Paper 2680-2015

Object-Oriented Program Design in DS2

Shaun Kaufmann, Farm Credit Canada

ABSTRACT

The DS2 programming language was introduced as part of the SAS® 9.4 release. Although this new language introduced many significant advancements, one of the most overlooked features is the addition of object-oriented programming constructs. Specifically, the addition of user-defined packages and methods enables programmers to create their own objects, greatly increasing the opportunity for code reuse and decreasing both development and QA duration. In addition, using this object-oriented approach provides a powerful design methodology where objects closely resemble the real-world entities that they model, leading to programs that are easier to understand and maintain. This paper introduces the object-oriented programming paradigm in a three-step manner. First, the key object-oriented features found in the DS2 language are introduced, and the value each provides is discussed. Next, these object-oriented concepts are demonstrated through the creation of a blackjack simulation where the players, the dealer, and the deck are modeled and coded as objects. Finally, a credit risk scoring object is presented to demonstrate the application of this approach in a real-world setting.

INTRODUCTION

The DS2 programming language was introduced as part of the Base SAS 9.4 release to better enable advanced data manipulation. One of the most powerful features of the language was the addition of the package and method programming constructs. These constructs allow the programmer to write code using a much more structured approach than previously possible, better enabling both a modular design and data encapsulation. These new tools allow a programmer to break up a complex problem into a set of smaller modules that are easier to design, implement and test. Additionally, a modular design better lends itself to code reuse and standardization while simultaneously making a program easier to understand and maintain.

However, the power of the package and method programming constructs goes beyond simply providing the ability to write a well-structured program. Using packages and methods, the programmer is able to employ an object-oriented approach to program design. Specifically, object-oriented programming methodology is a development approach based on the concept that systems can be built from a collection of reusable components called objects. This software development approach leads to the creation of code that is more reusable and readable. It is the opportunity to employ an object-oriented approach to program design and implementation that is the focus of this paper.

It is worth noting that working with objects is not an entirely new concept for many SAS programmers. Anyone who has developed applications with SAS/AF software and the SAS Component Object Model (SCOM) is likely completely comfortable with object-orented programming concepts. Additionally SAS provides five predefined component objects for use in the DATA step. These include the hash and hash iterator objects, the Java object, as well as the logger and appender objects. These component objects provide the programmer with powerful prebuilt tools that can be leveraged to reduce the time required for code development. The blackjack example program presented later in this paper uses hash objects to implement much of its required functionality while at the same time further reinforcing object-oriented concepts. However, what makes the addition of package and method constructs within the DS2 language revolutionary is that for the first time the SAS programmers are capable of developing their own user-defined objects for the purpose of data preparation.

The goal of this paper is to demonstrate the power and elegance of object-oriented design. The reader is assumed to have a basic understanding of the DS2 language. However, to make the content accessible to a broader audience, an effort was made to make this paper as self-contained as possible. The next section reviews some fundamental concepts of the DS2 language that are necessary to facilitate a discussion of object-oriented design.

A FOUNDATION IN DS2

As a level-setting exercise, this section reviews the concepts necessary to facilitate an introduction to object-oriented design. For brevity, a list of definitions is presented in Table 1. Readers who are comfortable with these definitions may wish to skip the remainder of this section. However if any of these concepts are unclear, it is worthwhile to continue through the subsequent example program where these concepts are demonstrated in a more concrete fashion.

Term	Description
Package	DS2 packages are language constructs that bundle variables and methods into named objects that can be stored and reused by other DS2 programs. A DS2 package is a template to construct an instance of the package (an object). With a DS2 package you can approximate the encapsulation and abstraction of object-oriented classes.
Package Instance	A package instance (often referred to as an object in other languages) is a particular instance of a package. The object can be a combination of variables, functions, and data structures. For the purposes of this paper the terms *package instance* and *object* will be used interchangeably.
Instantiate	To create a package instance (object) from a package.
Package Variable	A variable whose data type is a reference to a type of package.
Encapsulation	The grouping of related concepts into one item, such as a class or package
Constructor	A special package method that is used during construction of a package instance. A package's constructor method has the same name as the package and initializes a newly constructed package instance.
Default Constructor	A constructor that takes no parameters.
Parameter	A special kind of variable used in a call to a method to refer to one of the pieces of data provided as input to the method.
Attribute	Something that an object knows (data/information)
Method	A procedure associated with a package object
System Method	Also known as predefined methods, these methods provide the structural and functional framework for your program to execute. These methods provide special functionality such as implicit looping.
User-Defined Method	Similar to functions, procedures, and subroutines in other languages, these are the methods that you create or that someone else created for reuse.

