

Service Based Architecture

by William T. Green
Castle Software Engineering

Revision: 5.45
Rev-Date: April 18, 2000

Table of Contents

| | |
|----------------------------------------------------------------------|-----------|
| SECTION 1 - EVOLUTION..... | 8 |
| CHAPTER 1 - WINDOW BASED ARCHITECTURE | 9 |
| <i>Benefits</i> | 12 |
| <i>Drawbacks</i> | 12 |
| <i>Identification of Need</i> | 12 |
| <i>Summary and Conclusions</i> | 12 |
| CHAPTER 2 - OBJECT BASED ARCHITECTURE..... | 13 |
| <i>Main Object Based Hierarchy</i> | 13 |
| <i>Benefits</i> | 15 |
| <i>Drawbacks</i> | 16 |
| <i>Identification of Need</i> | 17 |
| <i>Summary and Conclusions</i> | 17 |
| CHAPTER 3 - SERVICE BASED ARCHITECTURE | 18 |
| <i>Power</i> | 20 |
| <i>Performance</i> | 21 |
| <i>Encapsulation</i> | 22 |
| <i>Reuse and Maintainability</i> | 23 |
| <i>Ease-of-use</i> | 24 |
| <i>Flexibility</i> | 25 |
| <i>Market Growth</i> | 26 |
| <i>Drawbacks</i> | 27 |
| CHAPTER 4 - MOVING FROM 2 ND GENERATION MODEL TO SBA..... | 28 |
| <i>Direct Method Extraction</i> | 28 |
| <i>Re-Architecting from the ground up</i> | 29 |
| <i>Re-Architecting from a new Base</i> | 30 |
| SUMMARY..... | 32 |
| SECTION 2 - SBA MECHANICS..... | 33 |
| INTRODUCTION | 33 |
| CHAPTER 5 - CLASS TYPES | 34 |
| <i>Virtual Classes</i> | 34 |
| <i>Abstract Classes</i> | 34 |
| <i>Concrete Classes</i> | 34 |
| <i>Extension Classes</i> | 34 |
| <i>Client Classes</i> | 35 |
| <i>Service Classes</i> | 35 |
| CHAPTER 6 - CLIENT-SERVICE INTERACTION | 37 |
| SECTION 3 - SBA SPECIFICATION..... | 38 |
| INTRODUCTION | 38 |
| <i>Key: Structure</i> | 38 |
| CHAPTER 7 - THE SERVICE CLASS HIERARCHY | 41 |
| CHAPTER 8 - BASIC SERVICES | 43 |
| <i>Typed Implementation</i> | 43 |
| <i>Tight Coupling</i> | 44 |
| <i>Drawbacks</i> | 44 |

| | |
|------------------------------------------------------------------------------------------------|-----------|
| CHAPTER 9 - MANAGED SERVICES | 45 |
| <i>Benefit of Managed Services</i> | 46 |
| <i>Usage Count Management</i> | 48 |
| CHAPTER 10 - THE SERVICE CLASS MANAGER..... | 49 |
| <i>The Service Load method</i> | 50 |
| <i>The Service Unload method</i> | 52 |
| CHAPTER 11 - BROKERED SERVICES..... | 55 |
| <i>Client specifies which services will be used</i> | 55 |
| <i>Service Manager instantiates services and stores service class reference pointers</i> | 56 |
| <i>Service Manager requests that service “export” available service methods</i> | 56 |
| <i>Exported methods are linked to the service class reference</i> | 58 |
| <i>Developer calls a method by utilizing a method reference</i> | 58 |
| <i>Specification: Brokered Service Class</i> | 59 |
| SUMMARY..... | 60 |
| SECTION 4 - SBA BEST PRACTICES..... | 61 |
| *** Coming Soon *** | 61 |
| CHAPTER 12 - CODING BEST PRACTICES | 61 |
| <i>Auto-Instantiate</i> | 61 |
| <i>Constants</i> | 61 |
| <i>Attributes</i> | 63 |
| CHAPTER 13 - USING METHODS..... | 63 |
| <i>Get & Set</i> | 63 |
| <i>Overloading</i> | 63 |
| <i>Overriding</i> | 63 |
| <i>Returning Data</i> | 63 |
| CHAPTER 14 - EXTERNAL FUNCTIONS | 63 |
| <i>Stubbing</i> | 63 |
| <i>Integrated Platform Services (cross platform capability)</i> | 63 |
| CHAPTER 15 - EXTENDING CLASSES IN A SERVICE BASED FRAMEWORK..... | 66 |
| SECTION 5 - SBA TECHNIQUES | 67 |
| CHAPTER 16 – BUILDING AND IMPLEMENTING A SERVICE | 67 |
| CHAPTER 17 - THE APPLICATION-SCANNER SERVICE | 67 |
| <i>Setting the Goal</i> | 68 |
| <i>The Design</i> | 68 |
| <i>Extending the Design</i> | 71 |
| <i>Building the Objects</i> | 75 |
| <i>Integrating the Services</i> | 79 |
| <i>Summary</i> | 80 |
| CHAPTER 18 - SERVICE CLASS FAMILIES - THE TRANSACTION OBJECT | 81 |
| <i>Goal</i> | 81 |
| <i>Design</i> | 81 |
| CHAPTER 19 - THE DATAWINDOW BUTTON (COMING SOON)..... | 83 |
| <i>The Goal</i> | 83 |
| <i>The Design</i> | 83 |
| <i>Expanding the Design</i> | 83 |
| <i>Building the Objects</i> | 83 |
| <i>Integrating the Objects</i> | 83 |
| FUTURE ATTRACTIONS | 84 |
| APPENDIX A | 85 |
| NAMING CONVENTIONS | 85 |
| <i>Service Class Naming</i> | 85 |

| | |
|-----------------------------|----|
| DEVELOPMENT GUIDELINES..... | 86 |
|-----------------------------|----|

Introduction

Service Based Architecture is taking the PowerBuilder world by storm. This is the fourth edition of this white paper, continually updated to provide you with as much information about the architecture as possible.

PowerBuilder has matured to the point where the developers of a few years ago, eagerly diving in to produce applications as quickly as their allotted budgets would allow, are looking back and asking that age old question, "How could I have done this better". I'm one of those developers, and the answer to the question, for me, is in architecture. We tried the traditional inheritance model and achieved a great deal of reuse, oftentimes at the expense of ease of use. As the technology and our knowledge grew, we continued to add new features, extending the inheritance model to it's limits. Now it's time to adapt. We need to utilize the technology for what it can provide. For this we needed a new architecture, and with this architecture comes the requirement for a new methodology. This white paper details the architecture and the methodology we derived.

PLEASE NOTE: We have deliberately avoided using specific notations, such as Booch, Coad-Yourdon, OMT etc.. There are, at my last count, twelve such notations documented in various books. It would lead to confusion. You may substitute notation of choice.

About the Author

William (Bill) Green is President of Castle Software Engineering, a software development company based in New Jersey. Bill is a charter member of TeamPowersoft, writer for PBDJ and columnist for PB Advisor. Bill has been involved with the writing of the PowerBuilder 4.0 Developers Guide and PowerBuilder 4.0 Secrets of the PowerBuilder Masters. Bill is co-author of the book PowerBuilder 5.0: Object-Oriented Design through Deployment(McGraw-Hill-1996). Bill can be contacted on CompuServe @ 71203,1414, or at bgcastle@mail.concentric.net. Feedback is welcome and requested.

Acknowledgments

Although I wrote the first edition of this paper more than a year ago, the reason for it's growth and popularity is the continued feedback, guidance and advise I have gotten from many people. Some of these people have contributed greatly to the success of the

paper, and to forwarding the architecture used in PowerBuilder today. I would be remiss if I did not acknowledge their efforts. My thanks to all those who have contributed, but especially to:

Sandra Barletta

Steve Benfield

Millard Brown

Jon Credit

Boris Gasin

Steve Katz

Mark Pfeiffer

William Rompala

Mark Overbey - Project Manager, Powersoft

Alex Whitney - Director, Companion Products, Powersoft

Section 1 - Evolution

Technology evolves just as everything else does. The difference is that the evolution is measured in milliseconds rather than in years

No matter what language is used, it seems development starts off with the same approach. In the world of Windows development, the key component is the window. So we begin with the window as our most important component. In some languages, windows are called forms, but no matter what you call it, the window is an integral part of the architecture we will use to develop Windows programs.

I've worked with, and continue to work with, several object-oriented languages and the common denominator in all of them is the window. My windows programming began with COBOL. This was a language which, at the time, did not support event-driven programming, and the "windows" were really screens that looked much like the mainframe counterparts they succeeded. Then began the trend which began to change the face of programming.

My introduction to event-driven programming was Visual Basic. This was a total paradigm shift for me, as each control placed on a form had a series of events which were fired in reaction to a user action. Now I had to start thinking about what needed to happen when the user entered a value in a field, or clicked on a button. This shift in thinking didn't take that long to adjust to, much to my surprise, as this was the way we reacted as humans. We react to an event, or an action.

The hard part was in trying to picture the complete program as an assembly of the events programmed to react to the user. When we began to learn that reactions, or events, were similar from window to window if the same type of controls were present, we began to discover the need for reuse. At about this time I was introduced to my first object-oriented language, PowerBuilder.

Now we were able to assemble components that contained both the code for the events we wanted a reaction to, and the data it was to act upon. These components could be reused from one window to another, (if functionality was similar enough), and eventually, reused between applications. I began to learn about class libraries and what they could do.

Chapter 1 - Window Based Architecture

Evolution in architecture is like all evolution. It has to start somewhere. PowerBuilder began with the window - the focal point of GUI based applications.

The first class libraries I studied, built, or used, had one common denominator. The window. As the window was the focal point of our application, we learnt that we can reuse windows if the code contained within the window was generic and implemented only behavior of the window style. The windows were all based on one common window, the base window. The base window more often than not, grew out of necessity rather than by design. Typically, we created a window class that suited a particular implementation. For example, my first generic window displayed a list of data retrieved from a database. It doesn't matter which language you use, one of the first uses of the development tool will be to display data retrieved from a database. So I created this window, which we'll call `w_display_data`. I pre-coded the window to automatically retrieve data based on a set of criteria. I added to the window some buttons to control scrolling, printing and refreshing. Figure 1 shows this initial generic window.

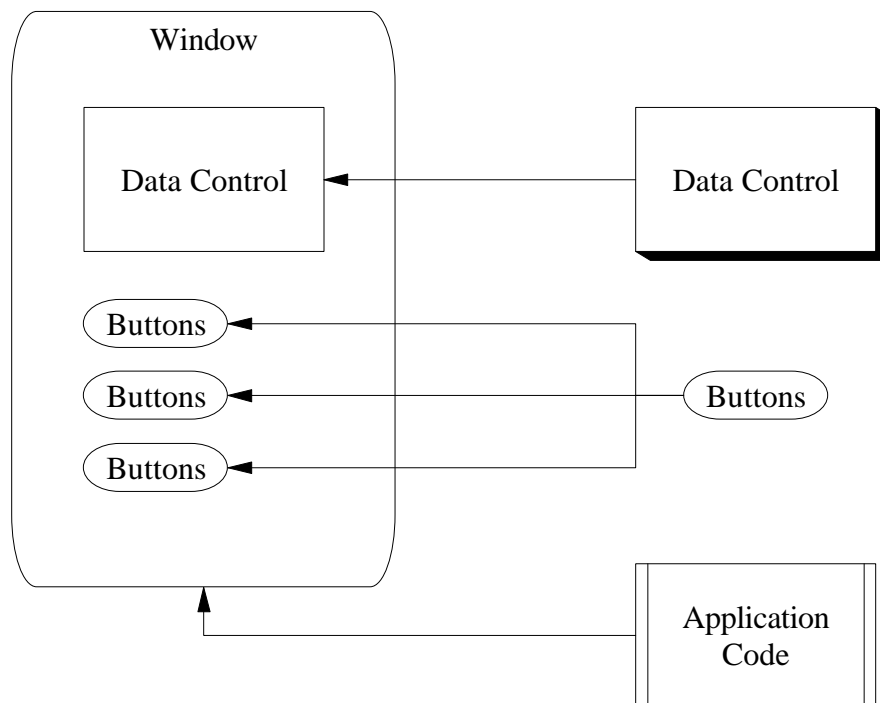


Figure 1 - Generic Detail Display Window

Now that I had this window, it got used a lot. Soon, however, I found a need to be able to update data. This required a new window class which could not only retrieve

data, but also update it. It would also not usually be in the form of a list, but more typically as a single row update presentation. Deciding that the update window was not a descendant of the display window led me to create a new window class, w_update_data. So now I had two window classes, and I was on my way to object-oriented reuse heaven....or so I thought.

I soon discovered that there were a lot of similarities between the update window and the display window, although they were not the same. This is the point where I decided I needed a Base window where I would place all of the common code, and descend the display and update windows from this base. The base window would contain one data control, and Print and Exit were still common. I was indeed proud of the resultant hierarchy. I was beginning to really reuse objects! Figure 2 shows how the hierarchy now appeared.

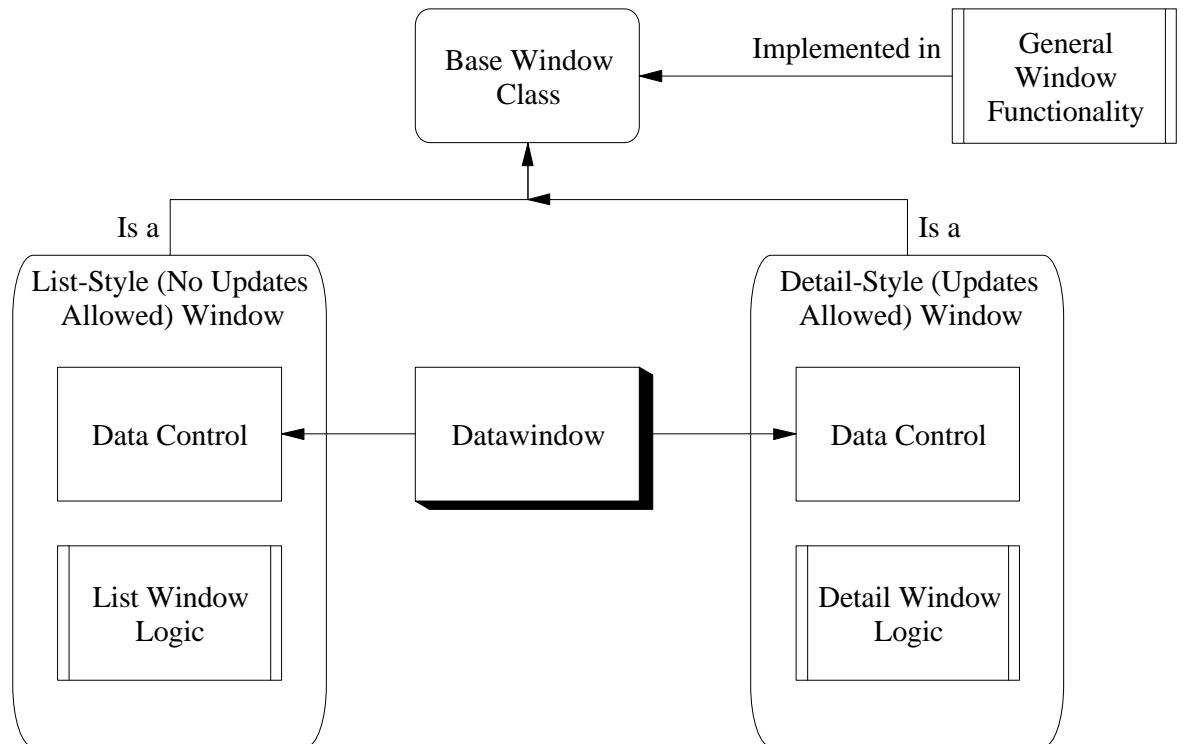


Figure 2 - Window hierarchy begins to form. Base window is used as the generalization of the two descendent windows

As time went on, I continued to find new advances in the techniques I used, and new “styles” of window that included one or more data controls. Elements such as linkage, transaction management and inter-object messaging were built in, sometimes at the wrong level, and then moved up the hierarchy to become more generic. Location of code was little more than guesswork, and most of it ended up in the Base Ancestor because it was common to both styles anyway.

The hierarchy continued to expand as I identified more and more styles that I most often used in my applications. Eventually, I reached a plateau where I felt I had most of the basic styles of windows I would need in applications I had worked with, but the resultant hierarchy was beginning to get messy. Figure 3 shows what this hierarchy looked like at this point.

