria. Each navigation unit type should be represented with one of the 3 car execution variants and one of the 3 truck execution variants.

Test the program to insure it still behaves as expected.

8 Objects 301 – Design Patterns: Singleton pattern

This section describes the requirements for the exercise programs associated with the chapter covering the topic of the Singleton design pattern in the book Object Oriented Design with ABAP. This design pattern has an object scope and a creational purpose. Intent of the Singleton pattern:

- **Singleton Pattern** – Ensures a class has only one instance, and provides a global point of access to it[7].

8.1 Exercise 19

Program: ZOOT301A

Title: Objects 301: Design Patterns: Singleton pattern, step 1

Functional requirements

The requirement for this version is identical to the previous version – that is, the user sees no discernible difference. All changes are technical in nature.

Technical requirements

Until now the report class has been defined as a static class. This is acceptable since we have no reason for there to be more than one report instance – this program produces only one report. This version shows how to make the static report class into a class enabling defining an instance and to insure that only one instance is ever defined for the life of this program execution. This is known as the Singleton design pattern.

Refer to the associated UML class diagrams. Copy one of the previous versions forward to your student copy of this program and make the following changes:

- Remove the “abstract” qualifier from the definition of the “report” class, but be certain to retain the “create private” qualifier. This will enable the class to be instantiated (because it no longer has the “abstract” qualifier) but only the “report” class will be capable of creating instances of the “report” class (because it still has the “create private” qualifier).

- In the “report” class, change the static methods to instance methods and change the static attributes to instance attributes.

- Within the methods of the “report” class, change references to the static attributes and methods now to use the self-reference selector qualifier. For example:

```
Change   loop at report=>vehicle_stack
To       loop at me->vehicle_stack

Change   call method report=>set_column_titles
To       call method me->set_column_titles
```

- In the private visibility section of the “report” class, define a new static variable as type reference to “report”. Though any name may be chosen for this variable, the name “singleton” reinforces the purpose it serves.

7 Design Patterns: Elements of Reusable Object-Oriented Software – Gamma, Helms, Johnson, Vlissides (aka GoF), p. 127.

- Define a static constructor for the “report” class which is to create a “report” object using the new static variable defined in the previous step.
- Create for the “report” class a new static public method “get_singleton_instance”, enabling external entities to obtain a reference to the singleton “report” instance.
- Create a new global variable “report” defined as type ref to report. Place this new global variable after the screen definition.
- Change classic event block “initialization” to invoke static method “get_singleton_instance” of class “report”, using the returned instance reference to populate the new global variable.
- For callers to the methods of the “report” class, change references to what had been static public methods now to use the new global variable to reference instance public methods. For example:

```
Change   call method report=>register_car_entry
To       call method report->register_car_entry

Change   call method report=>show_report
To       call method report->show_report
```

The “create private” qualifier along with the static constructor creating the instance of the report class is what guarantees there will be only one instance of the report class existing during execution.

The decision whether to define a class as static or a singleton object often will be determined by whether the class inherits from another class. When inheritance is involved, then a purely static class eliminates the option of using of polymorphism because static methods cannot be redefined.

Test the program to insure it still behaves as expected.

8.2 Exercise 20

Program: ZOOT301B

Title: Objects 301: Design Patterns: Singleton pattern, step 2

Functional requirements

The requirement for this version is identical to the previous version – that is, the user sees no discernible difference. All changes are technical in nature.

Technical requirements

With the previous version we introduced a new global variable. You may recall that during the earliest exercise programs we made quite an effort to eliminate all global variables from the program; yet with the previous version we found ourselves defining a new one. This version demonstrates how we can eliminate this new global variable as well.

The UML diagram does not change with this exercise program. Copy one of the previous versions forward to your student copy of this program and make the following changes:

- Change class "report" to move the static singleton attribute from the private section to the public section, and assign to it the "read-only" modifier, which grants external entities "read" access to this public attribute but not "write" access. Having provided public read access to this attribute, there no longer is a need for static method "get_singleton_instance", so it is to be deleted.

- For callers to the methods of the “report” class, there no longer is a reason to use the global variable to reference its instance public methods; instead, use the now publicly visible static singleton attribute of the “report” class to access these methods. For example:

  ```
  Change   call method report->register_car_entry
  To       call method report=>singleton->register_car_entry

  Change   call method report->show_report
  To       call method report=>singleton->show_report
  ```

  Note: This format of method call, where multiple class (=>) and/or instance (->) selectors are used in the same statement, represents an example of a compound reference. In this case the “report” class contains a public static attribute named singleton. The public static attributed named singleton is itself a reference to an instance of a “report” object, which provides public instance methods named “register_car_entry” and “show_report”. Accordingly, these statements are causing the public instance methods “register_car_entry” and “show_report” of a “report” class instance to be invoked through the public static attribute “singleton” of the “report” class.

- Since the singleton instance of the “report” class is now publicly visible, there no longer is a need for the global variable defined by the previous exercise, so delete it.

- Now that the global variable has been deleted, there no longer is a need to populate it, so delete from the “initialization” classic event block the call to the now-defunct method "get_singleton_instance" of the "report" class.

Test the program to insure it still behaves as expected.

9 Objects 302 – Design Patterns: Strategy pattern

This section describes the requirements for the exercise programs associated with the chapter covering the topic of the Strategy design pattern in the book Object Oriented Design with ABAP. This design pattern has an object scope and a behavioral purpose. Intent of the Strategy pattern:

- **Strategy Pattern** – Defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.[8]

The student may be pleased to find there are no new exercises associated with the Strategy design pattern because this design pattern already is being used in the exercise programs, and has been since exercise program ZOOT105C. It was that program which enabled the user to choose a navigation unit to be used with each corresponding car and truck.

The most recent exercise program includes this design pattern. The navigation unit the user chooses becomes an instance of a “simple_navigation” interface, which is defined as an attribute of the “vehicle” class. Accordingly, we say that the “vehicle” class “has a“ instance of a navigation unit. Because all the navigation unit classes – a) “navigator”; b) “gps”; c) “dead_reckoning” – implement the “simple_navigation” interface, all of them are interchangeable with each other. The code in the “vehicle” class accessing its corresponding navigation unit does so via references to the methods of the “simple_navigation” interface, and so can remain oblivious to the exact class type of the navigation unit with which it is working.

The use of the Strategy design pattern illustrates an example of using **class composition**. In this case, the “vehicle” class is “composed” with a reference to an instance of one of the three navigation unit classes. Class composition is an object-oriented design alternative to using inheritance. Without composition, we might have used inheritance to define three inheritors to the “vehicle” class – one for “vehicle with navigator”; one for “vehicle with gps”; one for “vehicle with dead_reckoning” – and would consequently have defined the “car” and “truck” classes as inheritors to these instead of inheritors to “vehicle”. Figure 9.1 illustrates these combinations using inheritance:

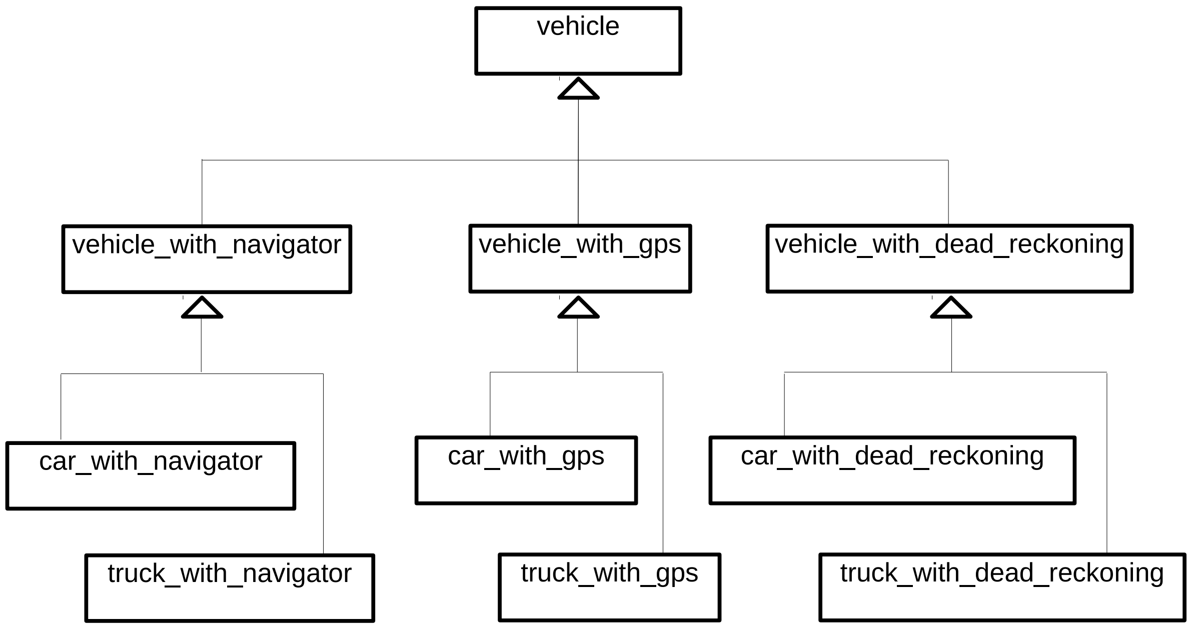


Figure 9.1

8 GoF, p. 315.

With inheritance, classes have an "*is a"* relationship between them (e.g.: a "truck_with_gps" *is a* "vehicle_with_gps", and a "vehicle_with_gps" *is a* "vehicle").

In contrast, with class composition, classes have a "*has a"* relationship between them (e.g.: a "vehicle" *has a* "gps"). With class composition, we can add a "bus" class as a subclass to "vehicle" without making any corresponding changes to the various navigation unit subclasses, and, likewise, can add new navigation unit classes without making any corresponding changes to the various "vehicle" subclasses. Indeed, our use of the Strategy design pattern makes our exercise program conform with two of the object-oriented principles:

- **Favor object composition over class inheritance**[9]

- **Program to an interface, not an implementation**[10]

Figure 9.2 shows a UML diagram illustrating these combinations using class composition:

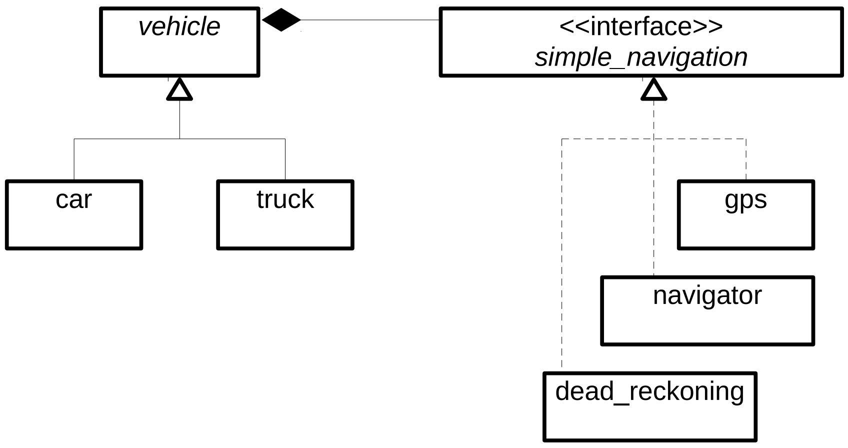


Figure 9.1

9 GoF, p. 20.

10 GoF, p. 18

10 Objects 303 – Design Patterns: Observer pattern

This section describes the requirements for the exercise programs associated with the chapter covering the topic of the Observer design pattern in the book Object Oriented Design with ABAP. This design pattern has an object scope and a behavioral purpose. Intent of the Observer pattern:

- **Observer Pattern** – Defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.[11]

The Observer pattern has been implemented into the ABAP language itself. The corresponding statements are:

- EVENTS
- METHOD … FOR EVENT …
- RAISE EVENT …
- SET HANDLER …

10.1 Exercise 21

Program: ZOOT303A

Title: Objects 303: Design Patterns: Observer pattern

Functional requirements

Users have become concerned that the weight being registered for some of their 2-axle trucks may exceed the national maximum weight limit for a 2-axle vehicle. They have asked to be notified with the number of registered trucks which exceed this 2-axle weight limit. An informational message is to be presented to the user prior to displaying the ALV report, and the message is to indicate the number of trucks exceeding the 2-axle weight limit along with the current national weight limit for such vehicles. When the number of trucks is zero, the message is to be displayed as informational; otherwise it is to be displayed as a warning.

Technical requirements

ABAP language statements are applied to facilitate one class observing the changes of another class.

Refer to the associated UML class diagrams. Copy one of the previous versions forward to your student copy of this program and make the following changes:

- Define a new class “truck_axle_weight_monitor” using the singleton pattern, with constants indicating the weight limit for 2-axle vehicles is 40,000 pounds (two attributes: one for the weight; the other for the weight unit), and an instance variable to hold the number of vehicles exceeding the 2-axle weight limit. Place the definition of the new “truck_axle_weight_monitor” class between the “truck” class and the "report" class.

- For the new class “truck_axle_weight_monitor”, define an instance method which will add 1 to the instance variable to hold the number of vehicles exceeding the 2-axle weight limit. Consider the visibility level at which this method should be defined (see more explanation following this bullet list).

11 GoF, p. 293.

- For the new class “truck_axle_weight_monitor”, define a static constructor which will create a singleton object of class “truck_axle_weight_monitor” and register (via SET HANDLER) its instance method (see previous bullet) to be invoked whenever any “truck” instance raises the “weight_exceeds_2_axle_limit” event.

- For the new class “truck_axle_weight_monitor”, define a public static method which will return the national maximum weight and unit for 2-axle vehicles.

- For the new class “truck_axle_weight_monitor”, define a public instance method which will display an informational message showing the number of vehicles exceeding the 2-axle weight limit, displayed as informational when the number of vehicles is zero, but displayed as warning when the number of vehicles is not zero.

- In the “truck” class, include an “events” statement in the public section defining an event called “weight_exceeds_2_axle_limit”.

- In the “truck” class, define a static attribute to hold the weight limit for 2-axle vehicles along with an attribute for its corresponding weight unit.

- In the “truck” class, define a static constructor which invokes the method in class “truck_axle_weight_monitor” to get the values for the maximum weight and its unit (see more explanation following this bullet list).

- In the “truck” class, define an instance method “check_axle_weight” which checks the gross weight of the truck against the maximum gross weight for 2-axle vehicles, converting between dissimilar weight units, if necessary, and raise the event “weight_exceeds_2_axle_limit” when the gross weight is found to be in excess of the limit.

- In the “end-of-selection” classic event block, invoke the new method of class “truck_axle_weight_monitor” which shows the number of vehicles exceeding the 2-axle weight limit prior to presenting the report.

In this case, class “truck_axle_weight_monitor” is the observer and class “truck” is the observed. The observed “truck” instance is oblivious to any observers it may have. It merely knows it is to raise an event when a certain situation occurs. It does not know which or even whether any observers are registered to respond to that event. Instances of other classes, and even static classes, which register as observers to the “truck” instance are said to be “loosely coupled” with the “truck” instance. This means observers will be notified once event “weight_exceeds_2_axle_limit” of the “truck” class is raised, but there is no direct method call by the “truck” instance to a method of a registered observer.

It is easy to recognize the similarities between an observer class responding to an event of an observed class and a public method of a class being invoked by an external caller. Accordingly, it may be natural to conclude that a method of the observer class responding to an external event must have public visibility. In fact, the visibility level of the responding method of the observer class is of no consequence to its ability to be invoked in response to an event raised externally. Since there is no direct call to the responding method from the class raising the event, it is not necessary for responding method to be defined in the public visibility section of its class.

In the corresponding exercise program, the responding method “add_to_over_2_axle_limit_count” of observer class “truck_axle_weight_monitor” is defined in the private visibility section. This conforms with the concept of encapsulation, where visibility should be as restricted as possible and class members should be only as visible as necessary. Unless the responding method is invoked directly by an inheriting subclass, which would justify its visibility as protected, or is invoked directly by an external caller, which would justify its visibility as public, the private visibility assigned still enables it to be invoked in response to a raised event. This is possible because the SET HANDLER statement registering class “truck_axle_weight_monitor” as an observer of class “truck” events is processed during the static constructor for class “truck_axle_weight_monitor”, and the static constructor has access to all visibility sections of its own class.

As noted above, the new static constructor of the “truck” class invokes a method in new class “truck_axle_weight_monitor” to get the values for the maximum weight and its unit. This serves an additional purpose than simply getting these corresponding values – it also causes the static constructor method of class “truck_axle_weight_monitor” to be invoked, enabling it to register itself as an observer of all “truck” instances during the processing of the “truck” static constructor (before any “truck” instances are created).

It is recommended to update at least one of the truck execution variants to accommodate exceeding the 2-axle weight limit.

Test the program to insure it still behaves as expected and now includes the new message to precede the presentation of the report.

11 Objects 304 – Design Patterns: Factory patterns

This section describes the requirements for the exercise programs associated with the chapter covering the topic of Factory design patterns in the book Object Oriented Design with ABAP.

All factory patterns deal with creating instances of classes. There are three commonly recognized variations of the factory pattern:

1. Simple Factory Pattern
2. Factory Method Pattern
3. Abstract Factory Pattern

Some books claim the **Simple Factory Pattern** is not so much a design pattern as it is a programming idiom, but it usually is included in discussions on the concept of factory patterns. We have been using the simple factory pattern since the first exercise program (ZOOT101A), to create an instance of an ALV table object (cl_salv_table=>factory).

Intents for the remaining two Factory patterns, both of which are patterns explained in the book Design Patterns: Elements of Reusable Object-Oriented Software are:

- **Factory Method Pattern** – Defines an interface for creating an object, but lets subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.[12]

- **Abstract Factory Pattern** – Provides an interface for creating families of related or dependent objects without specifying their concrete classes.[13]

Both patterns have a creational purpose. The Factory Method design pattern has a class scope while the Abstract Factory pattern has an object scope.

The exercise programs associated with this section will illustrate both the Simple Factory pattern and the Factory Method pattern.

11.1 Exercise 22

Program: ZOOT304A

Title: Objects 304: Design Patterns: Simple factory pattern

Functional requirements

The requirement for this version is identical to the previous version – that is, the user sees no discernible difference. All changes are technical in nature.

Technical requirements

Refactoring is applied to simplify creating instances of classes using the simple factory pattern.

The UML diagram does not change with this exercise program. Copy one of the previous versions forward to your student copy of this program. Implement the simple factory pattern for the “car” and “truck” classes:

12 GoF, p. 107.

13 GoF, p. 87.

- Include the qualifier “create private” for both the “car” and “truck” class definitions. This means no other entities are able to create instances of these classes other than the “car” and “truck” classes themselves.

- Define a static method “create” for both the “car” and “truck” classes, accepting the same parameters as the constructor parameters used with the CREATE OBJECT statements where the “car” and “truck” classes are created and returning an object of type reference to “vehicle”. The implementation for this new method should use the same “create object ...” statement found in the respective “register_car_entry” and “register_truck_entry” of the “report” class.

- Move the definitions for the constructor methods of the “car” and “truck” classes into the private visibility section.

- Change the register car and register truck methods of the “report” class to no longer create instances of “car” and “truck” objects but to call the respective “create” methods for this.

The “create private” qualifier appearing on a class definition insures that only the class itself can create instances of the class. Such classes always will have an accompanying public static method defined for the class by which external callers can request to have the class create an instance of itself, since otherwise there would be no way to request the class to create such an instance.

Test the program to insure it still behaves as expected.

11.2 Exercise 23

Program: ZOOT304B

Title: Objects 304: Design Patterns: Factory method pattern, step 1

Functional requirements

The requirement for this version is to notify the user that the corresponding navigation unit is calibrated and registered with each new vehicle. In the case of a gps unit, it also is to indicate that software has been downloaded to the unit.

Technical requirements

Compared to other factory patterns, the significant distinction of the factory method pattern is that subclasses determine the type of object to be instantiated. Here we will be creating 5 new classes, three of which inherit from one of the other two.

Refer to the associated UML class diagrams. Copy one of the previous versions forward to your student copy of this program and make the following changes:

- Create a new abstract class “navigation_accessories_maker” with a public method “make_navigation_unit”, protected methods “calibrate_unit” and “register_unit” and protected abstract method “create_unit”. Place the definition of this new class between the “dead_reckoning” class and the "vehicle" class.

  The method “calibrate_unit” is to have no signature and to issue an informational message “Navigation unit successfully calibrated”. The “register_unit” method is to have no signature and to issue an informational message “Navigation unit has been registered.

Thank you".

Method "make_navigation_unit" is to accept an initial heading and to return both a reference to "simple_navigation" interface and the type of navigation unit provide in that interface reference (type seoclsname). Abstract method "create_unit" is to have exactly the same signature as method "make_navigation_unit". The "make_navigation_unit" method is to invoke methods "create_unit", "calibrate_unit" and "register_unit", in that order. Because method "create_unit" is defined as abstract, its implementation is provided by a subclass. Accordingly, method "make_navigation_unit" is unaware of the type of navigation unit with which it is working.

- Create a new class "dead_reckoning_unit_maker", inheriting from class "navigation_accessories_maker" and redefining the abstract method "create_unit" of its superclass. Place the definition of this new class between the new "navigation_accessories_maker" class and the "vehicle" class. Its implementation of method "create_unit" is to set the unit type to the class id of the "dead_reckoning class", and to use the specified initial heading to create a navigation unit of that type and pass it back to the caller via the reference to interface "simple_navigation".

- Create a new class "navigator_unit_maker", inheriting from class "navigation_accessories_maker" and redefining the abstract method "create_unit" of its superclass. Place the definition of this new class between the new "dead_reckoning_unit_maker" class and the "vehicle" class. Its implementation of method "create_unit" is to set the unit type to the class id of the "navigator" class, and also to use the specified initial heading to create a navigation unit of that type and pass it back to the caller via the reference to interface "simple_navigation".

- Create a new class "gps_unit_maker", inheriting from class "navigation_accessories_maker" and redefining the abstract method "create_unit" of its superclass. Place the definition of this new class between the new "navigator_unit_maker" class and the "vehicle" class. Also, it is to redefine public method "make_navigation_unit" of its superclass, and to include an additional method "download_software_to_unit". Its implementation of method "create_unit" is to set the unit type to the class id of the "gps" class, and also to use the initial heading to create a navigation unit of that type and pass it back to the caller via the reference to interface "simple_navigation". Its implementation of method "make_navigation_unit" is to be the same as that implemented in its superclass except it is to invoke method "download_software_to_unit" between the invocations of methods "calibrate_unit" and "register_unit". Its implementation of method "download_software_to_unit" is to issue an informational message "Software successfully downloaded to navigation unit".

- Create a new static class "vehicle_accessories_store" with 3 attributes to retain references to "navigation_accessories_maker" objects – one for the "dead_reckoning_unit_maker"; one for the "navigator_unit_maker"; one for the "gps_unit_maker" – and a public method "get_navigation_unit", which is to accept an initial heading and 3 flags to correlate to the 3 types of navigation units – basic navigation, gps navigation, no navigation – that can be selected, and return both a reference to "simple_navigation" interface and the type of navigation unit it is providing in that interface reference (type seoclsname). Place the definition of this new class between the new "gps_unit_maker" class and the "vehicle" class.

  The implementation of method "get_navigation_unit" is to create whichever type of navigation unit is indicated by invoking the method "make_navigation_unit" of the respective maker of that type of navigation unit objects. When it is determined that a corresponding object does not yet exist for that maker of navigation units, then it is to create an object of that type and retain the reference in the appropriate one of the 3 attributes with reference to “navigation_accessories_maker” objects before using that reference to invoke its “make_navigation_unit” method.

- In the constructor of the “vehicle” class, remove all code related to creating a navigation unit object and instead invoke static method “get_navigation_unit” of class “vehicle_accessories_store”, passing it the relevant heading and unit type selection flags, and receiving a reference to a navigation unit object as well as the navigation unit type.

Test the program to insure it still behaves as expected and that it presents the appropriate pop-up messages when navigation units are created.

11.3 Exercise 24

Program: ZOOT304C

Title: Objects 304: Design Patterns: Factory method pattern, step 2

Functional requirements

Users have learned there is a commercial gps unit available which accommodates information about low overpasses and restricted weight limits for bridges. They want this always to be used as the navigation unit to be registered when they indicate a truck with a gps.

Technical requirements

A new type of navigation unit is made available through a new class and a new factory to produce instances of it.

Refer to the associated UML class diagrams. Copy one of the previous versions forward to your student copy of this program and make the following changes:

- Copy the definition of class “gps” to class “commercial_gps”, changing the value of the public constant class_id accordingly. The definition otherwise is identical. Place the definition of this new class between the “gps” class and the "dead_reckoning" class.