Table 1. Definition of Terms.

Additionally, readers are encouraged to the explore the SAS DS2 Language Reference (available on the SAS website) for a complete introduction to the language. Specifically, chapter 3 of the document provides an excellent introduction to DS2 by presenting a series of example programs which introduce syntax and concepts. Here we follow a similar approach in an attempt to illustrate the foundational concepts presented in Table 1. Examine the "Hello World" program in Figure 1. Using this sample code we will reinforce the above concepts by linking each to a concrete application.

A First Example - "Hello World!"

```
proc ds2;
   package logMessage /overwrite=yes;
        dcl varchar(100) message;

        FORWARD setMessage;
        FORWARD putMessage;

        method logMessage();
             setMessage('Hello World - from the constructor.');
             putMessage();
        end;

        method setMessage(varchar(100) message);
             this.message = message;
        end;

        method putMessage();
             put message;
        end;
   endpackage;
   run;

data _null_;
   dcl package logMessage m();

   method init();
        m.setMessage('Hello World - from the init() method.');
        m.putMessage();
   end;

   method run();
        m.setMessage('Hello World - from the run() method.');
        m.putMessage();
   end;

   method term();
        m.setMessage('Hello World - from the term() method.');
        m.putMessage();
   end;

enddata;
run;

quit;
```

Figure 1. The “Hello World!” example.

When examining the above program, the reader is encouraged to make note of the following key points;

First, when writing DS2 code in Base SAS, the code is contained within a *DS2 procedure*. To be specific, all the code is placed within the framework presented in Figure 2.

```
proc ds2;
... DS2 statements ...
run;
quit;
```

Figure 2. The Proc DS2 Framework.

Additionally, a DS2 procedure contains a data program portion that is similar to the original SAS DATA step in many ways. Within the DS2 framework the data program code block begins with the DATA statement and ends with the ENDDATA statement. The reader is encouraged to see chapter 2 of the *DS2 Language Reference* (entitled *DS2 and the DATA Step*) for a complete description of the similarities and differences between the DS2 language and DATA step.

```
data _null_;

    method init();
    end;

    method run();
    end;

    method term();
    end;

enddata;
```

Figure 3. Simplified Data Program.

A simplified data program is show in Figure 3. Notice that it contains three system methods: the INIT, RUN and TERM methods. Every DS2 program will contain either, implicitly or explicitly, these three methods. These methods provide a structured framework for the data program.

In DS2 the implicit loop is represented by the RUN method, the INIT method contains initialization code and TERM method contains finalization code. The reader is encouraged to see chapter 6 of the DS2 Language reference for a complete explanation of DS2 programming semantics.

The data program portion of the “Hello World!” program is show in Figure 4.

```
data _null_;

    dcl package logMessage m();

    method init();
          m.setMessage('Hello World - from the init() method.');
          m.putMessage();
    end;

    method run();
          m.setMessage('Hello World - from the run() method.');
          m.putMessage();
    end;

    method term();
          m.setMessage('Hello World - from the term() method.');
          m.putMessage();
    end;
enddata;
```

Figure 4. The "Hello World!" Data Program.

As a first step, the package reference variable *m* was declared and an instance of the package instantiated. It is worth noting that there is a difference between declaring a package reference variable and instantiating an instance of the package. A package variable *m* of type logMessage is declared in Figure 5. Note the lack of parenthesis following the *m*.

```
dcl package logMessage m;
```

Figure 5. Declaring a Package Variable.

To be clear, the package variable *m* can now be used to reference a specific instance of the logMessage package, but it doesn't yet since no instance of the package has been created. It is first necessary to instantiate an instance of the package for *m* to reference, as shown in Figure 6.

```
m = _NEW_ logMessage();
```

Figure 6. Instantiating an Instance of the User-Defined logMessage Package.

Here we instantiate an instance of the logMessage package by calling the packages default constructor logMessage() using the _NEW_ operator. These two steps can alternatively be combined in a single line of code as show in Figure 7. There it is important to note that the code presented in Figure 7 differs from that in Figure 5 since here the *m* is followed by a set of parenthesis. This denotes the calling of the constructor at the same time that the variable is declared.

```
dcl package logMessage m();
```

Figure 7. Combining Declaration with Instantiation.