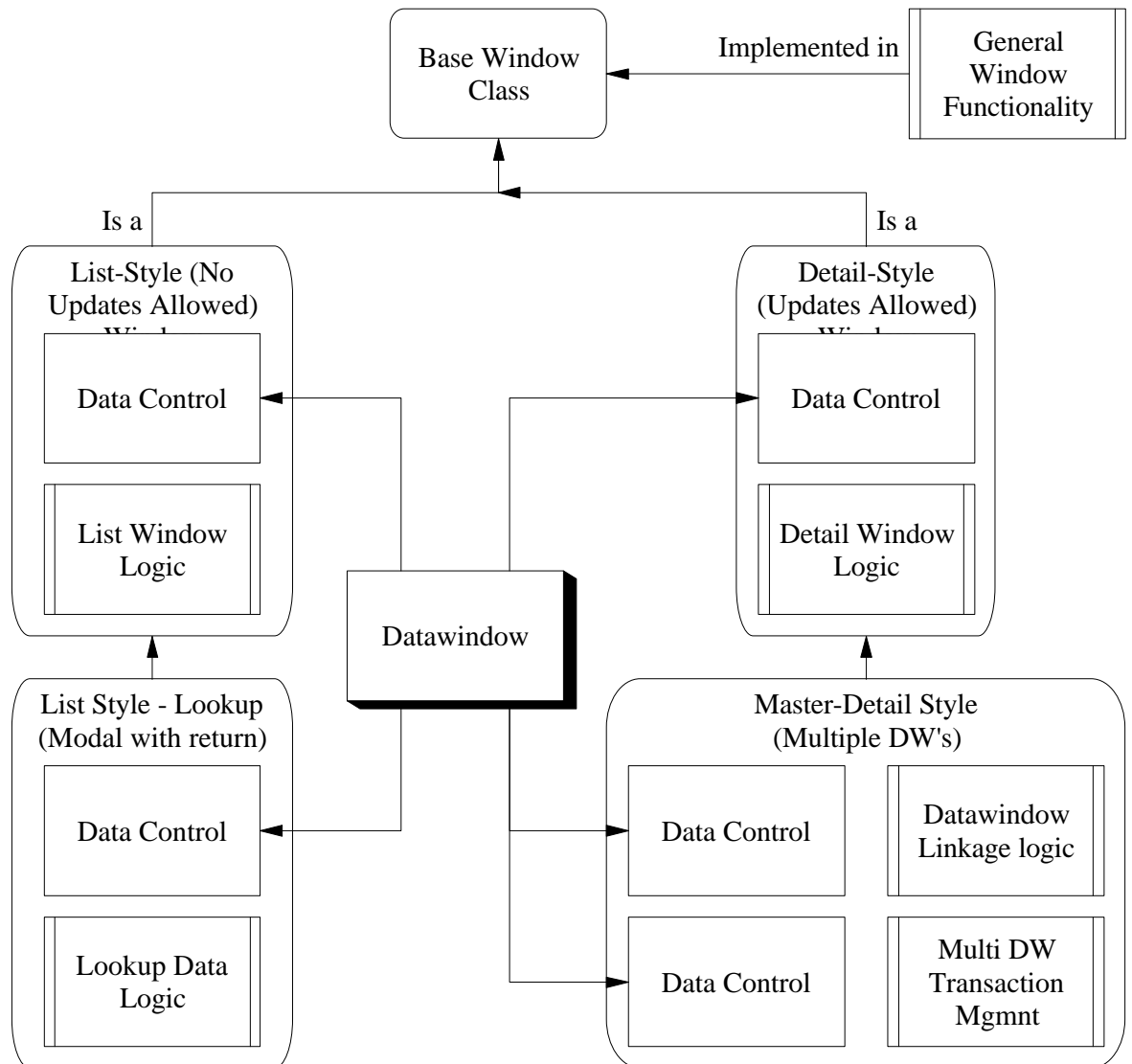


Figure 3 - Window styles expanding and incorporating more features.

What I began to realize at this point was that every time a new style of window was required, I needed to expand the hierarchy of the window class. In addition to that, I often found that I needed only a piece of the functionality from one branch added to a piece of functionality in another branch, and this type of piecemeal multiple inheritance was not only not available, but almost sure to never be available to me.

Benefits

The benefits of this technology was that it introduced us to the world of reuse. We could build reusable classes, or templates of what we wanted to accomplish. Because the architecture was window based, it was a relatively simple task to place code into the correct level of the hierarchy. This type of architecture was admittedly partially a result of a lack of capability within the tool being used, (in my case, PowerBuilder was the only tool with which I could achieve reuse as Delphi and Optima were not even available and I was not a C programmer ready to try the new world of C++), but I noticed that with the introduction of Delphi and later Optima, the exact same pattern evolved.

Drawbacks

The drawbacks to this style of architecture are immediately obvious.

- Every time a new style of window is needed, the hierarchy grows.
- There is a lot of redundant code being placed in different levels of the hierarchy
- Developers face almost impossible choices when trying to decide which object within the hierarchy should be utilized in a particular situation
- Once a choice was made, it was next to impossible to change your mind and select a new object class to use without re-writing the entire window

Identification of Need

What this architecture taught me was that the objects we placed on the window played a much more vital role in our development process than the window itself. We needed to separate the client window from the controls we wanted the window to manage. We also wanted fewer window classes to have to choose from in order to achieve greater reuse and enhance ease of use.

Summary and Conclusions

In conclusion we found that what we needed was an object-based model where our window classes could act as containers, or managers of the objects they housed. This moved us into the second generation architecture, the Object based model

Chapter 2 - Object Based Architecture

Evolving to the object-based model was natural. After all, wasn't this an Object-oriented language? Suddenly the datawindow becomes an object rather than a control.

The object based model was a natural progression for me as a PowerBuilder developer. I was firmly entrenched in the belief that 4GL tools were the wave of the future and object-oriented technology was to be the surfboard to ride that wave. As a PowerBuilder developer, I realized that the datawindow played a huge part in my development efforts and that the datawindow should be the main object class rather than the window.

My first task was still to reduce the complexity of the window based hierarchy and introduce a new suite of reusable objects which would operate in tandem with the window upon which they were placed, and yet retain their autonomy and allow me to expand on the object-based hierarchy with ease.

Main Object Based Hierarchy

I began with simplifying my window hierarchy. I developed a set of window classes which reflected the styles of windows I was comfortable with, and which I found to satisfy most of my application requirements. Figure 4 shows the new window hierarchy which now appears much flatter and more style specific.

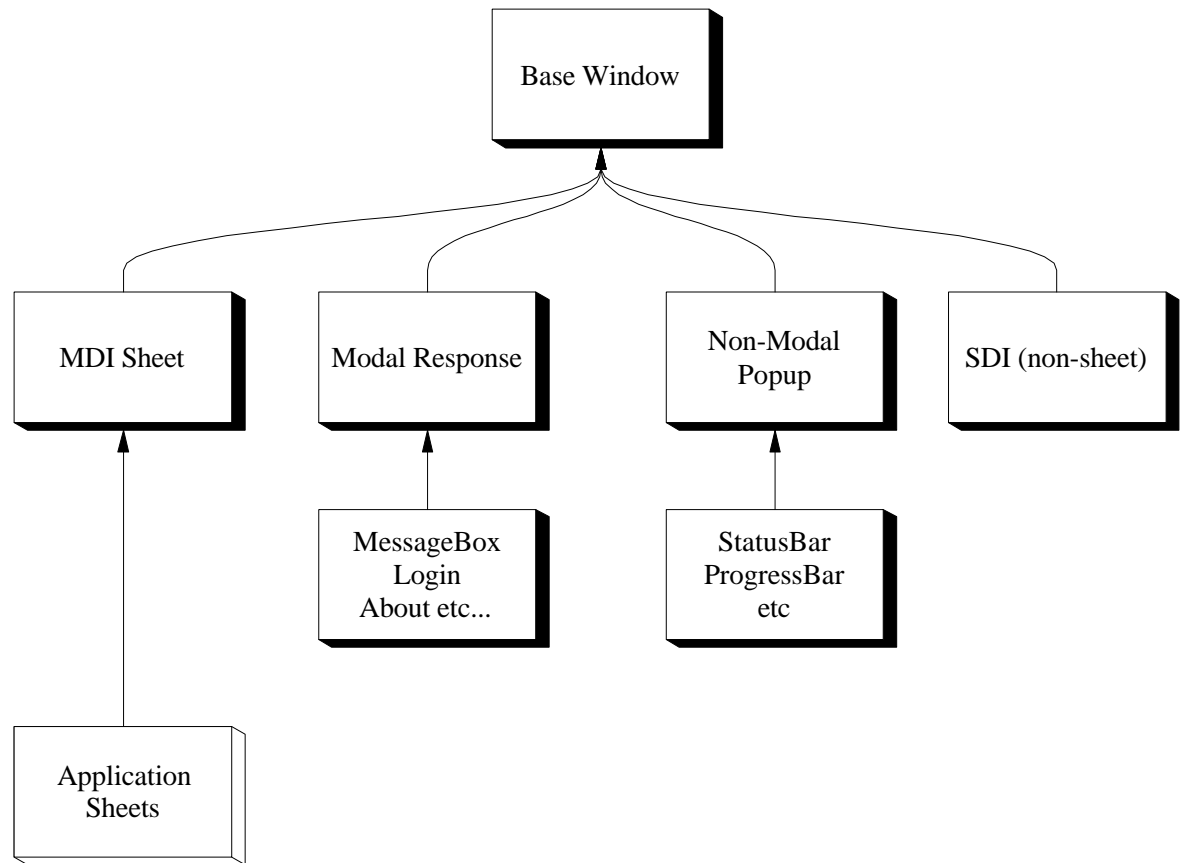


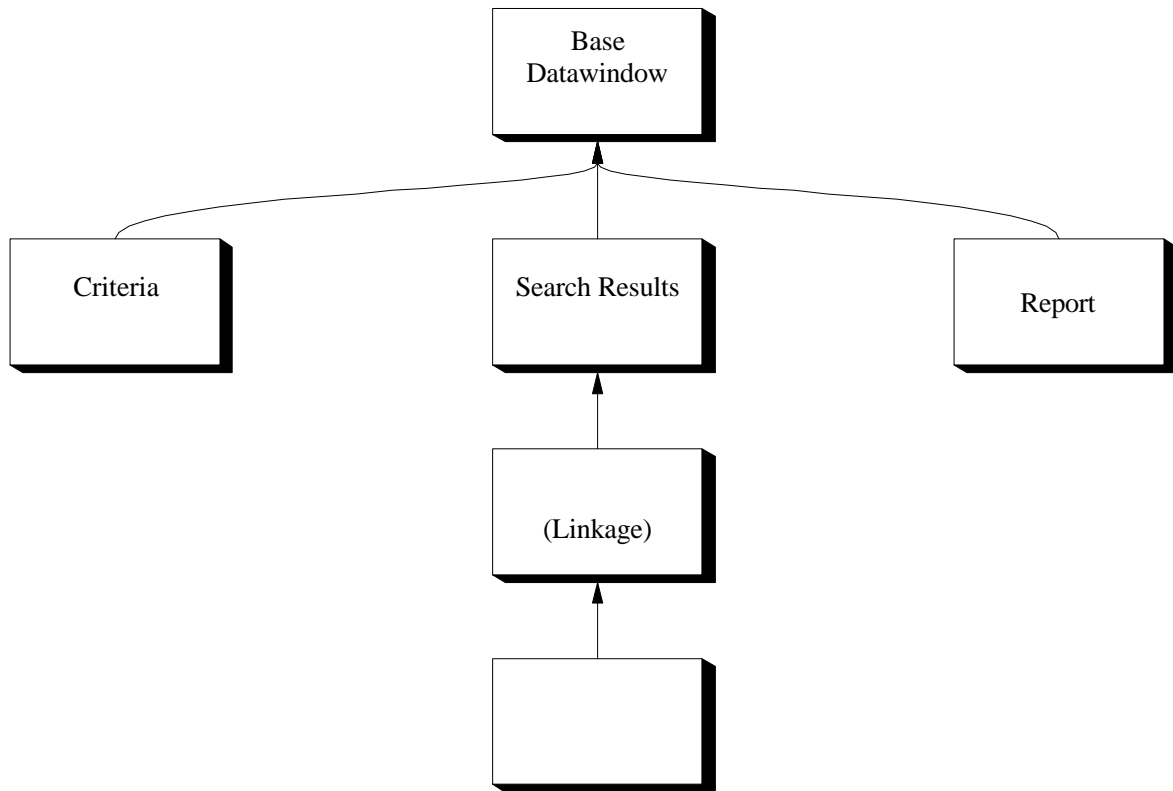
Figure 4

What I now needed was a new hierarchy of object classes which would become the mainstay of the objects that I would ordinarily place on these windows. In order to

being used. While windows are common between tools, the objects being placed on the window or form are very different, often being what distinguishes various tools.

ever seen. It's capabilities and various styles that could be used based on a single class, was often overwhelming. This was the obvious candidate for the class that would most

I began developing a hierarchy of datawindows which implemented behavior at various levels, based on what I saw was the logical extension of the parent class. Figure 5



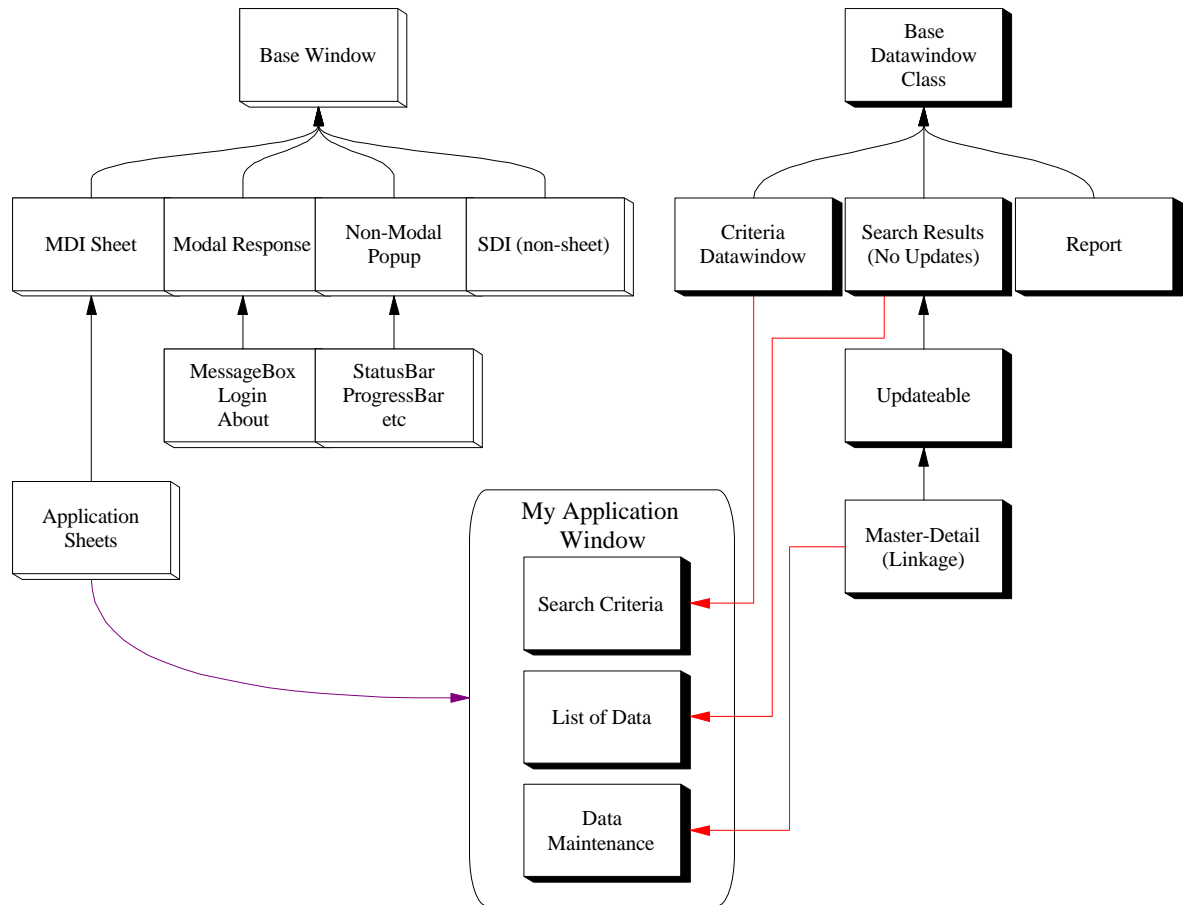


Figure 6 - Building an application utilizing available components.

Drawbacks

What was not immediately obvious, was that the shift of responsibility would result in larger and more complicated “functionally aware” data access controls. It was difficult to master the mindset that the datawindow was simply a client object which required it’s functionality to be a little more user-definable.

PowerBuilder is an object-oriented language with few limitations, but one of those limitations is the lack of multiple-inheritance-ability, which would certainly have allowed me to develop a more comprehensive set of smaller, and more efficient, datawindow classes. The results of my efforts left me staring at a relatively direct hierarchy, certainly not the intention of the inheritance phenomena, and these classes simply grew in size to the point where they became cumbersome and very difficult to maintain. I needed to be able to “separate” the behavioral content of the datawindow class into a more usable and maintainable hierarchy.

I quickly discovered that the pitfalls of the object-based approach was a complicated tangle of “fat client” objects where object choice became an important aspect of my development effort, as it was not easily changed. Another pitfall was that the more functionality that got built into the object class, the more difficult it became to figure out where to add the next set of functionality.

Identification of Need

The need was for a more comprehensive solution that would still allow me to take advantage of the natural power of controls such as the DataWindow, but to be able to decide the level of imbedded functionality, or ability, that the object would need, dependent upon where it was to be used.

We also needed to put our objects on a weight loss program if the objects we designed today were to ever be useful to us in future applications, such as Internet or Distributed applications.

Summary and Conclusions

We came to the conclusion that although the object based approach was certainly a useful step in our evolution as object-oriented programmers of business applications, there was certainly a lot left that we could do to further make use of the capabilities of the paradigm. This led us into the next generation of architecture, Service Based Architecture.

Chapter 3 - Service Based Architecture

There are several nomenclatures for Service Based Architecture. SBA, as it has become to be called, is simply the shifting of responsibility from one object to another yet again, but this time in such a fashion that the behavioral aspects of the object could be isolated into very specific groups, known as Services. The object that used the services were Clients. It fit into the Client-Server paradigm which most of use are trying to conquer in the business world today. But what exactly is Service Based Architecture, and how does it help alleviate the problems we have seen in the previous architectures?

Service Based Architecture dictates that there are two distinct roles that the various classes assume. One is that of a Client, the other of a Service. The client class becomes a requester of services that are not contained within the client object. The client class is capable of deciding which services it needs to implement at any given point in time, and is therefore much more in control of it's capabilities, without carrying any overhead for those services it does not need. Service classes become delegated behavioral classes. The service object should be compact, combining functionality which relates specifically to it's purpose.

The architecture grew more out of a need to separate functionality (isn't separation similar to encapsulation?), provide more capability and flexibility to developers while retaining the common access point to the objects functionality. SBA provides developers of class libraries and applications alike, the ability to standardize an interface to object functionality like never before. The benefits of this architecture are tremendous, but can be categorized into 7 main areas, each of which we will discuss in more detail. They are:

- Adds power to the already powerful object-oriented development tools.
- Improves performance over "fat client" styled objects
- Improve Encapsulation allowing further componentizing of classes, enhancing the possibility of code portioning
- Make reusable classes easier to use. Common component development techniques promote knowledge transfer
- Make code more maintainable while still promoting reuse

- Enhance flexibility by allowing the developer to select what functionality is wanted on a particular object
- Create a new market for component classes

To begin discussing these benefits, we need to look at the big picture. Figure 7 shows the Service Based Architecture model.