- Copy the implementation of methods constructor, “change_heading” and “get_heading” for class “gps” to class “commercial_gps”, changing any qualifiers as necessary but otherwise leaving the logic the same.

- Copy the definition of class “gps_unit_maker” to “commercial_gps_unit_maker”. Place the definition of this new class between the “gps_unit_maker” class and the "vehicle_accessories_store" class. Add a new method definition for “download_bridge_data_to_unit” with no signature.

- Copy the implementation of methods “create_unit”, “download_software_to_unit” and “make_navigation_unit” for class “gps_unit_maker” to class “commercial_gps_unit_maker”, changing any qualifiers as necessary. For method “make_navigation_unit”, include an invocation of method “download_bridge_data_to_unit” between the invocations for “download_software_to_unit” and “register_unit”.

- Create new method “download_bridge_data_to_unit” for class “commercial_gps_unit_maker” which will issue an informational message stating: “Bridge data, indicating locations of low overpasses and restricted weight limits, successfully downloaded to navigation unit".

- Change class "vehicle_accessories_store" to accommodate a reference to the "commercial_gps_unit_maker", similar to those for the other 3 unit makers.

- Change method "get_navigation_unit" of class "vehicle_accessories_store" to accept a parameter indicating the classification of the vehicle.

- Change the implementation of method "get_navigation_unit" of class "vehicle_accessories_store" to check the vehicle classification when a gps unit is requested. When the vehicle classification is for "truck" (using the public "class_id" constant defined for class "truck"; see subsequent bullet), then create a navigation unit of type "commercial_gps", which includes bridge information applicable primarily to commercial drivers to ensure safety; otherwise continue to create a navigation unit of type "gps". Be certain to check for a valid reference to an object of "commercial_gps_unit_maker", and to create one if one does not yet exist.

- Change the constructor for the "vehicle" class to accept a parameter indicating the classification of the vehicle and to pass the vehicle classification parameter to the call to create a navigation unit.

- Change the "car" and "truck" classes so that each one now has a publicly visible constant named class_id holding the upper-case value of its class name (type seoclsname). Change each of their constructors to pass to the super constructor the vehicle classification, using its own class_id constant.

Test the program to insure it still behaves as expected and that when a gps navigation unit is selected for a truck it 1) presents the appropriate extra pop-up message about bridge data and 2) the report provides the correct descriptor for a commercial gps in the column indicating the type of navigation unit.

11.4 Exercise 25

Program: ZOOT304D

Title: Objects 304: Design Patterns: Factory method pattern, step 3

Functional requirements

The users have noticed that they are getting calibration and registration messages for navigation units even when they indicate no navigation unit is to be used. They want these extraneous messages to be suppressed.

Technical requirements

The program is changed to utilize the principles of inheritance by overriding the implementation of a method provided by a superclass.

The UML diagram does not change with this exercise program. Copy one of the previous versions forward to your student copy of this program and make the following changes:

- Change the definition of the "dead_reckoning_unit_maker" to redefine inherited methods "calibrate_unit" and "register_unit".

- Add implementations for methods “calibrate_unit” and “register_unit” to the “dead_reckoning_unit_maker” class. Each one is to be an empty method – it does nothing – but it overrides the corresponding method in the super class issuing the associated informational message.

Test the program to insure it still behaves as expected and that it no longer issues informational messages about calibration and registration when the user specifies no navigation unit.

12 Objects 305 – Design Patterns: Adapter pattern

This section describes the requirements for the exercise programs associated with the chapter covering the topic of the Adapter design pattern in the book Object Oriented Design with ABAP. There are two variations of this design pattern, both of which have a structural purpose; one variation has a class scope, utilizing inheritance to provide the pattern, and the other variation has an object scope, utilizing class composition to provide the pattern. Intent of the Adapter pattern:

- **Adapter Pattern** – Converts the interface of a class into another interface the client expects. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.[14]

12.1 Exercise 26

Program: ZOOT305A

Title: Objects 305: Design Patterns: Adapter pattern, step 1

Functional requirements

The manufacturers of the commercial gps have notified their customers that they no longer will support the simple_navigation interface, and now will require users of the unit to specify their requests using a bearing number instead of a compass point. The users do not want to change the way they provide navigational information but want to continue to use the commercial gps. During discussions with the developers, the users have been told that this is still possible but that they would see a different value appear in the Navigation column of the report – it would now indicate it is a commercial gps adapter. The users have accepted this solution.

Technical requirements

The change implemented here is to continue to use the commercial gps even though the manufacturer has announced dropping its support of the “simple_navigation” interface. The solution is to use an adapter class which can accept the requests using the “simple_navigation” interface and to convert these into requests which the commercial gps can handle. This exercise is an example of an adapter designed using class inheritance.

Refer to the associated UML class diagrams. Copy one of the previous versions forward to your student copy of this program and make the following changes:

- Rename class “commercial_gps” to “commercial_gps_adapter_a”, inheriting from “commercial_gps”, changing the value of the public constant class_id accordingly. In addition, remove the constant for 360 degrees (see subsequent steps before doing this) and remove the instance attribute for bearing.

- In the implementation of method “change_heading” of the former class “commercial_gps” (now named “commercial_gps_adapter_a”) change accordingly references for the old class name to the new class name, if necessary, and add a local field to hold the bearing change (integer). In the case statement adding or subtracting degrees based on the turn type, apply this arithmetic to the new bearing change field (it will be zero upon entry to the method, so it may get a negative number). Remove the code checking for underflow and overflow, replacing it with a call to method “change_bearing” of the current class, inherited from the super class, sending it the local field holding the bearing change value.

14 GoF, p. 139.

- In the implementation of method “get_heading” of the former class “commercial_gps”, define a local field for bearing value (integer). Get the bearing by calling method “get_bearing” of the current class, inherited from the super class. Use this value to calculate the compass offset, and use that to set the corresponding heading (compass point).

- In the implementation of method constructor of the former class “commercial_gps”, define a local field for bearing value (integer). Set the bearing using the same process where previously the heading attribute was set. Use the bearing as a parameter to call the super constructor.

- Define a *new* class named “commercial_gps” which enables inheritance (that is, it is not to include the “final” qualifier on the class definition statement). Place the definition of this new class between the “gps” class and the renamed "commercial_gps_adapter_a" class. It is to have public methods constructor, “change_bearing” and “get_bearing”. The constructor method is to accept an initial bearing value. It is to have a constant defining 360 degrees (this can be taken from the class renamed in a previous step) and an attribute to hold the bearing (integer).

- For the implementation of method constructor of new class “commercial_gps”, set the bearing attribute to the value of the corresponding bearing parameter.

- For the implementation of method “change_bearing” of new class “commercial_gps”, add the value of the bearing parameter to the bearing attribute, then adjust for underflow or overflow to keep the value between zero and 359 degrees.

- For the implementation of method “get_bearing” of new class “commercial_gps”, return the bearing parameter.

- In method “create_unit” of class “commercial_gps_unit_maker”, change the reference to the class id used to set the unit type from reference to “commercial_gps” to reference to “commercial_gps_adapter_a”.

When completed, the “commercial_gps” class has been transformed to perform all of its processing based solely on bearings. The new adapter class “commercial_gps_adapter_a” enables callers to continue using the “simple_navigation” interface, which it converts into calls to the methods it inherits from class “commercial_gps” using bearing numbers it converted from the compass headings and turns provided by the callers.

Test the program to insure it still behaves as expected and now shows commercial gps adapter in the report for those rows describing trucks with gps navigation.

12.2 Exercise 27

Program: ZOOT305B

Title: Objects 305: Design Patterns: Adapter pattern, step 2

Functional requirements

The requirement for this version is identical to the previous version – however, the user will see a difference in the name presented in the Navigator column of the report.

Technical requirements

The change here is to implement an adapter for class “commercial_gps” using class composition instead of class inheritance. This will result in the new adapter class wrapping the commercial gps class, so that calls to the commercial gps class must go through the adapter.

Refer to the associated UML class diagrams. Copy one of the previous versions forward to your student copy of this program and make the following changes:

- Copy the definition of “commercial_gps_adapter_a” to “commercial_gps_adapter_b”, changing the definition statement so that it does not indicate any inheritance. Change the value of the class_id accordingly and add a new attribute adapted_gps to hold a reference to an instance of class “commercial_gps”. Place the definition of this new class between the “commercial_gps_adapter_a” class and the "dead_reckoning" class.

- Copy the implementation of “commercial_gps_adapter_a” to “commercial_gps_adapter_b”.

- In the implementation of method “change_heading” of new class “commercial_gps_adapter_b”, change references to constants defined in “commercial_gps_adapter_a” to their counterpart constants defined in “commercial_gps_adapter_b” and change the statement invoking method “change_bearing” from one invoking this method inherited from its superclass to one invoking this method of the object held in attribute adapted_gps.

- In the implementation of method “get_heading” of new “class commercial_gps_adapter_b”, make the same types of changes as were made for method “change_heading” for constants and reference to adapted_gps.

- In the implementation of method constructor of new class “commercial_gps_adapter_b”, make the same types of changes as were made for method “change_heading” for constants, and replace the call to the super constructor with a statement to create an instance of class “commercial_gps” into attribute adapted_gps.

- In method create_unit of class “commercial_gps_unit_maker”, change the reference to the class id used to set the unit type from reference to “commercial_gps_adapter_a” to reference to “commercial_gps_adapter_b”.

When completed, adapter class “commercial_gps_adapter_a”, based on class inheritance, has been abandoned and replaced by “commercial_gps_adapter_b”, based on class composition. Because class “commercial_gps_adapter_b” encapsulates an instance of class "commercial_gps" as a private attribute, it means that an instance of one class -- "commercial_gps" -- is used to compose the content of another class -- "commercial_gps_adapter_b". Stated another way, the "composition" of class "commercial_gps_adapter_b" includes an instance of class "commercial_gps". This is what is meant by "class composition". The conventional wisdom within the object-oriented programming community now considers a design based on class composition to be superior and more flexible than one based on class inheritance.

In this case, since class "commercial_gps_adapter_b" essentially “wraps” the instance of class "commercial_gps", it means that the instance of class "commercial_gps" can be accessed only by going through the instance of class "commercial_gps_adapter_b". Also, in the previous version we changed the definition statement for class "commercial_gps" to show it is no longer “final”. This is only required now because we still have the unused "commercial_gps_adapter_a" class inheriting from it. If we were to remove the definition and implementation of class "commercial_gps_adapter_a" altogether, then there would no longer be a reason for "commercial_gps" to remain defined as an inheritable class, and we could reinstate the "final" qualifier in the class definition.

Test the program to insure it still behaves as expected and now shows commercial gps adapter in the report for those rows describing trucks with gps navigation.

13 Objects 306 – Design Patterns: Decorator pattern

This section describes the requirements for the exercise programs associated with the chapter covering the topic of the Decorator design pattern in the book Object Oriented Design with ABAP. This design pattern has an object scope and a structural purpose. Intent of the Decorator pattern:

- **Decorator Pattern** – Attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.[15]

The decorator pattern uses four different types of classes as participants:

1. abstract component
2. concrete component
3. abstract decorator
4. concrete decorator

13.1 Exercise 28

Program: ZOOT306A

Title: Objects 306: Design Patterns: Decorator pattern, step 1

Functional requirements

The users have learned there are 2 vehicle options which can be applied to each vehicle:

- Vehicle locator option
- Cold climate option

Each option has a 2-character descriptor code along with a weight the option adds to the vehicle. The users want to be able to assign vehicle options to registered vehicles, and they want the gross weight in the ALV report to reflect the additional weight the option contributes as well as appending the corresponding vehicle option descriptor code to the Descriptor column in the report.

Technical requirements

The existing “vehicle” class will serve as the abstract component. The existing “car” and “truck” classes will serve as the concrete components. The abstract decorator and concrete decorator are new classes defined here.

Refer to the associated UML class diagrams. Copy one of the previous versions forward to your student copy of this program and make the following changes:

- Define a new abstract class “vehicle_option”, inheriting from class “vehicle” and redefining all its inherited methods. Place the definition of this new class between the “truck_axle_weight_monitor” class and the "report" class. It also is to have protected attributes for the decorated object (type ref to vehicle), the option weight (same type as vehicle weight), option weight unit (type msehi) and option abbreviation (same type as description for vehicle). This acts as the abstract decorator class.

15 GoF, p. 175.

- For class “vehicle_option”, the redefined methods “accelerate”, “change_heading”, “get_characteristics”, “get_heading” and “get_speed” are to pass control through to the same named methods of the decorated object.

- For redefined method “get_description” of class “vehicle_option”, pass control through to the same named method of the decorated object, then append to the returning description, separated by a comma, the 2-character abbreviation for its corresponding vehicle option.

- For redefined method “get_gross_weight” of class “vehicle_option”, pass control through to the same named method of the decorated object, then add to the returning gross weight the weight represented by the vehicle option. Be certain to accommodate conversion if the vehicle weight unit is not the same as the weight unit for the vehicle option.

- Define a new class “vehicle_option_vl”, inheriting from class “vehicle_option”. Place the definition of this new class between the “vehicle_option” class and the "report" class. It is to have a public constant class_id with the value, in upper case, of the name of its class and an instance constructor accepting a reference to type “vehicle” to be regarded as its wrapped object. It is to have private constants representing the weight of this option (student to decide), the weight unit of this option (LB), and the 2-character abbreviation for this option (VL). The constructor method is to invoke the super constructor with blank or zero values, as type appropriate, for each parameter; set the decorated object attribute using the corresponding wrapped object parameter passed to it; and set the protected fields option weight, option weight unit and option abbreviation from its private constants counterparts. This acts as a concrete decorator.

- Define a new class “vehicle_option_cc”, inheriting from class “vehicle_option”. Place the definition of this new class between the “vehicle_option_vl” class and the "report" class. It is to be a virtual copy of the class “vehicle_option_vl” changing where necessary (class_id value, abbreviation and option weight). Its constructor can be an exact copy as that for “vehicle_option_vl”. This acts as another concrete decorator.

- Change the “create” methods of both the “car” and “truck” classes to accept check-box values for these new vehicle options. Change their implementations to create an internal table of vehicle options using the names of the concrete decorator classes corresponding to the selected check boxes. A vehicle may have zero, one or two vehicle options associated with it, depending on the flags corresponding to the selected check-boxes, so the internal table of vehicle options will have up to two concrete decorator class names. After creating the “vehicle” object of type “car” or “truck”, loop through this internal table of vehicle options and create an object of the corresponding class type, sending the current “vehicle” object as the constructor's wrapped object parameter (a helper variable is necessary to achieve this; see comments in the sample exercise program). The first concrete decorator will wrap the “car” or “truck” object. The second concrete decorator will wrap the first concrete decorator. After completing the loop, one of the following will be the state:

    - There were no vehicle options specified so the “car” or “truck” object is the end result.
    - There was one vehicle option specified so the “car” or “truck” object is wrapped by the vehicle option object.
    - There were two vehicle options specified so the “car” or “truck” object is wrapped by one of these vehicle option objects, and that object is wrapped by the second vehicle option object. It does not matter the order in which the vehicle option

objects are created, so long as the innermost wrapped object is the “car” or “truck” object.

- Change the methods “register_car_entry” and “register_truck_entry” of the “report” class to accept the new parameters and pass them on through to the create methods of the “car” and “truck” classes.

- The selection screen is to be changed to include check-boxes for the following options:

    - Vehicle locator option
    - Cold climate option

  These check-box values are to be passed as parameters through the method calls as necessary.

Upon completing this exercise, the internal table defined in class report for holding references to vehicles will continue to hold such references, but now each of those references has the potential to be one of the following:

1. a “car” or “truck” object (as we had already)
2. a vehicle option object which wraps a “car” or “truck” object
3. a vehicle option object which wraps another vehicle option object which wraps a “car” or “truck” object

In those cases were vehicle option objects are involved, the attributes of the wrapped “car” or “truck” are accessible only by going through all its wrapping objects. The “car” or “truck” object becomes the final link in a chain of objects in which each link is an object inheriting from “vehicle”, and the entry registered in the internal table of reference to vehicles is the first link in that chain.

It is recommended to update some or all execution variants to accommodate the new selection criteria.

Test the program to insure it still behaves as expected and that now the gross weight of a vehicle and its descriptor is affected by any associated vehicle options. You should notice that serial numbers appearing in the report no longer are sequential when vehicle options had been specified for the entry appearing in the prior row.

13.2 Exercise 29

Program: ZOOT306B

Title: Objects 306: Design Patterns: Decorator pattern, step 2

Functional requirements

The users have learned there are yet another 6 vehicle options which can be applied to each vehicle:

- Mountainous terrain option
- Off-road operation option
- Corrosion resistance option
- Extended range option
- Chrome galore option

- Light spectacular option

They want to be able to use these along with the 2 vehicle options already implemented in the preceding version.

Technical requirements

The heavy lifting of enabling the decorator pattern already has been done. This change simply requires accommodating more vehicle options.

Refer to the associated UML class diagrams. Copy one of the previous versions forward to your student copy of this program and make the following changes:

- Change class “vehicle” to extend to at least 30 characters the size of the description type.

- Create 6 new concrete decorator classes:

    - “vehicle_option_mt” (for Mountainous Terrain)
    - “vehicle_option_oo” (for Off-road Operation)
    - “vehicle_option_cr” (for Corrosion Resistance)
    - “vehicle_option_xr” (for Extended Range)
    - “vehicle_option_cg” (for Chrome Galore)
    - “vehicle_option_ls” (for Light Spectacular)

  These last two options provide some literal context for "decorator" since many truck owner-operators take great pride in decorating their rigs with elaborate chrome enhancements and lighting arrangements. Place the definitions of these new classes in the sequence shown above between the “vehicle_option_cc” class and the "report" class. Create each one using the following process:

    - 1a) Copy the class definition of an existing concrete decorator.
    - 1b) Change the final two characters accordingly in the class name and attribute values for class_id and this_option_abbreviation (and the relevant comments).
    - 1c) Change the attribute value for this_option_weight accordingly.

    - 2a) Copy the class implementation of an existing concrete decorator.
    - 2b) Change the final two characters accordingly in the class name (and the relevant comments).

- Change the “create” methods of both the “car” and “truck” classes to accept check-box values for these new vehicle options. Change the processing of building the internal table of concrete decorator names to accommodate these new options.

- Change the methods “register_car_entry” and “register_truck_entry” of the “report” class to accept the new parameters and pass them on through to the create methods of the “car” and “truck” classes.

- The selection screen is to be changed to include check-boxes for the following options:

    - Mountainous terrain option
    - Off-road operation option
    - Corrosion resistance option
    - Extended range option
    - Chrome galore option

  - Light spectacular option

These check-box values are to be passed as parameters through the method calls as necessary.

It is recommended to update some or all execution variants to accommodate the new selection criteria.

Test the program to insure it still behaves as expected and that now the gross weight of a vehicle and its descriptor are affected by any of the new associated vehicle options.

13.3 Exercise 30

Program: ZOOT306C

Title: Objects 306: Design Patterns: Decorator pattern, step 3

Functional requirements

The users have learned there are some truck owner-operators who have been buying and applying to their rigs more than one of the vehicle options for “Light spectacular” and “Chrome galore”, so they want to have the capability to indicate multiple selections for each of the vehicle options. Those truckers! Many of them think it impossible to have too much chrome or too many lights. The users want to have the check-boxes on the selection screen for each vehicle option changed to a single digit in which they can indicate the number of times the option is to be applied to the vehicle.

Technical requirements

This change simply requires enabling each vehicle option to be specified multiple times.

The UML diagram does not change with this exercise program. Copy one of the previous versions forward to your student copy of this program and make the following changes:

- Change the “vehicle” class to define a new single-digit type to denote a vehicle option count.

- Change the “create” methods of both the “car” and “truck” classes to accept single digit values instead of check-box values for the vehicle options. Change the processing of building the internal table of concrete decorator names to accommodate these options as single digits instead of check-mark values. Replace the “if … endif” statements checking whether the option is specified with “do … enddo” statements to append the corresponding vehicle option as many times as the single digit parameter indicates.

- Change the methods “register_car_entry” and “register_truck_entry” of the “report” class to accept the new types for these parameters.

- The selection screen is changed so that each vehicle option may be specified as a single digit instead of a check-box.

Now that each of the 8 vehicle options may be selected multiple times, and, with a single digit screen parameter, each option may be selected a maximum of 9 times, it is now possible to build a decorator chain consisting of one concrete component (vehicle) and a series of 72 concrete decorators (vehicle options). Just image the possibilities if the screen parameter were to be defined as 2 digits!!

Also, consider now that the field on the report to hold the Descriptor is only so wide, and should the user indicate a combination of vehicle options causing the description to exceed this field length, some portion of it would be truncated. With the current configuration of 8 vehicle options, a single digit to indicate the number of times for an option (maximum 9), an option descriptor of 3 characters (counting its comma separator) and a longest vehicle descriptor of 5 characters ("Truck"), the field would need to be 221 characters in length to accommodate an untrucated descriptor for the longest decorator chain.

It is recommended to update some or all execution variants to accommodate the new selection criteria.

Test the program to insure it still behaves as expected and that now the gross weight of a vehicle and its descriptor are affected by any of the associated vehicle options being specified more than once.

13.4 Brain teaser #1

Let us consider expanding the numeric screen parameter field implemented in the preceding exercise from a single-digit field to a two-digit field.

1. What is the maximum number of objects that can be contained in the longest decorator chain (a single vehicle object wrapped by multiple vehicle decorator objects) when the maximum number of times a decorator can be specified is now limited to two digits?[16]
2. What would be the maximum number of characters required for a descriptor field to fully describe such a longest decorator chain, where the descriptor is to include the same information and format it already is using (option descriptor of 3 characters, counting its comma separator, and a longest vehicle descriptor of 5 characters)?[17]
3. What is the minimum modification required to implement this change?[18]

16 The formula is 1 (concrete vehicle) plus 8 (different types of concrete decorators) multiplied by 99 (number of times a single decorator can be specified): 1 + 8 * 99 = 793.

17 The formula is 8 (different types of concrete decorators) multiplied by 99 (number of times a single decorator can be specified), multiplied by 3 (the length of a single descriptor for a decorator, including its comma-separator) plus 5 (the length of the longest concrete vehicle descriptor – “Truck”): 8 * 99 * 3 + 5 = 2381.

18 We would need to make two extremely minor changes: 1) the definition of vehicle=>option_count would need to be changed from length 01 to length 02, and 2) the definition of vehicle=>description_type would need to be changed to a character field of length 2381. Depending on the number of vehicles we expect to be registered by users of this program, we might also consider changing the definition of vehicle=serial_type from num4 to num5 or num6. Getting such significant new capability by applying such trivial changes should provide convincing evidence that our efforts to use the object-oriented programming paradigm coupled with the use of design patterns greatly simplifies our maintenance efforts.

14 Objects 307 – Design Patterns: Chain of Responsibility pattern

This section describes the requirements for the exercise programs associated with the chapter covering the topic of the Chain of Responsibility design pattern in the book Object Oriented Design with ABAP. This design pattern has an object scope and a behavioral purpose. Intent of the Chain of Responsibility pattern:

- **Chain of Responsibility Pattern** – Avoids coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.[19]

Natasha, recently hired and the most experienced object-oriented programmer on our staff, has had a chance to look at the preceding version and has offered some suggestions for improving its maintainability. She found that class “vehicle_accessories_store” is doing too much work. Method “get_navigation_unit” does more than just get a navigation unit. First, it determines the type of navigation unit to get, then it establishes contact with the correct navigation unit maker, and only then does it actually get a navigation unit. She told us about a fundamental principle of object-oriented programming, known as the Single Responsibility principle:

- **A class should have only one reason to change.[20]**

She explained that the name of the method was fine, but it should do only what its name suggests it does. We had loaded it with more processing than it should have the responsibility to handle. The way we had written it, this method would need to change whenever a new type of navigation unit becomes available as well as whenever a new manufacturer of navigation unit is defined. Clearly that is more than one reason for its class to be changed.

Her recommendation was to eliminate from class “vehicle_accessories_store” all the extra processing of having the methods of the class make the determination of the type of navigation unit to get, and leave this up to the caller to resolve. Also, eliminate all the extra processing of contacting the various navigation unit makers and keeping track of each one for future reference. Here is where she suggested to define a new “agent” class which can provide this service on behalf of the vehicle accessories store, so the store can make one call to the agent, and the agent will determine the correct navigation unit maker. The agent uses the Chain of Responsibility design pattern, building a chain of navigation unit makers, where each maker knows its successor maker. The agent will pass the request to make a particular type of navigation unit to the first manufacturer. If that manufacturer does not specialize in that type of unit, then it passes the request on to its successor. This process continues until 1) one of the manufacturers proclaims it is the one capable of making that type of unit, or 2) the request reaches the final successor in the chain, which will satisfy the request by default making whatever type of unit is the specialty of that manufacturer. This guarantees that a unit maker will be located, even though it may not be for the type of unit we originally requested.

