Rapid Application Development of a Decision Support System using Object Oriented Programming

Amy Roehrig, Trilogy Consulting Corporation, Pleasant Grove, UT

Abstract

Constantly changing business needs require flexible, easily modifiable Decision Support Systems (DSS). An added level of complexity is introduced by changes to the underlying data location, structure, meaning, or scale. Object Oriented design techniques provide us with a mechanism for crafting DSS which are scalable as well as easily modified. Utilizing specific design patterns, we can provide a system that will work with changing data structures and meanings, and the growth of business logic.

This paper presents an Object Oriented design approach to the development of a Decision Support System. It presents solutions to the problems of providing flexibility, usability, and maintainability in one package. The techniques explored are of general interest to anyone involved in application development. Extensive knowledge of SAS/AF, other SAS products involved, or Object Oriented design is not required to derive benefit from the theories presented here.

Introduction

The task: design a decision support system (DSS) for a company which had previously operated on human hunch, and do it quickly. The complicating factors: the data, the business logic, and the users. The data supporting the new system was in a state of flux, with much of its development delayed beyond the need for decision support. Thus, the system needed to be flexible in its handling of the data: in structure, scale, and location. Additionally, the business logic was embryonic, and the DSS needed to be able to grow with the sophistication of its users. Supporting models of varying complexity needed to be incorporated into the system without requiring programmer action. Lastly, the user interface needed the ability to be changed and customized to the needs of the users.

The specifications for the task began to describe a system which could have any piece changed at will without adverse affects on the rest of the system. The solution: an object oriented design. By using objects with standardized interfaces (Norton, 1997), we can design a system that is flexible in both form and function. This type of approach provides us with the ability to plug in objects (or groups of objects) to meet new requirements while minimizing ripple effects. Thus, changes to one part of the the system will have little or no effect on other parts. Additionally, Object Oriented Programming (OOP) provides us with the ultimate in scalability. We can prove the decision support concept with a small amount of data in a prototype, and then scale it up to work with larger amounts of data for larger decisions. As the practice of DSS gains wider acceptance with the company, we can port it to other business units that have become envious of our technology. OK, wishful thinking, but with separation of business logic, supporting data, and the interface, we could!

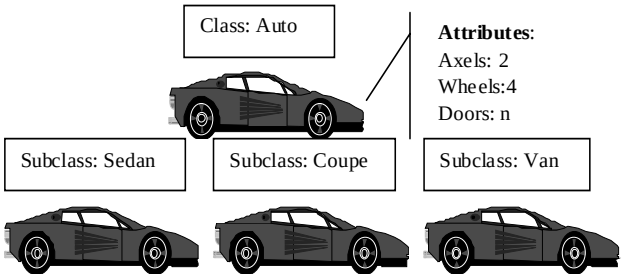
SAS and OOP

While the design screamed for an object oriented approach, the decision to use OOP was not a simple one. There is a certain technical difficulty in leaving behind traditional modular design. Encapsulating functions and data together? What could the world be coming to? Pretty soon we'll have dogs and cats living together, then mass hysteria! (Bill Murray, Ghostbusters). Be that as it may, the need was there and all that remained was to create a functional design and implement it. Simple, right?

Well, sort of. SAS/AF provides us with extensive tools for object oriented programming. Some of the OOP buzzwords that SAS supports are *inheritance*, *delegation* and *event driven action.* These concepts are basic to the design, so I will briefly discuss them here.

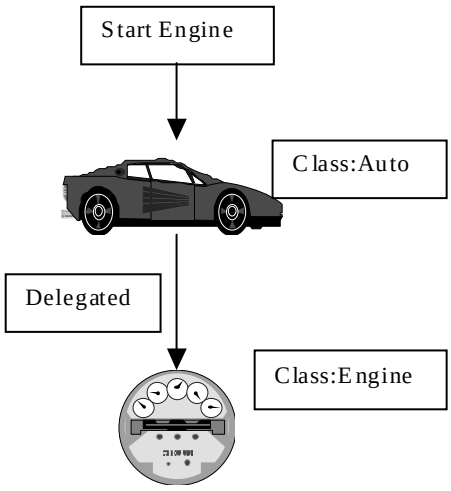
Inheritance provides us with the ability to subclass a particular object and have it inherit characteristics of the parent. Thus, if I create an object called an automobile, I can subclass it into sedans, coupes, minivans, and dune buggies. All of my subclasses will inherit the traits of 2 axles, 4 wheels, a steering wheel, an internal combustion engine, and a body with *n* doors. They will also inherit the ability to start the engine, stop the engine, go forward, go backward, turn left, turn right, enter passengers and exit passengers. For my electric car, I can override the internal combustion engine to give it an electric motor and batteries. Also, I can override the method I use for start and stop of the electric motor as opposed to the parent class, which used an internal combustion engine. However, my other characteristics and methods apply to all of my automobiles, so these will be inherited. These classes and subclasses are then used to create objects, such as my car, your car, and my brother's dune buggy.

