The Reference List Formatter: An Object-Oriented Development Project

Kevin R. Parker
Idaho State University, Pocatello, Idaho, USA

parkerkr@isu.edu

Abstract

Course projects that are manageable yet still sufficiently comprehensive are often difficult to find or develop. This can be especially true for a course in object-oriented development, since it involves a wide range of critical topics such as abstraction, encapsulation, inheritance, and polymorphism, as well as aggregation and composition, arrays of objects, abstract classes and interfaces, and object persistence. It is difficult to find a project that is broad enough to cover all the topics but at the same time also narrow enough to cover them thoroughly in a one semester course. One alternative is the use of individual "toy" problems for each concept, but that approach has been criticized as simplifying problems to the point where they are no longer realistic and lack useful substance. Another alternative is to use real-world projects in a class, but that approach can introduce unmanageable complexity or ambiguity into the classroom. This paper communicates the details of an object-oriented course project that has been developed and refined to provide a project-based learning component to reinforce course content. The project's deliverables have been designed to cover every concept that is included in an object-oriented development course and provide students with experience with each. Its refinement has taken place in multiple Java-based and VB.Net-based object-oriented development courses.

Keywords: object-oriented development, semester project, experiential learning, programming education, project-based learning.

Introduction

Object-Oriented (OO) Programming is a programming language paradigm based on the concept that procedures and the data on which they operate can be viewed as two parts of the same thing: an object. An object-oriented programming language is generally characterized by four distinguishing features –abstraction, encapsulation, inheritance, and polymorphism.

- abstraction: ignoring irrelevant features, properties, or functions and emphasizing those that are relevant to the given project.
- encapsulation: the act of grouping into a single object both the data and the operations that affect that data.
- inheritance: the process of creating new classes from existing classes by absorbing their attributes and behaviors and enhancing these with capabilities the new classes require.

Editor: Eli Cohen

- polymorphism: the ability to perform the same operation on different types of objects, each one implementing that operation in a way appropriate to them.

The above principles are common topics in many object-oriented design classes. Other essential concepts that should be addressed in an OO design course include aggregation and composition, arrays of objects, abstract classes and interfaces, and object storage.

It can be difficult to find or formulate a project for an object-oriented development class that covers all the fundamental concepts. Over the course of several years the author has developed and refined a course project that provides a project-based learning component to supplement course materials. It is designed to cover and reinforce every major concept in a typical object-oriented development course and provide students with experience with each, and has been refined over the course of multiple semesters in classes using both Java and Visual Basic .NET.

Background

A common problem in software development education is that many programming courses limit student exercises to "toy" problems, a term that is often used to refer to unrealistically easy and ultimately useless exercises (Ghezzi, & Mandrioli, 2005). This practice can lead students to underestimate the complexities of real-world programming applications (Ensmenger, 2004). Many researchers believe that toy problems simplify a real-world problem to such an extent that nearly all interesting aspects are altered. Moreover, because most toy problems focus on only one or two techniques, solving them does not entail exploring how different approaches might interact with one another or how solutions might be integrated. In addition, exclusively solving toy problems prevents students from realizing how various techniques will scale up to larger problems (Davis, & Morgenstern, 2004). To avoid this problem it was decided to develop a more realistic and intricate project. Problems that are of a large enough scale and are well chosen can be quite useful and challenging. There are many successful examples often derived from real-life problems but cleansed of irrelevant and distracting details (Jazayeri, 2004).

Once the decision was made to use a larger-scale representative project, a review of project-based learning was undertaken. A literature search shows a longstanding tradition for "doing projects," or incorporating "hands-on" activities (Thomas, 2000). A review of literature concerning project-based learning reveals considerable diversity of defining features. There are similarities between models referred to as project-based learning and models referred to with other labels, such as "intentional learning" (Scardamalia, & Bereiter, 1991), "design experiments" (Brown, 1992), and "problem-based learning" (Gallagher, Stepien, & Rosenthal, 1992). Additional similar approaches include "project-focused," "experiential education," and "active learning" (Thomas, 2000). While purists might argue whether the project described here is project-based learning, experiential education, intentional learning, or something else entirely, it will be referred to herein as project-based learning, because the approach used is in line with the discussion that follows.

Project-based learning is an instructional method that organizes learning around projects (Thomas, 2000). Projects are generally complex tasks, based on challenging questions or problems, and involve students in design, problem-solving, decision making, or investigative activities. Further, they culminate in realistic results or presentations (Jones, Rasmussen, & Moffitt, 1997; Thomas, Mergendoller, & Michaelson, 1999). Project-based learning at universities arose in the 1970s in Europe, based on the idea that the best form of professional development is learning by doing (Von Kotz, & Cooper, 2000). Project-based learning approaches are based on constructivist theory (Henze & Nejdl, 1997). The focus of constructivism is what actual people in a knowledge domain and in a real-life context typically do (Bednar, Cunningham, Duffy, & Perry, 1992). The basic foundations of constructivism include situated cognition and cognitive apprenticeship, both of which are incorporated into project-based learning. Situated cognition is a form of thinking

that is anchored in real-world contexts, with the learning of content embedded in the use of the content. In project-based learning, students learn content knowledge, skills, and dispositions in the process of working through a realistic project modeled on a scenario that might be encountered in the "real world." Cognitive apprenticeship is a teaching approach in which the assignment models the processes students are learning, and students are directed toward expert performance (Cavanaugh, 2004).

The nature of project-based learning is developing skills and content by engaging in tasks that involve the skills and content to be learned and that provide real-world context for learning (Warlick, 1999). Project-based learning requires students to assume a real-life role and apply the tools of a knowledge domain in completing a project. Project-based learning provides a context in which students move toward thinking as a knowledge domain expert might think (Cavanaugh, 2004). The opportunity to apply learning to a real-life situation facilitates the transfer of learning (Fisher, & Frey, 2007). Project-based learning requires students to deal with complex questions and undertake projects that involve synthesizing understandings and considering real-world issues.

Because of the complexity and size of the intended project it was decided that the project would be made up of a series of deliverables, each of which builds on previous deliverables. In this context deliverables are the "measurable results of intermediate activities within the project" (Association of Project Management, 2000). In educational projects a deliverable is a task that is instrumental in learning how to apply concepts and methodological approaches related to a particular task, i.e., an educational objective is reinforced by the corresponding deliverable (Piccinini, & Scollo, 2006). A single final deliverable, coupled with students' tendency to procrastinate, might lead to incomplete or shoddy results. Further, a series of deliverables gives students manageable "chunks" to process, permits those chunks to be assigned in close conjunction with relevant course content, and enables students to develop a more realistic system than most toy projects allow. The fact that each deliverable builds on the previous deliverable can lead to a problem known as deliverable dependencies, which is defined by Turk and Vaishnavi (2000) as a situation in which deliverable_n cannot be undertaken until deliverable_{n-1} has been completed. This issue has been somewhat successfully addressed in the classroom by providing a representative solution for each deliverable as soon as all students have turned in their assignments. This approach helps to level the playing field and avoid compounding student grade deductions if they erred on earlier deliverables. It also helps the student to see good programming practices, like well documented code, good variable naming practices, etc., so that they learn to develop code with an eye toward maintainability and modifiability.

Overview of Requirements

The objective of this project is to develop an object-oriented program that generates a formatted reference list. A reference list appears at the end of most research papers to provide the information necessary for a reader to locate and retrieve any source cited in the body of the paper. Most students enrolled in tertiary institutions have experience writing term papers and should, therefore, be familiar with reference lists. A reference list generally includes the author name or names, the title of the work, the resource type, and publication information in a consistent and recognizable form. References often include a wide range of resource materials – books, journal articles, conference proceedings, Internet sources, films, information services, and many others. Unlike a bibliography, which is often either a comprehensive or in-depth survey of works available on a particular subject or research area or a list of all materials that have been consulted during the course of conducting research for a project or a paper, a reference list usually includes only those works actually referenced in a paper or manuscript (Oberon Development, 2006).