14.1 Exercise 31

Program: ZOOT307A

Title: Objects 307: Design Patterns: Chain of Responsibility pattern, step 1

19 GoF, p. 223.

20 Single Responsibility Principle; Robert C. Martin; See http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod.

Functional requirements

The requirement for this version is identical to the previous version – that is, the user sees no discernible difference. All changes are technical in nature.

Technical requirements

Refactoring is applied so that the program adheres more to the Single Responsibility principle.

Refer to the associated UML class diagrams. Copy one of the previous versions forward to your student copy of this program and make the following changes:

- Change class “navigation_accessories_maker” to include new public methods for constructor and “locate_manufacturer” as well as protected attributes for specialty and successor. The constructor is to accept a successor, a reference to type “navigation_accessories_maker”, and is to set its successor attribute using this value. Method “locate_manufacturer” is to accept a type of navigation unit and to return a navigation accessories maker, comparing its specialty against the specified type of navigation unit, and when equal, or when there is no successor, to identify itself as the handler of the request; otherwise it is to pass the request on to its successor.

- Change each class inheriting from “navigation_accessories_maker” to define a constant indicating its own class id (similar to classes “car” and “truck”) and to define a constructor method accepting a successor, a reference to type “navigation_accessories_maker”, which will, after invoking the constructor of the superclass, set the specialty attribute provided by the superclass with the corresponding type of navigation unit in which the maker specializes.

- Create new class “navigation_manufacturers_agent” which is to facilitate resolving the reference to a shop capable of producing the type of navigation unit requested. Place the definition of this new class between the “commercial_gps_unit_maker” class and the "vehicle_accessories_store" class. This class should be defined using the singleton pattern. It also should define a private attribute “first_link_in_chain” defined as type reference to “navigation_accessories_maker”. Its static constructor method should create the singleton instance. Its instance constructor method should create a chain of navigation unit manufacturers where each link in the chain holds a reference to its successor link in the chain. This chain should include one instance of each class inheriting from “navigation_accessories_maker”, with the final link to indicate the manufacturer for units of class “dead_reckoning”. Define for it method “locate_manufacturer”, accepting a navigation unit type and returning a reference to the manufacturer that can satisfy the request, and implemented to invoke the “locate_manufacturer” method of the first link in the chain of manufacturers.

- Change class “vehicle_accessories_store” so that it no longer resolves or retains references to shops capable of producing navigation units (see subsequent bullet item before doing this). Instead it is to get the reference to a shop by invoking method “locate_manufacturer” of new class “navigation_manufacturers_agent”. Also, for its method “get_navigation_unit”, remove the extraneous check-boxes and vehicle classification parameters, and now that it is both used and set, its unit type parameter is to be modified from “exporting” to “changing”.

- Change the constructor of the “vehicle” class to resolve the type of navigation unit needed (this code can be taken from class “vehicle_accessories_store”, where it was removed), and change the call to method “get_navigation_unit” of class “vehicle_accessories_store” accordingly based on the changes to its method signature.

Test the program to insure it still behaves as expected.

Note: An argument can be made that the previous implementation of the “navigation_accessories_maker” was more efficient because it did not create an instance of a unit maker class until it was determined that the instance was needed. This is known as the “lazy initialization” technique[21], where the initialization or creation of a resource is delayed until it is required, often applied to entities which are expensive to initialize or create. In contrast, the new implementation creates one class instance for every type of unit maker class, in a chain of responsibility, before it is even known whether or not each of those class instances will be used. The case for creating all of these unit maker class instances up front is based on the fact that these simple classes are not resource-intensive to instantiate and, more important, the task of maintenance is made easier by not having to add yet another class attribute and its associated processing to class “vehicle_accessories_store” to accommodate each new unit maker which may become required.

14.2 Exercise 32

Program: ZOOT307B

Title: Objects 307: Design Patterns: Chain of Responsibility pattern, step 2

Functional requirements

The users have asked for a change to allow selecting a new iPhone sextant navigator which makes use of an iPhone application. The initial selection screen is to be changed to include an additional radio button by which the user indicates the iPhone sextant navigation unit applies to the vehicle:

- Equipped with iPhone navigat`n

Technical requirements

In the preceding exercise we learned from Natasha that implementing some changes which included the Chain of Responsibility pattern would simplify our maintenance efforts for the “vehicle_accessories_store” class. No sooner have we refactored the code accordingly than we find the users asking for a change to allow selecting a new type of navigation unit. Let's see just how much our refactoring efforts have simplified this maintenance task.

Refer to the associated UML class diagrams. Copy one of the previous versions forward to your student copy of this program and make the following changes:

- Define a new class “iphone_sextant”. Place the definition of this new class between the “navigator” class and the "gps" class. It is to be an exact copy of the “gps” class, changing accordingly the value of its class_id attribute and also changing to use the “iphone_sextant” class selector where necessary.

- Change the constructor of class “vehicle” to accommodate a new check-box parameter corresponding to the iPhone Sextant navigator selection. Do this also for the “create” and constructor methods of classes “car” and “truck”, and of methods “register_car” and “register_truck” of class “report”, as well as the constructors for each of the vehicle option classes (“vehicle_option_vl”, “vehicle_option_cc”, etc.).

21 The Lazy Initialization technique is explored more thoroughly in a subsequent exercise.

- Include a new radio button on the selection screen to correspond to selecting an iPhone Sextant as the navigation unit, and include this as a parameter when invoking the “register_car” and “register_truck” methods of class “report”.

Upon completing all these changes, one thing we notice is that we made no changes to class “vehicle_accessories_store”, so Natasha was correct that the refactoring we did with the preceding version made maintenance unnecessary for this class. But this exercise also illustrated that there are many places where we pass values through method signatures so they can be passed on yet again to subsequent methods. This is known as "tramp" data, indicating that the parameter values merely are "along for the ride" with the methods through which they pass, a term presumably referencing the hobos of the 1930s who would hop freight trains for passage but would neither pay for the ride nor contribute anything consequential at the destination. So we can conclude that maintenance is indeed easier, but that there remain opportunities for continued refactoring to make it easier still.

Test the program to insure it still behaves as expected and now enables the user to make a selection to indicate iPhone Sextant as the type of navigation unit.

Note: You may have noticed that although we defined a new class for the iPhone Sextant navigation unit, we did not define a corresponding maker class. This is intentional. You should find that while a user may select the iPhone Sextant for a vehicle, the corresponding navigation unit built to satisfy that request is not an iPhone Sextant.

14.3 Exercise 33

Program: ZOOT307C

Title: Objects 307: Design Patterns: Chain of Responsibility pattern, step 3

Functional requirements

The users immediately have noticed that their requests for iPhone Sextant navigation units are resulting in Dead Reckoning navigation units being defined. They have asked to have this problem rectified.

Technical requirements

Upon investigating we found that we neglected to create a class for a manufacturer of iPhone sextant navigation units. Indeed, we have found that our "builder of last resort" – the dead reckoning manufacturer – is the maker servicing requests for iPhone sextants simply because it is the final link in the chain of responsibility and no other manufacturer preceding it in the chain is taking responsibility for the request. At least we know that *this* is working properly, but we need to resolve this oversight.

Refer to the associated UML class diagrams. Copy one of the previous versions forward to your student copy of this program and make the following changes:

- Define a new class “iphone_sextant_unit_maker”. Place the definition of this new class between the “navigation_unit_maker” class and the "gps_unit_maker" class. It is to be an exact copy of the “gps_unit_maker” class, changing the value of its class_id, changing to use the “iphone_sextant_unit_maker” class selector where necessary and changing its constructor and “create” methods to indicate the iPhone Sextant as its specialty and the corresponding unit it will create.

- Change the constructor of class “navigation_manufacturers_agent” to include the new “iphone_sextant_unit_maker” in the chain of responsibility it builds.

Upon completing all these changes, again we notice that we made no changes to class “vehicle_accessories_store”, and it was a breeze to change the “navigator_manufacturers_agent” class to accommodate the new unit maker. We now can appreciate more fully the advice we received from Natasha about organizing components for ease of maintenance.

Test the program to insure it still behaves as expected and now enables the user not only to make a selection to indicate iPhone Sextant as the type of navigation unit, but to create a corresponding unit as well.

A Pause For Reflection

Consider that since we first began refactoring this program with recommendations from Natasha, we are now seeing how multiple design patterns complement and interact with each other to offer solutions. In this case:

- We are using the **Singleton pattern** with the new navigation manufacturers agent class
- We have defined the new iPhone Sextant class to use the “simple_navigation” interface so that it can be regarded as yet another navigation unit to participate in the **Strategy pattern** of composing a vehicle class with one of the compatible navigation unit classes selected at run time
- Having established the use of the **Factory method pattern** with the “navigation_accessories_maker” class, we continue to use this with the new navigation unit maker specializing in iPhone Sextant units, which, like all other navigation unit makers, inherits from the “navigation_accessories_maker” class
- After reorganizing the navigation unit maker classes to use a **Chain of Responsibility pattern**, it was a simple matter to include the new iPhone Sextant manufacturer in that chain

This illustrates just how much the principles of object-oriented programming and the use of design patterns contributes to simplifying our maintenance efforts.

15 Objects 308 – Design Patterns: Iterator pattern

This section describes the requirements for the exercise programs associated with the chapter covering the topic of the Iterator design pattern in the book Object Oriented Design with ABAP. This design pattern has an object scope and a behavioral purpose. Intent of the Iterator pattern:

- **Iterator Pattern** – Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.[22]

Natasha has found other ways for us to refactor the program to further comply with the Single Responsibility principle. She has explained that object-oriented designers use the term “cohesion” to measure how well an encapsulation unit (a class, an interface, a module) adheres to the Single Responsibility principle. Those encapsulation units which are designed around a set of related functions are said to have *high* cohesion, while those which are designed around a set of unrelated functions are said to have *low* cohesion. Here we are using the term “cohesion” in the sense that it means “to be logically connected.”

After again looking at our program, Natasha has noted that most of the classes adhere nicely to the Single Responsibility principle, but one glaring exception remains the “report” class, which she says is responsible not only for producing the report but also for registering the vehicles. She recommended first splitting the report class into three modules:

1. a class to handle the reporting requirements
2. a class to handle the vehicle registration requirements
3. an interface to define the constants contained in the “report” class but which are not used by the “report” class or any other class

Once we separate these components accordingly, our “report” class as well as the new components now offer high cohesion. However, with our initial attempt at separating the retention of registered vehicles from the reporting of them, we will see that we have established a tight relationship between these two classes – specifically, the “report” class knows how the registered vehicles are being retained in the new class which registers those vehicles. Natasha has told us that we also should endeavor to adhere to the following object-oriented principle:

- **Strive for loosely coupled designs between objects that interact.[23]**

She explained that there is a design pattern called **Iterator**, enabling us to conceal from the “report” class the technique being used in the new class to retain registered vehicles, and that this would result in two more loosely coupled classes[24].

The Iterator design pattern makes use of the following four types of class and interface participants:

1. An abstract iterator (via interface)
2. An abstract aggregate (via interface)
3. A concrete iterator (via class)
4. A concrete aggregate (via class)

22 GoF, p. 257.

23 The statement is from Head First Design Patterns (Freeman, et al. 2004; p. 53), a paraphrase of the *Tight Coupling* cause of redesign expressed by GoF, p. 24.

24 We already had seen this concept of “loosely coupled classes” once before, when we implemented the code associated with the Observer Design Pattern, where a “truck” instance would issue state change notifications to its observers, but remained unaware which other instances were its observers.

The concrete aggregate implements the abstract aggregate interface, and it creates an instance of a concrete iterator which implements the abstract iterator interface. In effect, the concrete aggregate, implemented here by the new class responsible for registering vehicles, "aggregates" the set of "vehicle" objects to be made available to the "report" class and builds an instance of an iterator the "report" class can use to get each next item, one after the other, until the list is exhausted. Accordingly, with the Iterator design pattern the "report" class knows nothing about how the new class is managing these "vehicle" objects, and changes the relationship between these two classes from tightly coupled to loosely coupled.

15.1 Exercise 34

Program: ZOOT308A

Title: Objects 308: Design Patterns: Iterator, step 1

Functional requirements

The requirement for this version is identical to the previous version – that is, the user sees no discernible difference. All changes are technical in nature.

Technical requirements

Refactoring is applied to begin the process of purging the "report" class of those members which cause it to exhibit low cohesion.

Refer to the associated UML class diagrams. Copy one of the previous versions forward to your student copy of this program and make the following changes:

- Define a new interface "registration_screen" and move to it those constants defined in the "report" class but used solely to control activity in the classic event blocks of the program (commands of type sy-ucomm and the name of the selection screen status). Place the definition of this new interface between the "simple_navigation" interface and the "navigator" class.

- In the classic event blocks, change the corresponding class selector used to reference these constants from "report" to "registration_screen".

This step removes from the "report" class only some of the elements with which it has no cohesion. In this case, these constants have cohesion only with the classic event blocks. While they could be defined as global variables in the program, we avoid the use of global variables unless there is no alternative.

This also illustrates the use of an interface to contain attributes but no behaviors. Another option would have been to define a static class which contains only attributes and no behaviors. The decision to use an interface instead of a static class was based upon choosing the entity which offers the least capability while still supporting the visibility required to its members.

Note: The use of an interface to contain only constants is known as a Constant Interface, and is derisively regarded by some object-oriented experts as an object-oriented design anti-pattern. This negative connotation usually arises due to the Constant Interface becoming a dumping ground for many constants, some of which are unrelated to others, thus offering the interface low cohesion. Here we are collecting into a Constant Interface only those constants which relate to registration screen processing. Accordingly, it offers high cohesion due to the relationship between the constants and does not succumb to the deficiency usually ascribed to Constant Interfaces: that they offer only low cohesion between members.

Test the program to insure it still behaves as expected.

15.2 Exercise 35

Program: ZOOT308B

Title: Objects 308: Design Patterns: Iterator, step 2

Functional requirements

The requirement for this version is identical to the previous version – that is, the user sees no discernible difference. All changes are technical in nature.

Technical requirements

More refactoring is applied to continue the process of purging the “report” class of those members which cause it to exhibit low cohesion.

Refer to the associated UML class diagrams. Copy one of the previous versions forward to your student copy of this program and make the following changes:

- Define a new class “fleet_manager” which contains those attributes and members of the “report” class which have cohesion with registering vehicles, and remove these members from the “report” class (it may be easiest to copy of the “report” class definition to the “fleet_manager” and then remove what does not belong from each one). The attribute vehicle_stack in the new fleet_manager class should have its visibility raised from private to public so it can be referenced externally (see next bullet). Also, the new “fleet_manager” class should use the singleton design pattern, just like is used with class “report”. Place the definition of this new class between the “vehicle_option_ls” class and the "report" class.

- Change the “report” class method “build_report” to loop through the vehicle stack which now is defined as a public attribute in the “fleet_manager” class.

- In the classic event blocks, change the corresponding class selectors referencing moved attributes and methods from “report” to “fleet_manager”.

This step removes from the “report” class the remainder of the members with which it has no cohesion. In this case, the attribute vehicle_stack and the methods “register_car_entry” and “register_truck_entry” are moved from the “report” class to a new class specifically created for retaining and registering vehicles.

An undesirable consequence of this activity is that we now have two classes – “report” and “fleet_manager” – that are tightly coupled via attribute vehicle_stack of the “fleet_manager” class, which “report” uses and which “fleet_manager” exposes, weakening its attribute encapsulation. The weakness is not so much that the “report” class has access to the vehicle_stack attribute defined in class “fleet_manager” as it is that the “report” class is aware that this attribute is an internal table of type ref to vehicle. Accordingly, the “fleet_manager” class cannot change how it manages vehicles, to use something other than an internal table of ref to vehicle, without this change also affecting the “report” class. We will address this weakness in the next exercise.

Test the program to insure it still behaves as expected.

15.3 Exercise 36

Program: ZOOT308C

Title: Objects 308: Design Patterns: Iterator, step 3

Functional requirements

The requirement for this version is identical to the previous version – that is, the user sees no discernible difference. All changes are technical in nature.

Technical requirements

In this version we address the weakness of having the “report” class and the “fleet_manager” class tightly coupled. To resolve this, we no longer will have the “report” class directly access the vehicle_stack attribute of the “fleet_manager”, but instead the “fleet_manager” will produce an iterator object for the “report” class. The iterator object will provide the series of registered vehicles to the “report” class, one by one, until the list is exhausted.

Refer to the associated UML class diagrams. Copy one of the previous versions forward to your student copy of this program and make the following changes:

- Define a new interface named “iterator”. This represents the abstract iterator. It is to define two flags to indicate true and false, and to define two methods: “get_next” and “has_next”. The signature for “get_next” simply is to return an object of type ref to object and the signature for “has_next” is to return a flag indicating true or false to reflect whether or not there is a next object to be returned. Place the definition of this new interface between the “registration_screen” interface and the "navigator" class.

- Define a new interface named “aggregate”. This represents the abstract aggregate. It is to define a method “create_iterator” with a signature which returns an object of type “iterator”. Place the definition of this new interface between the new “iterator” interface and the "navigator" class.

- Define a new class named “fleet_iterator” which implements the “iterator” interface. Place the definition of this new class between the “vehicle_option_ls” class and the "fleet_manager" class. This constitutes the concrete iterator class. Include with this a series of aliases for each of the behaviors of the iterator interface enabling the developer to refer to those method names within the “fleet_iterator” class without the necessity to prefix each one of them with the interface component selector (“iterator~”). It is to have a private attribute vehicle_stack defined the same way as it is defined in class “fleet_manager”, and is to include attributes to keep track of the number of entries in its vehicle stack as well as the index of the next entry. Implement method “get_next” to pass back to the caller the next entry in the list, and then adjust the index to the next entry to point to its successor entry. Implement method “has_next” to indicate true or false to reflect whether or not the index to the next entry has exceeded the number of entries in the list. Define a constructor method which accepts a table of “vehicle” object references passed to it, copying this table to its own internal table and then setting the tracking attributes accordingly.

- Change class “fleet_manager” to implement the “aggregate” interface. This constitutes the concrete aggregate class. Include with this an alias for the “create_iterator” method of the “aggregate” interface enabling the developer to refer to the method name within the “fleet_manager” class without the necessity to prefix it with the interface component selector (“aggregate~”). Implement the “create_iterator” method by creating and passing back to the caller an “iterator” object containing a copy of the current vehicle_stack attribute. Also, since the “report” class now will be calling the method “create_iterator” instead of directly referencing the vehicle_stack, the vehicle_stack attribute is to be reassigned its former visibility of private.

- Change class “report” so that the “build_report” method no longer refers directly to the vehicle_stack attribute in class “fleet_manager” (indeed, once the visibility of this attribute is changed back to private, any attempt by “build_report” to reference this will cause a syntax error). In place of looping through the no-longer visible vehicle_stack, the method is first to call the “create_iterator” method of “fleet_manager”, receiving from it an “iterator” object, then loop using a “while” construct for as long as there remains a next entry to be provided by the “iterator” object. Specifically,

  Change:
  ```
        loop at fleet_manager=>singleton->vehicle_stack
           into                           vehicle_entry.
  ```

  To:
  ```
        data: fleet_iterator          type ref to iterator
            , iteration_object        type ref to object
            .
              o
              o
              o
        call method fleet_manager=>singleton->create_iterator
          receiving
            iterator                  = fleet_iterator.
        while fleet_iterator->has_next( ) eq iterator=>true.
          iteration_object            = fleet_iterator->get_next( ).
  ```

  This is the first time in any exercise program we see the use of invoking a class method as an operand of a condition (in the “while” construct).

  Also, the “iterator” object will be returning an object of type ref to object. This enables the iterator interface to be used to return to any caller any type of object[25]. We cannot reference any of the attributes or methods of the actual “vehicle” object while it is in this reference variable; it needs to be moved to the existing variable defined as type ref to vehicle (into which formerly we were looping at vehicle_stack). Moving a value from a variable defined as type ref to object into one defined as type ref to vehicle constitutes specialized casting. With specialized casting, we are moving from a reference variable offering a more general view to one offering a more specialized view (such a moving from a reference to class animal to a reference to class dog). Accordingly, it requires that we use the specializing cast assignment operator (“?=”) and embed this assignment in a try-endtry construct. If the reference of the object to be moved is compatible with the target variable, then the assignment will work. If not, then a move-cast exception is raised. A catch clause within the try-endtry construct intercepts the move-cast exception and causes processing for that object to be ignored (that is, continue with the next object). So after we get the next object from the “iterator” object, we need some code to facilitate the specialized casting move from the field of type ref to object to the field of type ref to vehicle, as well as to intercept a failure of the move and ignore that entry, such as the following:

  ```
        try.
          vehicle_entry              ?= iteration_object.
  ```

25 A variable defined as type ref to object (its static type) can hold an object of any dynamic type, but can access none of its members.

```
catch cx_sy_move_cast_error.
  continue. " with next iteration object
endtry.
```

The set of changes described for this step removes the tight coupling between the “report” and “fleet_manager” classes.

Test the program to insure it still behaves as expected.

As noted above, the signature of the “get_next” method of the “iterator” class returns a reference to type object, and we needed to implement some extra processing to perform the specializing cast necessary for us to regard the instance as a “vehicle”. We could have made this easier on ourselves here and simply defined the “get_next” signature to return a reference to type vehicle, which would have alleviated the need for the specializing cast. While that might be tempting, it would constrain our maintenance efforts when later we want to use an iterator to return something other than references to instances of vehicles, such as references to all the instances of navigation_unit_makers. Though we could do this simply by defining two iterator interfaces – one iterator for returning vehicles and another for returning navigation_unit_makers – we also would be faced with defining two aggregate interfaces to go along with them – one to create iterators for vehicles and another to create iterators for navigation_unit_makers.

15.4 Exercise 37

Program: ZOOT308D

Title: Objects 308: Design Patterns: Iterator, step 4

Functional requirements

The requirement for this version is identical to the previous version, however the user will notice that the vehicle entries in the report are now appearing in the reverse order than they were registered.

Technical requirements

In this version we demonstrate the advantage of having loose coupling between the “fleet_manager” and “report” classes as implemented in the previous version. The “fleet manager” class is changed to manage the “vehicle” objects as a linked list instead of an internal table.

Refer to the associated UML class diagrams. Copy one of the previous versions forward to your student copy of this program and make the following changes:

- Change the “vehicle” class to include a protected attribute called next, defined as type ref to vehicle. Also, define two public methods, “assign_next_in_chain” and “get_next_in_chain”, which will set or get, respectively, the new “next” attribute.

- Change the “fleet_iterator” class so that it no longer manages an internal table of “vehicle” objects, but simply has a pointer to a “next” vehicle object. Change its constructor so it no longer accepts a table of “vehicle” objects, but accepts only a single “vehicle” object which it retains as the “next” vehicle object. Change its “get_next” method to pass back to the caller the “next” vehicle object it has, then to invoke the “get_next_in_chain” on that “vehicle” object to get its successor “vehicle” object, which then becomes the new “next” vehicle object. Change its “has_next” method to check whether or not its “next” vehicle object is bound, which no longer will be the case once the “get_next_in_chain” method is invoked on the final object by the “get_next” method.

- Change the “fleet_manager” class so that it also no longer manages an internal table of “vehicle” objects, but simply has a pointer to a single “vehicle” object representing the first-in-chain. Change the implementation of the “create_iterator” method so that the “create object” statement creating the “iterator” object accepts only a single “vehicle” reference instead of a table of “vehicle” references. Change both the “register_car_entry” and “register_truck_entry” methods so that instead of appending, as before, each newly created “vehicle” object to the internal table, it now sets the next-in-chain of the newly created “vehicle” object to the first-in-chain that it has, then replaces the first-in-chain object with the new “vehicle” object, effectively placing the new “vehicle” object at the front of the chain.

The changes described for this step allow us to change the technique by which the “fleet_manger” is managing the list of vehicles while avoiding any changes to the “report” class which uses the vehicles in that list. Again we see the value in loose coupling between classes since it is simplifying our maintenance efforts by allowing the “fleet_manager” class to vary independently of the “report” class.

Test the program to insure it still behaves as expected.

15.5 Brain teaser #2

Upon noticing that the list of vehicles is now being shown in reverse order from the sequence in which they were registered, the users now are screaming that this is simply unacceptable! They want their report to work exactly the way it used to work. They are not interested with details about how *this* object-oriented principle or *that* design pattern enables them to get software changes better, faster and cheaper – Oh, no!! – they want their report to operate the same way every day until such time as *they* request a change. These irate users want the report changed back to showing the vehicles in the same sequence in which they were registered … NOW!!