Inheritance



Another important concept is delegation. Suppose I create a class called motor and have subclasses which are the internal combustion engine and the electric motor. My automobile could actually be a composite made up of many parts (or objects) of which the motor is one. Thus, my automobile object will delegate the start and stop engine actions to its motor object, providing me with the flexibility to add a solar power subclass in a future model. The constant across all of the subclasses of motor is the interface to the object, which I've called start engine and stop engine. Once again, the classes cue the automobile and the motor. These are used to create the actual objects. My car is an object of the automobile class/sedan subclass. It delegates certain functions to its engine, which is of the internal combustion class.

Delegation



A last OOP concept which SAS supports in part is event-driven action. Thus, when a particular object changes its state, it can send an event declaring that change to another object(s). It is up to the second object to determine what response, if any, is appropriate for that change. How is this different than a traditional modular programming procedural call? For one thing, the first object doesn't have to wait around for the second object to do its business. For another, the changing state of the second object will dictate the responding action. The first object has no knowledge of the state of the second object at the time of the event because they are encapsulated. Rather than passing parameters to tell the second object what to do, only an event and descriptive information about the event is sent. I said that SAS supports this concept only in part because certain events user initiated events (such as a mouse button being clicked and held) are not directly programmable in SAS, v6.12.

Design Patterns

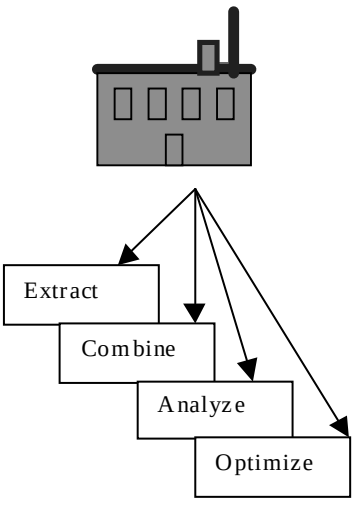
Since the tools are available, there are no excuses for avoiding an OOP design, except for a lack of knowledge and experience. This is where the book Design Patterns (Gamma, et. al., 1995) becomes invaluable. For a reasonable price, one can buy the knowledge and experience of experts. The patterns provided are a collection of useful deigns to solve many common programming dilemmas. Two patterns which were of immediate use are the Abstract Factory and the Template Method. The Abstract Factory is class of objects whose sole purpose is to create other objects. Each object that it creates in this design is a modified Template Method object to execute a particular part of the decision support algorithm. The main thrust of a Template Method is that it allows the designer to vary parts of a large or complex algorithm without changing the entire thing each time. This allows for maintenance programming to be broken down into more manageable pieces and minimize adverse ripple effects. Thus, our system design allows for the Abstract Factory to examine the current state (location, scale, etc.) of the data or business logic and to create the appropriate objects to process that data. The usefulness of this arrangement in a changing system should be apparent. As the data or business logic changes, the abstract factory can create different objects to manage the changing conditions.

Pattern Specifics-Abstract Factory

In the case of our decision support system, the supporting data could be in many locations, individual or summary in structure. Our abstract factory, EST_MGR, or estimate manager, creates engines depending on these conditions and then delegates responsibilities to these new objects. The four engines are:

Extract, which obtains the external data;
Combine, which combines external data with internally stored variables;
Analyze, which calculates performance measures;
Optimize, which actually performs the decision support.

All three objects are affected by the structure (individual vs. aggregate) of the data, which is what they have in common with each other. However, only Extract needs to determine the location of the data, only Combine needs knowledge of internally stored data, and only Analyze needs to interact with the business logic. The engines, once created, are ready to perform specific tasks depending on the state of the data. This state was provided by the abstract factory during creation of the engines (instantiation). After the factory has created these objects, all further commands to the factory are *delegated* to the appropriate engine. The events and other objects which drive these commands have no need to keep track of the engines, the factory which created them will do that.



Pattern Specifics-Template Methods

Furthermore, the factory can choose from a menu of different combinations of extract, combine, and analyze, because these three templates can have an unlimited number of different incarnations. Each version of Extract will support the same interface with the decision support system and respond to the command to obtain the data. However, the location and structure of the data will determine which subclass will be substituted in by the factory during engine creation. Likewise, as our business logic matures, different Analyze engines can be developed without affecting performance of Extract and Combine. There are bonus advantages to this design which we have yet to realize. Because we can have an infinite number of engines for business logic, we can maintain backward compatibility in future versions. This means that 3 years after a marketing campaign was run we will be able to re-evaluate it using the same logic we used 3 years earlier without having to search up old software. Version Control! How about that?