There is a myriad of styles for reference formatting. Among the most common are (Delaney, 2009):

- APA (American Psychological Association): used in psychology, education, and other social sciences.
- MLA (Modern Language Association): used in literature, arts, and humanities.
- AMA (American Medical Association): used in medicine, health, and biological sciences.
- Turabian: designed for college students for use with all subjects.
- Chicago: used with all subjects in the "real world" by books, magazines, newspapers, and other non-scholarly publications.

There are several additional specialized formatting styles like IEEE (Institute of Electrical and Electronics Engineers), ACS (American Chemical Society), CSE (Council of Science Editors), SAA (Society of American Archaeology), ASA (American Sociological Association), APS (American Physical Society), ASABE (American Society of Agricultural and Biological Engineers) , AAA (American Anthropological Association), and others even more specialized and obscure.

The project focuses on the development of a software application to format a reference list. In its current form it includes a base class called clsReference, as well as a clsBook and clsJournal that inherit from clsReference. There is also a clsChapter that inherits from clsBook. All classes implement an abstract interface that provides MLA, APA, and Chicago formatting capabilities. The project is extensible in that it can easily be expanded (and has been in classroom use) to include additional reference types as well as additional formatting styles.

The project consists of a series of deliverables, each of which builds on previous deliverables until a complete system has been developed and tested. In this way students gain experience developing a larger system than most toy projects allow. Project requirements were carefully constructed so that each deliverable addresses specific design concepts. The deliverables focus on, in order, abstraction, encapsulation, arrays of objects, aggregation, abstract interfaces and inheritance, polymorphism, and object persistence. Error handling can be interwoven throughout the project or included as an additional deliverable.

Table 1 provides a summary of the deliverables, noting the concept that each deliverable is designed to reinforce, and providing an overview of the deliverable.

Deliverable 1 focuses on abstraction. The student will gain experience designing classes. It requires the abstraction of a programmer-defined clsName class to allow the storage and display of author names in various formats. In addition, it requires the design of a programmer-defined clsDate class to allow the storage and display of publication dates in different formats.

Deliverable 2 focuses on encapsulation. The student will gain experience encapsulating classes and instantiating objects from those classes. It requires the creation of a programmer-defined clsName class to allow the storage and display of author names in various formats. In addition, it requires development of a programmer-defined clsDate class to allow the storage and display of publication dates in different formats.

Deliverable 3 provides the student with experience modifying classes, setting up an ArrayList of objects, and sorting the ArrayList. Specifications require the modification of the clsName class so that it implements the IComparable interface and includes a CompareTo method. It also requires instantiating an ArrayList of clsName objects, providing values for the names, and then sorting the ArrayList of objects in order by last name and first name.

Table 1: Summary of Deliverables

Concept Reinforced	Brief Overview
Abstraction	The student will gain experience designing classes.
Encapsulation	The student will gain experience encapsulating classes and instantiating objects from those classes.
Data Structures of Objects	Provides the student with experience modifying classes, setting up an ArrayList of objects, and sorting the ArrayList.
Aggregation	Provides students with additional experience with classes and object instantiation, as well as utilizing aggregation to create classes comprised of existing classes to demonstrate code reuse.
Inheritance and Interfaces	Reinforces the concept of abstract classes and abstract interfaces and provides students with experience implementing inheritance through building new classes that are based on existing classes.
Polymorphism	Students will gain experience manipulating data structures of objects in order to gain an appreciation of the fact that polymorphism allows objects derived from the same base class to be manipulated based on their actual type, in spite of the fact that the specific type may be unknown.
Persistent Objects	Requires that students learn how to create persistent objects using the binary formatter or Simple Object Access Protocol (SOAP) and data files.

Deliverable 4 focuses on the concept of aggregation. It provides students with additional experience with classes and object instantiation, as well as utilizing aggregation to take advantage of code reuse. Students are required to develop a new class called clsReference that includes as instance variables an ArrayList of clsNames to represent authors and a clsDate object reference to represent the publication date. Upon completion of this deliverable the student will have developed a structure capable of handling information shared by most publication types, including authors, title, and publication date.

Deliverable 5a reinforces the concept of abstract classes and abstract interfaces. The reference class developed in the previous deliverable must be modified so that it can serve as an abstract class. The next step is to develop an abstract interface that will be implemented by the reference class as well as all classes derived from it. The interface will include method signatures for formatMLA, formatAPA, and formatChicago, the subset of reference style approaches selected for this project. The reference class should be modified to implement the new abstract interface.

Deliverable 5b focuses on the concept of inheritance. It requires the development of derived classes from the reference class. A book class and a journal class will be derived directly from the reference class, and a text chapter class will be derived from the book class. Additional classes, such as a proceedings class, can easily be added to extend the scope of the project. There is a single deliverable for Deliverables 5a and 5b.

The focus of Deliverable 6 is on polymorphism. As the user enters individual references, such as book, chapter, and journal details, they will be stored in an ArrayList of generic references. They will then be manipulated based on their actual type to demonstrate the concept of polymorphism, which requires both method overriding and dynamic binding. Students will see that while they

may not know the specific type of an individual item in an ArrayList, the appropriate methods each object will be invoked correctly. Type casting will also be demonstrated by this exercise. The program should also implement a preview feature using a control like a RichTextBox or JTextPane if one is available in their development environment.

Deliverable 7 requires that students learn how to create persistent objects using the binary formatter or Simple Object Access Protocol (SOAP) and data files. Students will write their ArrayList of references to a binary or SOAP file and later retrieve it into another ArrayList and manipulate it to demonstrate that no object properties were lost in the transition. The ArrayList of objects will then be read from the file and the program will loop through the ArrayList applying the selected format and write the formatted output to a Microsoft Word file. At the instructor's discretion the student can also be directed to implement a user-friendly interface with menus, etc., to improve usability of the system.

Detailed Requirements

This section describes the detailed specifications for each deliverable. Details are given using terminology specific to VB.Net, since that was the language used in the latest classroom trial of the project. It is also written as a set of instructions for the student so that it can be used by instructors with minimal modifications. Instructor resources like sample code and grading rubrics will be addressed in a later section.

Deliverable 1: Abstraction

- Objectives: Gain experience using abstraction to design classes by assessing possible attributes and behaviors and capturing those that are relevant to the project at hand.
- Description: This deliverable is intended to provide students with experience designing classes (abstraction).

Recall that abstraction requires the designer to ignore irrelevant features, properties, or functions and emphasize only those that are relevant to the given project.

There are several different ways to format references in a research paper. Probably the most common ones are the APA style, the MLA style, and the Chicago style. You can find examples of each format in the Reference Styles help sheet in Appendix A.

This deliverable is the first component that you will develop in the process of designing and implementing a reference formatting system. Notice that every reference, regardless of the style, includes the author's name (or authors' names) and a publication date. As an exercise in abstraction you will consider each of those independently.

Part I: clsName

Begin by considering a name and asking what possible attributes can be associated with a name. Most individuals come up attributes like first name, last name, and middle name or middle initial. However, some names like J.R.R.Tolkien have two middle initials. There are also two word last names (e.g., St. James) as well as hyphenated last names (e.g., Smith-Cross). There is often a suffix, such as Ph.D., M.D., Sr., Jr., III, etc. And many names have a prefix like Mr., Mrs., Ms., and Dr., although it could be argued that they are not really part of the name. More on suffixes can be found in Wikipedia ("Suffix," 2009).