There are two options we might consider for implementing this change for our users:

1. In the “build_report” method of the “report” class, where we place each entry onto the output_stack, instead of using “append” we could use “insert” and indicate to insert the new entry into index position 1, placing it ahead of all other entries on the output_stack.

2. In the “build_report” method of the “report” class, after we have placed all entries onto the output_stack, we could include a “sort” statement to sort the output_stack entries by serial number.

After everything we've learned about object-oriented programming, principles and design patterns, which of these two options do you think is the better choice? and why? Consider these alternatives and make your decision, then refer to one answer appearing in the footnote below[26].

26 Both options require the “report” class to know something about the “vehicle” entries it is getting from the “fleet_manager” class. We just spent considerable time and effort implementing the Iterator design pattern so that the “report” class has become oblivious to how the “fleet_manager” class is managing the registered vehicle objects. If we were to use option 1, then we are right back to the same problem of having the report class know something about how the “fleet_manager” is handling these objects. If the “fleet_manager” were to change again and use some other technique to feed the “vehicle” objects to the “report” class, one which now produces them in their ascending serial number sequence, then the “insert at the front of the stack” option would no longer work as it would now again cause the sequence of “vehicle” entries to appear in reverse order. Option 2 requires only that the “report” class knows something about how “vehicle” entries are created, and not the internal workings of how other classes manage their private members. Accordingly, option 2 might be the safer choice.

16 Objects 309 – Design Patterns: Template Method pattern

This section describes the requirements for the exercise programs associated with the chapter covering the topic of the Template Method design pattern in the book Object Oriented Design with ABAP. This design pattern has a class scope and a behavioral purpose. Intent of the Template Method pattern:

- **Template Method Pattern** – Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.[27]

Natasha has continued reviewing our program looking for more ways in which it can be improved. She found that the program is more difficult to maintain than is necessary due to the multiple redefinitions of method “make_navigation_unit” (in “iphone_sextant_unit_maker”, “gps_unit_maker”, and “commercial_gps_unit_maker”) where each of these redefinitions replicates all the code from the superclass method and then adds only one or two extra method calls. Specifically, each redefinition restates the call to methods “me->unit_create”, “me->calibrate_unit” and “me->register_unit”. When asked why we did this, we responded that we needed to insert some extra processing after invoking “calibrate_unit” and before invoking “register_unit”, which we were able to achieve by redefining the method.

Natasha advised that we simply needed to include a stub method, also known as a "hook", in the original method definition of the superclass, placed between the invocations of “calibrate_unit” and “register_unit”, then provide an implementation for this method in the superclass which does nothing. Any subclasses needing to provide processing after the call to “calibrate_unit” and before the call to “register_unit” simply would need to redefine the hook method and provide for it an implementation which includes only the additional processing required. This means the superclass implementation of method “make_navigation_unit” is now a "template method", one which specifies the order in which to perform a series of steps. At those locations where extra processing might be required, the template method simply provides a call to a hook method, which the subclass uses to supply the necessary extra processing. If the hook method is not redefined, then its superclass implementation is used, which is an empty method.

16.1 Exercise 38

Program: ZOOT309A

Title: Objects 309: Design Patterns: Template Method pattern

Functional requirements

The requirement for this version is identical to the previous version – that is, the user sees no discernible difference. All changes are technical in nature.

Technical requirements

Refactoring is applied to reduce the replication of code.

The UML diagram does not change with this exercise program. Copy one of the previous versions forward to your student copy of this program and make the following changes:

- Change the "navigation_accessories_maker" class to include a new protected section method called "prepare_unit_for_installation", which has no signature. Change its definition of method "make_navigation_unit" to include the “final” qualifier, preventing subclasses from overriding it, and change its implementation of this method to include a

27 GoF, p. 325.

call to method "prepare_unit_for_installation" after the call to “calibrate_unit” and before the call to “register_unit”. Its implementation of method "prepare_unit_for_installation" is to contain no statements.

- Discard the redefinitions of method "make_navigation_unit" from classes “iphone_sextant_unit_maker”, “gps_unit_maker”, and “commercial_gps_unit_maker”, and instead for each class include an entry in the protected section for redefining method "prepare_unit_for_installation". Also for each of these classes, replace the overriding implementation of "make_navigation_unit" with an override for method "prepare_unit_for_installation", which is to contain only those statements that are additional when compared to the implementation of method “make_navigation_unit” in the superclass.

These changes enable us to avoid the unnecessary replication of redefinitions for overridden methods. Test the program to insure it still behaves as expected.

17 Objects 310 – Design Patterns: Command pattern

This section describes the requirements for the exercise programs associated with the chapter covering the topic of the Command design pattern in the book Object Oriented Design with ABAP. This design pattern has an object scope and a behavioral purpose. Intent of the Command pattern:

- **Command Pattern** – Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.[28]

With the command pattern, we acquire a reference to an object which "knows" how to apply some action upon yet another object. The caller does not need to know what the action is or how it is implemented, but simply knows that the action can be invoked by using the object which contains the "command".

17.1 Exercise 39

Program: ZOOT310A

Title: Objects 310: Design Patterns: Command pattern, step 1

Functional requirements

The users have requested the ability to reapply the last turn applied to the most recently registered vehicle. A new button is to be provided on the initial selection screen:

- Repeat last turn

Upon reapplying the last turn, an informational message is to be issued indicating “Last turn <turn> repeated for <license plate of vehicle>”, where <turn> is to indicate one of the valid turns (L, R, U) and <license plate of vehicle> is to indicate the license_plate attribute of the most recently registered vehicle.

Technical requirements

This exercise introduces the Command design pattern to facilitate the new user request.

Refer to the associated UML class diagrams. Copy one of the previous versions forward to your student copy of this program and make the following changes:

- Change GUI status SELECTION_SCREEN to include new function TURNAGAIN, with text "Repeat last turn" and icon ICON_SYSTEM_REDO, using Function Keys slot Shift-F11, Application Toolbar item 14 and no Menu Bar entry.
- Change the interface "registration_screen" to include a new constant for command repeat_last_turn. The value needs to match the counterpart value defined in the GUI status SELECTION_SCREEN.
- Create a new interface named "command" which contains only a method named "execute" with no signature. Place the definition of this new interface between the “aggregate” interface and the "navigator" class.
- Define a new class named "vehicle_turn_command" which implements the "command" interface and includes for the “execute” method of the “command” interface an alias called “execute”, so the method may be invoked without using the interface selector

28 GoF, p. 233.

(command~execute). Place the definition of this new class between the “vehicle” class and the "car" class. It is to include an instance constructor method which can accept both a “vehicle” reference and a last turn indication. It should have attributes for both the direction of the turn it is to apply to a “vehicle” and the reference to the corresponding “vehicle” to which the turn is to be applied. Its implementation of the constructor method is to accept the “vehicle” reference and its last turn indication and place these into its corresponding attribute fields defined to hold these values. Its implementation of the method "execute" is to invoke the "change_heading" method of the corresponding “vehicle” it holds in its “vehicle” reference attribute, then get the corresponding license plate of that vehicle to use to issue an informational message to the user indicating "Last turn T repeated for LLLLLL", where "T" and "LLLLLL" are replaced by the turn direction and license plate respectively.

- Change the "fleet_manager" class to include a new method named "repeat_last_turn" as well as a new attribute to hold a reference to a "command" interface, which will be used to facilitate repeating the last turn of the vehicle. Both the methods "register_car_entry" and "register_truck_entry" are to be changed so that after the last user-specified turn has been applied to the corresponding “vehicle”, it creates an object of type "vehicle_turn_command" into its new attribute for repeating the last turn, using both the last turn and the new “vehicle” as parameters to the constructor for the "vehicle_turn_command" instance. The new method "repeat_last_turn" simply is to invoke the "execute" method of the instance of class "vehicle_turn_command" it holds.

- Modify the classic status definition to include a new command to indicate repeating the last turn. The classic event "at selection-screen" is to be changed to accept the command for repeating the last turn and, when this command is issued, to invoke method "repeat_last_turn" of the singleton “fleet_manager”.

Test the program to insure it still behaves as expected.

Note: You may have noticed that the user can press the "Repeat last turn" button before having created an instance of a “vehicle”. If that were to occur, then the call to method “execute” of object “vehicle_turn_command” would be interrupted with an exception for CX_SY_REF_IS_INITIAL and the associated short dump would indicate "Access via 'NULL' object reference not possible." We will address this weakness in a subsequent exercise.

17.2 Exercise 40

Program: ZOOT310B

Title: Objects 310: Design Patterns: Command pattern, step 2

Functional requirements

The users now have requested the ability to reverse the last turn applied to the most recently registered vehicle. A new button is to be provided on the initial selection screen:

- Reverse last turn

Upon reapplying the last turn, an informational message is to be issued indicating “Last turn <turn> reversed for <license plate of vehicle>”, where <turn> is to indicate one of the valid turns (L, R, U) and <license plate of vehicle> is to indicate the license_plate attribute of the most recently registered vehicle.

Technical requirements

This exercise continues with the Command design pattern to facilitate the latest user request.

The UML diagram does not change with this exercise program. Copy one of the previous versions forward to your student copy of this program and make the following changes:

- Change GUI status SELECTION_SCREEN to include new function UNDOTURN, with text "Reverse last turn" and icon ICON_SYSTEM_UNDO, using Function Keys slot Shift-F12, Application Toolbar item 15 and no Menu Bar entry.
- Change the interface "registration_screen" to include a new constant for command reverse_last_turn. The value needs to match the counterpart value defined in the GUI status SELECTION_SCREEN.
- Change the interface "command" to include a new method named "undo" with no signature.
- Change class "vehicle_turn_command" to include for the “undo” method of the “command” interface an alias called “undo”, so the method may be invoked without using the interface selector (command~undo). Its implementation of the method "undo" is, like with method "execute", to invoke the "change_heading" method of the corresponding “vehicle” it holds in its “vehicle” reference attribute, then get the corresponding license plate of that vehicle to use to issue an informational message to the user indicating "Last turn T reversed for LLLLLL", where "T" and "LLLLLL" are replaced by the turn direction and license plate respectively. The significant difference between "undo" and "execute" is that the "undo" method is to determine the reverse of the last turn command and apply that to the vehicle. The reverse only needs to be applied to a left or right turn, since undoing a U-turn is the same as doing a U-turn. To do this, build a translation string holding the sequence of the turns for left, right, right and left, in that order, using the constants from the “simple_navigation” interface. Then place the last turn attribute for the instance into a local variable and perform a translation on it, so its value left changes to right and right changes to left.
- Change the "fleet_manager" class to include a new method named "reverse_last_turn", which simply is to invoke the "undo" method of the instance of class "vehicle_turn_command" it holds.
- Modify the classic status definition to include a new command to indicate reversing the last turn. Change the classic event "at selection-screen" to accept the command for reversing the last turn and, when this command is issued, to invoke method "reverse_last_turn" of the singleton “fleet_manager”.

Test the program to insure it still behaves as expected.

Note: This exercise exhibits the same weakness identified for the previous exercise. We will address this weakness in a subsequent exercise.

17.3 Exercise 41

Program: ZOOT310C

Title: Objects 310: Design Patterns: Command pattern, step 3

Functional requirements

The users now have found a weakness in the actions for “Repeat last turn” and “Reverse last turn”. Specifically, they can request “Reverse last turn” indefinitely, reversing more turns than had been originally applied to the vehicle. They would like the “Reverse last turn” request to be limited to the number of times the “Repeat last turn” had been requested, with extraneous requests for “Reverse last turn” ignored.

Technical requirements

This exercise introduces checking into the Command design pattern.

The UML diagram does not change with this exercise program. Copy one of the previous versions forward to your student copy of this program and make the following changes:

- Change class "vehicle_turn_command" to include a counter attribute, which is incremented once each time the "execute" method is invoked and decremented once each time the "undo" method is invoked. In the "undo" method, check that the counter is greater than zero before applying the corresponding activity.

Test the program to insure it still behaves as expected.

18 Objects 311 – Design Patterns: Null Object pattern

This section describes the requirements for the exercise programs associated with the chapter covering the topic of the Null Object design pattern in the book Object Oriented Design with ABAP. Intent of the Null Object pattern:

- **Null Object Pattern** – Provide an object as a surrogate for the lack of an object of a given type. The Null Object Pattern provides intelligent do nothing behavior, hiding the details from its collaborators[29].

Natasha has continued reviewing our program looking for more ways in which it can be improved. She finally discovered the potential execution interrupt and accompanying short dump awaiting the hapless user who requests the "Repeat last turn" or "Reverse last turn" prior to having registered the first vehicle. Upon having it brought to our attention, we suggested fixing this by changing both the "repeat_last_turn" and "reverse_last_turn" methods of the "fleet_manager" class, adding to each one some conditional code to check for a valid object bound to the vehicle_last_turn_command reference field. Natasha suggested a different approach using the Null Object design pattern.

The Null Object design pattern establishes an object which does nothing. In this case, a null object is defined which implements the "command" interface. Both the "execute" and "undo" method implementations in the null object do nothing – they contain no executable statements. Then, during execution of the constructor method of the "fleet_manager" class, a null object is created to satisfy its reference field "vehicle_last_turn_command". Accordingly, this field always will contain a valid reference to an object even before the user registers the first vehicle, so when methods "repeat_last_turn" and "reverse_last_turn" invoke their respective "execute" and "undo" methods of the null object, no activity takes place (including no execution interrupt and no accompanying short dump). Creating a null object is an alternative to implementing conditional code within the methods "repeat_last_turn" and "reverse_last_turn" to check for the existence of a valid reference in field "vehicle_last_turn_command".

18.1 Exercise 42

Program: ZOOT311A

Title: Objects 311: Design Patterns: Null Object pattern

Functional requirements

Some users have found that pressing either the "Repeat last turn" or "Reverse last turn" command buttons before registering the first vehicle will cause the program to be interrupted and cause a short dump. They want this to be corrected.

Technical requirements

This exercise introduces the Null Object design pattern as an alternative to using conditional logic.

Refer to the associated UML class diagrams. Copy one of the previous versions forward to your student copy of this program and make the following changes:

- Create a new class "null_command" which implements the "command" interface. Place the definition of this new class between the "vehicle_turn_command" class and the "car" class. Include for the "execute" and "undo" methods of the "command" interface aliases

29 Unlike all of the other design patterns presented so far, the Null Object design pattern is not included amongst those covered in the book Design Patterns: Elements of Reusable Object-Oriented Software – Gamma, Helms, Johnson, Vlissides , 1995, Addison-Wesley. This definition of this design pattern is taken from http://www.oodesign.com/null-object-pattern.html.

called “execute” and “undo”, so these methods may be invoked without using the interface selector (command~execute, command~undo). It is to have no class members other than the methods implicitly acquired by implementing the "command" interface. Its implementation for both the "execute" and "undo" methods are to be empty methods.

- Change class "fleet_manager" to include an instance constructor which will create in field "vehicle_last_turn_command" a null object of type class "null_command". Its methods "repeat_last_turn" and "reverse_last_turn" can have those comments removed which call attention to the problem resolved by the null command object.

Test the program to insure it still behaves as expected. Also, try pressing the “Repeat last turn” and “Reverse last turn” command buttons before registering the first vehicle.

19 Objects 312 – Design Patterns: State pattern

This section describes the requirements for the exercise programs associated with the chapter covering the topic of the State design pattern in the book Object Oriented Design with ABAP. This design pattern has an object scope and a behavioral purpose. Intent of the State pattern:

- **State Pattern** – Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.[30]

The users have been making more requests for changes to the program. Now they want the ALV report to include command buttons by which they may refresh the screen to update the distance traveled, to enable applying turns to selected vehicle rows and to indicate for some vehicles that they have stopped, have changed their speed, and other assorted actions the users consider helpful.

Although there are 12 exercise programs associated with this design pattern, the first 7 of them deal only with implementing into the program changes requested by the users, setting the stage for illustrating how the state design pattern can simplify the new processing.

19.1 Exercise 43

Program: ZOOT312A

Title: Objects 312: Design Patterns: State pattern, step 1

Functional requirements

The users have requested a change to the report to include a trip odometer column which can show the distance each vehicle has traveled since being added to the fleet of vehicles appearing in the report, along with a Refresh command appearing on the ALV report button bar by which the user can update the distance traveled for each vehicle.

Technical requirements

We now need to facilitate action commands issued from the ALV report. To do this we will be adding another GUI status to the program.

Refer to the associated UML class diagrams. Copy one of the previous versions forward to your student copy of this program and make the following changes:

- Copy GUI status SALV_STANDARD from SAP-delivered program SALV_DEMO_TABLE_FUNCTIONS to your new version as GUI status REPORT_SCREEN. Replace its existing Function keys and Menu Bar entries for function MYFUNCTION (Shift-F10) with function REFRESH, accompanied by text “Refresh” and icon ICON_REFRESH, using Application Toolbar item 19 (also replacing the entry for function MYFUNCTION).
- Create a new interface "report_screen" to contain the constants associated with the new GUI status REPORT_SCREEN, similar to how interface registration_screen contains the constants associated with GUI status SELECTION_SCREEN. Place the definition of this new interface between the “registration_screen” interface and the "iterator" interface. The value for the constant representing the Refresh command needs to match the counterpart value defined in the GUI status REPORT_SCREEN.

30 GoF, p. 305.

- Change class “vehicle” to define public types for trip odometer (type p length 7 with 3 decimal positions) and timestamp (type timestamp). Change the constructor method to accept a parameter timestamp representing when the vehicle started moving, with default zero. Define a private attribute to retain the timestamp provided to the constructor. Define new public method “get_distance_traveled” to calculate the distance the vehicle has traveled since it began moving, the formula for which is:

    - distance = current speed * number of seconds traveling / number of seconds in 1 hour

  For instance, a vehicle traveling at 50 kilometers per hour which has been traveling for 30 minutes (1800 seconds) would have calculated for it a distance traveled of 25 kilometers:

    - 25 kilometers = 50 kilometers per hour * 1800 seconds traveling / 3600 seconds in 1 hour

  Use standard SAP class cl_abap_tstmp to perform timestamp arithmetic.

- Change the constructor methods for both the “car” and “truck” classes to get the current timestamp and pass this value along on the call to the super constructor.

- Change the “report” class to include new columns in the report layout: one for the new trip odometer value and another for a reference to the vehicle object (a column that by default will be excluded from the display of the report). Define two new private methods:
    - Method “on_user_command” is to respond to the event “added_function” of class “cl_salv_events”[31], invoking method “refresh” when the command triggering the event is the Refresh command. Its signature is to accommodate importing parameter e_salv_function defined by the “added_function” event.
    - Method “refresh” is to facilitate refreshing the trip odometer value for each report row by invoking the “get_distance_traveled” method for each row, then invoking the “refresh” method of the ALV grid object once.

  Change the definition of the ALV object reference from the local variable defined in method “present_report” to a private attribute of the class, a change made necessary by the processing within method “refresh”, which needs access to the ALV object reference. Change method “build_report” to place the vehicle object reference retrieved from the iterator directly into the report row, and remove the local variable that previously had received this value. Change method “present_report” to enable the new GUI status REPORT_SCREEN to appear with the ALV report and to enable responding to user events raised by ALV, as shown in the following snippet of code:

```
    data         : grid_events    type ref
                                    to cl_salv_events_table
                 .
    o
    o
*   Enable report buttons:
    me->alv_grid->set_screen_status(
      pfstatus                   = report_screen=>report_status_name
      report                     = sy-repid
      set_functions              = me->alv_grid->c_functions_all
```

31 Recall that *events* is the way the Observer design pattern has been implemented directly into the ABAP language itself. Here we are indicating that our instance of “report” is to be registered as an observer of changes to class “cl_salv_events”, specifically for the changes arising from event “added_function” available to that class. Accordingly, when the user clicks on an added function to the ALV report instance, class “cl_salv_events” will raise the “added_function” event, at which point all registered observers for that event, which will include our “report” class, will be notified of the event through a call to the method which had been registered to be invoked when that event was triggered (in our case, method “on_user_command” of the “report” instance).

```
                                       ).
*   Enable responding to user command:
    grid_events                  = me->alv_grid->get_event( ).
    set handler me->on_user_command for grid_events.
*   Display alv grid:
    o
    o
```

Also, though not necessary, change within method “present_report” those references to the ALV object so each one now includes the “me->” self reference prefix, reinforcing that the visibility of this reference has been changed from local method variable to private attribute of the class. Change method “set_columns” to facilitate the new trip odometer column.

Test the program to insure it still behaves as expected and will cause distance to be updated with each press of the Refresh command button, **but in doing so do not provide any vehicle options for any of the vehicles to appear in the report**.

Note: This exercise contains a bug. It would cause a program interruption (short dump) should any one of the vehicles appearing in the report have any vehicle options provided for it and the user presses the Refresh command button. This bug is fixed in the the next exercise.

19.2 Exercise 44

Program: ZOOT312B

Title: Objects 312: Design Patterns: State pattern, step 2

Functional requirements

The users have discovered the bug introduced by the preceding version of the program – specifically, they pressed the Refresh button on the report which contained rows for vehicles which included vehicle options, and program execution was interrupted with a short dump. They want this fixed immediately.

Technical requirements

We have found that the problem does not occur when none of the vehicles include any vehicle options. This gives a clue that the presence of vehicle options is a contributing factor. After some investigation with the help of Natasha, we found that we were allowing vehicle decorator instances to perform distance calculations, but found during debugging that their timestamp value, representing when the vehicle started moving, was set to zero. The "get_distance_traveled" method of class "vehicle" attempts to calculate the distance traveled using the number of seconds between the current date/time and the year 0000, month 00, day 00. Accordingly, the program encounters an unhandled CX_PARAMETER_INVALID_RANGE exception.

The problem here is that we did not consider that the Decorator design pattern also was being used. Class "vehicle_option", a class inheriting from class "vehicle" and facilitating our implementation of the Decorator design pattern, redefines all the public methods inherited from "vehicle" that control the vehicle (accelerate, change_heading) or that enable us to get attribute information about the vehicle (get_characteristics, get_heading, etc.) and passes control from the concrete decorators through to the concrete component. We did not accommodate this with the new public method “get_distance_traveled" we defined for class "vehicle" in the previous exercise.

The UML diagram does not change with this exercise program. Copy one of the previous versions forward to your student copy of this program and make the following changes:

- Change class “vehicle_option” to redefine public method “get_distance_traveled" and to pass control on to the next inner “vehicle” concrete object, similar to the redefined implementation for method “get_speed”.

Test the program to insure it still behaves as expected, and now should be able to facilitate refreshing the ALV report even when it includes rows representing vehicles which contain vehicle options.

Finally we have encountered our first programming problem contributed by the use of design patterns. We need to consider that using design patterns presents a double-edged sword: they help us in solving design problems but they also expose us to the pitfalls of their misuse. After this sobering lesson, we recognize the need to be cautions in how we approach the maintenance of programs which contain implementations for multiple design patterns.

19.3 Exercise 45

Program: ZOOT312C

Title: Objects 312: Design Patterns: State pattern, step 3

Functional requirements

The users have requested the ability to change the direction of travel assigned to each vehicle appearing in the report. They want new command buttons “Turn left” and “Turn right” to appear on the ALV report button bar which can be applied to selected rows.

Technical requirements

A technique is implemented by which turns can be applied to vehicles appearing in the report.

The UML diagram does not change with this exercise program. Copy one of the previous versions forward to your student copy of this program and make the following changes:

- Change GUI status REPORT_SCREEN to include new function TURNLEFT, with text "Turn left" and icon ICON_ARROW_LEFT, using Function Keys slot F9 and Application Toolbar item 20; and new function TURNRIGHT, with text "Turn right" and icon ICON_ARROW_RIGHT, using Function Keys slot Shift-F2 and Application Toolbar item 21. Both of these functions are to appear in the Menu Bar ahead of the entry for the List > &RNT_PREV function.

- Change interface “report_screen” to include two constants to facilitate the commands to Turn left and Turn right. These need to match the counterpart values defined in the GUI status REPORT_SCREEN.

- Change class "report" method "present_report" to enable row selection buttons to appear for each row shown in the ALV grid, using the following snippet of code:

```
*   Set grid selection mode to rows and columns:
    me->alv_grid->get_selections( )->set_selection_mode( cl_salv_selections=>row_column ).
```

Define new private method "turn" to facilitate changing the headings of selected vehicle rows, invoking the "change_heading" method on each vehicle instance, obtaining each vehicle instance directly from the selected ALV grid row, then to show the updated heading values in the report via an invocation of the "refresh" method. Issue an informational message when no rows have been selected. Identify the selected rows using the following snippet of code:

```
*   Get selected rows of grid:
    selected_rows_stack = me->alv_grid->get_selections( )->get_selected_rows( ).
```

Prior to invoking the "refresh" method, reset the selected rows using the following snippet of code:

```
    clear selected_rows_stack.
    me->alv_grid->get_selections( )->set_selected_rows( selected_rows_stack ).
```

Note: Each of the snippets of code provided above illustrates a technique known as a *chained method call*. In a chained method call scenario, the object returned from a preceding method call provides the object against which the next method call is invoked. It avoids not only multiple statements to facilitate the multiple method calls but also eliminates the need to define helper variables to receive the objects returned for each of the leading method invocations.