Now that an instance of the logMessage has been instantiated, we can make use of the instance's (object's) methods. Each of the system methods in the "Hello World!" program performs two calls to the logMessage object referenced by *m*. The package's methods are accessed by what is referred to as *dot notation* as show in Figure 8.

```
m.setMessage('Hello World - from the init() method.');
m.putMessage();
```

Figure 8. Calling an Object's Methods using Dot Notation.

In the first line of code the setMessage method is called with a character constant of 'Hello World - from the init() method.' passed as a parameter. In the second line of code the putMessage method is called. Note that, unlike the setMessage method, the putMessage method does not require a parameter. Whether a method requires a parameter or not is determined by its declaration in the user-defined package. This can be observed by examining the logMessage package in Figure 9.

```
package logMessage /overwrite=yes;

 dcl varchar(100) message;

 FORWARD setMessage;
 FORWARD putMessage;

 method logMessage();
       setMessage('Hello World - from the constructor.');
       putMessage();
 end;

 method setMessage(varchar(100) message);
       this.message = message;
 end;

 method putMessage();
       put message;
 end;

endpackage;
```

Figure 9. The User-Defined logMessage Package.

The user-defined logMessage package introduces a number of important concepts:

- Unlike traditional DATA step, DS2 declares variables using a *declare* or *dcl* statement. The declaration requires the user to specify a datatype. This datatype can be one of the standard DS2 datatypes such as a varchar datatype or a user-defined package datatype as illustrated previously.
- Forward declaration is also demonstrated. Here using the FORWARD statement enables the setMessage method to be defined after methods that reference it in the code.
- The setMessage() method uses the THIS operator to distinguish the global variable MESSAGE from the parameter that is named MESSAGE.

Hopefully this review of DS2 language syntax and concepts will help the reader avoid some common pitfalls and provide a foundation sufficient to enable the discussion of object-oriented program design that follows in the next section.

THE CHALLENGE WITH TEACHING OBJECT-ORIENTED PROGRAMMING

Before we start to explore object-oriented design it is useful to note a common pitfall often encountered when teaching object-oriented concepts. Specifically, there is a tendency to start the process by equating methods with functions, procedures and subroutines. On the surface this appears to be a logical approach, since using this commonality as a starting point introduces the subject area by drawing parallels to concepts that the programmer already understands. It then often follows to mention that packages (or classes) are collections of related methods. It is certainly true that relating new concepts to known parallels is an effective teaching approach. However, the problem with the above approach is that it is too effective in getting the programmer up and running in the new language. This approach fails to communicate how object-oriented programming is fundamentally different from non object-oriented approaches. By failing to address this point, we run the risk of enabling programmers to continue to write non object-oriented code using the new object-oriented constructs. In this case we are quickly on-boarding the programmer in the short-term, but failing them in the long run because much of the power of the object-oriented paradigm is obscured.

A more effective approach to teaching object-oriented programming is to start fresh by first introducing the concept of the object. Consider the following description of object-oriented programming taken from Wikipedia:

"Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects", which are data structures that contain data, in the form of fields, often known as attributes; and code, in the form of procedures, often known as methods. "

By starting with the concept of the object having attributes (data) that can persist for the life of the object we can start to communicate the uniqueness of the object-oriented approach. Once this concept is absorbed we can introduce methods (procedures), but framing their introduction in the context of how we use methods to manipulate (change the state of) an objects attributes. So in order to focus on the object, it is helpful to start with a focus on design instead of code.

WHAT IS OBJECT-ORIENTED DESIGN?

Objects are the key to understanding object-oriented design. Simply put, an object (in program design) is a representation of a real-world object. Real-world objects have two characteristics: state and behavior. For example, consider your dog. Dogs have state (name, color, etc) and behavior (barking, running, etc). Identifying the state and behavior for real-world objects is a great way to begin thinking in terms of object-oriented programming.

In an object-oriented approach, requirements gathering is organized around objects and these objects are modeled after the real-world objects that they represent. Therefore, the primary tasks in object-oriented design can be thought of as: identifying objects, defining objects' attributes, defining objects' actions and identifying how objects interact with each other. To illustrate this approach it is helpful to work through a concrete example. To this end, we will consider the card game blackjack as an example problem domain to model. For the purpose of this demonstration, we will focus on the somewhat simplified representation of the game. Specifically for our purposes we will assume that the game is played between a *dealer* and one *player*, where each have one *hand* of *cards* obtained from a single *deck* (no shoe).