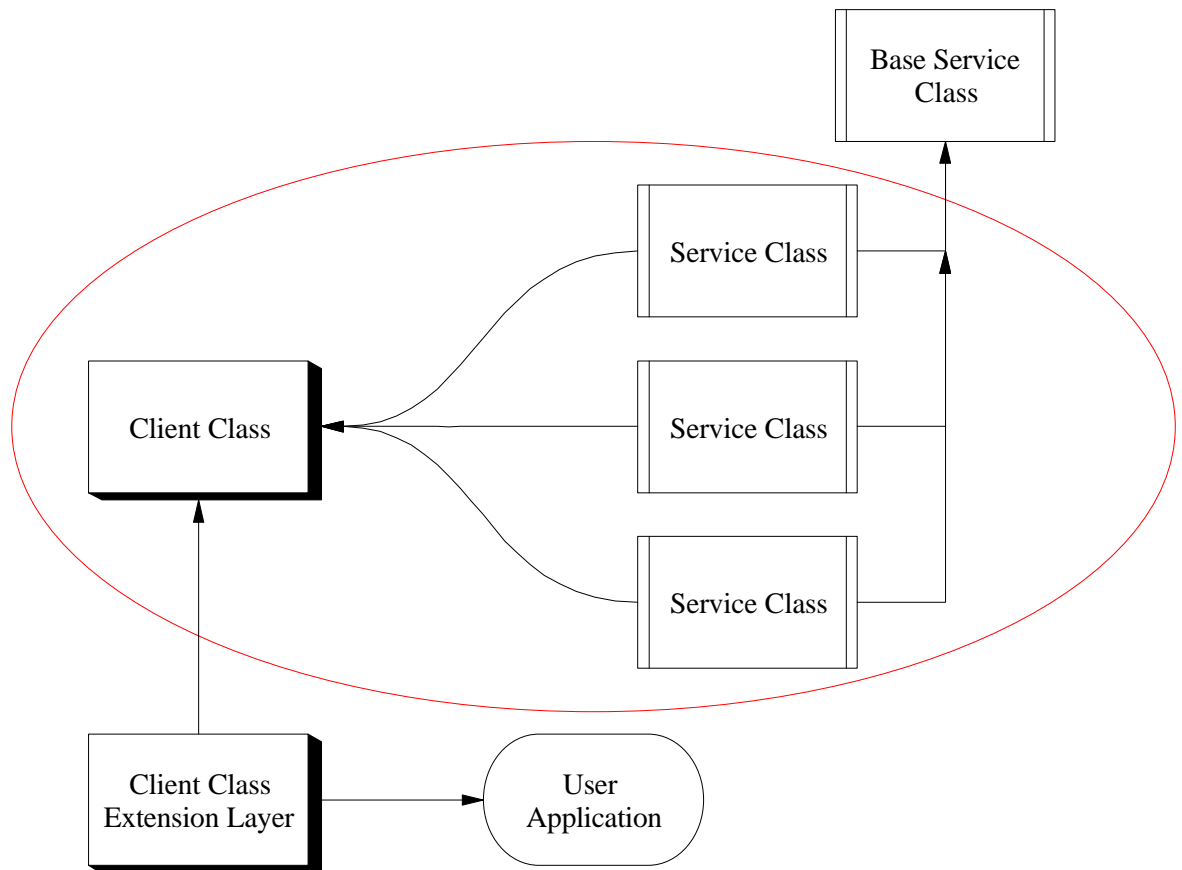


Figure 7 - The Service Based Architecture Model

Note that the “client” object is part of the flat inheritance model while the “services” are part of a traditional model. The client object is referred to as such because it does not implement a lot of functionality, rather it serves as the container for selected services. The service classes are typically nonvisual in nature, but are not required to be.

Note also that we still rely on the traditional inheritance model to promote reuse within the service hierarchy, but the developer does not have to be aware of this. What does this achieve? Let’s take a look at those seven categories one at a time.

Power

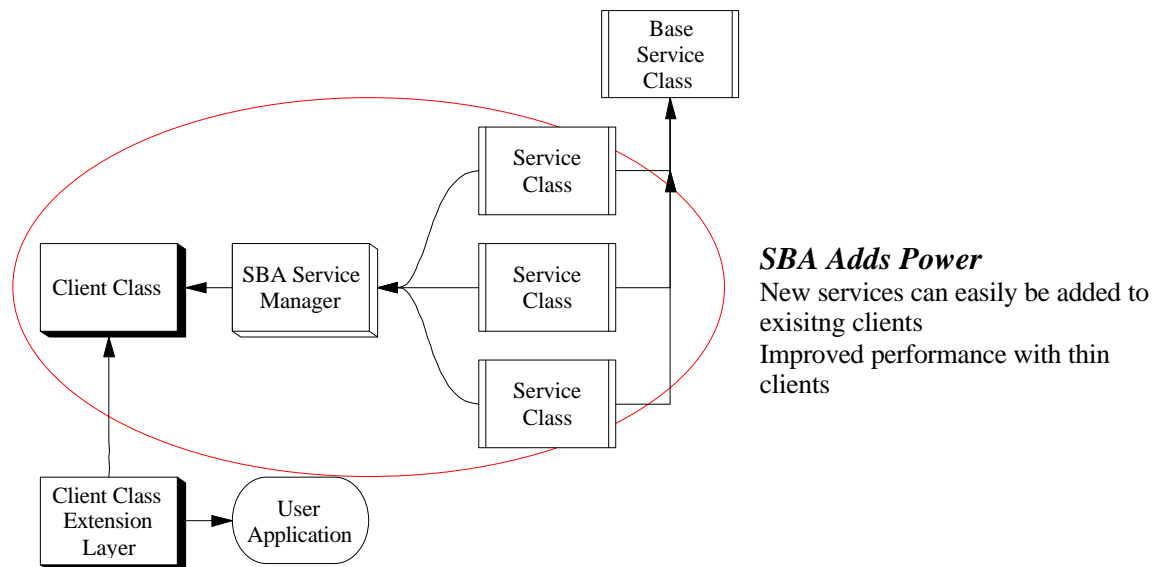


Figure 8 - SBA Adds Power

SBA adds power to the already powerful features of object-oriented development by providing the developer with the means to encapsulate specific behavior in very specific classes. SBA is not a form of inheritance, rather it is the implementation of delegated behavior in a developer user-definable fashion. New services can easily be added to an existing client, allowing you to make new, tested service classes, available to your developers without affecting their existing code-base. For example, we recently added generic stored procedure services to our datawindow client class allowing developers to change from their utility generated, or often hand-written stored procedure code, to a single service class declaration, the service taking care of the remainder of the requirements.

Performance

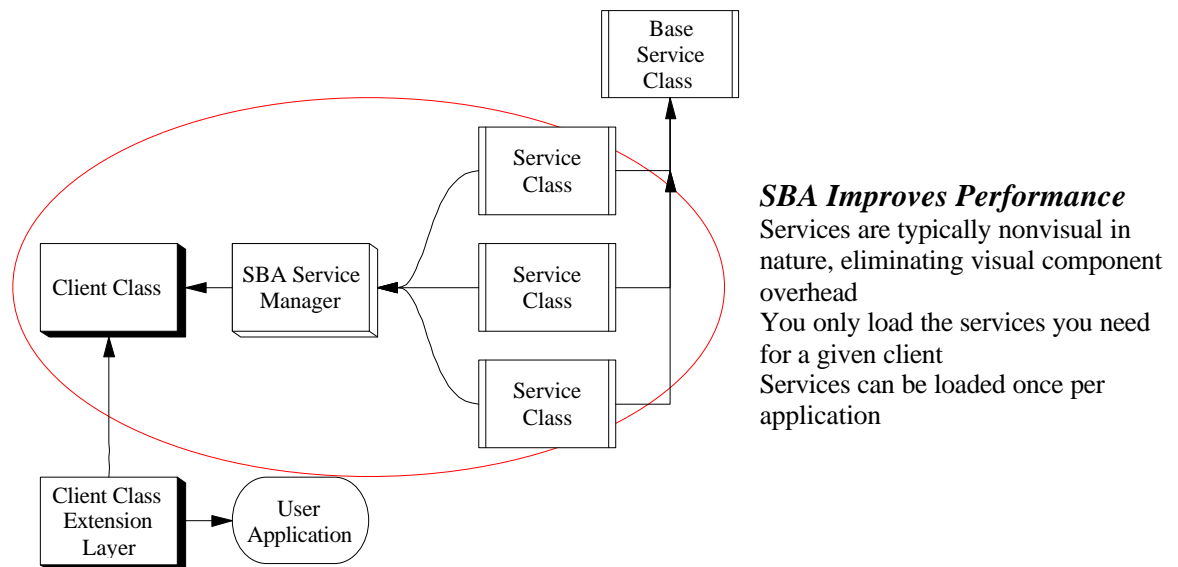


Figure 9 - SBA Improves Performance

The architecture improves performance. How many times have you heard that said? This time, however, it's for real. Consider the following example. We used to have a datawindow object hierarchy as follows:

| | |
|------------------------------|---------------------------------------------------------|
| Base Object | 90K of code/attributes |
| Search Results | 30K of code/attributes (Added SQL capabilities etc) |
| Parent-Child management etc) | 80K of code/attributes (Added parent/child, transaction |

To open up a parent-child object in development took 8-12 seconds on my 486/75¹. In run time, the simplest parent-child window took 1.5 seconds to open, a complex window taking up to 8 seconds to open. We re-architected using SBA and we now have a hierarchy such as this:

| | | |
|---------------|-----|---------------------|
| Base Object | 9K | (Object Management) |
| Client Object | 33K | (Service Control) |

¹ Note that the performance differences were distinctly measurable on a slower machine. The same objects on a Pentium 100, loaded in less than 2 seconds.

We apply up to 360K of services in any datawindow class now, so we also added functionality during the re-architecting. In development mode, our application extension level client object opens in about 2 seconds. In production, our simple windows open up in sub-second times, while our more complex windows still take up to 3 seconds - We're still working on these. The difference is almost staggering - We were not quite SBA aware when we did this and missed some crucial areas of performance improvement, especially the 64K limit.

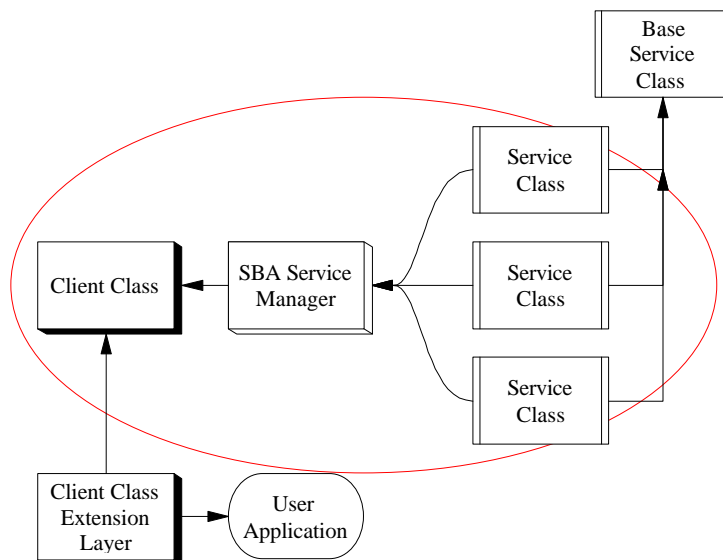
Under win 3.1, objects over 64K (any single object, not a hierarchy total), can take up to 35% longer to open (at run-time), than an object smaller than 64K. There is a pointer allocation algorithm invoked when memory allocation requirements exceed 64K. This limit is not as apparent under win 95/NT, but it is still measurable. This does mean that you should be careful to not exceed the 64K per object limit when deploying to the Intel 16-bit platform. (Service objects are included).

Note

This is not a physical limit. It will not stop your objects from functioning correctly. It does have an effect on object load time however.

One reason is obvious - Size. The second reason is that nonvisual classes are not impeded by graphics processing, often the bottleneck in windows applications. Finally, instantiation of services can be delayed (for the most part), until after the client object is instantiated (and removed when not needed - Implementation Options discusses this in more detail later).

Encapsulation



SBA Enhances Encapsulation

- Services are typically small, focused classes with a very specific purpose
- Services should not maintain data about the client allowing the service to be used across different client classes

Figure 10 - SBA Enhances encapsulation and encourages use of OO techniques

SBA further promotes encapsulation - this is the premise of the service objects in the first place. Encapsulate processing into small, manageable chunks which are completely independent of each other, other than by ancestry. For example, row processing can be encapsulated into a single service. Row Selection services may be one service that could be implemented as part of row processing, allowing even further encapsulation. For example, you might create a family of row selection services, say Single-Row selection, or multi-row selection, and based on developer choice, implement one of these as part of row processing. Now the developer can choose to alternate the service by datawindow, or even within the datawindow. When a change to multi-row select is required, the service class developer knows it is contained within the multi-row selection service class. Which brings us to the next benefit, Maintenance.

Reuse and Maintainability

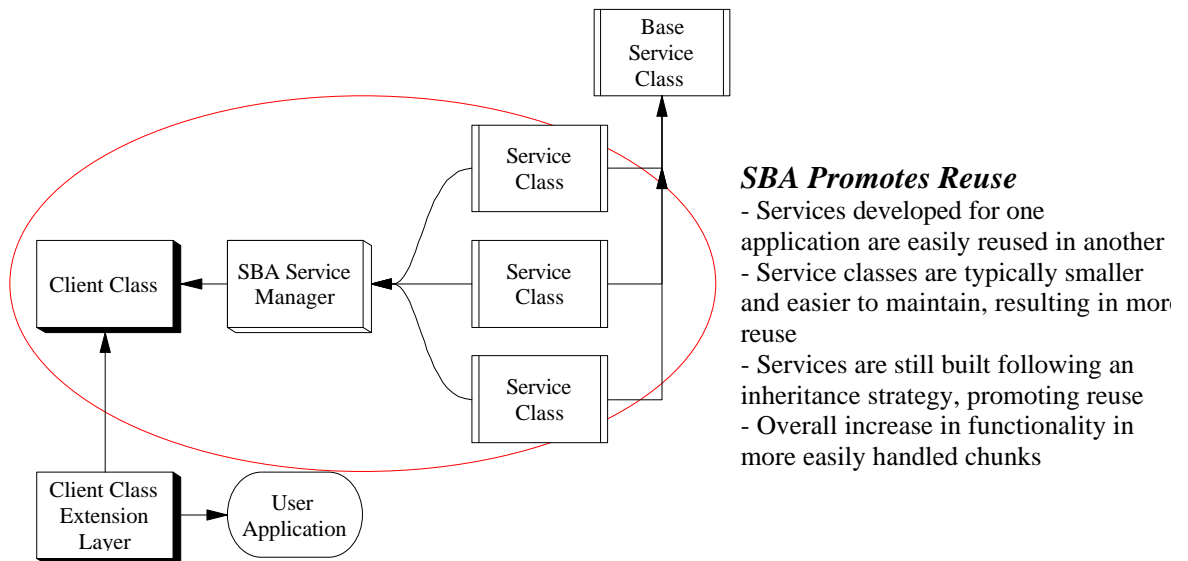


Figure 11 - SBA promotes easier maintenance of reusable classes.

If services are well-encapsulated, the benefit of easier maintenance is almost a given. It's one of the OO fundamentals that reuse will reduce maintenance. What's less well publicized is that good encapsulation promotes easier maintenance. It also means that when a change is made to a service class, the developers that use them are less likely to be impacted, because smaller chunks of code are easier to write, test and document, almost assuring you of higher quality. I said almost. Bad code in any size object causes severe problems in a reusable class environment - It is simply easier to trace the problem to a specific service, correct, re-test that service, and redistribute the corrected object. Service classes still follow a traditional inheritance model however, promoting reuse at the service class level. One thing this means is that your classes will be easier to use.

Ease-of-use

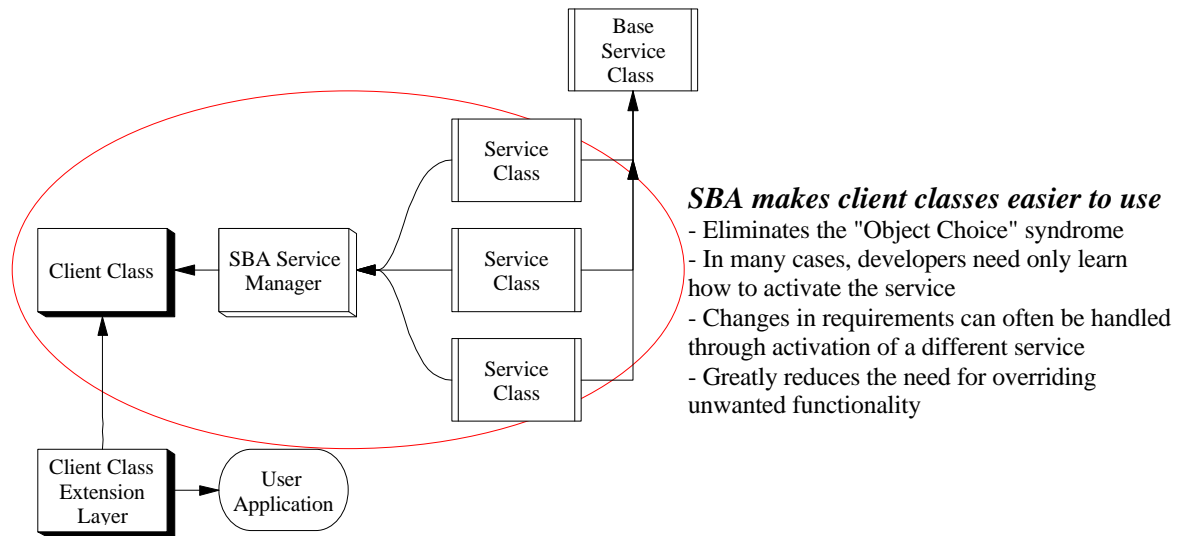


Figure 12 - SBA improves ease-of-use

Have you used a hierarchy which seems simple enough at first, but realized just how complicated the simple hierarchies can become? For example, I mentioned our old hierarchy of Base -> Search Results -> Parent-Child. Seems simple enough, but it causes masses of confusion, and enhancement nightmares. Let me elaborate.