Use of the chained method call technique is controversial. It facilitates a method calling sequence which violates another design principle known as the Principle of Least Knowledge, also known as the Law of Demeter[32]. The Law of Demeter suggests that objects should not reach through a called object to access yet another object since this suggests that the object doing the reaching knows something about the structure of the object through which it is reaching. The Law of Demeter is not violated so long as the invoking method invokes only those methods belonging to:

- its own instance
- an object referenced by an attribute of its own class or instance
- an object passed to the method through its signature
- an object the method instantiates

- Change method "refresh" of class "report" to retrieve the current heading of each vehicle, which could have been changed by the new Turn left or Turn right commands. Change method "on_user_command" of class "report" to intercept the commands for Turn left and Turn right and to invoke the new "turn" method, passing a parameter indicating whether the user had selected to turn left or right.

Test the program to insure it still behaves as expected and provides the user with the capability of applying turns to selected vehicle rows.

19.4 Exercise 46

Program: ZOOT312D

Title: Objects 312: Design Patterns: State pattern, step 4

Functional requirements

32 See http://www.ccs.neu.edu/home/lieber/LoD.html.

The users have requested a change to the report to include a new column indicating the state of the vehicle, with each vehicle defaulting to “cruising” state when the report is first presented. They want new ALV report button bar commands for “Stop”, which can be applied to selected vehicle rows to change the vehicle state from “cruising” to “stopped”, and “Resume”, which also can be applied to selected vehicle rows to change the vehicle state from “stopped” to “cruising”.

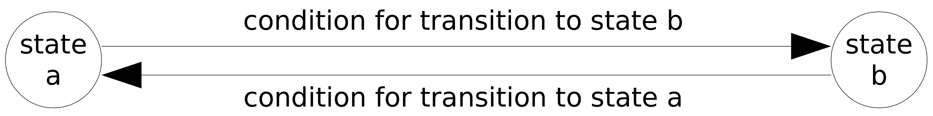
This is the first program where we are introducing the concept of “state” to the vehicles. Prior to this request the vehicles always were moving. Now the vehicles can be in either one of 2 states – “stopped” and “cruising”. The commands “Stop” and “Resume” constitute transition commands to change a vehicle from one state to the other.

The users have presented the following chart describing the transitions between these states:

Current state	**Transition command**	**New state**
cruising	Stop	stopped
stopped	Resume	cruising

States and their transitions often are depicted in what is known as a *state diagram*[33], as shown in the following diagram, where:

- a *state* is represented by a circle containing the description of the state
- a *state transition* is represented by a single-headed arrow line connecting the state circles between which a transition is valid from the old state to the new state
- a *transition condition* is represented by a word or phrase to indicate what must occur for a transition from one state to another, and appears close to the state transition connecting line with which it is associated


condition for transition to state b
state a
state b
condition for transition to state a

For this new user request, refer to the associated state diagram.

Technical requirements

This exercise introduces the concept of *state* to the vehicles appearing in the report.

The UML diagram does not change with this exercise program. Copy one of the previous versions forward to your student copy of this program and make the following changes:

- Change GUI status REPORT_SCREEN to include new function STOP, with text "Stop" and icon ICON_BREAKPOINT,using Function Keys slot Shift-F4 and Application Toolbar item 22; and new function RESUME, with text "Resume" and icon ICON_RELEASE, using Function Keys slot Shift-F5 and Application Toolbar item 23. Both of these functions are to appear in the Menu Bar ahead of the entry for the List > &RNT_PREV function.

33 See https://www.ee.usyd.edu.au/tutorials/digital_tutorial/part3/t-diag.htm for more information.

- Change interface “report_screen” to include two constants to facilitate the commands for Stop and Resume. These need to match the counterpart values defined in the GUI status REPORT_SCREEN.
- Change class “vehicle” to include a new public type defined for current state (character, length 16) and constants to represent state cruising, with value "cruising", and state stopped, with value "stopped". Change its constructor to accept a parameter to represent the initial state, with default of space. Define a private attribute to hold the current state of the vehicle and set it to the initial state parameter value accepted via the constructor. Define two new public methods to "get_current_state" and "set_current_state" which can get or set, respectively, the new private vehicle state attribute.
- Change the constructors for both the "car" and "truck" classes to pass to the super constructor an initial value of "cruising" for the current state.
- Change class "vehicle_option" to include redefinitions for inherited methods "get_current_state" and "set_current_state", each of which is to pass control on to the next inner “vehicle” concrete object.
- Change class "report" to include a new column in the ALV output row layout to show the current state value. Change method "build_report" to get the current state of a vehicle and place this value into the new column of the ALV report row being built for the vehicle. Change method "set_column_titles" to accommodate the title for the new ALV column for current state. Change method "refresh" to get the current state of each vehicle. Define new methods “resume” and “stop” which will invoke the “set_current_state” method on the corresponding vehicle instance for all selected rows, setting the current state to “cruising” or “stopped”, respectively, then to show the updated state values in the report via an invocation of the “refresh” method. Change method "on_user_command" to intercept the commands for Stop and Resume, passing control to the corresponding new methods facilitating these actions. Model the implementations of the new “stop” and “resume” methods using the implementation for the “turn” method.

Test the program to insure it still behaves as expected and provides the user with the capability of changing the state of selected vehicle rows.

19.5 Exercise 47

Program: ZOOT312E

Title: Objects 312: Design Patterns: State pattern, step 5

Functional requirements

The users have noticed that upon pressing the Refresh command button, vehicles in state “stopped” are continuing to accumulate distance traveled as though they were still cruising, and they also are able to apply turns to stopped vehicles. They want stopped vehicles to be prohibited from turning. They also want the Refresh command to be applicable only to vehicles in state “cruising” – stopped vehicles are to continue to show their distance traveled until reaching the “stopped” state, and to begin again accumulating distance traveled upon transitioning to the “cruising” state.

Technical requirements

Additional conditional logic is introduced for checking the state of a vehicle to determine whether continuing to log additional distance traveled is applicable.

The UML diagram does not change with this exercise program. Copy one of the previous versions forward to your student copy of this program and make the following changes:

- Change class “vehicle” to include a new private attribute to record the total distance traveled before stopping, and new public methods "resume" and "stop". Method "resume" will reset the timestamp at which the vehicle begins moving again and set the current state to "cruising", while the "stop" method will record the total distance traveled before stopping then set the current state to "stopped". Existing method "get_distance_traveled" now will check the state of the vehicle, returning only the distance traveled before stopping when the vehicle is in state "stopped", and including the distance traveled before stopping when calculating the distance traveled since the resumption of movement when the vehicle is in state "cruising".

- Change class "vehicle_option" to include redefinitions for inherited methods "resume" and "stop", each of which is to pass control on to the next inner “vehicle” concrete object.

- Change class "report" methods “resume” and “stop” to no longer directly invoke the “set_current_state” method on the corresponding vehicle instance, but now to invoke the corresponding “vehicle” methods “resume” and “stop”, respectively. Change method “turn” so that it will ignore applying a turn to a vehicle unless it is in the “cruising” state.

Test the program to insure it still behaves as expected and will no longer enable users to apply turns to stopped vehicles or to continue accumulating distance traveled when in the stopped state.

Notice that we have implemented conditional logic in methods of the vehicle class to check whether processing is applicable.

19.6 Exercise 48

Program: ZOOT312F

Title: Objects 312: Design Patterns: State pattern, step 6

Functional requirements

The users have noticed that although vehicles in state “stopped” no longer are accumulating any distance traveled with Refresh, they still are showing a non-zero speed. They want these vehicles to show a speed of zero when in the “stopped” state.

The users also have requested a change to the report whereby they can indicate that a vehicle is in a state of heavy traffic, accompanied by a new ALV report button bar command for “Slow” which can be applied to selected vehicle rows to change their vehicle states from “cruising” to “in heavy traffic”, accompanied by a decrement in the speed by half.

Now that there will be 3 states, along with the respective commands to cause transitions from one state to another, the combinations of state transitions has become more complicated. The users have presented the following chart describing the valid transitions between the 3 states:

Current state	Transition command	New state

cruising	Slow	in heavy traffic
cruising	Stop	stopped
in heavy traffic	Stop	stopped
in heavy traffic	Resume	cruising
stopped	Resume	cruising
stopped	Resume	in heavy traffic

Notice there are 2 transitions from state stopped via the Resume command. The previous state the vehicle had before being assigned the stopped state will determine the new state applicable with the Resume command.

Refer to the associated state diagram.

Technical requirements

More conditional logic is introduced to facilitate the new state requested by the users.

The UML diagram does not change with this exercise program. Copy one of the previous versions forward to your student copy of this program and make the following changes:

- Change GUI status REPORT_SCREEN to include new function SLOW, with text "Slow" and icon ICON_PS_WBS_ELEMENT, using Function Keys slot F7, Application Toolbar item 24, and to appear in the Menu Bar ahead of the entry for the List > &RNT_PREV function.

- Change interface “report_screen” to include a constant to facilitate the command Slow. This needs to match the counterpart value defined in the GUI status REPORT_SCREEN.

- Change class "vehicle" so that now it includes a new state constant defined to describe "in heavy traffic". Define new private attributes to retain both the previous state and the previous state speed. Define a new public method "slow" which will change the current state to "in heavy traffic" and decrement the current speed by half. Change methods "resume", "slow" and "stop" to change the values of attributes previous state and previous state speed, as well as resetting the current state and current speed. Change method "resume" further so that it takes into consideration the current state when resetting the speed and the next state, using the previous state and previous state speed attributes when the current state is”stopped”, or simply doubling the speed when the current state is “in heavy traffic” (since we reduced speed by half to get to a state of "in heavy traffic"). Change method "get_distance_traveled" to recognize "in heavy traffic" as a valid state for calculating current distance.

- Change class "vehicle_option" to include a redefinition for inherited method "slow",which is to pass control on to the next inner “vehicle” concrete object.

- Change class "report" to define new method “slow” which will invoke the “slow” method on the corresponding “vehicle” instance for each selected row, obtaining each “vehicle” instance directly from the selected ALV grid row, then to show the updated state values in the report via an invocation of the “refresh” method. Model the implementation of the new “slow” method using the implementation for the “stop” method. Change method "on_user_command" to intercept the command for Slow, passing control to the corresponding new method facilitating this action. Change method "refresh" to get the value for the current speed. Change method "turn" to recognize "in heavy traffic" as a valid state for making a turn.

Test the program to insure it still behaves as expected and provides the user with the capability of changing the state of selected vehicles according to the state transition matrix.

Notice that we have implemented more conditional logic in methods of the “vehicle” class to check whether or not processing is applicable.

Note: This exercise contains bugs. A user could mark a vehicle row and press the Stop button for an entry which already is in state "stopped". Subsequent attempts to mark the row for Resume will have no effect. Similar problems occur when command buttons "Slow" and “Resume" are pressed and the vehicle already is in that state. These problems are fixed in the next exercise.

19.7 Exercise 49

Program: ZOOT312G

Title: Objects 312: Design Patterns: State pattern, step 7

Functional requirements

The users have discovered problems when selecting to apply to a row the same vehicle state to which the vehicle already is set. They found in some cases the status values were being cleared and in other cases it was impossible to apply a subsequent state transition to a selected row. They want this fixed.

Technical requirements

Additional conditional logic is introduced for checking the current state of the vehicle before applying a command intended to change its state.

The UML diagram does not change with this exercise program. Copy one of the previous versions forward to your student copy of this program and make the following changes:

- Change methods "resume", "slow" and "stop" of class "vehicle" so that the current state is checked upon entering the method, and when the current state is not compatible with the command associated with the method, then ignore the request.

Test the program to insure it still behaves as expected and no longer enables users to encounter problems when selecting to apply to a selected vehicle a state transition command which is not valid for the current state of the vehicle.

Notice that we have implemented even more conditional logic in methods of the “vehicle” class to check whether or not processing is applicable.

19.8 Exercise 50

Program: ZOOT312H

Title: Objects 312: Design Patterns: State pattern, step 8

Functional requirements

The requirement for this version is identical to the previous version – that is, the user sees no discernible difference. All changes are technical in nature.

Technical requirements

We have found ourselves adding conditional logic to methods of the “vehicle” class to facilitate the processing associated with the transition of a vehicle from one state to another. We have become concerned that requests by the users for more vehicle states and the combinations of valid transitions between those states could proliferate this conditional logic to an unmanageable level. We have discussed this with Natasha, who agrees and has suggested we consider using the State design pattern, which avoids the need for all the conditional logic in the “vehicle” class methods we have implemented so far as well and will alleviate the need to implement any new conditional logic with a request for a new state and its transition matrix. We have decided that now would be a good time to refactor the code to eliminate the conditional logic scattered throughout the methods of the “vehicle” class, before we get a request from the users for yet another new vehicle state.

Refer to the associated UML class diagrams. Copy one of the previous versions forward to your student copy of this program and make the following changes:

- Create new interface "state", which takes the responsibility away from class "vehicle" for defining types "current_state_type" (which is renamed "description_type" for the "state" interface) and "odometer_type", and methods "get_distance_traveled", "resume", "slow" and "stop". Place the definition of this new class between the “command” interface and the "navigator" class. Define new method "get_description", which will return to the caller the description of the state (and is not to be confused with the abstract method "get_description" defined for class "vehicle") and method "turn", which is defined exactly the same way as the definition for method "turn" of class "report".

- Change class "vehicle" to remove from its pubic visibility section those members now defined in the “state” interface, including the definitions for methods "get_distance_traveled", "resume", "slow" and "stop", as well as the constants defining the state descriptions for cruising, in heavy traffic and stopped. Completely remove the implementations for methods "get_distance_traveled", "resume", "slow" and "stop”, as these are to be taken over by the classes implementing the "state" interface (see next bullet). Change its attributes for current state and previous state now to be defined as references to the "state" interface, change its attributes distance traveled before stop and previous state speed now to be defined in terms of types provided by the "state" interface, and define 3 new attributes as references to the "state" interface, one for each of the current states - cruising, in heavy traffic, and stopped. Change its constructor method no longer to accept an initial state, but instead to create into the 3 new references of “state” attributes instances for each of the 3 states cruising, in heavy traffic and stopped, and then set its own current state to its cruising state. Change signatures of methods "get_current_state" and "set_current_state" to accepting references to a "state" interface. Define new methods to get and set its attributes for previous state, time started moving, distance traveled before stop, and previous state speed. In addition, define 3 new methods for getting references to instances of classes which implement the “state” interface, one for each of the current states - cruising, in heavy traffic and stopped.

- Define three new classes which implement the state interface:
  - "cruising_state"
  - "in_heavy_traffic_state"
  - "stopped_state"

Place the definitions of these new classes in the sequence shown above between the “vehicle” class and the "vehicle_turn_command" class. Each one is to contain:

- a constant, of type seoclsname, containing its own class id in upper case
- aliases for all of the methods contributed by the "state" interface
- an attribute to represent the instance of the vehicle to which it is associated
- an attribute holding a descriptor of the state it represents, taking this definition away from the “vehicle” class
- an instance constructor accepting a reference to a class "vehicle", to be put into the corresponding new attribute.
- an implementation for the "get_description" method to retrieve the corresponding state descriptor value

All method definitions other than the constructor are provided by the "state" interface, and only those methods applicable to its state are to have any code in their implementations. All other methods are to be left as empty methods, which when invoked will do nothing. For example, the "turn" method is not applicable to the "stopped_state" class, so this method remains an empty implementation in that class. For those methods which have code, remove from them the conditional logic to determine whether or not the action is applicable; this now is implicitly applicable by the fact that code for the action has been implemented in a specific state class.

- Define new class "cruising_state" with an implementation for its method "get_distance_traveled" which is basically a copy of that method implementation from the "vehicle" class, but changed it to invoke methods "get_time_started_moving", "get_dist_traveled_before_stop" and "get_speed" of the instance of its "vehicle" class to retrieve these respective values it uses in calculating the distance traveled. Its implementations for "slow" and "stop" similarly are to be copies of the "slow" and "stop" methods of the "vehicle" class, also adjusted to get from and set to the instance of its "vehicle" class the values for distance traveled before stop, current speed, previous state speed and time started moving, as well as invoking the "accelerate", "get_<next>_state" and "set_current_state" methods of its "vehicle" instance. Its implementation for "turn" is to invoke the "change_heading" method of its "vehicle" instance.

- Define new class "in_heavy_traffic_state" similarly to class "cruising_state", but whereas "cruising_state" has a null implementation for the "resume" method, class "in_heavy_traffic_state" is to have an implementation for method "resume" which is a copy of the "resume" method of the "vehicle" class, minus all the conditional logic, and adjusted similarly to the way methods "slow" and "stop" where adjusted for class "cruising_state". Its implementations for methods "get_distance_traveled", "stop" and "turn" are to be identical to those of class "cruising_state". Its "slow" method is to be a null implementation.

- Define new class "stopped_state" similarly to class "cruising_state", but provide null implementations for methods "slow", "stop" and "turn". Its implementation for method "resume" is to be similar to that for class "in_heavy_traffic_state" but does not need to set the distance traveled before stop, setting the current speed from the previous state speed and resetting the previous state speed to zero, all by invoking methods of its "vehicle" instance. Also, its implementation for method "get_distance_traveled" merely is to return the corresponding value it gets from method "get_dist_traveled_before_stop" of its "vehicle" instance, and does not need to adjust this value by any subsequent distance traveled since then.

- Change classes "car" and "truck" so their constructors no longer send a default state on the call to their super constructor.

- Change class "vehicle_option" to remove its redefinitions for methods "get_distance_traveled", "resume", "slow" and "stop" (since these have been removed from the “vehicle” class) and provide new pass-through methods for those new attribute getter and setter methods defined for the “vehicle” class.

- Change class "report" to define its state description and trip odometer fields in terms of types provided by the "state" interface. Change its "build_report" method to get a reference to the current vehicle state object from which it retrieves the description of the state to be placed into the report. Change its "refresh" method to get the current vehicle state object so that it can get both the current state description as well as the current trip odometer value. Change its methods "resume", "slow", "stop" and "turn" now to retrieve the current vehicle state object and use it to facilitate the action, relying on the state object to allow or disallow the corresponding action. These four methods now will have nearly identical implementations, with the exception of their name and the name of the corresponding method invoked on the state object.

Test the program to insure it still behaves as expected.

Notice that all the conditional logic formerly included in methods of the vehicle class for checking whether or not processing is applicable has been eliminated.

19.9 Exercise 51

Program: ZOOT312I

Title: Objects 312: Design Patterns: State pattern, step 9

Functional requirements

The requirement for this version is identical to the previous version – that is, the user sees no discernible difference. All changes are technical in nature.

Technical requirements

We noticed with the previous exercise program that there was code replicated between the three state classes. We have decided to consolidate this replicated code into a single class from which the three state classes can inherit.

Refer to the associated UML class diagrams. Copy one of the previous versions forward to your student copy of this program and make the following changes:

- Define new class "vehicle_state", using class "cruising_state" as a model, elevating the "vehicle" attribute to protected visibility and adding a new protected attribute "descriptor" which will hold the value of the class description. Define protected methods "halt" and "calculate_distance_traveled" into which the implementation for methods "stop" and "get_distance_traveled", respectively, are to be copied from class "cruising_state". Move to this new class the code for public method "get_description" from class "cruising_state", making it available to all inheriting classes, but change it to provide the value from the protected attribute "descriptor". Move to this new class the code for public method "get_distance_traveled" from class "stopped_state", where it will represent the default implementation unless overridden in one of the subclasses. The implementations for those methods provided by the "state" interface, other than "get_description" and "get_distance_traveled", are to have no code, effectively rendering these methods to null actions. Place the definition of this new class between the “vehicle” class and the "cruising_state" class.

- Change classes "cruising_state", "in_heavy_traffic_state" and "stopped_state now to inherit from class "vehicle_state", discarding from each their private attribute "vehicle", now defined as a protected attribute in the super class, as well as their interface and aliases statements, since these are now covered by their superclass. In addition, each class is to redefine those methods provided by the state interface which would require action other than the default null action inherited from the super class, providing the implementation for the method in the subclass itself, as "cruising_state" does for its method "slow", or to invoke the protected methods of the super class in those cases where the implementation has been migrated there, as method "get_distance_traveled" of class "cruising_state" does by invoking protected method "calculate_distance_traveled” and method “stop” of class "in_heavy_traffic_state" does by invoking protected method "halt” . Also, the constructors of these three classes now are to set the protected attribute "descriptor" to the value of their private attribute "description", in addition to invoking the superclass constructor.

Test the program to insure it still behaves as expected.

This is the first exercise program in which we have encountered an example where a superclass implements an interface.

19.10 Exercise 52

Program: ZOOT312J

Title: Objects 312: Design Patterns: State pattern, step 10

Functional requirements

The users have requested a change to the report whereby they can indicate that a vehicle is out of service, accompanied by a new ALV report button bar command “Place out of service” which can be applied to selected vehicle rows to change their vehicle states from “stopped” to “out of service”.

Now there are 4 states and 4 transition commands. The users have updated the chart describing the valid transitions between these states:

Current state	**Transition command**	**New state**
cruising	Slow	in heavy traffic
cruising	Stop	stopped
in heavy traffic	Stop	stopped
in heavy traffic	Resume	cruising
stopped	Resume	cruising
stopped	Resume	in heavy traffic
stopped	Place out of service	out of service

Refer to the associated state diagram.

Technical requirements

Although a new state requested by the users is introduced, the only new conditional logic to facilitate it is in the single class to handle the new command to transition a vehicle to that state. Accordingly, the preceding implementation of the State design pattern has simplified this change.

Refer to the associated UML class diagrams. Copy one of the previous versions forward to your student copy of this program and make the following changes:

- Change GUI status REPORT_SCREEN to include new function PLACE_OOS, with text "Place out of service" and icon ICON_DEFECT, using Function Keys slot F8, Application Toolbar item 25, and to appear in the Menu Bar ahead of the entry for the List > &RNT_PREV function.
- Change interface “report_screen” to include a constant to facilitate the command Place out of service. This needs to match the counterpart value defined in the GUI status REPORT_SCREEN.
- Change interface “state” to define new method “place_out_of_service”.
- Change class "vehicle" to define a new private attribute for “out_of_service_state” and define new public method "get_out_of_service_state" which will return the reference to the state interface held in the “out_of_state” attribute. Change the constructor to instantiate an object into the “out_of_service_state” new private attribute.
- Change class "vehicle_state" to include an alias for the "place_out_of_service" method contributed by the "state" interface, and implement this as a null method.
- Change class "stopped_state" to include a redefinition for method "place_out_of_service" and provide an implementation which first will save the current state by invoking method “set_previous_state”, then will reset the current state to the out of service state.
- Define new class "out_of_service_state" similarly to class "stopped_state", but it is to have only a constructor method defined for it, modeled after the constructor for “stopped_state”. Place the definition of this new class between the “stopped_state” class and the "vehicle_turn_command" class.
- Change class "vehicle_option" to include a redefinition for inherited method "get_out_of_service_state", which is to pass control on to the next inner “vehicle” concrete object.
- Change class "report" to define new method “place_out_of_service”, similarly to its implementation for the "stop" method, invoking the “place_out_of_service” method on the corresponding “vehicle” instance for each selected row, obtaining each vehicle “instance” directly from the selected ALV grid row, then showing the updated state values in the report via an invocation of the “refresh” method. Change method "on_user_command" to intercept the command for Place out of service, passing control to the corresponding new method facilitating this action.

Test the program to insure it still behaves as expected and provides the user with the capability of changing the state of selected vehicles according to the state transition matrix.

19.11 Exercise 53

Program: ZOOT312K

Title: Objects 312: Design Patterns: State pattern, step 11

Functional requirements

The users have requested a change to the report whereby they can indicate that a vehicle is any one of 3 new states:

- being towed
- in shop
- available

along with 5 new transition commands to cause the respective state change:

- Maintain
- Tow
- Repair
- Make available
- Start

The following chart describes the corresponding state transition matrix:

Current state	Transition command	New state
cruising	Slow	in heavy traffic
cruising	Stop	stopped
in heavy traffic	Stop	stopped
in heavy traffic	Resume	cruising
stopped	Resume	cruising
stopped	Resume	in heavy traffic
stopped	Place out of service	out of service
stopped	Make available	available
out of service	Tow	being towed
out of service	Maintain	in shop
being towed	Repair	in shop
in shop	Make available	available
available	Start	cruising

Refer to the associated state diagram.