The first step in constructing a simulation of the game is to determine what objects will be necessary. Notice in the above description of our simplified game that I highlighted dealer, player, hand, cards and deck in italics. These indicate the real-world entities that we will need to model as objects in the simulation of the game. We will consider each of these five, with a focus on: what are the objects' attributes, what are their behaviors, and how will each object interact with other objects.

What to Notice

As we work through the blackjack example below, you will see "What to Notice" commentary intermingled with the code. These comments are included to create linkages between somewhat abstract object-oriented concepts and actual implementation details. As an example, here are a few key concepts that are often overlooked:

- The _NEW_ and THIS operators and the role they play in encapsulation and scope.
- How packages become objects (the difference between declaration and instantiation).
- Special kinds of methods called constructors and what it means to overload them.
- Objects can be passed as parameters to methods and returned by methods.

DETERMINING THE OBJECT'S ATTRIBUTES

We will start modeling the card object by considering its attributes. Specifically, a card has:

a suit - (Hearts, Clubs, Spades, or Diamonds), and

a type - (Ace, Two, Three, …, Queen, King).

Therefore our card object will have the following attributes: a suit and a type.

Turning our attention to the deck and hand objects, both are comprised entirely of a collection of cards. When we later implement these objects, each will need a data structure capable of supporting a collection of cards. However at this point we are not worried about implementation details and for now it is sufficient to recognize that the only attribute these objects require is a collection of cards.

Next consider a real player in the game of blackjack. Generally speaking a player has a pool of money (or chips) for betting with and one or more hands of cards to play. However, in order to ensure that this example is as simple to understand as possible, we will ignore the betting component of the game. Additionally, we have further constrained our simulation by specifying that the player only has a single hand per game. Therefore, the only attribute that our player object requires is a hand.

Similar to the player object, our dealer object will require a hand to play as well and therefore will require a hand attribute. However, unlike the player, the dealer requires a deck to deal from in addition to the hand to play. To summarize, the dealer requires two attributes: a hand, and a deck.

At this point we should start to recognize a pattern. Our design process started by creating the concept of a card object. Next a collection of card objects is utilized as an attribute in both the hand and deck objects. Finally, both the dealer and the player make use of the hand object as an attribute. This provides a concrete demonstration of code reuse in object-oriented program design.

DETERMINING THE OBJECT'S METHODS

Having determined each object's attributes, we now turn our attention to the actions that each object must be able to perform.

The card object is an example of a very simple object. Specifically a card object actually does nothing, it just needs to exist and have attributes. The way we enable an object to come into existence is to implement one or more constructor methods. This will be a property of all our user-defined objects and this concept will be discussed further when examining each package's code later in this paper.

The hand object is slightly more involved than a card object in that it needs to be able to add each card that it is dealt to its collection of cards. To this end we will need an add_card method when we implement the package.

The deck object is conceptually similar to the hand object, but instead of adding a card it need to be able to remove the top card as part of the dealing process. We will need to implement a method to achieve this. It is the combination of adding a card to a hand object while simultaneously removing that card from the top of the deck object that will facilitate the act of dealing.

In our simplified version of the game, the only action that the player needs to be able do is to request an additional card. In blackjack this is referred to as a hit, so we will implement a hit method for the player object.

Since our simplified blackjack game forgoes betting, the dealer simply has to be able to deal a card, either to the player's hand or to the dealer's own hand.

IMPLEMENTING THE OBJECTS

Having completed the analysis and design exercise in the preceding section, we can now turn our attention to implementation. In DS2, user-defined objects are implemented using the package construct. In this section we present an implementation for the card, hand, deck, player and dealer objects. As always there are many different possible implementation approaches. The code presented below represents just one possibility.

```
package Card /overwrite=yes;
      DCL varchar(10)     suit type;

      /* Overloaded the Constructor     */
      method Card(varchar(10) suit_type, varchar(10) card_type);
             suit = suit_type;
             type = card_type;
      end;

      /* Default Constructor                  */
      method Card();
      end;
  endpackage;
```

Figure 10. The Card Package.

What to Notice – Figure 10.

- The card object's attributes are first declared. Both the suit and type attributes specify a varchar of length 10 as the datatype.
- The card package has two constructors: a default constructor capable of creating a new card without specifying the suit and card type at creation, and an overloaded alternative constructor that takes a suit and a type as constructor parameters.

```
package Hand /overwrite=yes;
    DCL package hash h_hand();
    DCL integer card_key;
    DCL package Card new_Card;

    /* Hand constructor */
    method Hand();
           card_key = 1;
           h_hand.defineKey('card_key');
           h_hand.defineData('card_key');
           h_hand.defineData('new_Card');
           h_hand.defineDone();
    end;
    method add_card(package Card in_card);
           new_Card = in_card;
           h_hand.add();
           card_key = card_key + 1;
    end;
endpackage;
```

Figure 11. The Hand Package.