I'm working on a project and I'm preparing a particular window which contains 3 data objects. A Criteria object (which is an element of our hierarchy which was not discussed, but descends from our base object as a peer to search results), a list of data driven by the criteria, and a maintenance object linked to the list of data. Seems simple. We mapped the criteria object to our Search Criteria, the list to Search Results and the maintenance object to the Parent-Child object. But what if we were told that the list of data might be updateable, and might be driven by a different lookup on occasion. Well, Search Results is not updateable, and Parent-Child is the master of linking datawindows together, so perhaps we want to go Search Criteria, Parent-Child, Parent-Child. OK. So we're carrying more than half of the Parent-Child functionality as overhead, never to be used. So what? It's OO right? We can override whatever is not needed!

So now the system is developed and the user comes back and says, "That list object will not be updateable after all. Make sure it's non-updateable". Do we go back and change the inheritance? We could, but with only 2 days to User Acceptance Testing (UAT), do we take a chance on the export-change inheritance-re-import capabilities? Multiply this a few times, and add in a few more complex scenarios and you can see that this process can quickly escalate to Nightmare on Inheritance Street.

With SBA, you no longer have to choose the level of functionality beforehand. In our new client hierarchy of Base -> Client -> Application Extension layer, you always use the Application extension Client object. No more choosing. Inside the application extension client, you can choose to activate the services necessary to Build SQL (Search Results), Manage Transactions and Manage Relationships (Parent-Child). When the user says, no, that list must be updateable, you open the object, activate the Update services, and viola! Updateable list object. Change your mind, deactivate the service. Now your application can adapt to user requirements much more quickly, and safely. This also means....Enhanced Flexibility!

Flexibility

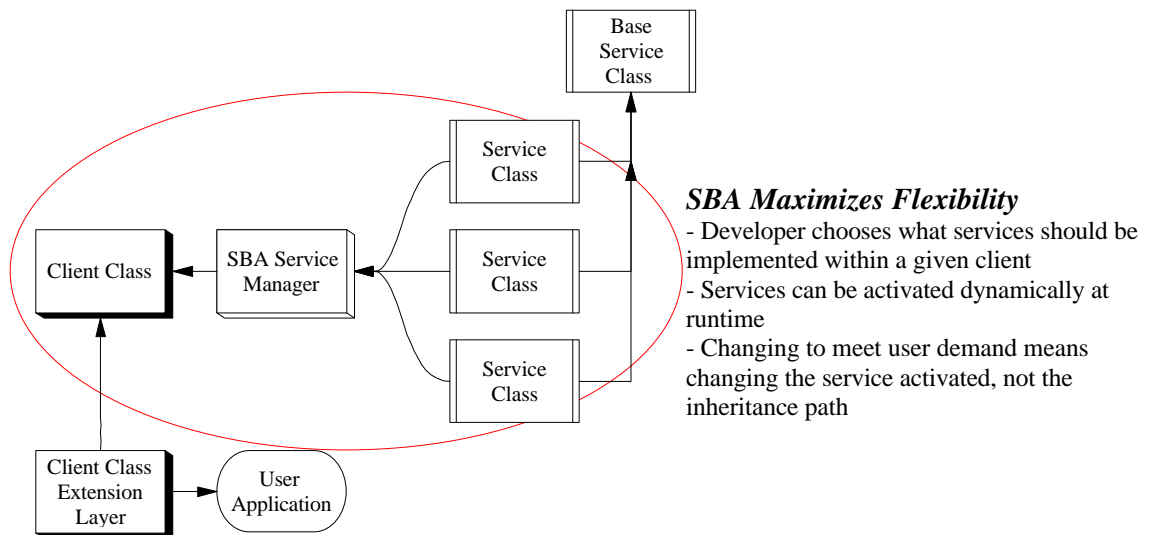


Figure 13 - SBA Enhances flexibility by giving the developer more choices as to what each individual object may or may not do

Two things spring to mind with flexibility. Be able to specify what each individual object is capable of doing, and be able to specify what an object is capable of doing at run-time. SBA allows the developer to choose what services to make available to the client object. This does not mean they are all activated immediately, although you could code them that way if you chose to. Using the service based approach means that you are able to set up the service implementation in a variety of ways, some of which might differ in actual implementation depending on the development tool you are using. Implementation Styles and scope are discussed in detail later.

Market Growth

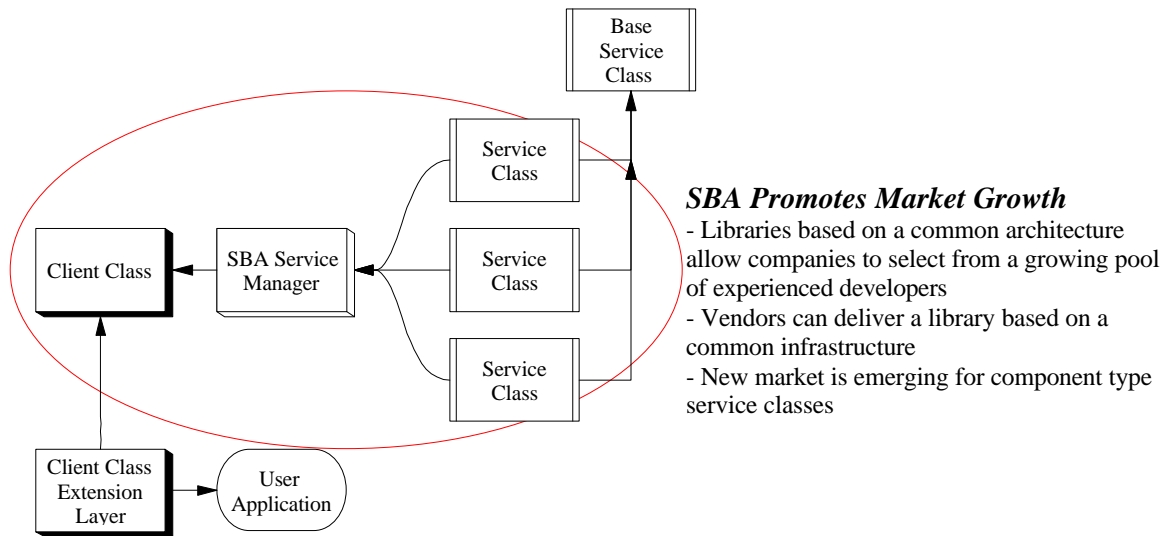


Figure 14 - SBA promotes market growth

With services being available in small, manageable components, I see an opening for a new market for components, be they PowerBuilder, Optima or Delphi, DLL, OCX, ActiveX or other forms. Specifically, two areas of component development will grow. Service components, and then Business Components.

In addition, with the growing adoption of Service Based frameworks, such as Powersoft's PFC library, developers learning the PFC library will become more marketable to organizations using any library based on the PFC. This is an area of increasing importance. Have you worked with a class library and been in the position of having to try and hire someone for your organization? I have, and a premium is being placed on people who have experience with your specific class library. Those with experience with any class library are rising to second on the list, while those without any reusable class experience, are often being overlooked.

With SBA, we might begin to see a standard being developed for service object components which will promote even further reuse. OCX's and more recently, ActiveX's are enjoying huge popularity at the moment because they offer a standard for design and implementation so developers and tool makers alike can utilize them. PowerBuilder, Optima and Delphi offer the capability to build inbound OLE automation servers, a fancy way of saying a class built on a specification for linking, opening up a door of huge opportunity.

Drawbacks

SBA is not without drawbacks. No architecture is. The biggest drawback to SBA is in introducing yet another paradigm to the clutter of client-server paradigms. Developers will have to learn a new method of developing, but I believe that the end justifies the means. The second drawback is that the costs of moving to the Service Based Architecture, for already established code bases, might be prohibitive. Below is a list of the drawbacks to the SBA architecture.

- Choosing implementation styles must be done carefully. While not extremely difficult to change, can cause confusion
- Requires a new way of thinking - adds one more learning curve for developers to overcome
- Existing libraries using 1st or 2nd generation architecture will not be easy to convert - Not cost effective over the short term
- Thousands of applications based on pre-SBA libraries suddenly become “legacy” systems with no simple solution to migrate them forward
- Evolution of the architecture leaves us wondering how long it will be before the next generation arrives to obsolete this one
- No industry standards for the development of services to support the architecture

Chapter 4 - Moving from 2nd generation model to SBA

Even once you adopt the paradigm, you will face issues. How do you get there, what about distributed processing etc.

This is becoming an increasingly important decision. More and more designers and developers are adopting the SBA approach, but re-architecting is not easy. There are many issues to address. We faced these issues when we re-architected our library a year ago (we have an in-house developed library developed jointly with an outside vendor). Our decision was this.

SBA forces a new paradigm on the developer. API entry points, (and yes, you can consider calls to object functions the API of the object as this is generally the objects interface), will change, making your old objects not backward compatible. This is a very real issue when dealing with an environment that has several projects already in production using the older library.

You have two options. Drop the old versions of the objects, replacing them with new re-architected objects and force development teams to upgrade. Benefit: Old objects are completely discarded. Drawback: Cost. Development efforts in maintenance do not have the same kind of financial backing as new development efforts. The second option is a little easier to bear. Maintain your old objects in a separate library for compatibility, allowing existing projects to continue using them. Develop the re-architected objects with a new hierarchy and name, and let new development efforts utilize the new objects. Maintenance efforts can convert objects individually when needed and justified. (Improved performance is a powerful justification tool).

Now your decision remains how to develop this architecture. One way to do this is to mirror your current functionality. A second is to re-architect the object completely from the ground up. We'll take a look at both methods.

Direct Method Extraction

Direct method extraction involves mirroring the functionality of your current client object using the SBA paradigm. Basically, this moves the functionality from the client object to the services objects, maintaining a very similar functional level as before. For example, if you have a datawindow userobject in which you implemented a method to execute a filter, you might have a function `OF_Filter()` which executes the filter operation. To move this to the SBA world, you would follow these steps.

Create new Client Object (`u_dw`)

Create service object (`nvo_filter`) based on your defined service hierarchy.

Create method in service (`of_filter (u_dw, args))`

In client object, create method to instantiate service (OF_SetFilter())

In client object, create method which calls service (OF_Filter() which does:
If invalid(nvo_filter) then nvo_filter.of_filter(this, args)

In client object, add an OF_SetFilter(False) call to your services Destructor method

Benefits

This method allows you to move pretty quickly from the datawindow model to the SBA model, realizing some of the benefits of SBA like more encapsulated functionality, improved performance etc. If you're under the gun to get from point A to point B, then this method is the best suited for you. The functionality you are using will be proven code, making test cases reusable. Conversion of applications using the classes will also be relatively simple as they will be able to replace existing method invocations with methods residing on a service class. As you move down your object hierarchy, you'll be able to do the same thing, adding descendant class functionality to existing services, or creating new ones.

Drawbacks

This is not truly an SBA approach. This is function load-shifting. The solution will not be the optimum that can be achieved, and you will probably find you end up with service classes that are rather large, some even exceeding that magical 64K barrier. You will also find that your service classes are tightly coupled to the client objects, restricting the ability to become true service classes.

Re-Architecting from the ground up

This is, of course, an expensive operation, and if you are not willing to accept the costs, this is not the right option for you. Re-architecting your client and service classes from the ground up will, however, allow you to build a true SBA environment where services can easily be moved between operation tiers, apply functionality in a fashion which is not dependent on the object it is servicing, (for example, Clipboard Services can be truly object independent etc.). What this means is that the architecture can become more loosely coupled so that inter-object dependencies can be reduced, if not eliminated.

If you already have a client class hierarchy, then you can still use this as the basis for developing your new SBA hierarchy. Collecting as much information as you can, you can build a set of requirements that you wish your client object be able to perform. Once you have this, forget about the existing class and how the methods were accessed. Re-design the services following a proven OOD design technique, (OMT, UML, CBM etc.) and then build your new hierarchy from the ground up. This is exactly what I ended up doing for myself after looking at the result of our first cut at this which used the method extraction techniques described above. What I ended up

with was significantly reduced inter-object dependency, even better performance, and also introduced me to the world of brokered services. Brokered services are simply those whose interface is defined to a broker rather than to the client object, allowing you to add in new services without actually touching the client object. Brokered services are described a little later

Benefits

The major benefit of re-architecting from the ground up is that you are able to clearly define the architecture and utilize the full power of SBA. You are not bound by previous definitions of functionality.

Drawbacks

The most obvious drawback here is cost. Class libraries are not cheap to design and develop correctly. I've heard of development efforts running into several worker-years before a usable product results. Of course, we know a lot more now than we did then, but it is still a very expensive process.

Re-Architecting from a new Base

If you're starting from scratch, I'd first suggest looking at the PFC. It already provides you with the groundwork for a sophisticated application framework, and already implements the SBA paradigm. If you really want to build everything yourself, define the requirements for your services and their clients², and enlist the help of an expert who knows PowerBuilder and, more importantly in my mind, OOA and OOD. Once you become familiar with the PFC and it's internals, you might want to design services to fit into the PFC mold, relying on Powersoft to maintain the central portion of your source code. Several class library vendors are doing exactly that, and my current client has followed this lead, embarking on a project to re-architect our existing class library to be based on the PFC.

Benefits

Using the PFC as your base enables you to shift the burden of maintenance for your library over to Powersoft. If you begin with the PFC, your core infrastructure is maintained by someone else, resulting in reduced costs, both during development, and during maintenance. A huge plus is that development resources will become more readily available. As more and more developers become familiar with the PFC, so will your options in the marketplace. You can find an experienced PFC developer which

² Methodology to design classes with SBA being a specific target is described in detail in PowerBuilder 5: Object-Oriented design & development (Green, Brown: McGraw-Hill 1996) and will not be discussed here

automatically ensures you that the developer already knows 50% (or more) of your class library. Resource availability cannot be underestimated as a benefit.

Drawbacks

The approach locks you into an architecture and an infrastructure from which a departure might mean another re-architecting and/or conversion overhaul. Having your base code maintained by someone else could also be seen as a drawback. Finally, revisions to the product have to be scrutinized much more closely as the impact of change will be more far-reaching.

Summary

Service Based Architecture is not the end of the road, rather I see it as one leg of the journey. I also think we'll see more evolution as more of our applications begin targeting the Internet/intranet worlds.

In conclusion, Service Based Architecture (SBA) is a powerful technique that can be utilized to make applications more efficient, maintainable and reusable. To summarize the facts presented, we have learned that SBA is:

- More powerful: Service based architecture allows developers to achieve implementations emulating capabilities inherent in other languages.
- Performs better: Smaller physical objects are loaded - Services can be loaded when needed, improving overall performance.
- More encapsulated: Processing is encapsulated at the service level - smaller objects with less code to maintain and reduced reliance upon global accessibility, improving componentizing
- Easier to use: Client objects follow flatter hierarchy model enhancing ease-of-use while services are selected using declarative attributes.
- More maintainable and promotes reuse: Services follow a traditional inheritance model promoting reuse and maintainability.
- Enhanced Flexibility: an object need not necessarily be instantiated - Developer can control using declarative attributes. Developer can change object capabilities by changing an attribute, not the objects ancestry.
- Promotes Market Growth: As more and more object classes are built into services, some services will begin to be offered as commercially available components. Indeed, we might see service class components being delivered for various OCX's etc..
- How to get to an SBA architecture from a 1st or 2nd generation architecture.

The sections that follow will break out the specific requirements of SBA based classes with detailed specifications for the development of SBA classes and managers.

Section 2 - SBA Mechanics

Understanding the interaction between client and service objects is tantamount to your success in utilizing the full power of the SBA approach.

Introduction

There are two elements involved in an SBA application/framework. A Client and a Service.

Even within this boundary I found sub-categories of classes which affect the way you look at the class. The sub-categories are not hard boundaries and an object may cross the boundaries at some point in it's lifetime, perhaps even dynamically, but it has helped me to keep a good vision of what I think the class should support and how I should approach extending it's capabilities.

In this section I will discuss these "types" of classes, as well as try to provide a good sense of what a client class is and what a service class is. I will also show the interaction between client and service classes.

I have identified four basic types of classes in use in a framework. Although the boundaries between these can become blurred, it is useful to lay these out so that they are easily identifiable. The four types of classes are:

- Virtual Classes
- Abstract Classes
- Concrete Classes
- Extension Classes

Let's take a look at each of these and try to isolate exactly what distinguishes the classes from each other.

Chapter 5 - Class Types

Virtual Classes

A virtual class is one which has no particular function other than to be the ancestor of a group of abstract classes. For example, a base window class (w_base) can be defined as being an abstract class if it has no descendants. It's intended for you to inherit from and specialize for your purposes. If, however, you create descendants of w_base such as w_sheet, w_dialog etc. (which are abstract classes themselves), then the intention is to use w_base *only* as the ancestor (or aggregate) class for the window classes. This makes it a virtual class.

NOTE: PowerBuilder does not support the C++ construct of virtual classes and functions, and the intent is not to confuse the two here

Abstract Classes

Abstract classes are those classes which are partially specialized, but still intended to be used through inheritance only. As above, w_sheet might be considered an abstract class as it is from this class that a developer will inherit an application sheet for use within their application. From a certain perspective, all of the standard PowerBuilder controls could be considered abstract classes, as simply by using them, you are inheriting from them. (Choosing "New" from a painter inherits the class for you).

Abstract classes include most visual components in a framework, and are also the typical implementation site for services, as this is the area where you, as the framework, (or library) developer will want to implement the behavior your classes are capable of supporting.

Concrete Classes

As the name suggests, concrete classes are those that are typically more specialized and are not inherited from, but used as is. Logon windows, About boxes, Sort dialogs are examples of concrete classes. Most services also fall into this category, if the service does not serve as the ancestor for more services. E.g. A Sort service is usually a concrete service class, but may be abstracted if you decide to develop the sort service as the aggregate class for DataWindow Sort services, DataStore Sort services, and Array Sort services.

Concrete classes typically form the end (or leaf) of a hierarchy of classes.