Those are just the types of names that students in English-speaking countries are most familiar with. When designing software one must take into account international usage that can impact design. For example, the list of possible suffixes must be expanded to include suffixes like Don,

Doña, Sr., and Sra. Further, a website that discusses name variations points out that "the rest of the world" doesn't necessarily conform to the "first name, middle initial, and last name" model (Frank, n.d). The reason is that the major components of people's names include the given names, which are the names given to children by their parents, and family names, or surnames, which are the names passed down from one family generation to another in most countries.

Here are several cases that should be considered:

Case 1: Ann Elizabeth Brown has two given names and one family name. If she calls herself Ann, then she has a first name, can use a middle initial, and has no problem with the standard format.

Case 2: Supposing, however, that she has been called Beth since birth, and goes by A. Beth Brown. In that case her name doesn't conform to the "first name, middle initial, and last name" model. Neither will that of J. Edgar Hoover, a former FBI director, and many others.

Case 3: If Beth Brown marries someone named Tom Adams she might then call herself Beth Adams, or perhaps Beth Brown, or even Beth Brown Adams. Although Ann is still her first given name she seldom uses it. What is now her middle initial?

Case 4: Romelia María Hernández Flores is from Mexico. Her names (nombres) are Romelia María (and she always uses both these names), her primer apellido (father's family name) is Hernández and her segundo apellido (mother's family name) is Flores. You would find her in a Mexican phone book under "Hernández Flores, Romelia María." She calls herself Romelia María Hernández.

Case 5: Romelia María's boyfriend is Jorge Eduardo Romero de León. He has two given-names (and uses the second of these), and two family-names (Romero and de León). You find him in a Mexican telephone book under "Romero de León, Eduardo." He calls himself Eduardo Romero.

Case 6: If Romelia María marries Eduardo, Romelia María Hernández Flores becomes Romelia María Hernández de Romero.

Case 7: Li Xiao Ping is from China. In China, Japan, Vietnam, Hungary, and some other countries, the family-name (Li) comes first. The two components (Xiao Ping) of his given name are used together as one name such that they could almost be written Xiaoping. You find him in a Chinese phone book as Li Xiao Ping (written in 3 Chinese characters with no comma).

Designing a name class suddenly seems much more complex. However, when engaging in object-oriented design you begin by considering all possible attributes and then narrow those attributes down to those that are relevant to the problem at hand. That simplifies the situation in this case.

The problem at hand is a reference formatter. Refer to the Reference Styles help sheet in Appendix A to see what information is required to represent an author's name. If it does not contain enough examples to help you get a feel for what is required, perform an Internet search for the terms "reference styles," "MLA," "APA," or "Chicago."

Abstraction is not limited to attributes. Once you complete the abstraction process with regard to attributes, you next need to address functional abstraction, a process in which you determine which functionality is important in much the same way as you determine which data items are important.

Functional abstraction takes into account such things as accessor methods (those used to return the values of instance variables), mutator methods (those used to change the values of instance

variables), constructors (those methods used to initialize the instance variables), and any necessary data validation or data conversion methods.

You should consider the necessary constructors. There will always be a last name, but first name may be a name or an initial, or may not even exist. Further, you can't always find an author's first name on a publication if they only use an initial. J.R.R. Tolkien is again a good example. Did Socrates have a first name? And will there always be a middle name or initial?

Consider mutator methods. Are they needed for individual attributes or do you want to provide values for all attributes, or a subset of them, at once? There are instances in which you want to create "read-only" instance variables in which case no mutator methods are provided, only accessor methods.

Shifting our focus to accessor methods, the different formatting styles often use variations on how the author's name is displayed, and this affects the components that you must consider. Even if you limited your scope to just first name, middle initial, and last name you have several combinations. The fact that middle initial is either omitted or must follow the first name if it is included narrows the possibilities. The fact that a first name may be the complete first name or a first initial broadens the possibilities.

Permutations with middle initial:

- Anthony T. Jones
- Jones, Anthony T.
- A.T. Jones
- Jones, A.T.

Permutations without middle initial:

- Anthony Jones
- Jones, Anthony
- A. Jones
- Jones, A.

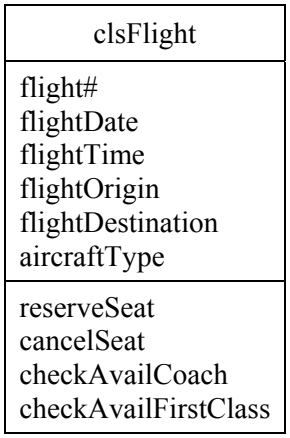
So there are eight combinations even if you ignore the suffix, two word last names, hyphenated last names, etc. Again, review the Reference Styles help sheet in Appendix A to see what formats are required to represent an author's name.

Now consider validation methods. How do you validate the data stored in a name? You can test the value character by character to be sure it is not blank, not numeric, or non alphabetic. Is that adequate? Sure, for most American names. However, apostrophes, hyphens, or spaces in names must be accounted for as well (Odriscoll, 2008). Therefore you should be sure that each character is uppercase (Unicode values 65-90), lowercase (97-122), or a space (32) for Dutch names like van Kemp, apostrophe (39) for Irish, French, Italian, and African names like O'Connor, period for names like St. James, or hyphen (45) for Arab names like Al-Hussein or hyphenated married names like Joyner-Kersee. Validating the middle initial is considerably easier, but still must be done. Note that only a single (valid) character or no middle initial at all is acceptable.

You should also provide utility methods like getInitial that will extract the first letter from a first or middle name.

A simple Unified Modeling Language (UML) class diagram depicts classes as boxes with three sections, the top one indicates the name of the class, the middle one lists the attributes of the

class, and the third one lists the methods. For example, a class diagram of clsFlight (from an airline reservation system) is shown below.



Part II: clsDate

Now consider how a reference formatting system stores and displays a date. Notice that the different formatting styles often use variations in how the publication or conference date is displayed. All references include at least a year of publication, while some, like proceedings, include a complete beginning date and a complete ending date, like October 31, 2008 - November 2, 2008 or November 7-10, 2008.

This deliverable will require you to create a programmer-defined clsDate class that will allow the storage and display of publication dates in different formats. Most OO programming languages provide a built-in date class. Can it be used to represent a publication date so that we don't have to develop our own? In most cases the answer is no. As noted above, some publications include only a year, some include only a month and year, and some include an entire date. Therefore, your class must allow the storage of a 0 value for day for instances in which only the month and year of publication are known, and should also allow the user to specify both a 0 or null day and a 0 or null month for instances in which only the year of publication is known. Most built-in date classes require non-zero values for month, day, and year so a user-defined date class is required.

Go through the same process that you followed for name to decide on the attributes and functionality needed for your date class. Discuss with your professor whether you want to store your date in month, day, and year format, with attributes for each, or in Julian date format with a single attribute for all. You can read more about Julian dates in Wikipedia (“Julian day,” 2009). As with names, you should consider international usage. There are many different formats that can be used to display dates (Frank, n.d.).

You should provide at least three constructors: (1) one that accepts actual parameters for month, day, and year, (2) one that accepts parameters for month and year, and (3) one that accepts an actual parameter for year.

Whether you use Julian dates or individual attributes for month, day, and year you are going to have to validate your month, day, and year. That will require validateMonth, validateDay, and validateYear methods. It will also require a function to determine if the year is a leap year, since that knowledge is required to validate the day. Recall that most years that are evenly divided by 4 are leap years. However, years that are evenly divided by 100 but not evenly divided by 400 are not leap years. Thus 1996 is a leap year, 1900 is not, and 2000 is.

As with clsName, design a UML class diagram to model your clsDate.

Keep in mind that although abstraction requires that you ignore irrelevant details, you must also take into account future expansion of the project. How might this affect your design decisions in this case?