Technical requirements

More new states are introduced, their implementations made simpler due to the presence of the State design pattern facilitating processing for the current states.

Refer to the associated UML class diagrams. Copy one of the previous versions forward to your student copy of this program and make the following changes:

- Change GUI status REPORT_SCREEN to include the following new functions:
    - TOW, with text "Tow" and icon ICON_TRANSPORT, using Function Keys slot Ctrl-Shift-F8 (occupied by &AQW) and Application Toolbar item 26.
    - REPAIR, with text "Repair" and icon ICON_TOOLS, using Function Keys slot Ctrl-Shift-F7 (occupied by &VEXCEL) and Application Toolbar item 27.
    - MAINTAIN, with text "Maintain" and icon ICON_TEST, using Function Keys slot Ctrl-Shift-F11 (occupied by &GRAPH) and Application Toolbar item 28.
    - MAKE_AVAIL, with text "Make available" and icon ICON_ACTIVATE, using Function Keys slot Ctrl-Shift-F2 (occupied by &VLOTUS) and Application Toolbar item 29.
    - START, with text "Start" and icon ICON_FOREIGN_KEY, using Function Keys slot Ctrl-Shift-F1 (occupied by &EB9) and Application Toolbar item 30.

  All of these functions are to appear in the Menu Bar ahead of the entry for the Goto > BACK function.

- Change interface “report_screen” to include constants to facilitate the commands Maintain, Make available, Repair, Start and Tow. These need to match their counterpart values defined in the GUI status REPORT_SCREEN.

- Change interface “state” to define new methods "maintain", "make_available", "repair", "start" and "tow".

- Change class "vehicle" to define 3 new private attributes for "being_towed_state", "in_shop_state" and "available_state", and define new public methods "get_<new_state_name>_state" which return the reference to the state interface held in these respective attributes. Change the constructor to instantiate an object into each of these new private attributes.

- Change class "vehicle_state" to include aliases for the new "make_available", ”maintain", "repair", "start", and "tow" methods contributed by the "state" interface, and implement these as a null methods.

- Change class "stopped_state" to include a redefinition for method "make_available" and provide an implementation which will reset the current state to the available state.

- Change class "out_of_service_state" to include redefinitions for methods "maintain" and "tow", and provide each with an implementation which will reset the current state to the in shop state or being towed state, respectively.

- Define new class "available_state" similarly to class "stopped_state", to have a constructor method and a redefinition for the "start" method, which is to cause a transition from the available state to the cruising state with an initial cruising speed of 5 speed units. Place the definition of this new class between the “out_of_service” class and the "vehicle_turn_command" class.

- Define new class "being_towed_state" similarly to class "stopped_state", to have a constructor method and a redefinition for the "repair" method, which is to cause a transition from the being towed state to the in shop state. Place the definition of this new class between the “available_state” class and the "vehicle_turn_command" class.

- Define new class "in_shop_state" similarly to class "stopped_state", to have a constructor method and a redefinition for the "make_available" method, which is to cause a transition from the in shop state to the available state. Place the definition of this new class between the "being_towed_state" class and the "vehicle_turn_command" class.

- Change class "vehicle_option" to include redefinitions for inherited methods "get_available_state", "get_being_towed_state" and "get_in_shop_state", each of which are to pass control on to the next inner "vehicle" concrete object.

- Change class "report" to define 5 new methods:
    - "maintain"
    - "make_available"
    - "repair"
    - "start"
    - "tow"

  Each of these is to be implemented similarly to the "place_out_of_service" method, invoking the same named method of the current state object of the corresponding "vehicle" instance for each selected row, obtaining each "vehicle" instance directly from the selected ALV grid row, then showing the updated state values in the report via an invocation of the "refresh" method. Change method "on_user_command" to intercept the commands for Maintain, Make available, Repair, Start and Tow, passing control to the corresponding new methods facilitating these action.

Test the program to insure it still behaves as expected and provides the user with the capability of changing the state of selected vehicles according to the state transition matrix.

19.12 Exercise 54

Program: ZOOT312L

Title: Objects 312: Design Patterns: State pattern, step 12

Functional requirements

The users have requested a change to the report whereby they can indicate that a vehicle is under police escort, along with new transition command "Assign police escort", applicable only to vehicles in state "cruising", with an accompanying increase in speed by a factor of 1.35. While in the state "police escort", the command Stop will cause a transition to state "stopped". Vehicles in state "stopped" which had transitioned to "stopped" from "police escort" will transition back to state "police escort" with the Resume command.

The users also have requested a change to the report enabling a change in speed for selected vehicle rows for which the current vehicle state is any of the moving states ("cruising", "in heavy traffic", "police escort"), accompanied by 4 new ALV report button bar commands for the following:

- Decelerate by 5
- Decelerate by 1
- Accelerate by 1
- Accelerate by 5

The users have specified that when a vehicle speed has been changed while it is in either the "in heavy traffic" or "police escort" states, resumption to the "cruising" state is to use the current speed, and not the previous state speed, as a baseline for determining the cruising speed. This means that a transition from state "police escort" to state "cruising" is to cause the current speed to be decreased by a factor of 1.35 (the same factor by which it was increased upon transitioning to the “police escort” state), and a transition from state “in heavy traffic” to state “cruising” is to cause the current speed to be doubled.

The following chart describes the corresponding state transition matrix:

Current state	**Transition command**	**New state**
police escort	Resume	cruising
police escort	Stop	stopped
cruising	Assign police escort	police escort
cruising	Slow	in heavy traffic
cruising	Stop	stopped
in heavy traffic	Stop	stopped
in heavy traffic	Resume	cruising
stopped	Resume	cruising
stopped	Resume	in heavy traffic
stopped	Resume	police escort
stopped	Place out of service	out of service
stopped	Make available	available
out of service	Tow	being towed
out of service	Maintain	in shop
being towed	Repair	in shop
in shop	Make available	available
available	Start	cruising

Refer to the associated state diagram.

Technical requirements

Yet another new state is introduced along with new commands to change the speed of vehicles in moving states, all of which is simplified by having the State design pattern in place.

Refer to the associated UML class diagrams. Copy one of the previous versions forward to your student copy of this program and make the following changes:

- Change GUI status REPORT_SCREEN to include the following new functions:
    - DECEL05, with text "-05" and icon ICON_CHANGE_NUMBER, using Function Keys slot Shift-F6 (occupied by &CRB) and Application Toolbar item 31.
    - DECEL01, with text "-01" and icon ICON_CHANGE_NUMBER, using Function Keys slot Shift-F7 (occupied by &CRE) and Application Toolbar item 32.
    - ACCEL01, with text "+01" and icon ICON_CHANGE_NUMBER, using Function Keys slot Ctrl-F2 (occupied by &CRL) and Application Toolbar item 33.
    - ACCEL05, with text "+05" and icon ICON_CHANGE_NUMBER, using Function Keys slot Ctrl-F3 (occupied by &CRR) and Application Toolbar item 34.

  - ESCORT, with text "Assign police escort" and icon ICON_ALARM, using Function Keys slot Shift-F1 (occupied by &URL) and Application Toolbar item 35.

  All of these functions are to appear in the Menu Bar ahead of the entry for the Goto > BACK function.

- Change interface “report_screen” to include constants to facilitate the commands Assign police escort, -05, -01, +01 and +05. These need to match their counterpart values defined in the GUI status REPORT_SCREEN.

- Change interface “state” to define new methods "assign_police_escort", "decelerate_05", "decelerate_01", "accelerate_01" and "accelerate_05".

- Change class "vehicle" to define a new private attribute for "police_escort_state", and define a new public method "get_police_escort_state" which is to return the reference to the state interface held in this new attribute. Change the constructor to instantiate an object into this new private attribute.

- Change class "vehicle_state" to include aliases for the new "assign_police_escort", "decelerate_05", "decelerate_01", "accelerate_01" and "accelerate_05" methods contributed by the "state" interface, all of which are to be implemented as null methods. Also define constant “speed_change_factor” as a decimal field with 3 decimal positions and value 1.35. Define new method “accelerate” accepting a change in speed, with its implementation to invoke the “accelerate” method of the corresponding “vehicle” object, but insuring first that the speed will be reduced to a minimum of 01 speed unit -- that is, a reduction in speed will never allow the speed to go to zero or less, but will use only as much reduction in speed to reduce the speed to 01; for example, the deceleration by 05 speed units for a vehicle moving at 03 speed units is to have its speed reduced to only 01 speed unit, for a net speed reduction of 02 speed units instead of the full 05 speed units specified. Use the implementation for method "slow" in class "in_heavy_traffic" as a model for this new "accelerate" method.

- Change class "cruising_state" to redefine method "assign_police_escort", which is to cause a change of state to the “police_escort_state” after increasing the vehicle speed by attribute "speed_change_factor". Redefine methods "decelerate_05", "decelerate_01", "accelerate_01" and "accelerate_05" to invoke the “accelerate” method provided by the “vehicle_state”, in each case passing to the “accelerate” method a change in speed applicable to the name of the method – for instance, method “decelerate_05” passes the value -05 to the “accelerate” method.

- Change class "in_heavy_traffic_state" to redefine methods "decelerate_05", "decelerate_01", "accelerate_01" and "accelerate_05", with implementations the same as their counterparts in “cruising_state”.

- Define new class "police_escort_state" similarly to class "in_shop_state", with a constructor method and redefinitions for the "get_distance_traveled" "resume", "stop" and "turn" methods, which are to be implemented similarly to the way these methods are implemented in the “in_heavy_traffic_state”, with the processing for the “resume” method changing the speed by dividing the current speed by the new “speed_change_factor”. Also, redefine methods "decelerate_05", "decelerate_01", "accelerate_01" and "accelerate_05" with implementations the same as their counterparts in the “cruising_state”. Place the definition of this new class between the “in_shop_state” class and the "vehicle_turn_command" class.

- Change class "vehicle_option" to include a redefinition for inherited method "get_police_escort_state", which is to pass control on to the next inner “vehicle” concrete object.

- Change class "report" to define 5 new methods:
  - “assign_police_escort”
  - “decelerate_05”
  - “decelerate_01”
  - “accelerate_01”
  - “accelerate_05”

  Each of these is to be implemented similarly to the “tow” method, invoking the same named method of the current state object of the corresponding “vehicle” instance for each selected row, obtaining each “vehicle” instance directly from the selected ALV grid row, then showing the updated state values in the report via an invocation of the “refresh” method. Change method "on_user_command" to intercept the commands for Assign police escort, -05, -01, +01 and +05, passing control to the corresponding new methods facilitating these action.

Test the program to insure it still behaves as expected and provides the user with the capability of changing the state of selected vehicles according to the state transition matrix and changing the speeds of vehicles in those states accommodating speed changes.

Notice that the state transition chart provided by the users now has 17 rows describing valid state transitions. Consider for a moment all the conditional logic this would have required in the various methods of the “vehicle” class had we not implemented the State design pattern.

20 Objects 313 – Design Patterns: Lazy Initialization

This section describes the requirements for the exercise programs associated with the chapter covering the topic of the Lazy Initialization in the book Object Oriented Design with ABAP. Intent of Lazy Initialization:

- **Lazy Initialization** – Delay the creation of an object or performance of an expensive operation until the moment it is needed[34].

20.1 Exercise 55

Program: ZOOT313A

Title: Objects 313: Design Patterns: Lazy Initialization, step 1

Functional requirements

The performance folks have become concerned with the number of state objects that exist during an execution of this report. They would like a way to show how many state objects do exist once the report is presented, and have requested a new command "Show state objects count" that can be selected from the report menu which will show this number via a simple informational message presented to the screen.

Technical requirements

This exercise begins the process of optimizing the use by the program of object-oriented resources.

The UML diagram does not change with this exercise program. Copy one of the previous versions forward to your student copy of this program and make the following changes:

- Change GUI status REPORT_SCREEN to include the following new functions:
- SHOW_SOC, with text "Show state objects count" and icon ICON_WD_NUMERIC_VALUE, using Function Keys slot Ctrl-Shift-F12 (occupied by &CRDESIG), Application Toolbar item 35 (see next sentence) and to appear in the Menu Bar ahead of the entry for the Goto > BACK function. Application Toolbar item 35 was just assigned to the ESCORT function in the previous change to this status. To make this slot available *without losing any other of the functions already assigned*, right-click on Application Toolbar item 18 (vertical bar) and select Cut. This will remove the vertical bar and shift all subsequent items down by one item, thus freeing item 35 for use with the SHOW_SOC function.

- Change interface "report_screen" to include a constant to facilitate the command Show state objects count. This needs to match the counterpart value defined in the GUI status REPORT_SCREEN.

- Change class "vehicle_state" to include a new private static attribute for retaining the number of state objects instantiated. Implement an instance constructor method which will increment the new private static attribute with each new instantiated state object. Define a new public static method "get_state_objects_count" which will provide to callers the number of state objects instantiated.

34 Lazy Initialization is not so much a design pattern as it is a performance optimization technique. Accordingly, it is not included in the book Design Patterns: Elements of Reusable Object-Oriented Software – Gamma, Helms, Johnson, Vlissides , 1995, Addison-Wesley.

- Change class "report" to define new method “show_state_objects_count”, which is to retrieve the count of the number of state objects from vehicle_state and issues an informational message showing this number. Change method "on_user_command" to intercept the command for Show state objects count, passing control to the corresponding new method facilitating this action.

Test the program to insure it still behaves as expected and provides the user with the capability to reveal the number of state objects that have been instantiated.

20.2 Exercise 56

Program: ZOOT313B

Title: Objects 313: Design Patterns: Lazy Initialization, step 2

Functional requirements

The previous version provided the performance folks with information on the number of state objects being created. They found some examples where a user could run the program using only about 6 vehicles but resulting in the creation of many hundred state objects. Upon closer inspection it was found that a vehicle instance is having all of the 8 state objects created for it -- cruising state, in heavy traffic state, stopped state, out of service state, being towed state, in shop state, available state and police escort state -- regardless whether the vehicle transitions to any of these states. Worse, the decorator pattern is causing the creation of vehicle option concrete decorators for every option the vehicle has, and each of these also is having all 8 state object instances created for them. Accordingly, a single vehicle which includes one of every vehicle option will result in 72 state objects created for the vehicle: 8 for the concrete vehicle object itself and 8 more for every one of the 8 concrete vehicle option objects. For a vehicle that never transitions from its initial cruising state, this constitutes 71 unnecessary state objects. The performance folks have requested we try to reduce the overhead of extraneous state objects.

Technical requirements

This exercise continues the process of optimizing the use by the program of object-oriented resources.

The UML diagram does not change with this exercise program. Copy one of the previous versions forward to your student copy of this program and make the following changes:

- Change class “vehicle” so it no longer longer instantiates all of its state objects in its constructor method, but makes a single call to its own "get_cruising_state" method, retrieving the returned object into its current state attribute. Change methods "get_<state_name>_state" so that each one first checks whether its own state object attribute has an active object, and, if not, then and only then creating an object. The "create object" statements in these "get_<state_name>_state" methods are the same as those previously residing in the constructor method. In this way, only those state objects that actually are required are instantiated. This is known as the "lazy initialization" technique, where entities are not initialized (objects instantiated) until the moment when it is known the entity is needed.

Test the program to insure it still behaves as expected.

Run this program and the previous exercise program side by side, using the same criteria for generating the report in both programs. You should find the program which controls state object instantiation through use of the Lazy Initialization technique offers a significant reduction in the number of state objects created.

20.3 Brain teaser #3

When running the program to create only a single vehicle entry to appear in the report, what is the maximum number of state objects that could have been created before implementation of the Lazy Initialization technique reduced this number?[35]

When running the program to create 10 vehicle entries to appear in the report, where each vehicle entry eventually transitions through every possible state, what is the maximum number of state objects that can be created after implementation of the Lazy Initialization technique?[36]

35 The formula is the maximum number of state objects created by a vehicle option instance (acting as concrete decorator of the decorator pattern) multiplied by the maximum number of vehicle options available multiplied by the maximum number each vehicle option may be selected for a vehicle multiplied by the number of vehicle entries in the report plus the number of state objects created by a vehicle instance (acting as concrete component of the decorator pattern) multiplied by the number of vehicle entries in the report: 8 * 8 * 9 * 1 + 8 * 1 = 584

36 The formula is the same, but since state objects no longer are created by concrete decorators, the number is significantly reduced: 0 * 8 * 9 * 10 + 8 * 10 = 80

21 Objects 314 – Design Patterns: Flyweight pattern

This section describes the requirements for the exercise programs associated with the chapter covering the topic of the Flyweight design pattern in the book Object Oriented Design with ABAP. This design pattern has an object scope and a structural purpose. Intent of the Flyweight pattern:

- **Flyweight Pattern** – Use sharing to support large numbers of fine-grained objects efficiently.[37]

21.1 Exercise 57

Program: ZOOT314A

Title: Objects 314: Design Patterns: Flyweight pattern, step 1

Functional requirements

The requirement for this version is identical to the previous version – that is, the user sees no discernible difference. All changes are technical in nature.

Technical requirements

The performance folks are happier with the reduction in objects achieved through the use of Lazy Initialization, but have asked if there are any further improvements that can be made to reduce the number of state objects. We have discussed this with Natasha, who has recommended we implement the Flyweight design pattern to manage our state objects. To do this, we need to change the state objects so they retain no attributes which cannot be shared by multiple users of the object. Currently each of our state objects holds a reference to the specific vehicle for which it manages the state transitions. Accordingly, with this design no two vehicles would be able to share the same state object. This vehicle reference attribute is an example of what is known as *intrinsic* data – data that resides within the object. To enable sharing of objects, all *intrinsic* data that cannot be shared by multiple users of the object needs to become *extrinsic* data – that is, data that can be passed to the object by its caller.

The UML diagram does not change with this exercise program. Copy one of the previous versions forward to your student copy of this program and make the following changes:

- Change interface “state” to include for every method other than “get_description” an importing parameter accepting a reference to a vehicle object. Also, now that this interface contains methods with references to class “vehicle”, but the definition of class “vehicle” occurs later in the source code, the interface definition is to be preceded by the statement "class vehicle definition deferred".

- Change class "vehicle_state" to remove its protected attribute vehicle, a value that now will be provided by the callers of its methods. Change its protected methods "accelerate", "halt" and "calculate_distance_traveled" now to include in their signatures an importing parameter for a reference to type vehicle. Change the implementations of its methods which formerly used the reference to its own vehicle object now to refer instead to the vehicle object reference provided by the method signature.

- Change subclasses "cruising_state", "in_heavy_traffic_state", "stopped_state", "out_of_service_state", "available_state", "being_towed_state", "in_shop_state" and "police_escort_state" so their respective constructor methods no longer accept a parameter of type reference to vehicle. Also, the implementations of their methods are to

37 GoF, p. 195.

be changed to account for the reference to the vehicle object now being provided by the method signature instead of its own instance attribute.

- Change class "vehicle" so it no longer provides its self-reference object as a parameter on the create object statements creating instances of state objects.

- Change class "report" to accommodate sending the vehicle object reference on calls to methods of the current state object.

Test the program to insure it still behaves as expected. Since all we did in this exercise was to remove intrinsic data attributes from state objects, running this program and the previous exercise program side by side should show no differences in the number of state objects created.

21.2 Exercise 58

Program: ZOOT314B

Title: Objects 314: Design Patterns: Flyweight pattern, step 2

Functional requirements

The requirement for this version is identical to the previous version – that is, the user sees no discernible difference. All changes are technical in nature.

Technical requirements

The previous version prepared for using the Flyweight pattern by removing all non-shareable intrinsic attributes from the state classes. This version introduces the Flyweight design pattern. Flyweight accommodates significantly reducing the number of objects required to facilitate processing.

In the previous version we changed the state objects so they no longer hold a reference to a vehicle instance. Until we did that, the reference to the vehicle instance was an example of "intrinsic" data in the state object -- that is, the reference to the vehicle was part of the object itself. Accordingly, the state object could not be shared by more than one vehicle. To enable sharing these state objects, any intrinsic data elements that could not be shared by multiple callers to the same object needed to be converted into "extrinsic" data elements. This is what was done in the previous version, by removing the reference to the vehicle held by the state object, and then providing this reference on all calls to methods of the state object. We now can transform these state objects to behave as Flyweights, enabling sharing of a state object amongst multiple vehicles, by converting each one of them into Singleton objects.

Refer to the associated UML class diagrams. Copy one of the previous versions forward to your student copy of this program and make the following changes:

- Change subclasses "cruising_state", "in_heavy_traffic_state", "stopped_state" , "out_of_service_state", "available_state", "being_towed_state", "in_shop_state" and "police_escort_state" in the following way:
    - Each class definition statement now is to include the "create private" qualifier, and its corresponding instance constructor is to be moved to the private visibility section.
    - Each class now is to include a private attribute "singleton" defined as type reference to its own class.
    - Each class now is to include a static constructor method which will create an instance of the class into the "singleton" attribute.

  - Each class now is to include a static public method "get_state_object" which will return the reference to its "singleton" state object.
  - Each class is to have its "class_id" constant removed; These constants were required only to facilitate class instantiation externally, which no longer is possible now with the addition of the "create private" qualifier on the class definition statement.

- Change class "vehicle" so each of its methods "get_<state_name>_state" no longer creates an instance of the corresponding state subclass (this capability was invalided with the addition of the "create private" on each corresponding state subclass) but now invokes the "get_state_object" static method of the corresponding state subclass.

These changes effectively transform each of the state subclasses into singleton objects, meaning that no matter the number of vehicle objects appearing in the report, each vehicle entry shares the same state objects with all other vehicle entries. In addition, for each vehicle entry, its concrete component (vehicle) of the decorator pattern shares the same state objects with all of its concrete decorators (vehicle options). Accordingly, there never will be more than 8 state instances, one for each of the state subclasses that can be instantiated.

Here is an example of two design patterns collaborating to achieve a desired result, in this case, the reduction of the number of state objects necessary to support program execution. The Singleton pattern is what insures there never will be more than one instance of any specific state object. The Flyweight pattern is what insures all vehicle instances are able to share an instance of a specific state object.

Test the program to insure it still behaves as expected and indeed can never show more than 8 state instances through the Show state objects count command, regardless of the number of vehicle entries in the report and regardless of the number of state transitions applied to each of those vehicle entries.

21.3 Exercise 59

Program: ZOOT314C

Title: Objects 314: Design Patterns: Flyweight pattern, step 3

Functional requirements

The requirement for this version is identical to the previous version – that is, the user sees no discernible difference. All changes are technical in nature.

Technical requirements

Now that our state objects are flyweights, with a static method for us to invoke to get an instance of the object, we no longer need all the private attributes in class "vehicle" to keep references to these objects. Accordingly, without these private attributes we no longer need its methods “get_<state_name>_state" to retrieve the reference to the “state” object. In addition, without these "get_<state_name>_state" methods defined in class "vehicle", we no longer need the corresponding pass-through methods redefined in class "vehicle_option". In short, there is a lot of code in the program to manage state transitions which no longer is necessary, and can be discarded.

The UML diagram does not change with this exercise program. Copy one of the previous versions forward to your student copy of this program and make the following changes:

- Change class "vehicle" to remove the private attributes holding references to each of the state objects and discard their corresponding methods "get_<state_name>_state" . Change its constructor method to invoke static method "get_state_object" of class "cruising_state" to set its current state.

- Change class "vehicle_option" to remove the methods "get_<state_name>_state" which had redefined methods no longer defined in class "vehicle".

- Change method "halt" in class "vehicle_state" to retrieve the reference to the "stopped_state" object by invoking the static method "get_state_object" of class "stopped_state" instead of invoking a method of class "vehicle" to do this. This same concept applies to these other methods:

"slow" and "assign_police_escort"	in class "cruising_state"
“resume"	in class "in_heavy_traffic_state"
"make_available" and "place_out_of_service"	in class "stopped_state"
“maintain" and "tow"	in class "out_of_service_state"
"start"	in class "available_state"
"repair"	in class "being_towed_state"
"make_available"	in class "in_shop_state"
"resume"	in class "police_escort_state"

  In each case the object to be retrieved is requested from the static method "get_state_object" of whichever class represents the next state to which the vehicle is to transition.