Extension Classes

Extension classes are those classes which are added to a library to aid developers in extending the capabilities of that library. Often, the extension class serves as the

insulation layer between the library and the application developer. Extension classes are often needed when the classes developed are tightly coupled, i.e. the extension class is defined by type to its client class. The PFC is an example of a library that utilizes the extension class as both an extension layer and as an insulation layer.

Despite its benefits to developers, the extension class is often seen as detrimental by more advanced developers, mostly because an extension layer approach tends to inhibit the flexibility of a library. In some cases, however, an extension class is a necessary approach to allow maximum flexibility in the use of your library, while preserving the integrity of the library itself. In other cases, it is possible to avoid the use of extension classes through other techniques, some of which are documented in this paper.

Client Classes

The client class is always the class that requests the assistance of a service class in any way. It's for this reason that some types of classes may be both a service and a client if part of its behavior is delegated out to a separate service class. Typically, however, we consider the visual portion of a library to be the client class and the nonvisual element to be the service classes. This is only an assumption however, as there is no restriction on the capabilities, or attributes of either a client, or a service class. The inhibitions are imposed by the framework itself. (If your base services class is inherited from the nonvisualobject, you are in effect imposing the decision that all service classes will be nonvisual). In a later discussion we will look at how a nonvisual class is often thought of as a service, but is in fact, a client.

Service Classes

Service classes are the focal point of SBA. Service classes implement behavior, which, when implemented within a client class, forms that classes *abilities*. Service classes are typically nonvisual in nature, but as stated previously, do not have to be. Some service classes implement other visual components (for example, a Debug service has the ability to display the debug information to the user).

On occasion, services are implemented as a *family* of classes. What this means is that there is more than one class defining a particular behavior, but only one of these classes is implemented in a client class at any given point in time. An example of a family of services is Transaction Connection services whereby a service class can be defined to handle the connection of a transaction object to a particular flavor of DBMS.

Keep in mind the following "rules" for developing service classes.

- Services can contain behavior which is required in multiple unrelated classes:
Example: Sort is required for Datawindows and DataStore.

- Services can contain behavior which is required in multiple instances of the same class: Example: RowSelection in Datawindows. It's not needed in all datawindows class instances, but can be required in more than one instance of the class.
- Services should not maintain information about a client. The client is responsible for it's state and identity. A reference to the client should be passed to the service for direct interaction.

Chapter 6 - Client-Service interaction

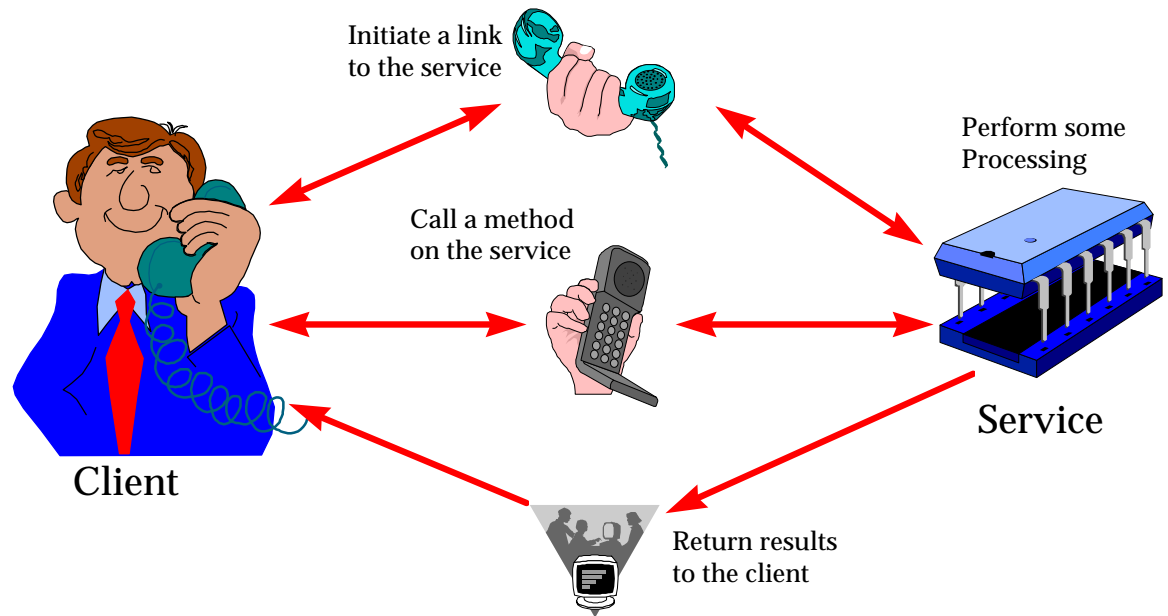


Figure 15 - Client-Service interaction

As seen in figure 15, there are three basic communications between a client and a service. The first is to link/unlink the service to/from the client. The second is to call methods on the service, and the third is to process results from the service.

Section 3 - SBA Specification

Even SBA is evolving. From simple delegated services, manually implemented into client classes, through extensions layers, managed services and brokered services, the architecture evolves. This chapter discusses the potential service styles and scope available today.

Introduction

There are a lot of questions and debates going on surrounding the PFC and the service based architecture it employs. The most often asked question I get is “How should services be created and what are the different implementation methods I can use”. Well, there is no simple answer. There are no formalized standards for developing PowerBuilder service classes.

One of the key aspects about implementing services is in deciding how to implement them. If we look at the PFC, the services are implemented as Instance services, i.e. a pointer to the service is declared as an instance variable within the client object. Furthermore, the services are tightly coupled. This means that the client object contains a pointer to a very specific service object. This is the simplest method of implementing services, but is not very flexible. In addition, even if the services were implemented as loosely coupled services, they still do not allow for varying implementation choices. Let's take a look at the different ways a service can be implemented and then we will discuss how services can be developed that can be implemented in any of these choices.

Key: Structure

This first thing we have to decide is what scope the service must have. There are three basic choices: Restricted to the instance, available to an entire class of objects or available to any object in the application. Figure 16 shows the implementation scope levels.

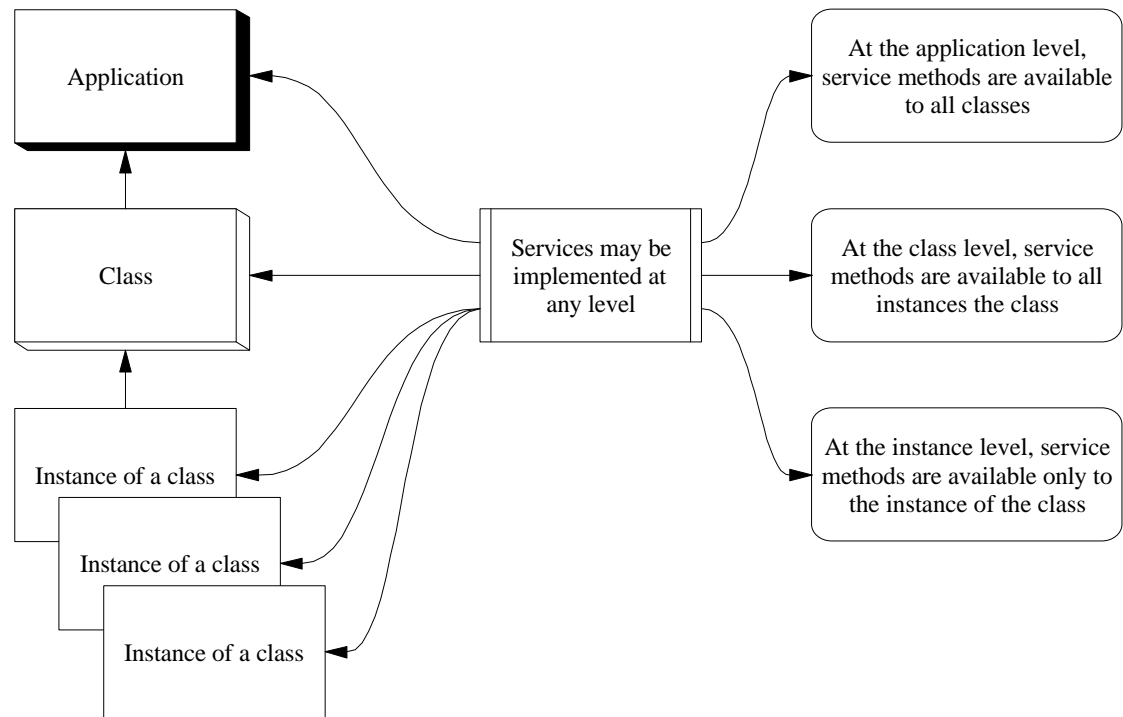


Figure 16 - Implementation scope determines the accessibility of the service methods to the client class

When you look at the PFC in this light, the majority of the PFC services fall into the Instance service category. Some services are implemented at a global level, and therefore, qualify as Application services. Some of the services in the PFC are auto-instantiable and not directly referenced within PFC client objects, and therefore, the implementation scope is up to you.

What drives whether a service class may be used at any level is the level of tight coupling built into the service. For example, if you write a service which maintains information about the client object it is servicing, that service can only be implemented at an instance level. An example is the datawindow linkage service within the PFC. It can only be used for datawindows, and it maintains specific information about the datawindow it is implemented in.

This type of forced implementation does provide benefits. It makes the development of the service less complicated, (introduction of any known parameter simplifies an equation), and it makes it easier to use. (If you want to link two datawindows together, you use the datawindow linkage service). It also has its drawbacks. If you want a window linkage service, it would be developed separately from the datawindow linkage service, even though many methods would be shared.

Any service that maintains information about an instance of a class, forces that service into becoming an instance service. In my opinion, this should be avoided. Objects like

an attribute manager service should not be developed as a service, but as an attribute container. Services, like any other reusable class, should be designed in such a way that they could be implemented at any scope level. For example, A linkage service should be able to link any two objects, a resize service should work with any client object(s). This will make us think more about what code is placed within a service, and make the implementation of the service a choice of the developer, rather than of the class designer. Services should be viewed as “abilities”, in that if a client object needs a certain ability, it requests the use of a Service.

This defines the “where” of service class design, at least in respect to where in the application hierarchy (application, class, instance).

Chapter 7 - The Service Class hierarchy

We discussed the various types of service classes that could be used in a typical Client-Service environment, these being a Local or Instance Service, a Managed Service and a Brokered Service. We also discussed the need for a Service Manager that could easily load these different types of services and make them available to the client class. To recap, let's take a look at figure 17 which shows the hierarchy of the SBA based service class.

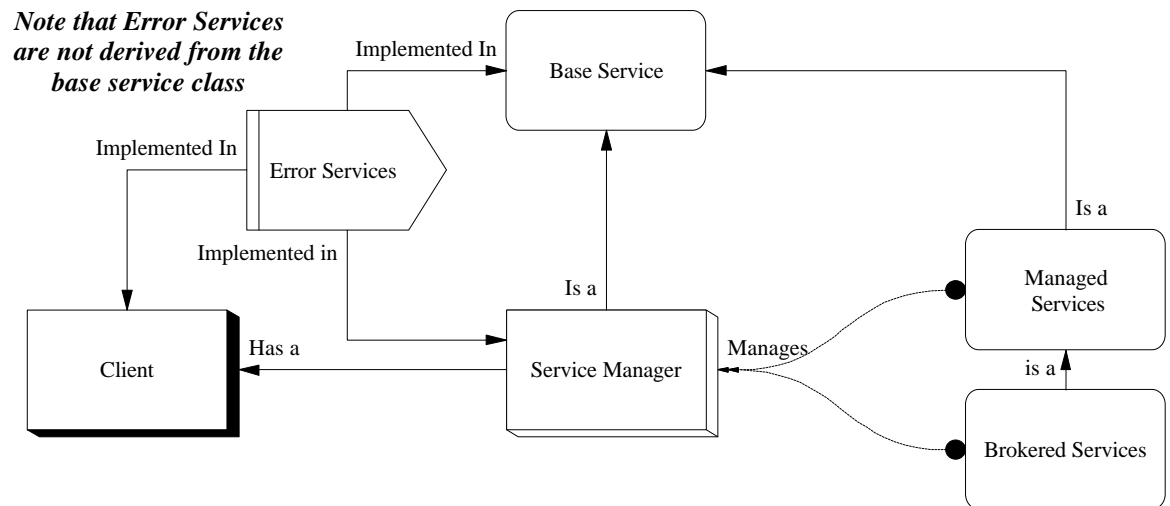


Figure 17 - The Service Class hierarchy

Now, as you can see from this diagram, the client class implements the service manager and an error service that is not managed. The reason for this is that error handling is arguably the most important aspect of any reusable code development, and error handling must be implemented at the highest level possible. Therefore, implementing an error handler in the base service class, and deriving the error handler service from the same base service class would create a recursive implementation that will cause much headache and confusion.

You can also see that the Base Service Class is also the service class that you will use for Local services. Local services are those implemented without management of any kind, relying upon the developer to issue the create/destroy statements, even though those statements may be implemented within a service Load and Unload function.

The Managed service class is the class I would most often use within class libraries such as the PFC and for implementing generic class library behavior. Managed services are not that complicated. They are simply Local Services which are managed by the Service manager, reducing some of the burden from the developer. You can be sure that the Service Manager will destroy all instantiated managed services for you.

Brokered services (and object brokers in general), are quickly becoming more and more popular. Brokered services allow you to completely eliminate any coupling between the client object and the service object. You never have to define a service class pointer in your client object. It is a little more complicated though, and performance has not been proven either way, but if you are planning on doing either distributed processing, or developing true business objects, brokered services and components are the way to go.

Now let's take a closer look at the elements of the SBA hierarchy.

Chapter 8 - Basic Services³

The Basic Service Class does not have much in the way of built in capability. In fact, this class implements your error handler for service classes and not much else. Figure 18 shows the Basic Service Class:

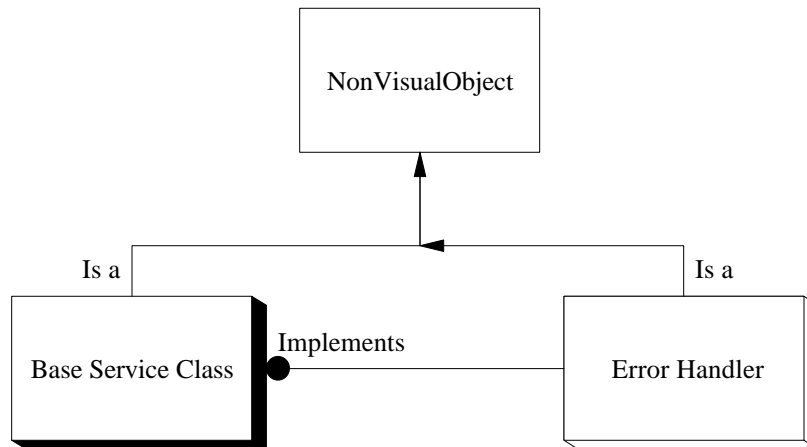


Figure 18 - Base Service Class

Basic Services are those which are intended to be implemented as Instance or Local services within a client object. Even though the Service Manager discussed later can handle loading and unloading of these services, they are not scoped as being application, or even class wide, and are often implemented by the developer without the use of a service manager. Most of the PFC services are implemented in this way.

Note: Error Handling is a critical element in any development effort. Please make sure you go to the time and effort to develop your error handling routines and procedures *before* you develop a service class hierarchy. Also, because there are so many schools of thought regarding error handling, this will not be discussed here, although my version of error handling will be seen in the sample classes included with this paper.

Typed Implementation

Typed implementation is the most common form of Service integration today. The service class is typically defined in the client class and the method of instantiation is targeted at creating the class specified. This often results in tightly coupled service classes.

³ In my implementation, this class is called `SBA_n_SvcBase` and the specification for this class is included as `SBA_n_SvcBase.DOC`

Tight Coupling

This means that the client class cannot exist or function without the service class being accessible, even though it may not be used, because the target service class is defined to the client class at design time. It also means that the service class cannot easily be extended using inheritance. In some cases, the service class being referenced may be an extension class, indicating that the method too extend the service class is by inserting a class between the functional class and the extension class. While this is effective and easy to accomplish, it detracts from the flexibility of a library developed in this manner. For example, if we have a client class which implements a dropdown datawindow search service, which we will call SBA_n_svc_dddwSearch, this class is defined to the client class at design time.

Instance Variable in Client

SBA_n_svc_dddwSearch *invo_dddwSearch*

Service Load Function

invo_dddwSearch = *CREATE SBA_n_svc_dddwSearch*

In some cases, the class referred to is a lower level than the functional class, for example, n_svc_dddwsearch, but that is irrelevant here. The point is that the client explicitly refers to the service class, both in the reference variable declared, and the service load function used to instantiate the service.

This is a typical implementation in many service oriented libraries today, including the PFC.

Drawbacks

The drawbacks to this approach may seem inconsequential until you try to extend the library. The tightly coupled approach results in reduced ability to extend the library without adding complexity. It is also impossible to replace built-in services with your own. For example, if you have a library containing the SBA_n_svc_dddwsearch service, and that service was declared and referenced using the typed implementation method, you cannot simply replace the SBA_n_svc_dddwsearch service with another. You'd have to implement your service using a different reference variable and load procedure. To solve this type of problem, we looked at Managed Services.

Chapter 9 - Managed Services⁴

Managed Services are exactly what they imply. A special service (the Service Manager) is developed which handles the instantiation and destruction of services. The client object will define this service, in addition to the services it already has, but all setup calls are directed to the service manager. The service manager takes care of creating and destroying services for you, and automatically keeps track of all services the client uses, alleviating this code from the developer using the services.

One key aspect to remember when using a managed service is that there is only one copy of the service class and it being used by many clients, so any Instance Variable data should pertain to the service class and not to the client. In fact, this brings up one of the rules of Service Class development in that the service should not maintain information about it's clients. Remaining autonomous has the benefit of being able to adapt to different needs. Did you ever want to implement a datawindow service within a DataStore? Typically you can't do that because the service maintains information about the client datawindow, and knows it is a datawindow. I prefer that the service implement the required method and overload the method if necessary to allow different object class to utilize the service. Figure 19 shows the Managed Service Class.