Submissions for this deliverable

- UML class diagram for clsName
- UML class diagram for clsDate

Deliverable 2: Encapsulation

- Objectives: Gain experience with encapsulation and instantiation by setting up classes and instantiating objects.
- Description: This deliverable is intended to provide students with experience "bundling" data and the processes that operate on that data into a single package. It will also provide experience using overloaded constructors, accessor methods, mutator methods, and utility methods.

Recall that encapsulation is basically a design issue that deals with how functionality is compartmentalized within a system. In object-oriented programming, encapsulation is the inclusion within a program object of all the resources needed for the object to function, i.e., related data and the methods that manipulate those data.

This deliverable will require you to create a programmer-defined clsName class to allow the storage and display of author names in various formats. In addition, it will require you to create a programmer-defined clsDate class that will allow the storage and display of publication dates in different formats. These classes were designed in the previous deliverable, but not implemented.

Part 1: clsName

The clsName class should include instance variables for (at least) first name, middle initial, and last name. (If your Deliverable 1 revealed additional variables like suffix then be sure to include them.) Take care to not include extraneous instance variables.

You should provide the necessary constructors (there will always be last name, but will there always be a first name and/or an initial?)

You must provide mutator methods. At a minimum, you should include the following:

```
Public Sub setFirstName (ByVal first As String)
Public Sub setLastname (ByVal last As String)
Public Sub setMI (ByVal mi As String)
Public Sub setName (ByVal last As String, ByVal first As String, ByVal mi As String)
```

Recall that mutator methods can perform three tasks:

(1) provide values for instance variables

(2) perform data validation, such as range checking

(3) translate between the form of the data used in the interface and the form used in the implementation

In this particular class the third task is not necessary, but you must remember to validate the first name, middle initial, and last name. Remember that it is not uncommon to know only the author's first initial and possibly no middle initial.

You should also provide accessor methods to getFirstInitial and method(s) to format names like Jones, Anthony T. and Jones, A.T. You may want to write a single method that reads the value of a parameter that you pass to determine which format to use, but it is not required that it be a single method.

Here are some possible combinations that you should provide accessor methods for:

A. Jones
A.T. Jones
Anthony Jones
Anthony T. Jones
Jones, A.
Jones, A.T.
Jones, Anthony
Jones, Anthony T.

In other words, include

```
Public Function getFiLast ( ) As String
Public Function getFiMiLast ( ) As String
Public Function getFirstLast ( ) As String
Public Function getFirstMiLast ( ) As String
Public Function getLastFi ( ) As String
Public Function getLastFiMi ( ) As String
Public Function getLastFirst ( ) As String
Public Function getLastFirstMi ( ) As String
```

To make your class more generic you may opt to include accessor methods called getFirst, getLast, and getMI.

Recall that not all methods need to be declared as Public. Those methods that are provided simply to support the Public methods but that do not need to be part of the interface are declared Private and are referred to as utility methods. Include these utility methods:

```
Private Function getFirstInitial () As String
Private Function getMiddleInitial () As String
Private Function validName (ByVal name As String) As Boolean
Private Function validInitial (ByVal initial As String) As Boolean
```

Part 2: clsDate

The clsDate class should store dates using month, day, and year. A new date class is necessary because the built-in data types for date do not allow zero values for month and day, but often that information is not available for a publication; hence the need for a specialized class. You can use the clsDate class provided in Appendix B as a basis.

The clsDate class should include instance variables for month, day, and year. Take care to include no extraneous instance variables.

You should provide at least three constructors: (1) one that accepts actual parameters for month, day, and year, (2) one that accepts parameters for month and year, and (3) one that accepts an actual parameter for year. The most efficient approach is to call the setDate method from each con-

structor and allow it to handle the 0 parameters. Why should a constructor call a mutator method? Recall again the tasks normally performed by mutator methods.

You must provide mutator methods. At a minimum, you should include the following:

```
Public Sub setDay (ByVal dd As Integer)
Public Sub setMonth (ByVal mm As Integer)
Public Sub setYear (ByVal yyyy As Integer)
Public Sub setDate (ByVal yyyy As Integer)
Public Sub setDate (ByVal mm As Integer, ByVal yyyy As Integer)
Public Sub setDate (ByVal mm As Integer, ByVal dd As Integer, ByVal yyyy As Integer)
```

Both setMonth and setDay should check to see if the argument received is 0. If it is, then the appropriate instance variable should be set to 0. Otherwise the argument should be validated and if it is acceptable the instance variable is set. Do not forget to take leap years into account. The setYear method simply needs to validate the year. The setDate method with three arguments sets the date values by first checking for and allowing a 0 month, and if it is non-zero it validates the month. It then checks for and allows a 0 day, and if it is non-zero it validates the day. If no month is provided but a day is provided, that is an error and the day should be set to 0. The approach used by the remaining overloaded setDate methods can be inferred from the above description.

You should also provide a variety of accessor methods. One accessor method, referred to below as dateToString, should return a string consisting of year, or a string consisting of month and year, or a string consisting of month, day, and year depending on the value of a parameter that is passed in. Note that this is a single routine, not three different methods. That requirement is included to test your programming ingenuity. It must NOT ever display a 0 month or 0 day, even if its parameter specifies it. You may also want to include getDay, getMonth, and getYear for completeness.

```
Public Function getDay () As Integer
Public Function getMonth () As Integer
Public Function getYear () As Integer
Public Function dateToString (ByVal MDY As String) As String
```

Include these utility methods:

```
Private Function validateMonth (ByVal mm As Integer) As Integer
Private Function validateDay (ByVal mm As Integer, ByVal dd As Integer,
                              ByVal yyyy As Integer) As Integer
Private Function validateYear (ByVal yyyy As Integer) As Integer
Private Function isLeapYear (ByVal yyyy As Integer) As Boolean
```

Submissions for this deliverable

Provide a main (form) method that instantiates a clsName object and tests each of the possible features. Also instantiate a clsDate object and test each of the possible features. The date on the form should default to the current date. Your clsName and clsDate should be stored in individual files. See Appendix C Figure 1 for a sample screen shot of the interface.

Deliverable 3: ArrayLists of Objects

- Objectives: Gain experience modifying classes, setting up ArrayLists of objects, and sorting those ArrayLists.

- Description: This deliverable is intended to provide you with additional experience with classes and object instantiation, as well as creating ArrayLists of objects and manipulating them.

It is common for a publication to have multiple authors. In Deliverable 2 you created a clsName class to store a single name, but in order to enable the storage of multiple author names a collection of clsName objects is needed. In order to accomplish this you will set up an ArrayList of clsName objects.

Modify the clsName class so that it includes the Imports System.Collections statement and implements the IComparable interface. The IComparable interface provides an abstract method by which objects can be compared so that sorting is possible. If the clsName class implements IComparable it must also be modified to include a CompareTo method. Recall that the CompareTo method must define a rule for comparing an instance of the clsName class to other objects to specify how sorting should be performed. Your test program should instantiate several clsName objects, assign values to the first and last name of each, call the ArrayList.Sort routine to sort the ArrayList of objects in order by last name and first name, and then print the sorted results in a text box.

Submissions for this deliverable

Provide a main form that includes an "Add Name to Author List" button and a "Sort Names" button. There should also be a "Clear Fields" button that clears all input and output fields. See Appendix C Figure 2 for a sample screen shot of the interface.

Note

Your professor will indicate whether you should use an array or an ArrayList for this deliverable and throughout the rest of this project, depending on the concept that he or she wants to emphasize. These specifications will refer to the required structure as an ArrayList.

Deliverable 4: Aggregation

- Objectives: Gain experience designing classes using aggregation.
- Description: This deliverable is intended to provide you with additional experience with classes and object instantiation, as well as utilizing aggregation to take advantage of code reuse.