Test the program to insure it still behaves as expected

22 Objects 315 – Design Patterns: Memento pattern

This section describes the requirements for the exercise programs associated with the chapter covering the topic of the Memento design pattern in the book Object Oriented Design with ABAP. This design pattern has an object scope and a behavioral purpose. Intent of the Memento pattern:

- **Memento Pattern** – Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to that state later.[38]

22.1 Exercise 60

Program: ZOOT315A

Title: Objects 315: Design Patterns: Memento pattern, step 1

Functional requirements

The users have realized that the resumption of speed from either the “in heavy traffic” or “police escort” states back to the “cruising” state should not be based on the current speed, which might have been adjusted by the Decelerate and Accelerate commands while in those states. They are requesting whether it is possible for speed resumption to use whatever was the current speed in the “cruising” state at the moment the vehicle transitioned out of it, even though they acknowledge having approved the current functionality back when the Decelerate and Accelerate commands first were made available.

Technical requirements

We have discussed this with Natasha, who has suggested we consider using the Memento design pattern. She explained that prior to making a transition to another state, we would be able with the Memento design pattern to take a snapshot of the current attribute values of a class or instance, and use these later to reset those attributes to the values that had been saved. This seems like a workable solution since the users would like the attributes of the vehicle instance to be reset to the same values they had been prior to transitioning from the “cruising” state to either the “in heavy traffic” or “police escort” states.

Refer to the associated UML class diagrams. Copy one of the previous versions forward to your student copy of this program and make the following changes:

- Precede the definition statement for the “vehicle” class with the following statement:

```
class vehicle_memento definition deferred.
```

- Change class "vehicle" to include public methods "create_memento", which is to create a new "vehicle_memento" instance (see next bullet) using the current speed and state values, and "reset_using_memento", which is to reset its speed and state using the values returned by the "get_speed" and "get_state" methods of the "vehicle_memento" instance.

- Define new class "vehicle_memento" with public methods "constructor", "get_speed" and "get_state", and with private attributes "speed" and "state", defined using references to class "vehicle" and interface "state", respectively. Its "constructor" method is to accept vehicle speed and state parameters, using these to set its speed and state attributes. Its

38 GoF, p. 283.

"get_speed" method is to return its speed attribute; its “get_state" method is to return its state attribute.

Note: Here we see that classes "vehicle" and "vehicle_memento" have a mutual dependency upon each other -- the new methods added to the "vehicle" class have signatures referring to class "vehicle_memento", and the "speed" attribute of class "vehicle_memento" is defined with reference to a public type defined by class "vehicle". This is why the definition of class "vehicle" needs to be preceded by the statement "class vehicle_memento definition deferred.", so that the signatures of its new methods can refer to a class definition the compiler has not yet encountered. This is necessary here only because these both are local class definitions; had both these been defined as global classes, we would not need to bother with this "class … definition deferred" statement.

Also, the arrangement of these class definitions here only works when the "vehicle" class precedes the "vehicle_memento" class. This is because the "class ... definition deferred" statement enables references only to the name of the deferred class, not to any of its members.

- Change method "halt" of class "vehicle_state" to request "fleet_manager" to set a vehicle memento (see changes to class “fleet_manager” below), then remove all the processing involved with setting previous speed and previous state.

- For class "cruising_state", change methods "slow" and "assign_police_escort" to request "fleet_manager" to set a vehicle memento, then remove all the processing involved with setting previous speed and previous state.

- For classes "in_heavy_traffic_state", "stopped_state" and "police_escort_state", change method "resume" to request the "vehicle_memento" instance from "fleet_manager" and to use it to invoke its method "reset_using_memento" for the corresponding vehicle. In addition, remove all processing involved with setting previous speed and previous state.

- Change class "vehicle_option" to include pass-through methods for the new "create_memento" and "reset_using_memento" methods defined by class "vehicle".

- Change class "fleet_manager" to include public methods "get_vehicle_memento" and "set_vehicle_memento", both of which are to accept a parameter indicating a reference to an instance of a “vehicle”, with "get_vehicle_memento" also returning a reference to an instance of “vehicle_memento”. Also, define a private internal table to associate a "vehicle" instance with a corresponding "vehicle_memento" instance. This table should be defined with two columns: the first column should hold a reference to a “vehicle” instance and the second column should hold the reference to its corresponding “vehicle_memento” instance. Method "get_vehicle_memento" is to read the new internal table to locate the "vehicle_memento" instance associated with the corresponding "vehicle" instance and to return it to the caller. Method "set_vehicle_memento" is to invoke method "create_memento" of the specified "vehicle" instance, then is to update the current internal vehicle memento table entry for a vehicle with its new "vehicle_memento" instance, or, if there is not yet an entry in the table for that vehicle instance, is to create one.

Test the program to insure it still behaves as expected and the use of the “Resume” command will cause the speed of a vehicle to be restored to the original value for that state, **but do not attempt to use “Resume” twice in succession for the same vehicle.**

Note: There is a bug in this program. Previously a user could request a vehicle to change from the "cruising" state to the "in heavy traffic" state, and then again change to the "stopped" state, then issue the Resume command twice against the vehicle and watch it move from "stopped" back to "in heavy traffic" and back again to "cruising". The internal table technique implemented in "fleet_manager" no longer handles moving back through a series of states like this since it accommodates only a single memento per vehicle. This problem is fixed in the next version.

22.2 Exercise 61

Program: ZOOT315B

Title: Objects 315: Design Patterns: Memento pattern, step 2

Functional requirements

The users have discovered the bug introduced by the preceding version of the program – specifically, they are complaining that they previously had been able to move a vehicle from states "cruising" to "in heavy traffic" to "stopped", and then press “Resume” to move back again to each previous state. Now they find this sequence leaves the vehicle in the "in heavy traffic" state, never returning to the "cruising" state. They would like the previous functionality to be restored.

Technical requirements

After some debugging we found that the problem is with the new memento internal table we introduced to class “fleet_manager”. A first memento entry for a vehicle is created to reflect its “cruising” state when moving from "cruising" to "in heavy traffic", and a second memento entry for the same vehicle is created to reflect its "in heavy traffic" state when moving from "in heavy traffic" to “stopped”, but the memento table defined by class “fleet_manager” is capable of retaining only a single entry per vehicle. Accordingly, when the second memento for the same vehicle is created, its value replaces the first memento reference when the table is updated. Repeated uses of the “Resume” command will always find the same memento entry for a vehicle, and it always will reflect the last one that had been created for it. We need this memento table to be able to handle multiple entries for the same vehicle in a way that enables stepping back through a sequence of vehicle state changes.

The UML diagram does not change with this exercise program. Copy one of the previous versions forward to your student copy of this program and make the following changes:

- Change the internal table defined in “fleet_manager” recording the vehicle memento associated with a vehicle to be regarded as a LIFO stack - last in, first out – where a row is inserted as the new first row of the table with each new memento instance. Method "get_vehicle_memento" now is to discard the located row from internal table "vehicle_memento_stack" upon finding it. Method "set_vehicle_memento" no longer is to look for a matching row for the specified vehicle, but simply to create a new entry as the first row in table "vehicle_memento_stack". Also, new method "discard_vehicle_mementos" is to be defined to clear any existing "vehicle_memento" entries being tracked for a vehicle.

- Change method "start" of class "available_state" to invoke method "discard_vehicle_mementos" of class "fleet_manager", which will remove any residual vehicle mementos remaining from its previous trip.

Test the program to insure it still behaves as expected and the use of the “Resume” command will cause the speed of a vehicle to be restored to the original value **even when it is used twice in succession for the same vehicle.**

22.3 Exercise 62

Program: ZOOT315C

Title: Objects 315: Design Patterns: Memento pattern, step 3

Functional requirements

The requirement for this version is identical to the previous version – that is, the user sees no discernible difference. All changes are technical in nature.

Technical requirements

The discovery we made with the previous version -- that a single entry for a previous state or speed cannot accommodate multiple resumptions back through a sequence of state changes – made us realize that our original design to record previous speed and previous state was similarly flawed. Although we were careful to set the previous speed and state values with each change of state, we never really used these values effectively, relying on tricks to achieve resumption though multiple states – specifically, we ignored the previous speed and state when processing the “Resume” command for the “in heavy traffic” state, since this might have caused us to resume speed to zero if we had returned to the “in heavy traffic” state from the “stopped” state. Instead, we simply *assumed* that the “in heavy traffic” state could only resume to the “cruising” state and consequently doubled the current speed to become the new speed for the “cruising” state. We now realize this was not a very robust design, so the “previous_state” and “previous_state_speed” attributes of the “vehicle” class, having been rendered virtually useless, and can be eliminated.

The UML diagram does not change with this exercise program. Copy one of the previous versions forward to your student copy of this program and make the following changes:

- Change class "vehicle" to remove the private attributes "previous_state_speed" and "previous_state", along with methods "get_previous_state_speed", "set_previous_state_speed", "get_previous_state" and "set_previous_state".

- Change class "vehicle_option" to remove the redefinition declarations and implementations for the same methods removed from superclass "vehicle".

- Change the following methods so they no longer invoke method "set_previous_state" of class "vehicle", which was removed from that class:
  - Method "make_available" of class "stopped_state"
  - Method "place_out_of_service" of class "stopped_state"
  - Method "maintain" of class "out_of_service_state"
  - Method "tow" of class "out_of_service_state"
  - Method "start" of class "available_state"
  - Method "repair" of class "being_towed_state"
  - Method "make_available" of class "in_shop_state"

Test the program to insure it still behaves as expected.

22.4 Exercise 63

Program: ZOOT315D

Title: Objects 315: Design Patterns: Memento pattern, step 4

Functional requirements

The users have requested the ability to change the state of a vehicle from "in heavy traffic" to "police escort". This change now will enable the state of a vehicle to follow a sequence through 3 other state changes and back again:

• from "cruising"	to "in heavy traffic"
• from "in heavy traffic"	to "police escort"
• from "police escort"	to "stopped"

Until now, the users had the ability to follow a sequence through only 2 other state changes and back again.

The following chart describes the corresponding state transition matrix:

Current state	**Transition command**	**New state**
police escort	Resume	cruising
police escort	Resume	in heavy traffic
police escort	Stop	stopped
cruising	Assign police escort	police escort
cruising	Slow	in heavy traffic
cruising	Stop	stopped
in heavy traffic	Assign police escort	police escort
in heavy traffic	Stop	stopped
in heavy traffic	Resume	cruising
stopped	Resume	cruising
stopped	Resume	in heavy traffic
stopped	Resume	police escort
stopped	Place out of service	out of service
stopped	Make available	available
out of service	Tow	being towed
out of service	Maintain	in shop
being towed	Repair	in shop
in shop	Make available	available
available	Start	cruising

Refer to the associated state diagram.

Technical requirements

So far the "cruising" class had been the only class to offer a change of state to "police escort". Now that class "in heavy traffic" also will offer this capability, we have decided to move the code to facilitate assigning a police escort from the public method in the "cruising" class to a protected method in the "vehicle_state" class, making it available to any subclasses to the "vehicle_state" class, similar to what was done with the code currently contained in the "halt" method of the "vehicle_state" class.

The UML diagram does not change with this exercise program. Copy one of the previous versions forward to your student copy of this program and make the following changes:

- Change class "vehicle_state" to provide a new protected method "engage_police_escort". Its implementation is to be the same code moved from the "assign_police_escort" method of the "cruising" class.

- Change class "cruising" to replace the former code for its "assign_police_escort" method, which has been moved to method "engage_police_escort" of the "vehicle_state" class, with a call to method "engage_police_escort".

- Change class "in_heavy_traffic_state" to include a redefinition for the "assign_police_escort" method. Its implementation is identical to new processing provided for this redefined method of the "cruising" class.

Test the program to insure it behaves as expected and enables moving from “in heavy traffic” to “police escort” and back again, as well as moving from “cruising” to “in heavy traffic” to “police escort” to “stopped” and back again.

22.5 Exercise 64

Program: ZOOT315E

Title: Objects 315: Design Patterns: Memento pattern, step 5

Functional requirements

The users have made no request for changes, but will be horrified to see that now the Resume command will cause the speed of the corresponding vehicle to be set to 999 and its state to be set to “being towed”, regardless of the expected speed and state.

Technical requirements

This example shows how a rogue developer can compromise the integrity of the memento objects created for vehicles, illustrating the security exposure within the program during the creation of memento objects.

The UML diagram does not change with this exercise program. Copy one of the previous versions forward to your student copy of this program and make the following changes:

- Change method "set_vehicle_memento" of class "fleet_manager" to ignore the "vehicle_memento" instance created for it by the "vehicle" instance and instead have it create its own "vehicle_memento" instance with deliberately corrupted values, using speed of 999 and state of “being towed”.

Test the program to insure it behaves as expected, changing a vehicle to these corrupted speed and state values upon issuing the Resume command.

22.6 Exercise 65

Program: ZOOT315F

Title: Objects 315: Design Patterns: Memento pattern, step 6

Functional requirements

The users have expressed their outrage with the faulty results of using the Resume command with the previous version. They want this fixed immediately!

Technical requirements

The previous version shows how memento instances can be created by virtually any other class within the program. In this case, it was a method of the “fleet_manager” class that, after requesting the “vehicle” instance to provide it with a memento, ignored it and created its own memento instance, using values *it* decided should represent speed and state, compromising the integrity of the functionality. Accordingly, we need to plug this security breach by insuring that only an instance of “vehicle” is capable of creating instances of “vehicle_memento”.

The UML diagram does not change with this exercise program. Copy one of the previous versions forward to your student copy of this program and make the following changes:

- Extend the definition of the "vehicle_memento" class to indicate "create private" and "friends vehicle". This prevents any classes other than the "vehicle_memento" itself and its friend "vehicle" from creating instances of it. Change the definition of "vehicle_memento" so that all methods and attributes are private. This otherwise would enable only instances of class "vehicle_memento" to access these private members, however the "friends" clause applied to the "vehicle_memento" class indicates the list of other classes that will be permitted to reference its members as though all of them had been defined with public access.

- Run the syntax checker at this point and you should find that the “create object …” statement in method “set_vehicle_memento” of class “fleet_manager” no longer is syntactically valid. It became invalidated upon including the “create private” clause on the “vehicle_memento” class definition. Specifically, the “fleet_manager” class is neither the “vehicle_memento” class nor one of its friends, the only entities now permitted to create instances of “vehicle_memento”.

- Change method "set_vehicle_memento" of class "fleet_manager" to discard the extra code used to create its own instances of "vehicle_memento" objects.

Note: While we can acknowledge that dealing with programmers who would pull such a stunt, whether deliberate or accidental, is beyond the scope of good program design, and that this solution merely moves the problem from originating in the “fleet_manager” class to originating in the “vehicle” class, at least this solution localizes any integrity corruption to the same class eventually using the values of the corresponding “vehicle_memento” instance, making it easier to locate such corruption when users complain of similar unexpected results spewing forth from the application.

The Memento design pattern is often described as having both a wide interface and a narrow interface. The originator participant – in this case, the “vehicle” class that creates the memento – uses the wide interface, one that permits access to all the members, and the caretaker participant – in this case, the “fleet_manager” class that holds all memento instances for later use – uses the narrow interface, one that allows only passing the memento to other objects. Prior to applying the “create private” clause to the “vehicle_memento” class, all classes had access to its wide interface. Afterward, only the memento and originator participants – in this case, “vehicle” and “vehicle_memento”, respectively – have access to its wide interface.

This is the first exercise to use of the “friends” clause on the class definition statement. This clause is used with a class definition to grant other classes access to its non-public members as though they had been defined as publicly accessible. Friendship can be offered, but is not automatically reciprocated – that is, when class A offers friendship to class B, it does not mean that class A automatically becomes a friend of class B, as though class B had offered friendship to class A. For class A to have access to the non-public members of class B, class B must directly or indirectly offer friendship to class A. Direct friendship is where class A offers friendship directly to class B by naming class B on a friends clause. Indirect friendship is where class A offers friendship to interface X by naming interface X on a friends clause, and class B implements interface X, resulting in a situation where class B now has access to the non-public members of class A.

Friendship often is used with ABAP Unit tests, where a class under test (that is, a class intended to go to production and is regarded as “under test” by the ABAP Unit feature) grants access to its non-public members to other specific classes and interfaces facilitating ABAP Unit testing of the class under test. This enables the ABAP Unit classes that have been offered friendship the ability to invoke protected and private methods and have access to protected and private attributes of the class under test for the purpose of testing the integrity of these members.

Test the program to insure it behaves as expected.

23 Objects 316 – Design Patterns: Visitor pattern

This section describes the requirements for the exercise programs associated with the chapter covering the topic of the Visitor design pattern in the book Object Oriented Design with ABAP. This design pattern has an object scope and a behavioral purpose. Intent of the Visitor pattern:

- **Visitor Pattern** – Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.[39]

We will not begin to use the Visitor pattern until the 5th exercise in this series. The first 4 exercises show how we might solve the problem without using this design pattern, setting the stage for illustrating how the visitor design pattern can simplify the new processing.

23.1 Exercise 66

Program: ZOOT316A

Title: Objects 316: Design Patterns: Visitor pattern, step 1

Functional requirements

The users have asked for a new command through which they can impose a high winds speed restriction to vehicles. When a vehicle is exceeding a maximum safe speed of 35 for high winds, the vehicle would have its current speed reduced to this maximum safe speed. They have indicated it would not be necessary first to select any vehicles since the geographic area associated with those vehicles represented in the report is small enough that the entire region would be affected by a high winds condition.

Technical requirements

Natasha is on vacation this week, so we are unable to discuss this change with her. Instead, we have decided to implement changes similarly to the way we had implemented the Accelerate and Decelerate commands, except we will not require the user to select any rows but simply will apply the high winds speed restriction to all vehicles appearing in the report.

The UML diagram does not change with this exercise program. Copy one of the previous versions forward to your student copy of this program and make the following changes:

- Change GUI status REPORT_SCREEN to include new function ISR_HIWIND, with text "High winds" and icon ICON_CATASTROPHE, using Function Keys slot Ctrl-F1 (occupied by &ABC), no Application Toolbar item, and to appear in the Menu Bar as Goto > Restrict speed > High winds, with the Restrict speed entry to appear in the Goto menu ahead of the entry for the BACK function.

- Change interface "report_screen" to include new constants to facilitate the command High winds. The value needs to match the counterpart value defined in the GUI status REPORT_SCREEN.

- Change interface "state" to include new method "impose_high_winds_restriction", which is to have the same signature as method "accelerate_05".

39 GoF, p. 331.

- Change class "vehicle_state" to define an alias for the new method "impose_high_winds_restriction" defined by the interface it implements, providing a default implementation to do nothing. It also is to define new protected method "apply_high_winds_restriction", which is to have the same signature as method "engage_police_escort", implemented to compare the current speed of the vehicle with a maximum safe speed for high winds of 35, and when the vehicle speed exceeds this, to reduce its speed to this specified maximum safe speed.

- Change classes "cruising_state", "in_heavy_traffic_state" and "police_escort_state" all to provide a redefinition for method "impose_high_winds_restriction", each one implemented to invoke inherited method "apply_high_winds_restriction".

- Change class "report" to define a new method "impose_high_winds_restriction", which is to loop through all the output entries calling method "impose_high_winds_restriction" of the corresponding state class associated with each vehicle. Its method "on_user_command" is to be changed to detect the command for high winds, invoking its own corresponding method to accommodate this functionality.

Test the program to insure it behaves as expected and will decrement the current speed of all vehicles which exceed a maximum safe speed of 35 for high winds.

23.2 Exercise 67

Program: ZOOT316B

Title: Objects 316: Design Patterns: Visitor pattern, step 2

Functional requirements

The users have stated that the previous feature they requested – to enable applying a high winds speed restriction to vehicles – was intended to apply only to trucks. They want cars to be exempt from the high winds speed restriction.

Technical requirements

With this exercise a technique is implemented by which a command can be restricted to applying only to instances of cars.

The UML diagram does not change with this exercise program. Copy one of the previous versions forward to your student copy of this program and make the following changes:

- Change method "apply_high_winds_restriction" of class "vehicle_state" to determine whether the corresponding vehicle is a truck, and applies the high winds speed restriction only to trucks. Implement this determination by defining a reference variable to class truck and at the start of the method performing a specializing cast move of the vehicle reference into this truck reference field. If the move is not successful, return to the caller; otherwise allow execution of the existing code to apply the speed reduction.

Test the program to insure it still behaves as expected and the use of the “High Winds” command will cause the speed of only trucks to be subject to this maximum safe speed, **but test this only with trucks that have no vehicle options selected.**

Note: It should be noted that this program has a bug awaiting discovery. Our implementation to impose a high winds restriction only to trucks relies on determining whether the dynamic type of the vehicle instance is one of type class "truck". We will find that this works only for those vehicles appearing in the ALV report which have no vehicle options defined for them (see values in Descriptor column). This intentional bug is fixed in the next version.

23.3 Exercise 68

Program: ZOOT316C

Title: Objects 316: Design Patterns: Visitor pattern, step 3

Functional requirements

The users have found the bug in the previous version – that it does not apply high wind restrictions to those trucks which have vehicle options selected. They would like this corrected as soon as possible.

Technical requirements

The implementation we had provided in the previous version to impose a high winds restriction only to trucks relies on determining whether the dynamic type of the vehicle instance is one of type class "truck". This check will work only for those trucks which have no vehicle options, since a "truck" instance with vehicle options will be wrapped within one or more instances of class "vehicle_option". Accordingly, the test for whether an instance of a "vehicle_option" is an instance of a "truck" will fail.

The UML diagram does not change with this exercise program. Copy one of the previous versions forward to your student copy of this program and make the following changes:

- Move the "descriptor" attribute of class "truck" from the private section to the public section to make it available to method "apply_high_winds_restriction" of class "vehicle_state".
- Change method "apply_high_winds_restriction" of class "vehicle_state" to remove the test for the dynamic type of the instance of vehicle, and replace it with code to retrieve the description of the vehicle and then checking whether the first few characters of this vehicle description contains the string "Truck", using the "descriptor" attribute of the "truck" class.

Test the program to insure it still behaves as expected.

Note: Though this works, it is not a very clean implementation since we are relying on a portion of text to control processing. Worse, the text upon which we now are dependent had been defined in the truck class as a private attribute, but we elevated its visibility to public so it could be accessible to the "apply_high_winds_restriction" method, where it is used to determine whether its value is contained within the description attribute of the vehicle. This dependency is removed in a subsequent version.

23.4 Exercise 69

Program: ZOOT316D

Title: Objects 316: Design Patterns: Visitor pattern, step 4

Functional requirements

The users have asked for a new command through which they can impose an icy road surface speed restriction to vehicles. When a vehicle is exceeding a maximum safe speed for icy road surfaces, the vehicle would have its current speed reduced to this maximum safe speed. Similar to their explanation about high winds, they have indicated it would not be necessary first to select any vehicles since the geographic area associated with those vehicles represented in the report is small enough that the entire region would be affected by an icy road surface condition. Furthermore, they have explicitly indicated that this is to apply to both cars and trucks, but where the maximum safe speed on icy roads is 15 for cars and 10 for trucks.

Technical requirements

Here we can use the model of the high winds command to make the additional changes to handle the ice command.

The UML diagram does not change with this exercise program. Copy one of the previous versions forward to your student copy of this program and make the following changes:

- Change GUI status REPORT_SCREEN to include new function ISR_ICE, with text "Ice" and icon ICON_ACTIVITY, using Function Keys slot Shift-F12 (occupied by &VCRYSTAL), no Application Toolbar item, and to appear in the Menu Bar as Goto > Restrict speed > Ice, following the entry for the ISR_HIWINDS function.

- Change interface "report_screen" to include new constants to facilitate the command Ice. The value needs to match the counterpart value defined in the GUI status REPORT_SCREEN.

- Change interface "state" to include new method "impose_ice_restriction", which is to have the same signature as method "impose_high_winds_restriction".

- Change class "vehicle_state" to define an alias for the new method "impose_ice_restriction" defined by the interface it implements, providing a default implementation to do nothing. It also is to define protected methods "apply_ice_restriction", which is to have the same signature as method "apply_high_winds_restriction", and "apply_speed_restriction", which is to accept one parameter of type vehicle and another for a maximum speed. New method "apply_speed_restriction" is to contain the logic to get the current speed of the vehicle and adjust it when it exceeds the maximum speed, code moved to it from method "apply_high_winds_restriction", but generalized so it also can be invoked by method "apply_ice_restriction". Change method "apply_high_winds_restriction" to remove the code for adjusting the speed of a vehicle, code that has been moved to method "apply_speed_restriction", and instead invoke new method "apply_speed_restriction". New method "apply_ice_restriction" also is to invoke "apply_speed_restriction" after determining the applicable maximum safe speed based on checking whether the vehicle is of type car or type truck, defining a maximum safe speed of 15 for cars and 10 for trucks, using the same technique as used with "apply_high_winds_restriction" for determining whether the vehicle is a car or truck.

- Change classes "cruising_state", "in_heavy_traffic_state" and "police_escort_state" all to provide a redefinition for method "impose_ice_restriction", each one implemented to invoke inherited method "apply_ice_restriction".