⁴ The specification for the Managed Service Class SBA_n_svcBase_Managed is included as SBA_n_svcbase_managed.doc

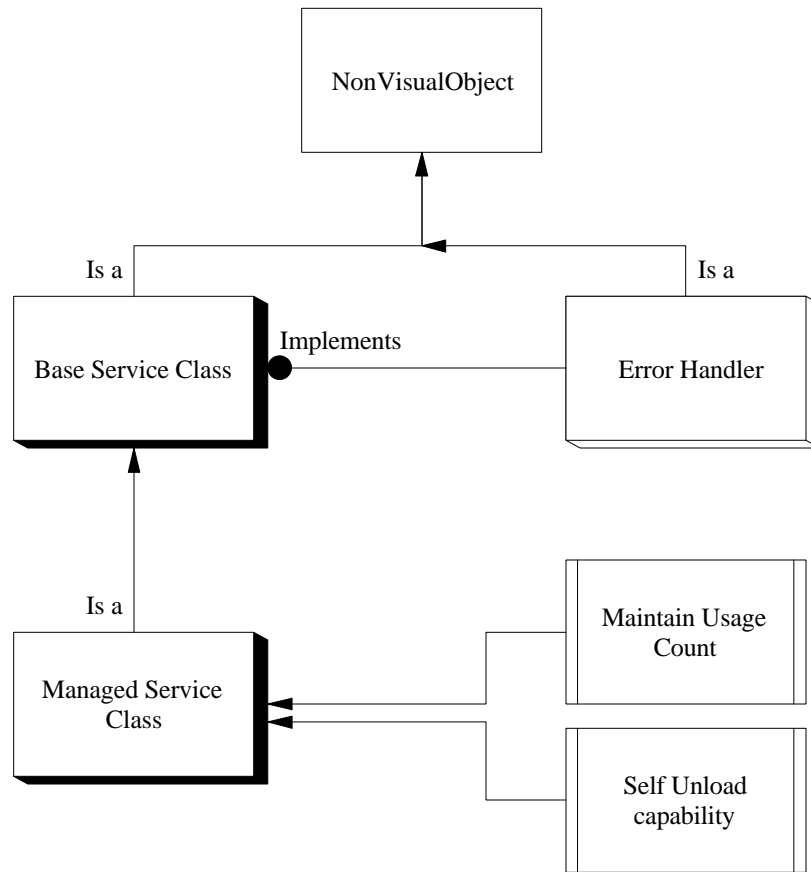


Figure 19 - Managed Service Class

Managed services still require that you declare the service class reference variable in the client object that can use it. In this regard it still seems like the Local Service, but the service manager allows you to specify the actual class to use, regardless of what reference variable you define.

Benefit of Managed Services

To give you an idea of the impact managed services can have, picture this.

In the PFC, there are 13 datawindow services. Each of these services has an instance reference variable assigned, an `of_Setxxxx()` function (containing 14-16 lines of code), a call to `of_setxxx(false)` in the Destructor of the client class to make sure that the service class is removed from memory when the client object goes out of scope, and an extension level class. Each of these categories of code increases when a new service class is added. Extending an existing class stirs a debate about how exactly should you do this, and using your own service instead of the predefined service class requires that you treat your service class as a new service, not a replacement.

In the case of managed services, you still have 13 services, and 14 reference variables, (you need one extra for the Service Manager), but no of_Set() functions, and nothing in the Destructor event. Furthermore, new services can be added by declaring the instance variable reference pointer, and then instructing the developer to call the LoadSvc() method on the Service Manager. Code increase is minimal, and the service manager allows you to define what service class to use in conjunction with a specific service reference variable, allowing greater flexibility in service class extension. To round this off, managed services do not require a pre-built extension layer, eliminating the debate on how to extend the service. If you want to extend it, inherit from it and then change the LoadSvc() call to specify the new service class. It's a very powerful technique that is in fact, not far removed from what we already have. Table 1 depicts this comparison in more measurable terms.

Table 1 - Code Difference between standard service implementation and Managed service implementation

| <i>Implementation Requirement</i> | <i>Standard Service</i> | <i>Managed Service</i> |
|--------------------------------------------------|--------------------------------|-------------------------------|
| Service Manager reference variable needed | No | Yes |
| Service reference variables needed | Yes | Yes |
| Service Load Function needed (of_Setxxx()) | Yes | No |
| Developer Activates Service | Yes | Yes |
| Service Instance created for every usage | Yes | No |
| Service Unload call needed in client destructor | Yes | No |
| Extension layer needed to maximize extensibility | Yes | No |
| Debate on best method to extend services | Yes | No |

Using this method would therefore have the following benefits:

- Eliminate almost 200 lines of code from the client object, and eliminate the need for service load/unload functions for any new services added
- Implement automatic garbage collection (services classes are managed by the service manager, not by the developer)
- Reduction of memory used as services are only loaded once

- Eliminate the need for an extension layer

Usage Count Management

The usage count management comes into play when you utilize one service in more than one place. Much like a DLL keeps track of how many “clients” it has, the managed service gets instantiated only once, regardless of how many service managers are instantiated. (Service Managers are instantiated in each client class instance, but they share service memory). As each service manager requests the service, a check is made to determine whether the service is already loaded. If not, it is created. Once created, the usage count is incremented. When a service manager destroys the service, it first decrements and checks the usage count. Once the usage count reaches zero for a particular service, the service will destroy itself.

Chapter 10 - The Service Class Manager

The Service Manager is a service itself, derived from the Base services class (to automatically implement error handling), and is a class that is defined as Auto-Instantiated, which allows you to drop the service manager into any client object and not worry about the create/destroy sequence. The service managers objective is to manage the services that are used by the client object. Figure 20 shows this strategy in operation.

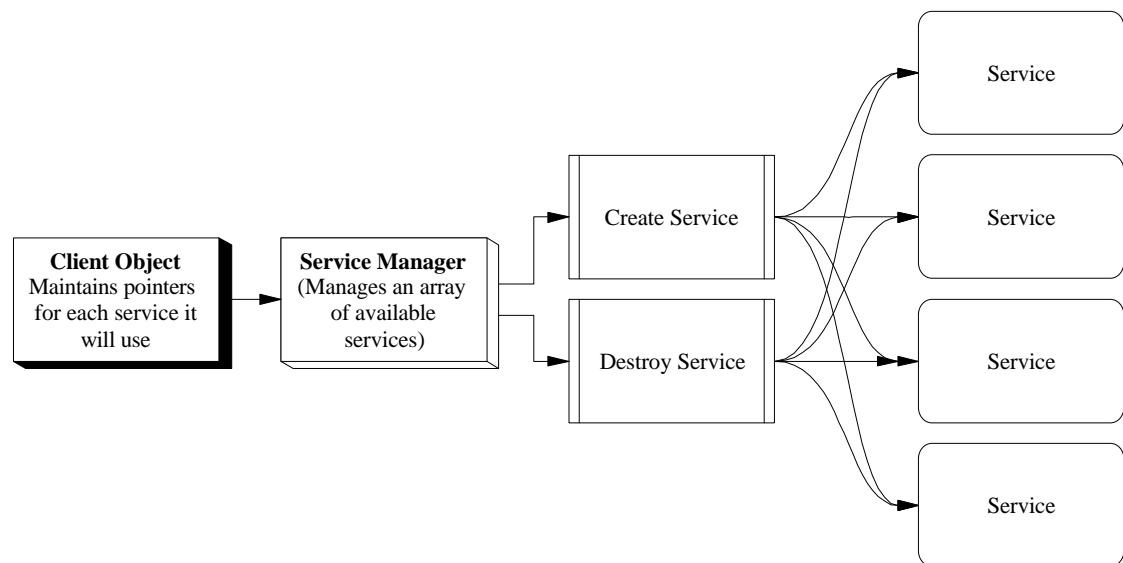


Figure 20 - Managed services are managed by a service manager unique to the instance of the client.

What's the benefit of this?

- The setup code resident in client objects can be moved out to the service manager, creating one place for maintenance of service setup code.
- The service manager itself can be auto-instantiated, allowing the client object to control when and where it gets created and destroyed without developer intervention,
- The service unload code is automatically fired by the service manager, and all services instantiated for the client object are automatically freed, at least for the current instance of the client.
- Implementation of the service within the service manager can be Scope and Style independent.

- Only one instance of a managed service is ever instantiated. Client instances obtain a reference pointer to the instantiated service.

It's a powerful technique which still allows the actual service access to be done through the clients instance pointer to the services.

The end-result is thinner, faster client objects, with less memory used (if some services are shared between classes), and definitely less maintenance.

The Service Load method

The service managers key method is the LoadSvc() function. This function is overloaded to support managed and brokered services. The way that it detects the difference is that a managed service provides both the service pointer and service name, while brokered services provide a "reference" to the service that will be used. (This is usually something more "english" than the class name of an object). The reference is resolved via an internal method/service repository.

Take a look at figure 21 which shows the Service Manager LoadSvc() function in action. You can see that most of the processing is common for managed and brokered services.

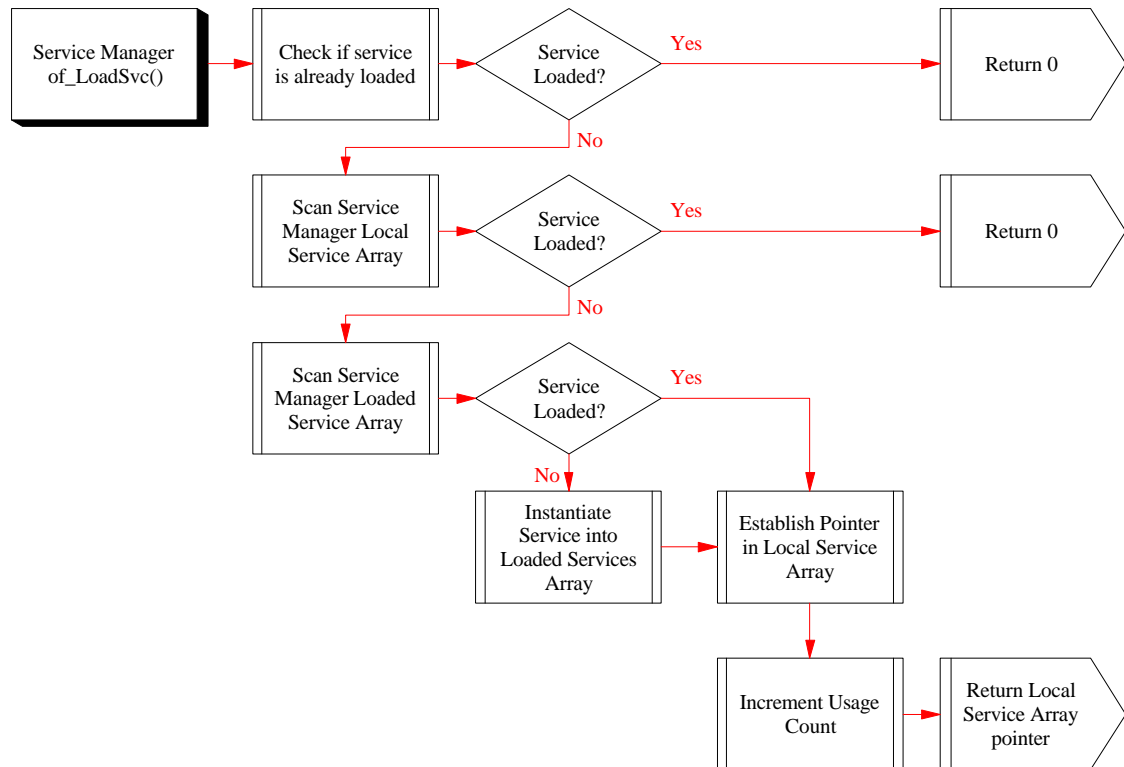


Figure 21 - The Service Manager Load function

The service manager manages service instantiation using two arrays. The first of these is the Loaded Services Array, which is an array containing the actual instantiated service classes. This array is implemented as a shared variable, allowing all service managers access to it, regardless of which class they might be implemented in.

The second array is the In-Use array which maintains a list of services that are in use by a particular client. This array is linked to the instantiated services array to speed up processing.

The service manager will first determine whether or not the service reference variable being passed is already loaded. If it is, the function terminates with a warning that the service reference variable is already populated.

If not, the manager scans the In-Use array to determine if the service requested has already been loaded for this client. If it is, the reference variable is loaded with a reference to the service and is returned to the client.

If not, the manager then scans the Loaded Services array to detect whether or not the service has been instantiated for any other client. If it has been instantiated, the manager loads a reference to the service in the In-Use array and also in the passed service reference variable which is then passed back to the client. If it has not, the manager

physically creates the service in the Loaded Services array, passes the reference to the loaded service back into the In-Use array, and finally back to the client.

The service manager is implemented as an Auto-instantiate class so that it is simple to drop the service manager into any client class and utilize it's power to reduce your overhead and complexity. It stores loaded services into shared memory space and, because all service managers will share the same space, it ensures that the service is only ever loaded once.

The Service Unload method

The other key aspect of the Service Manager is the UnloadSvc() method. The UnloadSvc() method is called under two conditions. When the client class issues a call to unload a particular service, and when the client class goes out of scope.

When the client class chooses to unload the service, the service manager locates the pointer in the in-use services array and calls the service class UnloadSvc() method. This will then attempt to decrement the usage counter, and if the usage count reaches zero, the service class “self-destructs”, or destroys itself, removing the service from the Loaded Services array. Figure 22 shows this operation.

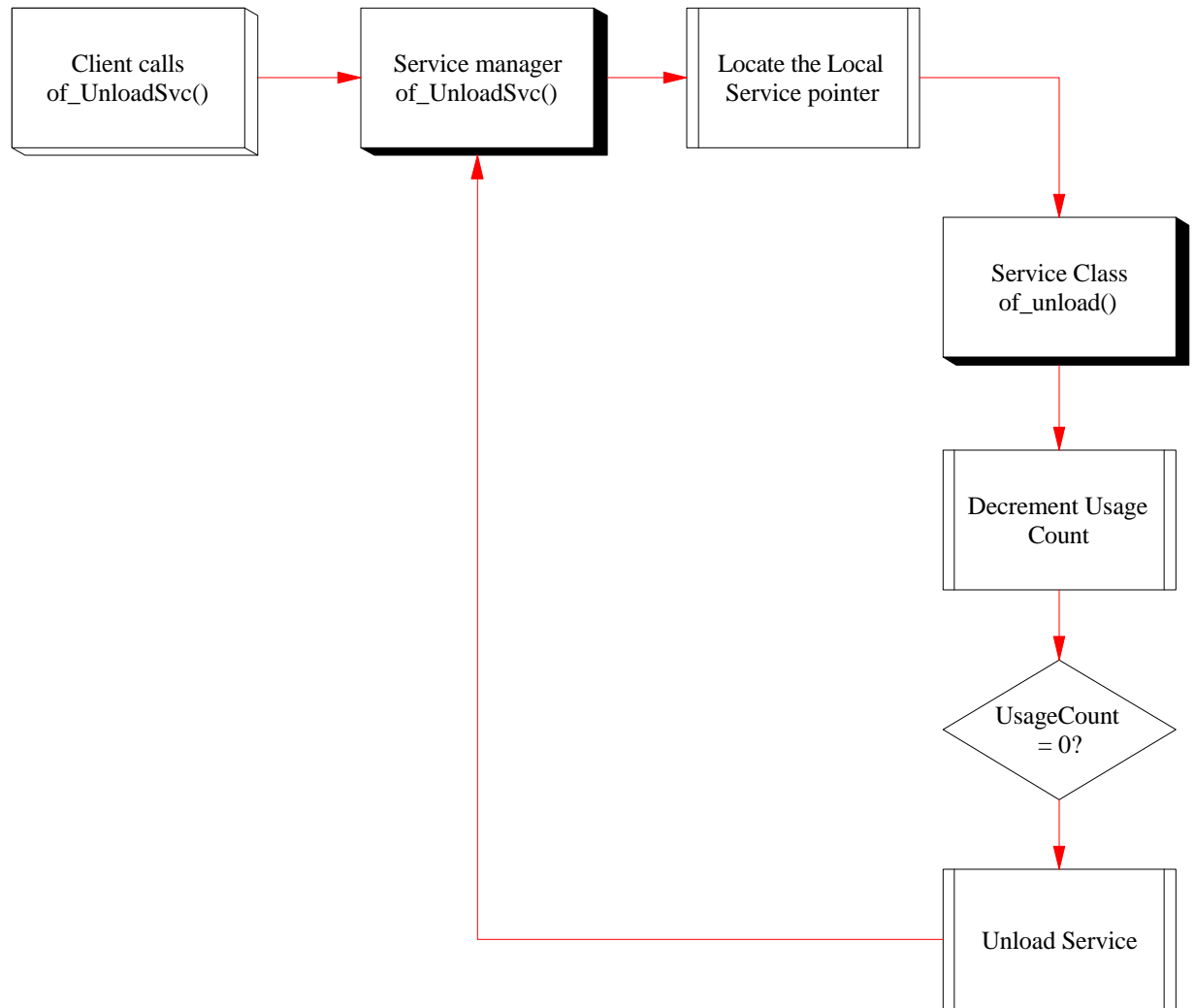


Figure 22 - Service Class unloaded by client

When a client class goes out of scope, it is basically relying on the service manager to unload any services it may have loaded for processing. In this case the service manager, as part of its Destructor method, loops through the array of in-use services and issues the `UnloadSvc()` call to each service that was loaded on behalf of the client. Figure 23 depicts this strategy.