Recall that an aggregation among classes exists when a class contains references to other classes. The features that are common to all references in a reference list, regardless of type, are author name(s), title, and publication date. A new class called clsReference will be used to capture these features and will include an ArrayList of clsName objects to represent author names, a String to represent the title, and a clsDate object to store the publication date. The class clsReference is an aggregation because the data types of its instance variables include other classes, namely clsName and clsDate.

This deliverable requires that you develop a new class called clsReference that has the following instance variables:

```
Private author As New ArrayList  (of clsName objects)
Private title As String
Private pubDate As clsDate
```

Your clsReference should also have the following methods:

```
Public Sub New ( )
Public Sub New (ByVal last As String, ByVal first As String, ByVal mi As String, _
    ByVal newTitle As String, ByVal mMonth As Integer, ByVal mDay As Integer, _
    ByVal mYear As Integer)
Public Sub setAuthor (ByVal last As String, ByVal first As String, ByVal mi As String )
Public Sub setTitle (ByVal newTitle As String)
Public Sub setPubDate (ByVal year As Integer)
Public Sub setPubDate (ByVal month As Integer, ByVal year As Integer)
Public Sub setPubDate (ByVal month As Integer, ByVal day As Integer,
                       ByVal year As Integer)
Public Function getAuthorLastFirstMi ( ) As String
Public Function getAuthorLastFiMi ( ) As String
Public Function getAuthorLastFiMi_FirstMiLast ( ) As String
Public Function getAuthorLastFirstMi_FirstMiLast ( ) As String
Public Function getTitle ( ) As String
Public Function getTitleLowercase ( ) As String
Public Function getPubDate (ByVal dateFormat As String) As String
Protected Function convertToLowerCase (ByVal changeText As String) As String
Protected Function convertToTitleCase  (ByVal changeText As String) As String
```

The accessor and mutator methods for author must be capable of handling multiple names, i.e., an ArrayList of names. Refer back to the Reference Styles help sheet in Appendix A for examples. Make sure you define the methods in the order listed above. See Appendix D for detailed descriptions of each method listed above.

If you have not yet learned about the Protected access modifier associated above with convertToLowerCase and convertToTitleCase, simply use it and wait for your instructor to discuss it when the topic of inheritance is covered.

Submissions for this deliverable

Appendix C Figures 3 and 4 show sample screen shots of the interface.

The Add Authors button should call the setAuthor method of the clsReference object and clear the textboxes. Display a message box or status line confirming what you have done.

The Save All Authors button should place the focus in the title textbox. Display a message box or status line confirming what you have done.

The Save Reference button should call the setTitle method of the clsReference object, the setPubDate method of the clsReference object, and then build the output string for the preview output box. Display a message box or status line confirming what you have done.

Deliverable 5: Abstract Classes, Abstract Interfaces, and Inheritance

- Objectives: Gain experience using abstract classes, abstract interfaces, and inheritance.
- Description: This deliverable is intended to provide you with experience designing abstract classes and interfaces, as well as inheritance.

Part I: Abstract classes

Previous deliverables have provided a beginning for the reference formatter system, but there is much work to be done. You recently created a clsReference class to serve as a base class for more specific derived classes. As it stands, clsReference is too generic to define real objects; we need to be more specific before we can think of instantiating actual references like books and journals. The first part of this deliverable requires you modify clsReference so that it can serve as an abstract class.

Recall that abstract classes are classes that are defined, but for which the programmer never intends to instantiate any objects. These normally serve as superclasses in inheritance situations, and are referred to as abstract superclasses. The sole purpose of an abstract class is to provide an appropriate template superclass from which other classes may inherit common interface and/or implementation details. Abstract classes facilitate reuse because they specify code common to all their derived classes.

Modify clsReference so that it can serve as an abstract class.

- Insert the keyword MustInherit.
- Include the keywords Protected and MustOverride with the methods if they are necessary.

Part II: Abstract interfaces

An abstract interface is a specification of a set of methods that are to be implemented in the class that inherits from it. An abstract interface provides common functionality across the classes that implement it. Abstract interfaces resemble abstract classes, but contain only abstract methods. An abstract interface contains only method signatures, while the abstract class can contain abstract methods as well as constants, variables, and concrete methods. If a class claims to implement an interface, all methods defined by that interface must appear in its source code before the class will successfully compile. Remember that the purpose of an abstract interface is to provide a common set of methods, or common interface, to all classes that implement it. In the context of a reference formatting system you should be able to format any reference, regardless of type, in the MLA, APA, or Chicago style.

Provide an abstract interface called IFormattableReference that includes the following method signatures:

- formatMLA
- formatAPA
- formatChicago

Modify clsReference to implement IFormattableReference. The classes that will be derived from clsReference in the next part of this deliverable will implement the interface.

Part III: Inheritance

Part III of this deliverable requires you to practice using inheritance by deriving classes from the abstract class clsReference and implementing the IFormattableReference interface. Recall that inheritance is the process of creating new classes, called derived classes, from existing or base classes by absorbing their attributes and behaviors and embellishing these with capabilities the new classes require. Therefore your new derived classes will inherit the instance variables and methods from clsReference, will implement every method specified in the IFormattableReference

interface, and will add instance variables and methods specific to whatever type of reference is being modeled by the derived class.

You will be implementing the inheritance structure shown in the UML class diagram in Figure 1. The class diagram may not include all instance variables or methods required.

Part III, Step 1: Create a new class called clsBook that is derived from clsReference. It should add new class members, including the following instance variables:

- publisher
- city
- state
- country

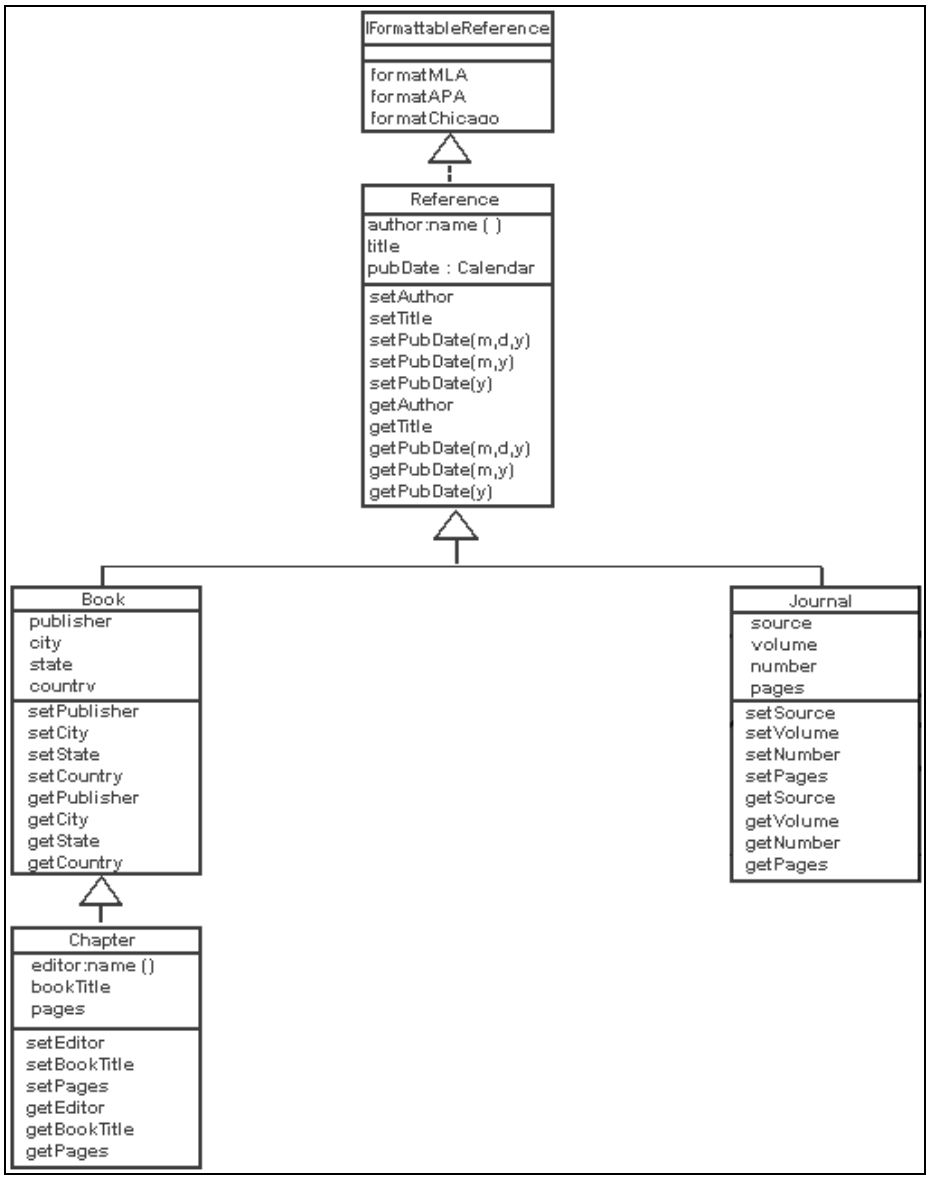


Figure 1: Class Diagram for the Reference List Formatter.

It should also include the following methods:

- any necessary constructors
- setPublisher
- setCity
- setState
- setCountry
- setBook (to set all clsBook instance variables. Provide three versions to handle all variations of dates or use optional parameters.)
- getPublisher
- getCity
- getState
- getCountry
- formatMLA (Overrides)
- formatAPA (Overrides)
- formatChicago (Overrides)

Part III, Step 2: Create a new class called clsChapter that inherits from clsBook. It should also add class members indicated in Figure 1, including the following instance variables:

- editor (ArrayList of names)
- bookTitle
- beginningPage
- endingPage

It should also add the following methods:

- setEditor
- setBookTitle
- setBeginningPage
- setEndingPage
- setChapterInfo (to set all clsChapter instance variables. Three versions to handle all variations of dates or use optional parameters.)
- getEditorLastFirstMi
- getEditorLastFiMi
- getEditorLastFiMi_FirstMiLast
- getEditorFiMiLast
- getEditorLastFirstMi_FirstMiLast
- getEditorFirstMiLast
- getBookTitle
- getBeginningPage
- getEndingPage
- resetNumEditors
- formatMLA (Overrides)
- formatAPA (Overrides)
- formatChicago (Overrides)

Part III, Step 3: Create a new class called clsJournal that inherits from clsReference. It should also add class members indicated in Figure 1, including the following instance variables:

- source
- volume

- number
- beginningPage
- endingPage

It should also add the following methods:

- setSource
- setVolume
- setNumber
- setBeginningPage
- setEndingPage
- setJournalInfo (to set all clsJournal instance variables. Three versions to handle all variations of dates or use optional parameters.)
- getSource
- getVolume
- getNumber
- getBeginningPage
- getEndingPage
- formatMLA (Overrides)
- formatAPA (Overrides)
- formatChicago (Overrides)

See Appendix E for detailed descriptions of each method listed above.

Submissions for this deliverable

Table 2 provides a description of the buttons shown in Figures 5-8 in Appendix C. Note: In this deliverable you are not required to add each reference entry to an array or ArrayList of references. That will be addressed in the next deliverable. For now you are still working with individual references.

Deliverable 6: Polymorphism

- Objectives: Use ArrayLists to demonstrate the polymorphic properties of objects.
- Description: This deliverable is intended to provide you with experience using polymorphism, ArrayLists, and simple file I/O.

Recall that polymorphism refers to the ability of a language to have duplicate method names in an inheritance hierarchy and to decide which method is appropriate to call depending on the class of object to which the method is applied. Polymorphism allows a number of different classes of objects to respond to the same request by providing subclasses with methods with the same name and signature as a method in the superclass.

Table 2: Button functions for Deliverable 5.

Button	Purpose
Add Author	Adds a single author to the ArrayList of authors instance variable in the book, chapter, or journal object, increments the author counter, updates the author counter label, and clears the author fields.
Save Authors	Depending on which radio button is clicked, moves to the correct input field.
Add Editor	Adds a single editor to the ArrayList of editors instance variable, increments the editor counter, updates the editor counter label, and clears the editor fields.
Save Editors	Depending on which radio button is clicked, move to the correct input field.
Submit Book	Calls the routines to set the book title, publication date, publisher, city, state, and country. It then calls the routine to save the updated object.
Submit Chapter	Calls the routines to set the chapter title, publication month, day, and year, publisher, city, state, and country where the publisher is based, book title, and beginning page and ending page. It then calls the routine to save the updated object.
Submit Journal	Calls the routines to set the paper title, the journal month, day, and year of publication, journal title, the beginning page and ending page, the journal volume and number. It then calls the routine to save the updated object.

In the previous deliverable you created the clsBook, clsJournal, and clsReference classes, each with its own specific versions of the formatMLA, formatAPA, and formatChicago methods. As the user enters individual references they will be stored in an ArrayList of generic references. When we later decide to format the reference list according to the MLA, APA, or Chicago style, polymorphism allow us to cycle through the ArrayList and call the correct formatMLA, formatAPA, or formatChicago method based on the actual type of each individual element in the ArrayList. In other words, if the user opted for APA style and the first element in the ArrayList is a book, the formatAPA method from clsBook is called. If the second element is a journal then the formatAPA method from clsJournal is called, etc.

You will demonstrate this by allowing the user to enter a series of references. As each clsBook, clsJournal, or clsReference is entered, it should be added to an ArrayList of clsReferences.

Set up a form (see Figures 9-12 of Appendix C) that includes radio buttons to allow the user to select the desired format (MLA, APA, Chicago). The program should also require the user to indicate where to save the formatted output using a file dialog. Be sure the new file has a .doc or .docx extension.

After the format is chosen and the file opened, the ArrayList should be sorted by first author's last name, first name, and middle initial, each element in the sorted ArrayList should have the requested formatting method (formatMLA, formatAPA, formatChicago) called, and the resulting string should be written to a new Word file. The output should be available for preview in a RichTextBox control, since a RichTextBox allows the viewing of a Word file with formatting preserved.

Sorting an array or ArrayList of clsReference objects will require that you modify clsReference to implement IComparable and to include the CompareTo method, much as you did with the clsName class in Deliverable 3.

Summary

1. Data entry
   a. enter reference details <from previous deliverable>
   b. store references in an ArrayList
2. Format references
   a. Specify destination file (Word file)
   b. Select format
      i. loop through ArrayList, applying the selected format, and writing to Word file.
      ii. close the Word file
3. View the doc file in a RichTextBox control

Submissions for this deliverable

Table 3 provides a description of the buttons shown in Figures 9-12 of Appendix C.

Table 3: Button functions for Deliverable 6.

Button	Purpose
Review Output	Displays the reference ArrayList in formatted form in a RichTextBox control.
Submit Book	Calls the routines to set the book title, publication date, publisher, city, state, and country. It then calls the routine to add the updated object to the reference ArrayList.
Submit Chapter	Calls the routines to set the chapter title, publication month, day, and year, publisher, city, state, and country where the publisher is based, book title, and beginning page and ending page. It then calls the routine to add the updated object to the reference ArrayList.
Submit Journal	Calls the routines to set the paper title, the journal month, day, and year of publication, journal title, the beginning page and ending page, the journal volume and number. It then calls the routine to add the updated object to the reference ArrayList.