- Change class "report" to define a new method "impose_ice_restriction", which is to loop through all the output entries and to call method "impose_ice_restriction" of the corresponding state class associated with each vehicle. Its method "on_user_command" is to be changed to detect the command for ice, invoking its own corresponding method to accommodate this functionality.

Test the program to insure it behaves as expected.

23.5 Exercise 70

Program: ZOOT316E

Title: Objects 316: Design Patterns: Visitor pattern, step 5

Functional requirements

The requirement for this version is identical to the previous version – that is, the user sees no discernible difference. All changes are technical in nature.

Technical requirements

Natasha has returned from vacation to learn about all the changes we had made in her absence. She was curious to know how we handled the problem of distinguishing between car and truck objects in our implementations for these two new commands. She seemed concerned upon hearing how we solved these challenges, and after looking at the code suggested we gather for yet another design discussion.

She had some suggestions about how the existing code could be improved for maintenance. We learned from her that there was no reason to check the state of each vehicle to determine whether to apply the commands Impose high winds restriction and Impose ice restriction since these only can apply to vehicles that are moving, meaning their current speed would be greater than zero, but that the requirement to determine the actual type of vehicle by methods of class "vehicle_state" introduced a less than robust implementation for a class that, until now, had no reason to distinguish between car and truck vehicles, introducing conditional logic that she thinks can be removed. She introduced us to a new object-oriented design principle known as the Open/Closed principle, which states:

- A class should be open for extension but closed for modification.

This principle suggests that once a class is defined it should not be changed to provide different behavior (the class is closed for modification) but instead the different behavior should be provided via a subclass inheriting from the class (the class is open for extension). It is the "but closed for modification" phrase with which she is most concerned regarding our current implementation since, so far, it had required changes to the existing class "vehicle_state" and its existing subclasses "cruising_state", "in_heavy_traffic_state" and "police_escort_state". All four of these classes were changed with the introduction of command Impose high winds restriction, and then all four were changed again with the introduction of command Impose ice restriction. She advised us that probably we would have to change these same four classes every time a new speed restriction command is introduced, violating the Open/Closed principle with each one.

She explained that if we were to implement the Visitor design pattern it would require one change to each of the “vehicle”, "car", "truck" and "vehicle_option" classes, but would not require further changes with the introduction of any new speed restriction commands, and it would eliminate the necessity for the conditional logic checking whether a vehicle is a car or truck, logic that even we concede is less than robust. As such, any subsequent user requests for speed restriction commands could be implemented without requiring any changes to the “vehicle”, "car", "truck", "vehicle_option", "vehicle_state", "cruising_state", "in_heavy_traffic_state" and "police_escort_state" classes, significantly simplifying our maintenance efforts and eliminating the chances of introducing bugs into these existing classes.

Refer to the associated UML class diagrams. Copy one of the previous versions forward to your student copy of this program and make the following changes:

- Remove methods "impose_high_winds_restriction" and "impose_ice_restriction" from interface "state". This effectively restores it to the pre-high winds version.
- Define new interface "visitor" with two methods: "visit_car" and "visit_truck". Each of these methods is to accept a single parameter of type "vehicle". Place the definition of the new “visitor” interface between the “state” interface and the “navigator” class.
- Define new interface "visitable" with method "accept", which is to accept a single parameter of type "visitor". Place the definition of the new “visitable” interface between the “visitor” interface and the “navigator” class.
- Change class "vehicle" to implement the "visitable" interface, indicating all methods this interface introduces are to be regarded as abstract methods, and define for it an alias for method "accept" introduced by the interface. Include an alias for the behavior of the “visitable” interface enabling the developer to refer to the interface methods within the “vehicle” class without the necessity to prefix it with the interface component selector (“visitable~”).
- Define new class "speed_restriction" with protected method "impose_speed_restriction", using the same signature and implementation defined for method "apply_speed_restriction" of class "vehicle_state". Place the definition of the new “speed_restriction” class between the “vehicle_memento” class and the “vehicle_state” class.
- Define new classes "high_winds_speed_restriction" and "ice_speed_restriction", both virtually identical, both inheriting from class "speed_restriction", both implementing the "visitor" interface and defining an alias for the two methods it introduces. Place the definition of these new classes one after the other between the “speed_restriction” class and the “vehicle_state” class. Define constants in the private section to indicate the car maximum speed and truck maximum speed for the corresponding restriction each class defines:

• high winds maximum safe speed for cars	500 (effectively no limit)
• high winds maximum safe speed for trucks	35
• ice maximum safe speed for cars	15
• ice maximum safe speed for trucks	10

  Their implementations for method "visit_car" are to be identical, both invoking superclass method "impose_speed_restriction", passing the vehicle provided and the corresponding maximum safe speed for car defined by the class. Similarly, their implementations for method "visit_truck" are to be identical, both invoking superclass method "impose_speed_restriction", passing the vehicle provided and the corresponding maximum safe speed for truck defined by the class.
- Change class "vehicle_state" to remove aliases for both "impose_high_winds_restriction" and "impose_ice_restriction" along with their respective implementations. Remove methods "apply_high_winds_restriction", "apply_ice_restriction" and "apply_speed_restriction". This effectively restores it to the pre-high winds version.

- Change classes "cruising_state", "in_heavy_traffic_state" and "police_escort_state" to remove their respective redefinitions and implementations for methods "impose_high_winds_restriction" and "impose_ice_restriction". This effectively restores them to the pre-high winds version.

- Change classes "car", "truck" and "vehicle_option" to redefine the "accept" method (provided to these classes through the implementation of the "visitable" interface by the "vehicle" superclass, which indicated that this method was to be regarded as abstract, thus requiring each subclass to provide an implementation). Each one is to implement method "accept" differently:
    - The "car" class is to invoke method "visit_car" of the "visitor" instance it is passed, passing itself (via the "me" self-reference qualifier) on the call.
    - The "truck" class is to invoke method "visit_truck" of the "visitor" instance it is passed, passing itself (via the "me" self-reference qualifier) on the call.
    - The "vehicle_option" class is to implement a pass-through method, calling the same method of the "vehicle" instance it wraps, in a manner similar to the other redefined methods of this class that serve only as pass-through methods.

- Change class "truck" to return its "descriptor" attribute back to the private section.

- Change methods "impose_high_winds_restriction" and "impose_ice_restriction" of class "report" in exactly the same way: Instead of each one getting the "state" object associated with the "vehicle" instance corresponding to the output entry, and then invoking the respective "impose_high_winds_restriction" or "impose_ice_restriction" method on that "state" instance, each one now is to:
    - define a variable as type reference to "visitor"
    - create into that reference variable an instance of class visitor using subclass type "high_winds_speed_restriction" or "ice_speed_restriction", respectively
    - invoke method "accept" on the corresponding "vehicle" instance, passing to it the reference to the "visitor" instance it had created

Test the program to insure it still behaves as expected.

The use of the visitor design pattern implements a technique known as "double dispatch". The underlying concept is explained here:

> When method "impose_high_winds_restriction" or "impose_ice_restriction" of class "report" invokes method "accept" against the instance of the vehicle it holds, dynamic dispatch determines the actual type of class referenced by the vehicle reference variable, eventually reaching the implementation of "accept" for a "car" or "truck" instance once any wrapping "vehicle_option" implementations of "accept" are passed through to one of these instances. Accordingly, the resolution from static type "vehicle" to dynamic type "car" or "truck" is the first of the two double dispatches.
>
> Next, the parameter used with the call to "accept" is one of type "visitor", a reference to an interface. When the implementation of method "visit_car" of class "car" or "visit_truck" of class "truck" is executed, again dynamic dispatch will occur to determine the dynamic type of the object referenced by static type "visitor", which now will be one of either "high_winds_speed_restriction" or "ice_speed_restriction". Hence, the resolution from static type "visitor" to dynamic type "high_winds_speed_restriction" or "ice_speed_restriction" is the second of the two double dispatches.

Effectively, it becomes the combination of both the "vehicle" instance dynamic type and "visitor" instance dynamic type that determines the associated processing to be executed.

23.6 Exercise 71

Program: ZOOT316F

Title: Objects 316: Design Patterns: Visitor pattern, step 6

Functional requirements

The users have asked for a new command through which they can impose a fog speed restriction to vehicles. When a vehicle is exceeding a maximum safe speed for fog, the vehicle would have its current speed reduced to this maximum safe speed. Similar to their explanation about icy roads, they have indicated it would not be necessary first to select any vehicles since the geographic area associated with those vehicles represented in the report is small enough that the entire region would be affected by fog. Furthermore, they have explicitly indicated that this is to apply to both cars and trucks, but where the maximum safe speed in fog is 30 for cars and 25 for trucks.

Technical requirements

This exercise demonstrates how the Visitor design pattern minimizes the number of classes requiring changes when introducing a new capability.

Refer to the associated UML class diagrams. Copy one of the previous versions forward to your student copy of this program and make the following changes:

- Change GUI status REPORT_SCREEN to include new function ISR_FOG, with text "Fog" and icon ICON_TRANSFER_STRUCTURE_INA, using Function Keys slot Ctrl-F7 (occupied by %SL), no Application Toolbar item, and to appear in the Menu Bar as Goto > Restrict speed > Fog, following the ISR_ICE function.

- Change interface "report_screen" to include new constants to facilitate the command Fog. The value needs to match the counterpart value defined in the GUI status REPORT_SCREEN.

- Define new class "fog_speed_restriction" virtually identically to class "ice_speed_restriction", using 30 as the maximum safe speed for cars and 25 as the maximum safe speed for trucks. Place the definition of this new class between the “ice_speed_restriction” class and the “vehicle_state” class.

- Change class "report" to define new method "impose_fog_restriction", which is to be virtually identical to its method "impose_ice_restriction" except that it creates a visitor instance of type "fog_speed_restriction". Its method "on_user_command" now is to detect the command for fog, invoking its own corresponding method to accommodate this functionality.

Test the program to insure it behaves as expected.

In the previous version we changed existing classes “vehicle_state”, "cruising_state", "in_heavy_traffic_state" and "police_escort_state" only to restore them to previous definitions. We also changed classes "vehicle", "car", "truck" and "vehicle_option" to enable the use of the visitor design pattern. Natasha had advised us that changing these 4 classes in the previous version would be the last time we would need to change them to be able to handle any subsequent requests for new speed restriction commands, and we see that she was correct about this – we did not change any of these classes with this new version. Other than the changes required to interface "report_screen" and class "report" to recognize a new command, it was a snap to implement the processing required for the new Fog command. Accordingly, we can attest that our maintenance effort has indeed been simplified by avoiding changes to existing classes and by requiring only the addition of one new class to facilitate the processing for a new speed restriction commands. We have observed the Open/Closed principle for classes "vehicle", "car", "truck" and "vehicle_option" with this new request for a speed constraining command, and would be able to continue with this policy for other such speed constraining commands.

23.7 Exercise 72

Program: ZOOT316G

Title: Objects 316: Design Patterns: Visitor pattern, step 7

Functional requirements

The users have asked for a new command through which they can impose a sun glare restriction to vehicles. When a vehicle is heading into the sun, cars would have their speed reduced to a safer maximum speed of 05, whereas trucks would transition to the stopped state.

Technical requirements

This exercise introduces a new capability through the Visitor design pattern that affects truck instances differently than car instances.

Refer to the associated UML class diagrams. Copy one of the previous versions forward to your student copy of this program and make the following changes:

- Change GUI status REPORT_SCREEN to include new function ESG_DAWN, with text "Dawn" and icon ICON_OVERVIEW, using function slot Shift-F8 (occupied by &XML), and new function ESG_DUSK with text "Dusk", also with icon ICON_OVERVIEW, using function slot Shift-F9 (occupied by &XXL). Neither function is to appear on the Application Toolbar, and should appear in the Menu Bar as Goto > Expose to sun glare > Dawn and Goto > Expose to sun glare > Dusk, respectively, with the Expose to sun glare entry to appear in the Goto menu ahead of the entry for the BACK function.

- Change interface "report_screen" to include new constants to facilitate the commands:
    - Expose to sun glare > Dawn
    - Expose to sun glare > Dusk

  These value need to match the counterpart value defined in the GUI status REPORT_SCREEN.

- Define new class "sun_glare_mitigator", modeled after class "fog_speed_restriction". It is to retain a definition for the maximum safe speed for cars, at 05, but not for trucks. Place the definition of this new class between the “fog_speed_restriction” class and the “vehicle_state” class. It is to define a constructor that will set its heading for vehicles affected by its processing, and a private method "is_heading_into_sun", which will return a true or false value indicating whether the specified vehicle is heading into the sun. Its implementation for "visit_car" is to be the same as that defined for class "fog_speed_restriction" except first it is to invoke its own method "is_heading_into_sun" to determine whether a speed restriction is applicable. Its implementation for "visit_truck" also is to invoke "is_heading_into_sun", but then instead of applying a speed restriction it is to apply a change of state to "stopped".

- Change class "report" to define new methods "expose_to_dawn_sun_glare" and "expose_to_dusk_sun_glare", both of which are to be virtually identical to its method "impose_fog_restriction" except that each one creates a visitor instance of type "sun_glare_mitigator", with the method facilitating dawn sun glare specifying eastbound as the heading while the method facilitating dusk sun glare specifying a westbound heading. Its method "on_user_command" is to detect the commands for dawn and dusk sun glare, invoking its own corresponding methods to accommodate this functionality.

Test the program to insure it still behaves as expected and the use of the “Expose to sun glare” commands will cause the intended change to both cars and trucks, **but test this only with trucks that have no vehicle options selected.**

Note: It should be noted that this program has a bug awaiting discovery. Our implementation to set the state of a truck to "stopped" will work, but the subsequent attempt to "resume" these trucks will work only for those trucks which have no vehicle options defined for them (see values in Descriptor column). Those which have vehicle options will cause a program interrupt during the attempt to "resume". This intentional bug is fixed in the next version.

23.8 Exercise 73

Program: ZOOT316H

Title: Objects 316: Design Patterns: Visitor pattern, step 8

Functional requirements

The users have discovered the bug in the previous version and want this corrected immediately.

Technical requirements

This version fixes the bug found in the preceding version. The implementation we had provided to change the state of a truck used the truck instance itself. This works fine when the truck has no vehicle options, since the memento created for resetting the truck back to its previous state will be created using the truck instance; so when a subsequent "resume" command is issued, the truck instance will be the one for which a match will be sought in the memento table managed by class "fleet_manager". When the truck instance is wrapped by one or more instances of vehicle_option, then it will be the outermost instance of "vehicle_option" for which a memento match will be sought, and no match will be found. Consequently, an attempt to use the reference to the non-existing memento – a null pointer – will cause a program interrupt. To correct this, when the request is made to change the state of the truck to "stopped", the request must be made using the outermost instance of wrapping vehicle option, and not the innermost "truck" instance being used.

The UML diagram does not change with this exercise program. Copy one of the previous versions forward to your student copy of this program and make the following changes:

- Change method "visit_truck" of class "sun_glare_mitigator", for which its "vehicle" parameter is pointing to the innermost instance of a potentially decorated truck, to get the serial number of the truck, then to request the fleet manager for an iterator of its vehicles. Iterate through the vehicle objects provided by the iterator object, retrieving the serial number associated with each vehicle. Upon finding the vehicle object provided by the iterator having the same serial number as the truck object, use that vehicle reference to make the request to change the state of the truck and discontinue with the iterator.

Test the program to insure it behaves as expected and the program interrupt no longer occurs.

Here we see that, yet again, the use of two design patterns being used together – in this case, the memento and visitor – has caused us to reconsider the weaknesses of the simple implementation we might otherwise have used. We had seen the implications of this with the decorator pattern, which required us continually to provide pass-through methods for the “vehicle_option” class with each new method we added to the “vehicle” class (actually, the problem arising between the memento and visitor patterns also is a manifestation of using the decorator pattern). It is a reminder that while design patterns are very powerful and provide significant benefits with software maintenance endeavors, the use of design patterns is a double-edged sword, requiring us always to be vigilant in considering the consequences of multiple design patterns working in concert with each other.

23.9 Exercise 74

Program: ZOOT316I

Title: Objects 316: Design Patterns: Visitor pattern, step 9

Functional requirements

The requirement for this version is identical to the previous version – that is, the user sees no discernible difference. All changes are technical in nature.

Technical requirements

This version corrects a design flaw introduced in a previous version. We were content to indicate that class "sun_glare_mitigator" inherits from class "speed_restriction" simply because we can recognize the similarities it shares with classes "high_winds_speed_restriction", "ice_speed_restriction" and "fog_speed_restriction" which also inherit from class "speed_restriction" -- specifically, it needs to be able to apply a speed restriction to vehicles. The subtle design flaw here to that class "sun_glare_mitigator" applies a speed restriction only to cars; to trucks it applies not a speed restriction but a state change. Accordingly, it is inaccurate to regard class "sun_glare_mitigator" as a specialization of class "speed_restriction" since not all of its public behaviors involve restricting speeds.

This exemplifies an incorrect use of inheritance. Whereas inheritance is one way a class can acquire the ability to make use of the methods provided by another class, this superclass-subclass relationship only makes sense when the type of subclass "is a" type of superclass.

Another way we can provide this capability is through class composition. We can regard the relationship between class "sun_glare_mitigator" and class "speed_restriction" not that the former "is a" the latter, but that the former "has a" or “owns a” the latter. The "is a" relationship denotes class inheritance, whereas the "has a" and “owns a” relationships denote class composition. We already have seen this technique used by other classes defined in previous exercise programs, most notably with the exercises associated with the Adapter design pattern since we saw examples of adapters using both inheritance and composition.

Refer to the associated UML class diagrams. Copy one of the previous versions forward to your student copy of this program and make the following changes:

- Change class "speed_restriction" to expand the visibility of its method "impose_speed_restriction" from protected to public.

- Change class "sun_glare_mitigator" to indicate it no longer inherits from class "speed_restriction", and now includes a private attribute defined as a reference to an instance of class "speed_restriction". Its constructor method is to be changed to create an instance of class "speed_restriction" into this new private attribute, and its "visit_car" method is to be changed to invoke method "impose_speed_restriction" through the instance of "speed_restriction" provided by the new private attribute.

Test the program to insure it behaves as expected.

23.10 Exercise 75

Program: ZOOT316J

Title: Objects 316: Design Patterns: Visitor pattern, step 10

Functional requirements

The users have inquired whether it is possible, after having issued one or more of the speed restriction or sun glare commands in succession, to issue the Resume command repeatedly and have the previous sequence of vehicle speeds reinstated accordingly.

A meeting was held where some of the considerations were discussed. One of the programmers remarked that while the Resume command currently is applicable to all moving vehicle states, its implementation for the “cruising” state causes it to be ignored. Accordingly, if the previous vehicle speeds could be recorded, users could reinstate them via resume only while the vehicle is in the state of “in heavy traffic” or under “police escort”, both of which have overriding implementations for the Resume command. Another person noted that the implementation for the Expose to sun glare command for trucks already causes a state change to “stopped”, and using Resume afterward already reinstates the previous vehicle speed, but the same is not true for cars, which have their speeds further restricted rather than changing state.

After further discussion an agreement was reached where the capability to use Resume to step back through the previous speeds in effect after issuing multiple speed restriction or sun glare commands in succession would be provided in two increments, with only the second increment being made available to the user community:

- First, enable this capability while a vehicle is in a moving state where the Resume command already has associated processing – specifically, the “in heavy traffic” and “police escort” states.
- Second, enable the use of the Resume command while the vehicle is in the “cruising” state.

Technical requirements

This exercise begins the process of applying the Memento design pattern to enable reversing the effects of using the speed restriction or sun glare commands in succession, commands implemented through the Visitor design pattern.

The UML diagram does not change with this exercise program. Copy one of the previous versions forward to your student copy of this program and make the following changes:

- Change class "speed_restriction" to include a new private method “create_vehicle_memento", which is to invoke method "set_vehicle_memento" of the singleton instance of class "fleet_manager". Its implementation for method "impose_speed_restriction" is to be changed to invoke this new private method just before invoking method "accelerate" of the "vehicle" instance.

Test the program using the speed restriction and sun glare commands in the sequence offering the most vehicle speed changes; if you have been using the recommended speed values from the sample exercise programs, this sequence should be as follows:

Command	**Maximum car speed / new state**	**Maximum truck speed / new state**
Impose speed restriction > High winds	500	35
Impose speed restriction > Fog	30	25
Impose speed restriction > Ice	15	10
Expose to sun glare (Dawn will apply only to eastbound vehicles; Dusk will apply only to westbound vehicles)	05	stopped

It should be found that using the Resume command after this sequence for any vehicles in the "cruising" state has no effect, as expected.

It also should be found that using Resume command after this sequence for any vehicles ***with no vehicle options*** in either the "in heavy traffic" or "police escort" states works as expected, stepping back through the previous speeds in effect before the Impose or Expose command was used. Indeed, it should be found that changing from one of these two states to the other at any time during this sequence of commands still will cause the Resume command to reinstate the previous speed or state as applicable.

It also should be found that using Resume command after this sequence for any vehicles ***with vehicle options*** in either the "in heavy traffic" or "police escort" states will result only in a change of state when one is applicable, but **has no effect on reinstating speeds changed while in the same state.** This unexpected behavior is addressed in the next exercise.

23.11 Exercise 76

Program: ZOOT316K

Title: Objects 316: Design Patterns: Visitor pattern, step 11

Functional requirements

The requirement for this version is identical to the previous version – that is, the user sees no discernible difference, but only because this version is not intended to be made available to the users.

Technical requirements

Some unexpected results were found while testing the previous version. After applying a series of speed restriction and sun glare commands for vehicles in the "in heavy traffic" or "police escort" states, the Resume command did not back out through the sequence of changes as expected.

This is the same problem we had seen before. The Resume command works fine when the vehicle has no vehicle options, since the memento created for resetting the vehicle back to its previous speed will be created using the car or truck instance; so when a subsequent "resume" command is issued, the car or truck instance will be the one for which a match will be sought in the memento table managed by class "fleet_manager". When the vehicle instance is wrapped by one or more instances of vehicle_option, then it will be the outermost instance of "vehicle_option" for which a memento match will be sought, and the only match to be found for that is the one to change the state from “in heavy traffic” or “police escort” back to state “cruising”.

The UML diagram does not change with this exercise program. Copy one of the previous versions forward to your student copy of this program and make the following changes:

- Change method "create_vehicle_memento" of class "speed_restriction" to find the outermost decorator object of a decorated vehicle (one containing vehicle options) and use that as the object for which the memento is to be created. The logic it uses is virtually identical to the logic used with method "visit_truck" of class "sun_glare_mitigator", except after locating the outermost vehicle decorator object instead of calling its methods it requests the creation of a memento for it.

Test the program using the speed restriction and sun glare commands in the same sequence offering the most vehicle speed changes.

It should be found that using the Resume command after this sequence for any vehicles in the “cruising” state has no effect, as expected.

It also should be found that using Resume command after this sequence for any vehicles ***with or without vehicle options*** in either the “in heavy traffic” or “police escort” states works as expected, stepping back through the previous speeds in effect before the Impose or Expose command was used.

We are now ready for the next increment – enabling the use of the Resume command while the vehicle is in the “cruising” state.

23.12 Exercise 77

Program: ZOOT316L

Title: Objects 316: Design Patterns: Visitor pattern, step 12

Functional requirements

The users, after having issued one or more of the speed restriction or sun glare commands in succession, may now issue the Resume command repeatedly and have the previous sequence of vehicle speeds reinstated accordingly.

Technical requirements

This is the second increment of the two-increment approach to enable the Resume command to reinstate former speeds after having issued one or more of the speed restriction or sun glare commands in succession.

The UML diagram does not change with this exercise program. Copy one of the previous versions forward to your student copy of this program and make the following changes:

- Inherited method "resume" of class "cruising_state” is to be overridden. Its implementation is to be a copy of the "resume" method implemented for the "in_heavy_traffic" state, with one change -- it is to retrieve the corresponding memento object first and check whether it is bound. Reset processing using the memento is to continue only when the returned memento object is bound.

Checking for a bound memento is what enables backing out through the sequence of mementos for a specific vehicle and stopping when none remain. We never had to bother with this before because all vehicles start in the “cruising” state, and the Resume command always would be able to find a memento to return it to that state, and when once again in the “cruising” state would ignore the Resume command. Now that the Resume command is observed while in the “cruising” state, it becomes necessary to check for the existence of a memento holding a previous speed to be reinstated.

Test the program using the speed restriction and sun glare commands in the same sequence offering the most vehicle speed changes.

It now should be found that using the Resume command after this sequence for any vehicles in the “cruising” state will step back through the previous speeds in effect before the Impose or Expose command was used.