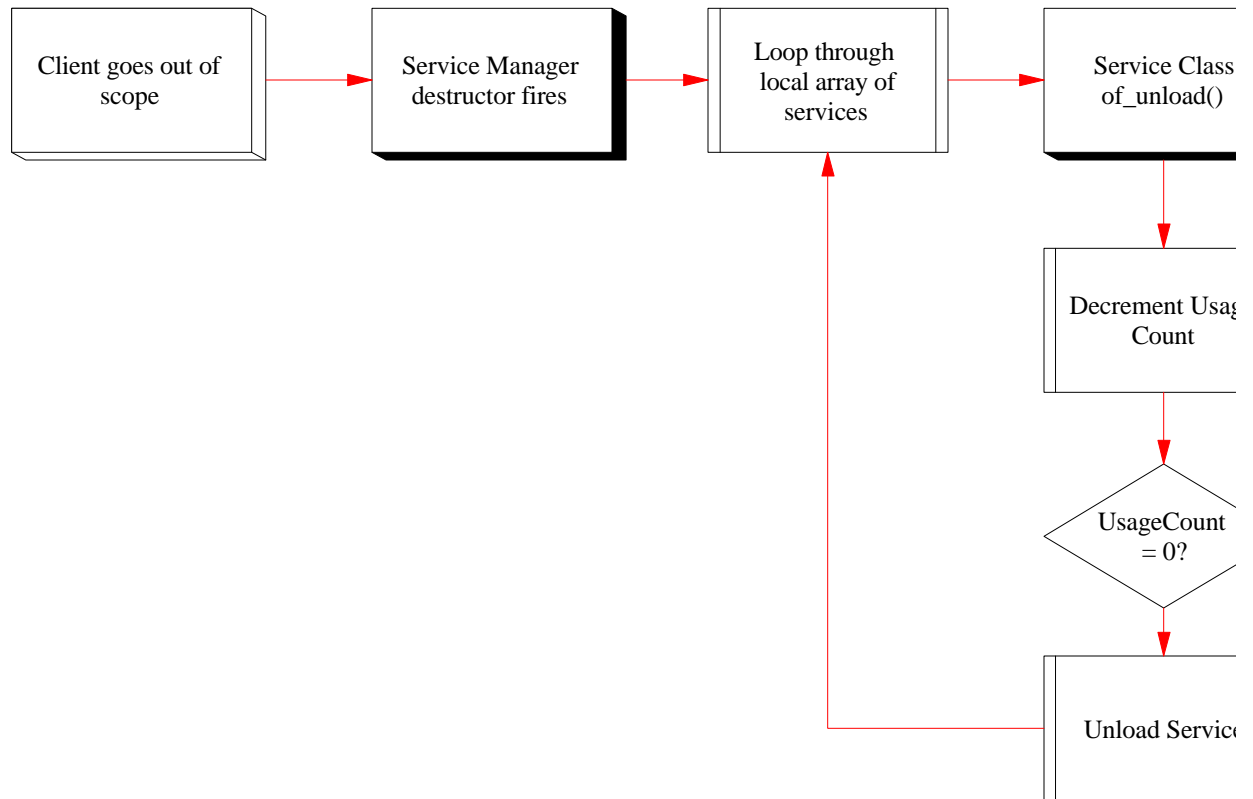


Figure 23 - Services unloaded by the service manager when the client goes out of scope

This ensures that client classes automatically implement garbage collection, and will prevent the dreaded memory leaks often experienced with the use of NVO's.

Managed services are, therefore, an extremely simple way to implement services that are self managing, simplifying the developers job.

Chapter 11 - Brokered Services⁵

The most loosely coupled approach to implementing services is to use a brokered approach. This approach is the most complex of those previously discussed, and we are hampered by certain limitations of the PowerBuilder product, but it can be done, and can be effectively used within the SBA framework.

Basically a brokered service is one which allows a client object to use an Object Request Broker (ORB) to invoke methods on the service. For example, if we coded the clicked event in a client object to call a function on a RowSelect service (not brokered) The Row Select service must be directly accessible to the client and your code would reference the method directly, such as `pSvc_RowSelect(arg_client, arg_row)`.

A brokered service works a little differently. Because the services are never declared to the client, the broker acts as a go-between for the client. The broker would have to know about the service object, and the client would have to be aware of the methods that are available. Methods are referenced through a single reference though, adding further flexibility to the brokered architecture. For example, row selection might be implemented as "Select Row". While the reference is actually quite meaningless, it is used to reference a declared method, which is what the broker does by resolving the reference to an actual method, and then calling that methods passing the desired arguments along. To do this, however, brokered services are required to accomplish several steps along the way.

First, the broker must find out which services will be used, so the client object must declare which services it will use. The first step would obviously be to declare the service broker, which is still a service manager. We do this with an instance variable reference to the service broker, which is an extended service manager and therefore, auto-instantiated.

Client specifies which services will be used

The client must specify which services they will utilize. To for this the client must declare those services which it will make use of. This declaration is done by calling the `LoadSvc()` function on the Service Manager just as any other service declaration, except in this case you pass *only* the name of the service, and not the service reference pointer as shown in the example below.

`pSvc_Broker.of_LoadSvc("Row Selection")`

⁵ Brokered service (SBA_n_svcbase_brokered) specification is included as SBA_n_svcbase_brokered.doc

Service Manager instantiates services and stores service class reference pointers

The service manager will instantiate the service just as it does for any other service, and like managed services, it stores the physical pointer to the service object itself. The difference in this case is that the pointer reference is not passed back to the client.

Service Manager requests that service “export” available service methods

Once instantiated, the service manager issues a request to each service to “export” its public methods. The brokered service class has an additional method defined to do this, as well as a method to allow the service class developer to indicate which of its methods are public and exportable. Figure 24 shows this strategy graphically.

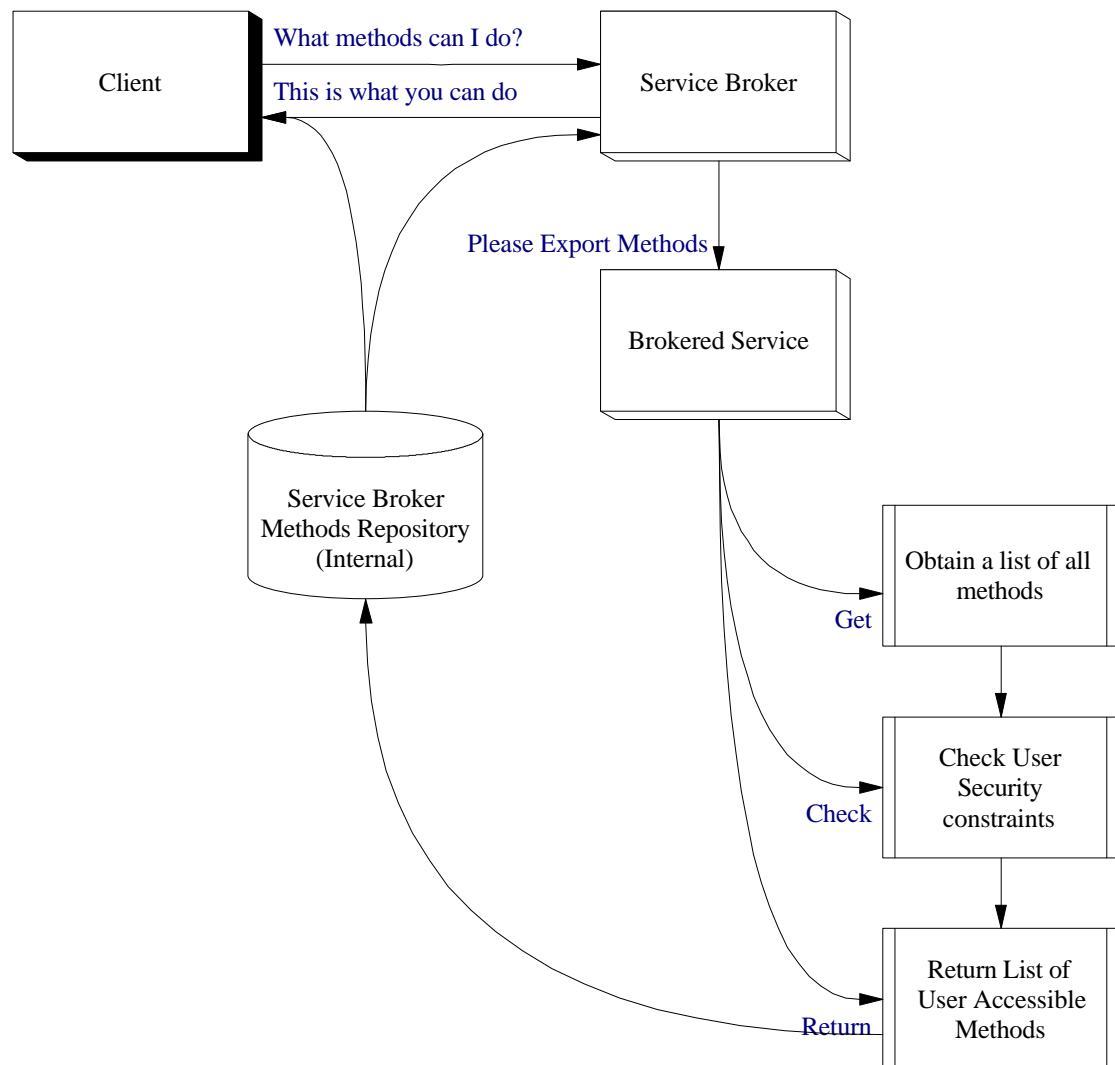


Figure 24 - Brokered Service load strategy

Now, when the client wants to highlight a row of data, it calls the broker and requests the method in question, such as:

```
pSvc_Broker.of_Execute("Highlight Current Row")
```

The broker will now scan it's set of available methods, and, upon detecting that this method is supported by the Row Selection Service, redirect the request to that method on the specified service. Figure 25 shows this strategy.

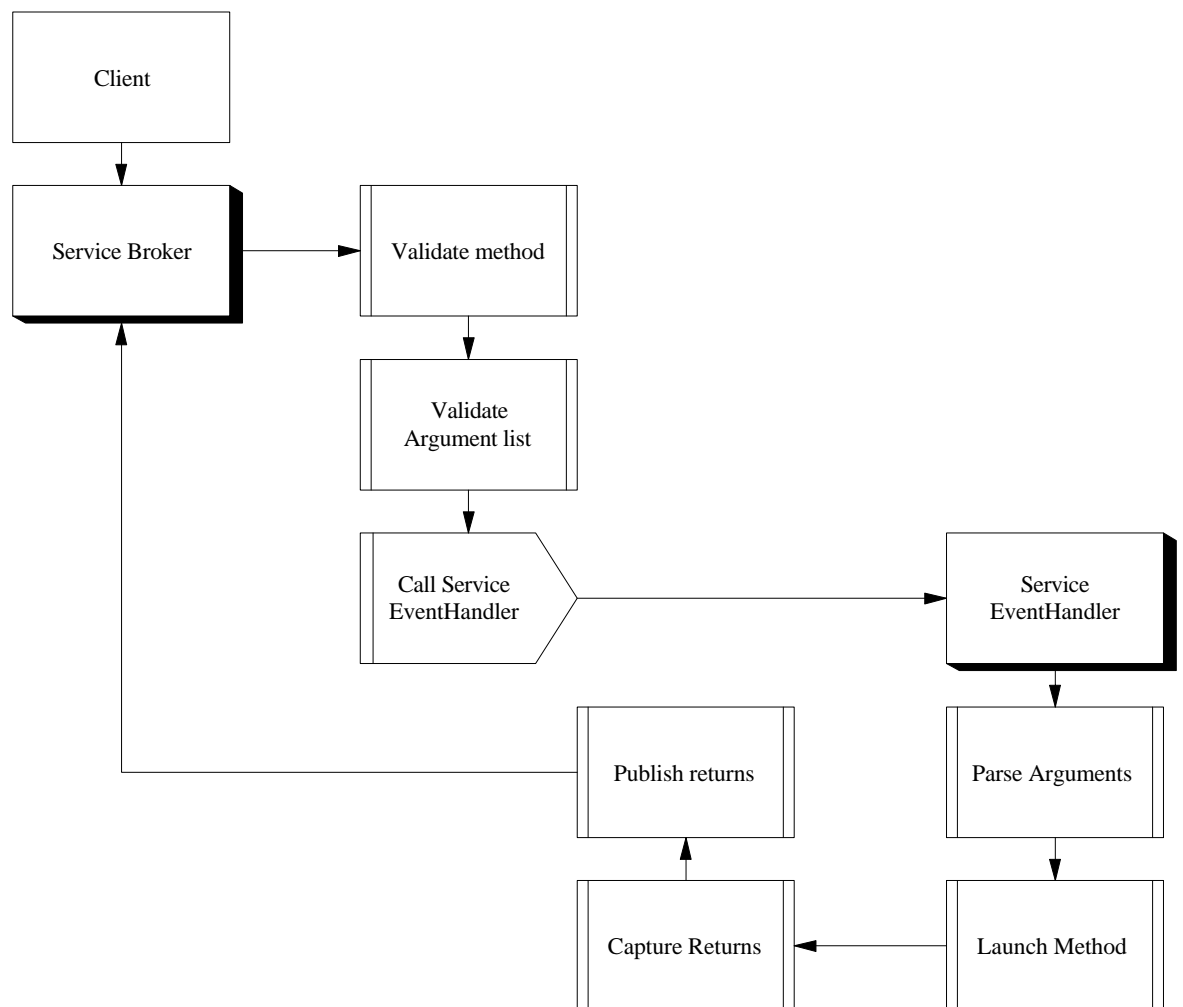


Figure 25 - Broker locates the method and fires the method on the service

This differs somewhat from the CORBA approach in that the services are not permanently defined to the broker when they are created. A CORBA approach would be that when the service is compiled, the Broker is notified and a permanent list of the service, it's location (and other details) and it's methods are registered and stored. Therefore, the Register function would not be required. When the method is requested,

the broker would locate the method, invoke the service if necessary, and then invoke the method on the service.

Exported methods are linked to the service class reference

The exported methods are linked to the service class to which they belong so that the service manager knows which service to invoke when a service method is called. What you will notice now is that the methods on the services are referenced using a method reference instead of an actual method name. For example, in a sort service, the method exported may be called “Perform Ascending Sort” rather than a function or event name such as `of_SortAscending()`. These exported method references are what the developer will now call from the client object. The flexibility in this approach is that the method will always have the same name, even though the method itself may change, or even be moved to a different physical service. In this way, the modifications to service classes do not affect the client object/application other than in the result of the execution of the method. Very flexible, very powerful. The drawback is that you lose a little bit of flexibility in that “methods” cannot be overloaded, at least not in the true sense. Methods can still be polymorphic however, as the service manager (who is the broker in this case), can detect that the client object calling the method could alter which service the method is invoked on. For example, if you had a Sort method residing in two separate services, (one for datawindows, one for DataStores), then when the client object invokes the “Sort” method on the broker, it detects that the client is either a datawindow, or a DataStore, and can invoke the method on the correct service.

| |
|------------------------------------------------------------------------------------------------------------------------------|
| (NOTE: I’m using Sort here as an example. You may choose to not implement basic services such as Sort as brokered services). |
|------------------------------------------------------------------------------------------------------------------------------|

Developer calls a method by utilizing a method reference

So now that you want to call a method, you will notify the Service Manager that you want to “Perform Ascending Sort” and attach any data elements to the request, e.g., in a datawindow, you pass a reference to the datawindow.

The service manager scans it’s Methods repository and locates the “Perform Ascending Sort” method, determines that the method exists on our Sort service, and then calls the Sort Service’s “Perform Ascending Sort” passing the data reference along. The way this is done is through a common interface in the brokered service called the Brokered Service Event Handler. This method on the service will then redirect the processing to the appropriate method in the service after parsing and arranging the arguments being passed. The service executes the necessary code to perform the sort and returns the results back to the service manager who in turn feeds the data back to the client.

Specification: Brokered Service Class

Based on the above description of the brokered service class we can see that we have added at least two built in methods, and allowing the developer to build the actual processing methods. Figure 25 shows the structure of the Brokered Service Class.

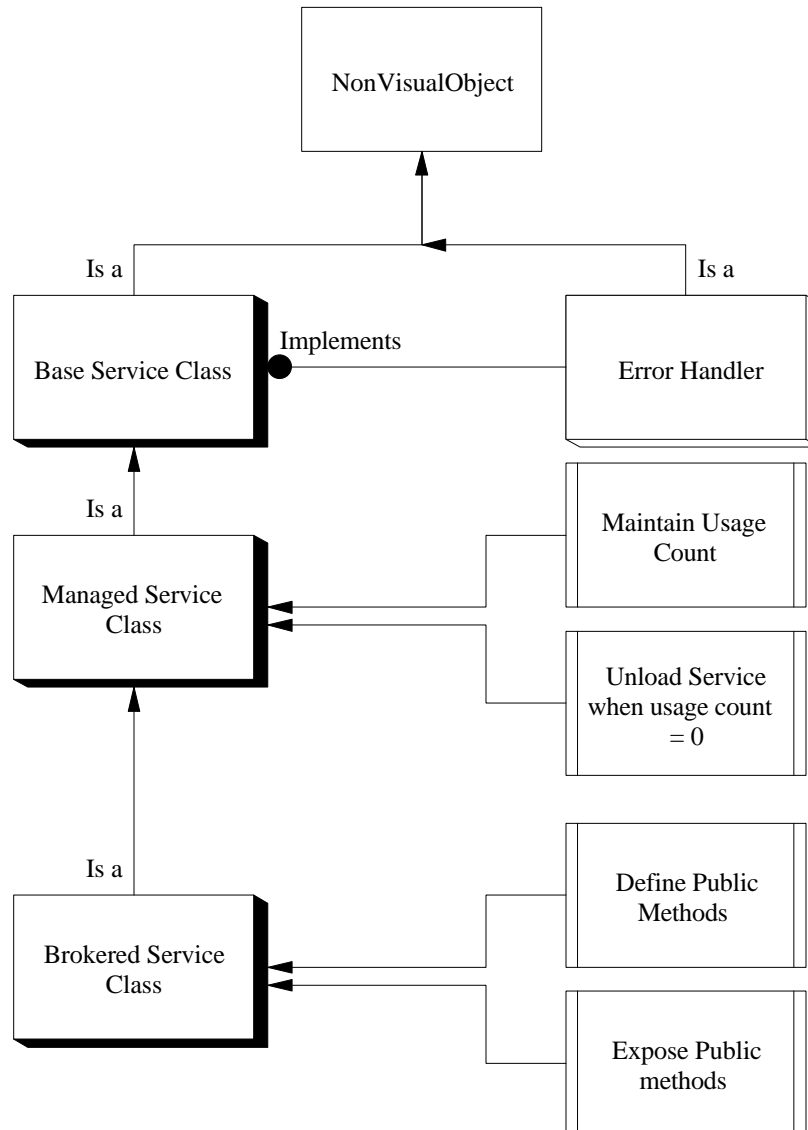


Figure 26 - Brokered Service Class specification

Brokered services also add flexibility in that the services and their methods can be maintained externally and loaded into the broker at run time. For example, a relatively simple repository could maintain the necessary data, thereby allowing you to alter the applications capabilities by manipulating the repository.