Deliverable 7: Object Persistence

- Objectives: Complete the reference formatter system.
- Description: Finalize the reference formatter system by saving the references to a binary or Simple Object Access Protocol (SOAP) file, and then open the file, format the references, and write them to a Word file for future use.

Object persistence refers to objects that outlive the execution of the program in which they were created. Without this capability, the objects and their data would only exist in memory as long as the program in which they were created continues running and would be lost when the program ends. Object persistence allows an object's state to be stored in long-term storage. In future sessions, the persisted objects can be restored to their previous state.

In the last deliverable you saved your references to an ArrayList as each one was entered. However, when you closed your program you lost the reference objects so you can never choose an alternate format in which to display them. Further, if you want to use that reference list again you will have to re-enter all of the data. This deliverable uses object persistence to deal with this problem.

When all references have been entered, write the entire ArrayList to a binary or SOAP file (your choice) with a .dat extension. Note that you are writing the unformatted data to a file, not the formatted output. Keep in mind that the user may enter his references one day, but not use them again for several days or weeks.

Therefore, the form should include a control that will allow the user to open the dat file of references that you created and saved in the previous step. The user will also want to specify the reference format that is desired. Thus, the form should also have radio buttons to allow the user to select the new format desired (MLA, APA, Chicago). The user will also need to specify the Word file in which to save the formatted output using a SaveFileDialog, as in the previous deliverable. Be sure the new file has a .doc or .docx extension.

After the format is chosen the dat file of references should be opened and the references should be read into an ArrayList of objects (or some type of "widened" reference). Then the array should be sorted by first author's last name, each element in the sorted array should have the proper formatting method (formatMLA, formatAPA, formatChicago) applied, and the resulting string should be written to a new Word file. You should provide a preview of the formatted output in a RichTextBox control.

Summary

1. Create source file
    a. enter reference details <from previous deliverable>
    b. store references in an ArrayList <from previous deliverable>
    c. save UNFORMATTED references to a binary or SOAP file (.dat file)
2. Format references
    a. Specify source file
        i. open file (.dat file)
        ii. read file into array
        iii. close file (.dat file)
        iv. sort ArrayList <from previous deliverable>
    b. Specify destination file (Word file) <from previous deliverable>
    c. Select format
        i. loop through ArrayList, applying the selected format, and writing to Word file. <from previous deliverable>
        ii. close the Word file <from previous deliverable>

Submissions for this deliverable

You will need to provide a user-friendly interface for this final deliverable similar to that shown in Figures 13-21 of Appendix C. Be sure to use menus and tab controls. Table 4 provides a description of the menu selections shown in Figures 13-21.

Table 4: Menu functions for Deliverable 7.

Menu Item	Purpose
File	**Opens the File submenu.**
Open Existing Source File	Open existing file of previously entered references, i.e., the binary or SOAP file that contains the unformatted references. It then reads the file contents into an ArrayList and then sorts the ArrayList. Additional references can be added by the Edit - Add References item.
Print Preview	View your reference list in various reference styles by selecting MLA, APA, or Chicago.
MLA	Selects MLA reference style for the preview.
APA	Selects APA reference style for the preview.
Chicago	Selects Chicago reference style for the preview.
Save Reference File	Select MLA, APA, or Chicago reference style and the application allows you to specify the destination doc file in which you want the sorted reference list to be stored. It then opens the file and returns the filename through a form-level variable.
MLA	Selects MLA reference style for the saved reference list. Loops through the ArrayList and applies the MLA format to every reference and writes the result to the destination file.
APA	Selects APA reference style for the saved reference list. Loops through the ArrayList and applies the MLA format to every reference and writes the result to the destination file.
Chicago	Selects Chicago reference style for the saved reference list. Loops through the ArrayList and applies the MLA format to every reference and writes the result to the destination file.
Exit	Exits the application.
Edit	**Opens the Edit submenu.**
Add References	Opens the reference entry form so new references can be added.
Help	**Opens a Help screen.**

Instructors' Resources

Instructor resources will be made available to those who adopt this project. Those who can provide proof of appointment as an educator can contact the author for access to resources such as downloadable demos for each deliverable, sample solutions (written in Visual Basic .NET) for each deliverable, sample code for each deliverable, and grading rubrics for each deliverable.

Conclusion

This project serves to integrate all of the critical concepts in object-oriented development – abstraction, encapsulation, inheritance, and polymorphism – as well other important concepts like aggregation, abstract interfaces, method overriding, dynamic binding, data structures of objects, and object persistence. Unlike most toy problems this project takes an imposing system to completion. The students start with one deliverable and continue to build upon and enhance that deliverable through each stage of the requirements. In this way students gain experience developing a larger system than most toy projects allow. The project is large enough that programming teams can be used if desired, as can agile development techniques. Further, the project is designed to ensure extensibility in that additional reference types as well as additional formatting styles can be easily incorporated. Unlike a "live" or real world problem, the complexity can be controlled by the professor and the project is able to involve all course concepts that the professor wants to reinforce.

The project is not without its shortcomings. As noted, if students are unable to complete a deliverable they can be provided a sample solution. However, this requires strict enforcement of deadlines because a sample solution cannot be handed out until all submissions are in. During testing it was found that allowing a cushion as little as a week leads to a detrimental delay before a sample solution is provided, because deliverable due dates were separated by at most two weeks, and sometimes only one week. The sample solution was sometimes not available until it was too late to be of use.

Further, students who failed to complete a deliverable often lacked adequate understanding to complete subsequent deliverables. For example, if a student is unable to successfully design a class, they most certainly will have difficulty with aggregation, which involves creating a class that includes instance variables that can themselves be classes. In another case, if a student has trouble creating an ArrayList or array of objects, then using those data structures later to demonstrate polymorphism is likely to be problematic. However, this problem can manifest itself in traditional programming assignments as well.

The user interface can also present students with problems. Past students were often unable to determine the purpose of some of the required buttons, such as the Add Authors button or Save Authors button in Deliverable 5. Detailed explanations such as that included in Table 2 can help to alleviate student confusion. Other user interface difficulties stem from student inexperience with lesser-used graphical user interface controls. Although controls like file dialogs, tab controls, and RichTextBoxes are available in many programming languages, they may have different implementations (e.g., RichTextBox versus JEditorPane). Further, students may not have been exposed to them in previous courses and may be intimidated when required to implement them as part of a deliverable interface. Again, this problem can be encountered in traditional programming assignments as well.

Finally, although it seems evident that a project that covers all core concepts of a course has intrinsic value, project effectiveness has not been evaluated. While it has been used in multiple classes, it has undergone refinement each time. During the semester in which it finally reached its finished state, the curriculum was revised and the course became an elective. Since then, there has not been adequate demand to teach the course again and therefore there has been no opportunity

to gather empirical evidence that the project enhances the learning experience of students. Future research will involve evaluating project effectiveness, but it will most likely require collaborating with colleagues at other universities who are willing to evaluate the project in their classes.

In spite of these limitations the project provides students with hands-on experience with not only the four principle concepts upon which object-oriented design and programming rest, but other key concepts as well. Student evaluations have noted that the project "matches the concepts being taught well." It also enables students to glimpse the complexity that they will encounter upon entering the work force. Students realize this, with one pointing out that the project deliverables "were practical and had real-world application." Another student commented, "I liked working on one major project throughout the semester. It allowed us to experience working on a large project." So while the project has not been empirically evaluated, it serves the purpose for which it was developed: to reinforce critical object-oriented concepts and to give students experience working through a realistic project modeled on a scenario that they might encounter in their future work environment.