Structural Data Objects

We now have a flexible design for the overall decision support, but what about some of the nitty gritty details? OOP helps us out there as well. For example, this DSS needs to allow user input for different scenario analyses, as well as certain variables used in the business model. The user population should be able to segment the data flexibly and provide analysis results at varying levels of summarization. Additionally, there were some variables which were absolutely required to be associated with certain data.

The result was a structural data object which was subclassed over half a dozen ways. Specific needs and required variables were handled in over-ridden methods, but each

instantiation (or object) of the data type *inherited* the ability to obtain its own structure, maintain and modify attributes, save, delete, select for analysis, and relate to other data in a parent/child relationship. By using these structural data objects, we were able to provide variability in segmentation schemes without losing items required by the business logic.

As the business needs change, new data subclasses can be introduced as necessary. In fact, by using different catalogs containing the class definitions, it is possible to substitute in entire groups of related objects simultaneously. The reason we have this flexibility is that the class definition for non-visible objects is loaded at run-time. By dividing data subclass definitions up into different catalogs, we can select the appropriate class definition on the fly. With the current set of structural data objects, our segmentation is already very flexible, as required in direct marketing. Users can define aggregates of data which are based on any scheme they desire, without any need for reprogramming.

To provide this capability, these data objects were managed through the use of another object, the structural data collection. While an individual data object only knows its children, the data collection can also keep track of siblings. This assists in building selection lists and maintaining referential integrity. Additionally, each data object can have collections of children that are related. This structure makes possible the ability to recursively copy or delete a data item and all of its dependents. It also allows the ability to select certain items into a collection while excluding others, thus providing for dynamic definition of relationships between data. An example of the use of this is the product object. Each product has a collection of child objects defining specific price points. These price points are siblings in the collection. The products are also collected as siblings for use by the campaign.

What about the users?

All of the items I've discussed have been non-visual objects, but this structure lent itself to an easy to implement graphical user interface (GUI). Because of the standard interface supported by the data and data collection objects, a generalized viewer was easily developed. Each frame created with the viewer class utilizes the generalized viewer, as well as any custom code of its own. The viewer provides a toolbox of capabilities including: clear the display, undo changes, save changes, create a copy, delete, and segment (create children). The viewer handles certain automatic tasks such as the creation of a data collection and interacting with the data collection to request a selection list or an action on the data. The frames which utilize the viewer are almost completely unrestricted in appearance and custom functionality, beyond the requirement for identifying the data being viewed. By maintaining a standard interface for the data and data collection classes, the generalized viewer can work with any subclass. Our general viewer is independent of the business logic, allowing for changes to either independently.

Additionally, for each data subclass, we have the option to customize the view in whatever way we desire to enforce business needs. While it isn't necessary to use the viewer to implement the non-visual objects, the viewer allowed for extremely rapid application development (RAD). Rather than spending development time on routine data acquisition and display programming for each individual subclass of the data type, with all of the associated debugging, I was able to concentrate on custom tasks. For example, in one data type there are required financials handled entirely by the general viewer. In another the user has the ability to interactively define their own variables and

values to provide for experimental business models.

The Bottom Line(s)

The arsenal of capabilities provided by SAS made it possible to transform classic object oriented design theory into a functioning application. While a DSS is normally custom designed and developed, this application combines flexibility with customizability in a neat, easily maintained pattern. Objects remain encapsulated and independent of each other wherever possible. In those cases where changing conditions affect a group of objects, we used an object factory to allow modification and growth without mixing the interface and the business logic. Even business logic was broken down into smaller parts to allow more minimization of ripple effects and ease maintenance woes.

Because we've provided options for our inputs and business functions, we can scale our DSS up when our needs exceed the current system. We also have the ability to go across platforms and to store our data in an external database management system. By merely plugging in a new input object that utilizes SAS/Connect® and/or SAS/Access®, our DSS can be used on data throughout the company. Last, but not least, the standard interfaces of these objects make rapid application development truly rapid, with interfaces rising naturally to fit the custom needs of users. Hooray for our side!

ACKNOWLEDGMENTS

Thank you to Sally Goostrey and Andy Norton for reviewing this paper and offering their helpful comments.

Amy Roehrig
Trilogy Consulting, Incorporated
1932 West 680 North
Pleasant Grove, UT 84062
(801) 796-0461
aimless@fiber.net

REFERENCES

Gamma, Erich, Richard Helm, Ralph Johnson, John Vlissides (1995), *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley Publishing Company.

Norton, Andy (1997), "Object Interfaces," in *Proceedings of the Twenty-Second Annual SAS Users Group International Conference*. Cary, NC: SAS Institute, Inc.