What to Notice – Figure 11.

- A hash object, called h_hand, is used to store the collection of cards in the hand.
- A new_Card package variable is declared. It is used as a hash data element and to reference the card object that is passed in as a parameter to the add_card method.
- The hand constructor handles the hash definition activities.
- The add_card method takes a card object as a parameter.

```
package Deck  /overwrite=yes;

    DCL package hash h_deck();
    DCL package Card new_Card;
    DCL integer card_index top_card_index number_of_items x y;
    DCL varchar(10) suit_list[4] card_type_list[13];

    method Deck();
           card_index = 1;
           top_card_index = 1;

           h_deck.defineKey('card_index');
           h_deck.defineData('card_index');
           h_deck.defineData('new_Card');
           h_deck.defineDone();

           suit_list := ('Hearts', 'Clubs', 'Spades','Diamonds');
           card_type_list := ('Ace', 'Two', 'Three', 'Four', 'Five', 'Six',
                             'Seven', 'Eight', 'Nine', 'Ten', 'Jack',
                             'Queen','King');

           do x = 1 to 4;
              do y = 1 to 13;
                new_Card = _NEW_ [THIS] Card(suit_list[x], card_type_list[y]);
                h_deck.add();
                card_index = card_index + 1;
              end;
           end;
    end;

    method top_card() returns package Card;
                  card_index = top_card_index;
                  h_deck.find();
                  top_card_index = top_card_index + 1;
                  return new_card;
    end;
endpackage;
```

Figure 12. The Deck Package.

What to Notice – Figure 12.

- A hash object, called h_deck, is used to store the collection of cards in the hand.
- A new_card package variable is declared to reference cards that are dynamically created using the _NEW_ operator and subsequently added to the h_deck hash. Using the [THIS] operator, specifies that each new instance of the card object has global scope.
- The hand constructor handles the hash definition activities, specifies the set of values possible for the suit and type array elements, creates a card for each of the 52 cards and adds them to the h_deck hash object.
- The top_card method returns a card object as a parameter.

```
package Player  /overwrite=yes;

    dcl package Hand my_hand();

    method hit(package Dealer in_Dealer, package Hand in_hand);
           in_Dealer.deal_card(in_hand);
    end;

endpackage;
```

Figure 13. The Player Package.

What to Notice – Figure 13.

- A Hand object, called my_hand, is declared and instantiated to store the player's cards.
- No constructor is explicitly implemented. Therefore the compiler will generate a default constructor.
- The hit method takes both a dealer object and a hand object as parameters, then calls the dealer's deal_card method to provide a card to the hand that was specified as a parameter.

```
package Dealer  /overwrite=yes;

    dcl package Deck the_deck();
    dcl package Hand my_hand();

    method Dealer(package Hand player_hand);
           put 'Card to Sam:';
           player_hand.add_card(the_deck.top_card());
           put 'Hole card to Dealer:';
           my_hand.add_card(the_deck.top_card());
           put 'Card to Sam:';
           player_hand.add_card(the_deck.top_card());
           put 'Card to Dealer:';
           my_hand.add_card(the_deck.top_card());
    end;
    method deal_card(package Hand in_Hand);
           in_Hand.add_card(the_deck.top_card());
    end;
endpackage;
```

Figure 14. The Dealer Package.

What to Notice – Figure 14.

- The dealer object has both a hand and a deck to deal from.
- The constructor takes as a parameter the player's hand that it will be dealing to, then sets up the game by dealing two cards each to the player's and dealer's hands.
- The hit method takes both a dealer object and a hand object as parameters, then calls the dealer object's deal_card method to deal a card to the hand object that was specified as a parameter.
- Put statements are not necessary for the functioning of the object, but are included to facilitate understanding when viewing the log.