References

Association of Project Management. (2000). *Syllabus for the APMP examination* (2nd Ed). High Wycombe, UK: Association of Project Management. Retrieved November 23, 2009 from http://www.pmir.com/html/pmdatabase/file/Ebook/APMP.pdf

Bednar, A., Cunningham, D., Duffy, T., & Perry, J. (1992). Theory into practice: How do we link? In T. Duffy & D. Jonassen (Eds.), *Constructivism and the technology of instruction* (pp. 17-34). Hillsdale, NJ: Lawrence Erlbaum Associates. Retrieved November 24, 2009 from http://books.google.com/books?id=7Uv8NHvKK44C&lpg=PA17&ots=XNfIXxU9rz&dq=%22Theory%20into%20Practice%3A%20how%20do%20we%20link%22%20Bednar%201992&pg=PA17#v=onepage&q=&f=false

Brown, A. L. (1992). Design experiments: Theoretical and methodological challenges in creating complex interventions in classroom settings. *Journal of the Learning Sciences*, *2*(2), 141-178. Retrieved November 24, 2009 from http://inkido.indiana.edu/syllabi/p500/brown1992.pdf

Cavanaugh, C. (2004). Project-based learning in undergraduate educational technology. In R. Ferdig et al. (Eds.), *Proceedings of Society for Information Technology & Teacher Education International Conference 2004* (pp. 2010-2016). Chesapeake, VA: Association for the Advancement of Computing in Education. Retrieved from http://www.coe.ufl.edu/faculty/cathycavanaugh/docs/PBL2040.pdf

Davis, E., & Morgenstern, L. (2004). Progress in formal commonsense reasoning. *Artificial Intelligence*, *153*(1-2), 1-12. Retrieved November 23, 2009 from http://www.ccs.neu.edu/home/futrelle/aiccis/seminar/papers/davis-morgenstern-common_sense.pdf

Delaney, R. (2009). *Citation style for research papers*. B. Davis Schwartz Memorial Library, C.W. Post Campus, Long Island University. Retrieved November 12, 2009 from http://www.liunet.edu/cwis/cwp/library/workshop/citation.htm

Ensmenger, N. L. (2004). Computing for the humanities and social sciences. In W. Aspray & A. Akera (Eds.), *Using history to teach computer science and related disciplines* (pp. 89-94). Washington, D.C.: Computing Research Association. Retrieved November 12, 2009 from http://www.cra.org/reports/using.history.pdf

Fisher, D., & Frey, N. (2007). Using projects and performances to check for understanding. In *Checking for understanding: Formative assessment techniques for your classroom* (pp. 79-97). Alexandria, VA: Association for Supervision and Curriculum Development. Retrieved November 23, 2009 from http://www.ascd.org/publications/books/107023/chapters/Using_Projects_and_Performances_to_Check_for_Understanding.aspx

Frank, J. H. (n.d.). *The K.I.S.S. Pages*. Gainesville, FL: University of Florida, Entomology and Nematology Department. Retrieved November 23, 2009 from http://entomology.ifas.ufl.edu/frank/KISS/

Gallagher, S. A., Stepien, W. J., & Rosenthal, H. (1992). The effects of problem-based learning on problem solving. *Gifted Child Quarterly, 36*(4), 195-200.

Ghezzi, C., & Mandrioli, D. (2005). The challenges of software engineering education. *Proceedings of the 27th International Conference on Software Engineering* (pp. 637-638). Retrieved November 12, 2009 from http://home.dei.polimi.it/mandriol/SitoItaliano/ICSE05-EDU-Paper.pdf

Henze, N., & Nejdl, W. (1997). A web-based learning environment: Applying constructivist teaching concepts in virtual learning environments. In F. Verdejo & G. Davies (Eds.), *The virtual campus: Trends for higher education and training* (pp. 63-77). New York, NY: Chapman & Hall. Retrieved November 24, 2009 from http://www.kbs.uni-hannover.de/paper/97/ifip97/paper15.ps

Jazayeri, M. (2004). The education of a software engineer. *Proceedings of the 19th IEEE International Conference on Automated Software Engineering* (pp. 18-27). Retrieved November 12, 2009 from http://www.inf.usi.ch/faculty/jazayeri/docs/papers/educationofSEASE.pdf

Jones, B. F., Rasmussen, C. M., & Moffitt, M. C. (1997). *Real-life problem solving: A collaborative approach to interdisciplinary learning*. Washington, DC: American Psychological Association.

Julian day. (2009, November 19). In *Wikipedia, the free encyclopedia*. Retrieved November 24, 2009, from http://en.wikipedia.org/wiki/Julian_day

Oberon Development. (2006). Bibliographies and reference lists. In *Citation Help*. Retrieved November 12, 2009 from http://www.citationonline.net/CitationHelp/c8i11diffstyles.htm

Odriscoll, S. (2008, February 21). Apostrophe in your name can cause a world O'trouble. *Denver Post*. Retrieved November 24, 2009 from http://www.denverpost.com/ci_8329730

Piccinini, N., & Scollo, G. (2006). Cooperative project-based learning in a web-based software engineering course. *Educational Technology &* Society*, 9*(4), 54-62. Retrieved November 23, 2009 from http://www.ifets.info/journals/9_4/6.pdf

Scardamalia, M., & Bereiter, C. (1991). Higher levels of agency for children in knowledge building: A challenge for the design of new knowledge media. *Journal of the Learning Sciences, 1*(1), 37-68.

Suffix. (2009, October 6). In *Wikipedia, the free encyclopedia*. Retrieved November 23, 2009 from http://en.wikipedia.org/wiki/Suffix_(name)

Thomas, J. W. (2000). *A review of research on project-based learning*. San Rafael, CA: The Autodesk Foundation. Retrieved November 23, 2009 from http://www.bie.org/files/researchreviewPBL_1.pdf

Thomas, J. W., Mergendoller, J. R., & Michaelson, A. (1999). *Project-based learning: A handbook for middle and high school teachers*. Novato, CA: The Buck Institute for Education.

Turk, D. E., & Vaishnavi, V. K. (2000). Software process models are software too: A domain class model for process models. *Proceedings of IRMA 2000: 11th International Conference of the Information Resources Management Association* (pp. 548-550). Retrieved November 24, 2009 from http://books.google.com/books?id=8DVcV9Smk3oC&lpg=PA548&dq=Process%20Models%20are%20Software%20Too&pg=PA548#v=onepage&q=Process%20Models%20are%20Software%20Too

Von Kotze, A., & Cooper, L. (2000). Exploring the transformative potential of project-based learning in university adult education. *Studies in the Education of Adults, 32* (2), 212-228.

Warlick, D. (2002). Raw materials for the mind. Raleigh, NC: The Landmark Project.

Appendixes

In order to conserve space appendixes are located online using the following link:
http://cobhomepages.cob.isu.edu/parkerkr/InSite2010/Appendixes.docx

Appendix A: Reference Styles Help Sheet
Appendix B: clsDate Using Generic Approach for Deliverable 2
Appendix C: Sample Screen Shots
Appendix D: Method Details for Deliverable 4
Appendix E: Method Details for Deliverable 5
Appendix F: VB Tips

Biography

Dr. Kevin R. Parker is a Professor of Computer Information Systems at Idaho State University. He has taught both computer science and information systems courses over the course of his nineteen years in academia. Dr. Parker's research interests include e-commerce marketing, competitive intelligence, knowledge management, the Semantic Web, and information assurance. He has published in such journals as *Informing Science: the International Journal of an Emerging Transdiscipline*, *Journal of Information Technology Education*, *Journal of Information Systems Education*, and *Communications of the AIS*. Dr. Parker's teaching interests include web development technologies, programming languages, data structures, and database management systems. Dr. Parker holds a B.A. in Computer Science from the University of Texas at Austin (1982), an M.S. in Computer Science from Texas Tech University (1991), and a Ph.D. in Management Information Systems from Texas Tech University (1995). Before entering academia Dr. Parker was a programmer/analyst with Conoco, Inc.