USING THE USER-DEFINED OBJECTS

The code presented in the preceding section provides an implementation for the five user-defied objects necessary to run a simulation of our simplified blackjack card game. In order to demonstrate the functioning of these objects a data program is required and is presented in Figure 15. This code simply scripts a simple game scenario. Figure 16 presents the log of this script along with commentary describing the game state at each point.

```
data _null_;
   dcl package Player Sam();
   dcl package Dealer our_Dealer(Sam.my_hand);

   method run();
         put 'Card to Sam:';
         Sam.hit(our_Dealer, Sam.my_hand);
         put 'Card to Sam:';
         Sam.hit(our_Dealer, Sam.my_hand);
         /* Sam decides to stay     */

         put 'Card to Dealer:';
         our_Dealer.deal_card(our_Dealer.my_hand);
         put 'Card to Dealer:';
         our_Dealer.deal_card(our_Dealer.my_hand);
   end;
enddata;
run;
```

Figure 15. The Data Program – Playing The Game.

```
Card to Sam:
Ace  of  Hearts           ← Sam is counting either 1 or 11.

Hole card to Dealer:
Two  of  Hearts           ← The Dealer’s hole card is a 2 of hearts.

Card to Sam:
Three  of  Hearts         ← Sam is now counting either 4 or 14.

Card to Dealer:
Four  of  Hearts

Card to Sam:
Five  of  Hearts          ← Sam is counting 9 or 19 and decides to stay.

Card to Dealer:
Six  of  Hearts           ← The dealer takes a card and is counting 12.

Card to Dealer:
Seven  of  Hearts         ← The dealer receives a seven and has 19 as well
                            and the game ends in a tie.
```

Figure 16. Log and Author Commentary.

The log output demonstrates that the functionality of our user-defined objects is sufficient for the purpose of this object-oriented design exercise. However, it also serves to suggest additional functionality that could be added to enhance the game. First, the deck is obviously not shuffled. Shuffle functionality could be added, possibly to the deck object's constructor. Second, it would be useful if the hand object could keep track of its current value. Finally, game strategy could be added to both the player and dealer objects. These enhancements could be explored by the reader as a learning exercise.

A REAL WORLD EXAMPLE – THE CREDIT SCORING OBJECT

The previous sections of this paper discussed how to use an object-oriented approach to program design. This section discusses some of the business reasons why one would want to. For this discussion we will leverage the problem domain of credit scoring models. Specifically, credit scoring is a function of financial institutions and includes such subdomains as application approval scorecards, probability of default models, loss given default models and pricing models. These subdomains have some subtle properties that might not be obvious to all readers. Two key properties are:

- Models of this type are proprietary and therefore it is important to guard the details of the model to ensure the continued effectiveness of the models.
- These models are of interest and of value to other areas of the company and therefore being able to share the execution of these models has business value.

So to summarize we have a desire to allow other business units to execute the models (for purposes such as scenario analysis) but also a need to not share the inner workings of the models. Given these two business objectives, let's examine how some of the fundamental concepts of object-oriented design can help to achieve these goals.

In the introduction we stated that an object-oriented approach provides for a modular design that enables encapsulation and standardization while building reusable components called object. Here it is important to note that encapsulation is synonymous with information hiding. When compiled, DS2 objects are encrypted by default. If the object is compiled to a shared location, then other business units can reuse the object in their code without having access to the implementation details. In this case code reuse is expanded to mean component reuse, achieving standardization of models across business units. All that is necessary is to document and share the object model so that the users understand how to run the objects. In this manner both the above business objectives are met by the object-oriented development approach in DS2.

CONCLUSION

With the addition of the DS2 language to the Base SAS environment, SAS programmers are for the first time able to utilize an object-oriented approach to program design. Using this object-oriented approach, programs are easier to design, write and maintain. Additionally code reuse can become component reuse, facilitating code standardization across business units while ensuring proprietary algorithms remain obfuscated through encapsulation and information hiding.

REFERENCES

SAS Institute Inc. "SAS® 9.4 Component Objects Reference." Accessed March 22, 2015.
http://support.sas.com/documentation/cdl/en/lecompobjref/67221/PDF/default/lecompobjref.pdf

SAS Institute Inc. "SAS® 9.4 DS2 Language Reference." Accessed March 22, 2015.
http://support.sas.com/documentation/cdl/en/ds2ref/68056/PDF/default/ds2ref.pdf

Wikipedia: The Free Encyclopedia. "Object-Oriented Programming." Accessed March 22, 2015.
http://en.wikipedia.org/wiki/Object-oriented_programming

ACKNOWLEDGMENTS

I'd like to thank Christine Gamble and Andrew Dyck for their efforts in editing this paper.

RECOMMENDED READING

- SAS® 9.4 DS2 Language Reference
- SAS® 9.4 DS2 Component Objects Reference