We could do this by having the broker service access it's "permanent list" from a database, which we could update via a special "Service Administration" program, which can scan a service object, detect what "methods" are available, (methods, in this case, would have to be coded as events). The Events can then further execute several other events and/or functions, but it would have to have an entry point as we cannot implement pointers to object functions just yet, nor can functions be dynamically built. It sounds complicated, but it really isn't and the benefits are immediately obvious.

- Client need know nothing about the service, not even it's location.
- Client stores no code or pointers to service objects further promoting the possibility of multi-tiered application partitioning.
- Broker handles all service activation/deactivation for the client
- Client can call multiple services with a single method
- New services and methods can be added with no additional code needed in the client or the broker. Simply call the new method.

The drawbacks of this approach are:

- More complicated for developers to learn and use. Use of Service Method reference names would have to be controlled and managed in a team development environment.
- No existing frameworks to support this approach
- Performance is slower than managed services
- Developers lose some control of what is actually being executed, and, based on the "Not Invented Here" syndrome, we can extend this to a new developer phobia, "Not invented here, and it worked yesterday, so it's not my problem" syndrome.

Summary

So now we have some standards to guide our development effort. We've not defined any standards for the contents of the service classes other than what's actually needed for operation. As this architecture evolves, we'll see more standards presented to handle other situations.

Section 4 - SBA Best Practices

*** Coming Soon ***

Chapter 12 - Coding Best Practices

Auto-Instantiate

Constants

Constants are a powerful ally in the world of coding. Constants allow you to assign an english-like name to a value, making it easier for you to remember what you want to use in a specific case, and also, constants make your code much more readable. For example, when you use a `MessageBox`, the value returned from the messagebox can sometimes lead to confusion, especially when reading the code later. If you define the messagebox as have the OK and Cancel buttons, you need to check the returned value to determine if OK (returned value = 1), or Cancel (return value = 2) was pressed.

By itself, this is not a big problem. If, however, in another part of your code you write another messagebox call, but this time you specify the Yes, No and Cancel buttons. Now the return values are a little different. Yes = 1, No = 2 and Cancel = 3! Cancel, when returned from different messagebox() functions has a different value.

Well, we can't change to code used by the `MessageBox()` (well, we can using function overloading, but that's another issue), but we can ensure that we don't fall into the same trap. For example, all of my object functions will return 1 of three values. -1 indicates that the function failed. 0 indicates that something happened which is a warning, but not fatal error. A 1 indicates that the function was successful. (Returned Data is done using Reference Variables, but that's also another article). To ensure consistent use, I created three constants. OK, Warning and Failed and assigned these constants the values 1, 0 and -1 respectively. Now, whenever my code needs to check or use the return value, it can check for the constants OK, Warning and Failed. Now my code is readable, (e.g. `IF CalledFunction() = Const.OK` then continue processing (where `Const` = the name of the object which houses my constants)). I am also assured that if my function wants to return a successful return code, I never have to stop and figure out if 0=success, or if 1=success. `Const.OK` = success regardless of it's value. If I write my own messagebox function, I can also use the `Const.OK` to signify that the OK button was pressed, reiterating that OK = OK and nothing else.

But how and where should you put the constants? There are several ways in which constants can be used. The first, and easiest way, is to simply declare a variable within the object you are working in, and assign the variable the keyword "Constant" which write-protects the variable and you may not change it's value. For example, in any object you can code: *Constant Integer OK=1.*

The method I prefer is to create one (or more) nonvisual objects to house my constants. For example, I have an object which houses all of my Service Class constants, *SBA_n_const_svc*, and another to house windows SDK constants, *SBA_n_const_winsdk*. I separate them to make it easier for me to locate the constant I am looking for. No more than that. As the constants classes are actually never implemented, (the reference to the variable within the object is resolved at compile time), the classes (*SBA_n_const_svc* and *SBA_n_const_winsdk*) are never physically created anywhere, which also alleviates the need for me to make sure they are destroyed afterward. All you have to do is reference the object which houses the constant to make it work. For example, in my *SBA_n_const_svc* class, I declare:

```
CONSTANT String svc_AppScanner="SBA_n_svc_AppScanner"
```

Then, to use this in my code, I write:

```
pSvc_Manager.of_Loadsvc(pSvc_AppScanner, SBA_n_Const_Svc.svc_AppScanner)
```

You do not have to instantiate the object that houses the constants. Referencing a constant this way allows me to change the class used for the App Scanner object by simply changing it in the services constants object. For example, if I change the constant to:

```
Constant String svc_AppScanner = "SBA_n_svc_NewAppScanner"
```

I've effectively changed the object that will be loaded without actually changing any code. Powerful, yet simple.

Constants are an effective way to write code in your application in a manner that makes the code readable, provides a centralized point to store the constants you use, and by virtue of the fact that the objects used for constants are never loaded into memory, their is absolutely NO runtime penalty. It's a win-win-win situation, and should be an important part of your application development effort.

NOTE: There is a problem using a constant as part of a concatenation of multiple values where the constant is the operand to the right and left of the operator. For example:

```
string = string1 + stringconstant //This works  
string = string1 + stringconstant + string2 //This works  
string = string1 + stringconstant + stringconstant //will fail
```

Attributes

Chapter 13 - Using Methods

Get & Set

Overloading

Overriding

Returning Data

Chapter 14 - External Functions

Stubbing

Integrated Platform Services (cross platform capability)

There are a couple of alternatives which provide the same end-result This is something I have been toying with for some time now, and I've come to the conclusion that PowerBuilder needs to take one more step and allow conditional compilation. I've heard rumors that it's being looked at, but cannot confirm whether or not it is being done.

First let's take a look at how the PFC (and many of us) have handled this predicament in the past. First we create a nonvisual object which will be the reference point for accessing the functions (nvo_base_api). Next we create two descendant objects, one to house 16-bit calls (nvo_winapi16), and one to house 32-bit calls (nvo_winapi32). We'll take a simple enough function as an example, GetWindowsDirectory

Function uint GetWindowsDirectory (ref string dirttext, uint textlen) library "KERNEL.DLL"

This function returns the directory path to the Windows directory in the variable passed as "dirttext". A simple enough function. We declare this as a local external function in our 16-bit userobject, nvo_winapi16. To be polymorphic, we also write a user function in our 16-bit userobject (call it of_GetWindowsDirectory), which will return the directory to us. We do this because the 32-bit function call is actually a little different from the 16-bit function in that it has an "A" appended to the end of the function name. The code snippet below shows this function declaration housed as a local external function in the 32-bit userobject.

Function uint GetWindowsDirectoryA (ref string dirttext, uint textlen) library "KERNEL32.DLL"

Again, we write a userobject function in the 32-bit userobject to call the GetWindowsDirectoryA function. We name the function the same as the 16-bit counterpart, of_GetWindowsDirectory. So now, depending on which object is instantiated, our application will always call of_GetWindowsDirectory().

In our application, we do not know whether to implement the 16-bit or 32-bit version until the application is running, so we declare a variable pointer of type nvo_base_api, which we name as inv_WinAPI. We write some code to detect the platform we are running on, (hopefully, we will do this right in the beginning of the application and store the information somewhere for later use, but for this article, we'll write the code out in full). The code below detects the environment and stored two pieces of information. The first is the operating platform, (Windows, Unix, Macintosh). The second is the platform operating system level (16-bit, 32-bit).

```
/* Check the environment */  
  
Environment Env  
GetEnvironment(ENV)  
  
Choose Case ENV.OSType  
    Case WindowsNT!, Windows!  
        is_OS = "Win"  
        IF ENV.Win16 THEN  
            ii_platform = 16  
        Else  
            ii_platform = 32  
        End if  
  
    Case Solaris!  
        Is_OS = "Unix"  
  
    Case Macintosh!  
        Is_OS = "Mac"  
  
End Choose
```

These variables are utilized

Now we can call the GetWindowsDirectory function from our application as inv_WinAPI.Dynamic of_GetWindowsDirectory() and be assured that we will access the correct function. To be able to call the function as non-dynamic, (for performance reasons perhaps), we would add a "stub" function of_GetWindowsDirectory() to the nvo_base_api object. When we instantiate the correct object, it's function (by virtue of overriding the stub function) will be called. All we have is three userobjects which control the API functions and three userobject functions.

Integrated Platform approach

Recently, I tried an alternative approach. I created the same hierarchy of userobjects, nvo_base_api, nvo_WinAPI16 and nvo_WinAPI32. I coded the same local external function declaration in the 16-bit userobject, namely:


```
Function uint GetWindowsDirectory (ref string dirtex, uint textlen) library "KERNEL.DLL"
```

In my 32-bit object, I coded the local external function declaration:

```
Function uint GetWindowsDirectory (ref string dirtex, uint textlen) library "KERNEL32.DLL"  
ALIAS FOR "GetWindowsDirectoryA"
```

The addition of the ALIAS keyword allows me to define the function with the same name as the 16-bit function, but actually refer to it using the GetWindowsDirectory() function call.

Make all of these objects AutoInstantiated (by turning AutoInstantiate on in the Base object. Now I'm ready to write the User API. To do this, I created a service object nvo_svc_sysapi in which I declare two instance variables:

```
nvo_winapi16    inv_api16  
nvo_winapi32    inv_api32
```

In the constructor event of this object, I placed the code to detect the platform I'm running on, and store the information in another instance variable ii_Platform.

I now code one userobject function in this object which looks like this:

```
Choose Case ii_Platform  
    Case 16  
        RETURN inv_API16.GetWindowsDirectory()  
    Case 32  
        RETURN inv_API32.GetWindowsDirectory()  
End Choose
```

Here, the decision regarding the platform is being made at the time the function is being called. AutoInstantiation ensures that the API objects are automatically created and destroyed for me.

I end up with 4 userobjects, one of which is empty (nvo_base_api), 2 which contain ONLY local function declarations (nvo_winapi16 and nvo_winapi32), and one userobject which issues the call to the external functions. I also end up with only 1 user function to maintain.

My goal is that one day we will get conditional compilation/execution which will allow me to declare the instance variables

```
Condition: 16bit: nvo_winapi16    inv_api  
Condition: 32bit: nvo_winapi32    inv_api
```

and be able to simply call the inv_api.GetWindowsDirectory().

Other Alternatives

The other alternative is one which will work based on your knowledge of the final system destination, and that the destination will not change. Create 2 userobjects called nvo_winapi, place them in separate libraries (lib16.pbl, lib32.pbl), and simply include the correct library in your search path when you compile the application. The more daring could also try dynamically setting the library list at runtime using SetLibraryList(), and include both the 16-bit PBD/DLL and 32-bit PBD/DLL with the executable, although I do not recommend this unless you are completely familiar with the SetLibraryList() and it's consequences.

Chapter 15 - Extending classes in a Service Based Framework

Section 5 - SBA Techniques

All too often, when we approach the development of services, we adopt the dive-in-and-see-what-happens philosophy, prevalent in the RAD approaches of the day. This can often be a, if not fatal, less than desirable approach as it leads to implementations that have not been thought through and designed correctly. This section will walk through the design and development of a service class and hopefully alert you to some of the elements that will drive successful Service-Based development. This is followed by a discussion on service vs. client detection, where the example is extending an existing object. Other examples will follow.

Chapter 16 – Building and implementing a service

Chapter 17 - The Application-Scanner service

One of the first needs of an application is the ability to detect if the application is already running. This example walks through a design for this ability to be added and function on any platform

One of the first things you will come up against in the PFC is to extend the PFC by adding a new service class. The typical approach is to copy one of the existing PFC service classes, remove the logic, and then build your service class using the shell. While this practice helped keep your class looking and feeling similar to the existing PFC classes, (a good thing), it also locks you into the PFC approach (perhaps not such a good thing). This means picking up both the good and bad features of the PFC architecture. Well, perhaps we can change all of that, and we're going to try by using a real-life example to do it.

Another element of service class design is what exactly the service is going to do. Again, the typical approach is to design the service to do what you specify it should do. Sometimes, a little extra thought and effort can raise the value of your service class. So let's take a look at what it is we want to accomplish, and then we'll figure out how we will get there.

Setting the Goal

If you're designing a service class, you should have a goal. When you have a goal, see if that will help others. Remember that a service class is intended to be useful to other classes, or applications, so follow the SBA design principles to ensure that you set the goals appropriately.

Our goal for this exercise is relatively simple. We want to detect whether or not our application is already running. It sounds simple enough, so the first thing we do is develop a Goal Statement.


Goal Statement

Develop a service which will detect whether or not the application is already running

Simple enough. This is a service that many have asked about since we entered the 32-bit world. Why? In the 16-bit world, we could use the statement

IF Handle(Application) > 0 THEN “application is already running”

This no longer works in the 32-bit world because each application on the 32-bit operating system would get it's own address space, and not be aware of other applications outside of it's memory space. This “protected” operating system made functions such as handle(application) obsolete. So, we figured the only other constant to applications in both the 16-bit and 32-bit worlds was the windows API.

 This is true even under the UNIX version of PowerBuilder, but the author is not familiar with Macintosh to make the same claim there.

The Design

We design the service class showing three important aspects. Client Class integration, Client Class interaction and Service Class API. The design does not have to be detailed, just informative.

When I do design work, the first thing I do is try to figure out how the whole thing is going to fit together. I use a flowchart to keep this all visual (which is more helpful for me), and in doing so, I often uncover a few aspects I would not ordinarily cover. I also try to ensure that I show all three Service Class Design requirements, Client Class

integration, Client Class interaction, and Service Class API. Figure 26 shows what this service class might look like.

Service to check if application is already running

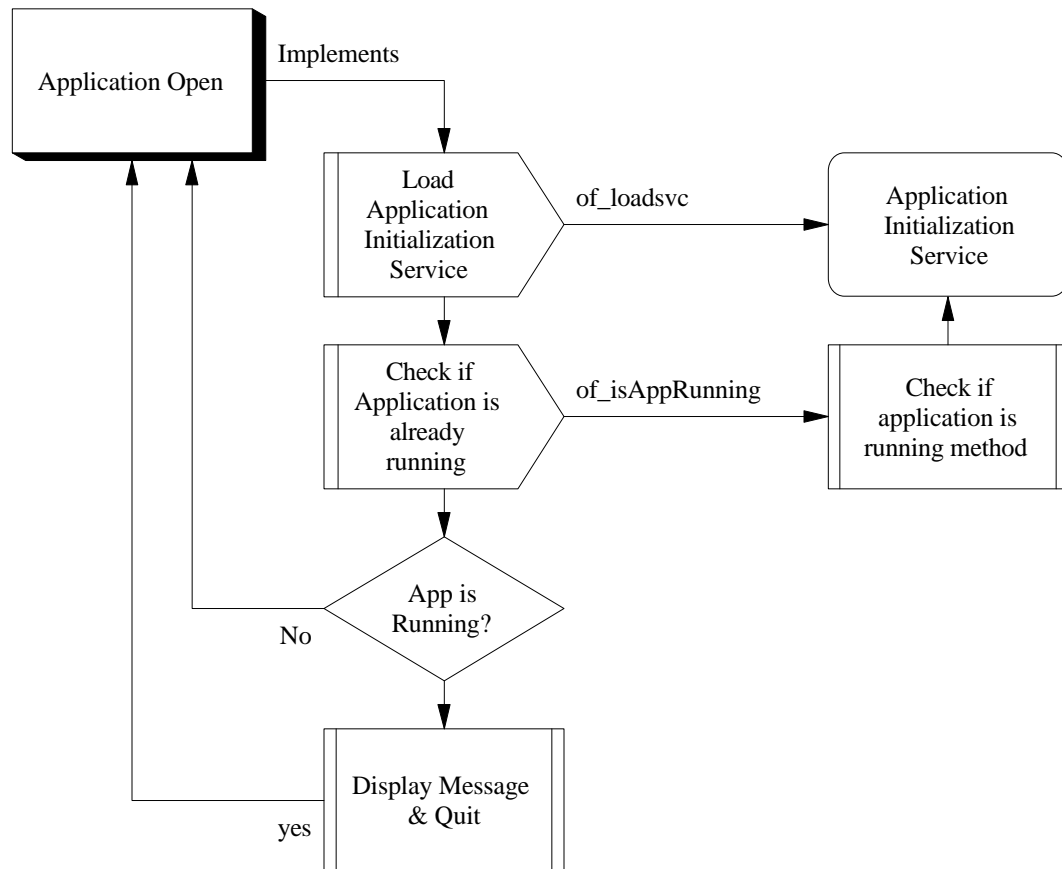


Figure 27 - Service Class Design for the "Check if application is running" service

Of course, this is only a skeleton view of the service. We still need to expand the method which will actually look to see if the application is running. The windows API offers a few alternatives to look at the applications that are running. One of the easiest to use is the FindWindow() function. FindWindow() will allow you to look for an application window by specifying either the window caption, (that is the title that appears in the applications main window title bar and may include the text for a subservient MDI child window), or the class name of the main window, or both. The problem I have with this is that it must be an exact match of the name you supply, and the name that appears on the top of the window you are looking for. 90% of the time, that would be enough.

Another method is to execute a DDE call to the application in question. If the call succeeds, the application is there. If not, it is not. Again, this will work about 90% of the time.

My preferred method uses another trio of windows API calls. GetDesktopWindow(), GetWindow() and GetWindowText(). GetDesktopWindow() accepts no arguments and simply returns that handle to the desktop window, the highest level window in a Windows, Windows 95 or Windows NT environment. By calling GetWindow() and passing arguments of 'DesktopWindowHandle' and 'Constant gw_child', you will get the handle of a Main window for an application.

GetWindowText('MainWindowHandle') will get the text from the window title and allow you to use this in determining if this is the window you are looking for.

Subsequent calls to GetWindow('MainWindowHandle', 'gw_next') will cycle through the main windows that are served by the desktop window and you can scan this until you find what you are looking for. Figure 27 shows the hierarchy of window handles in a typical windows environment and the function you use to interrogate it.

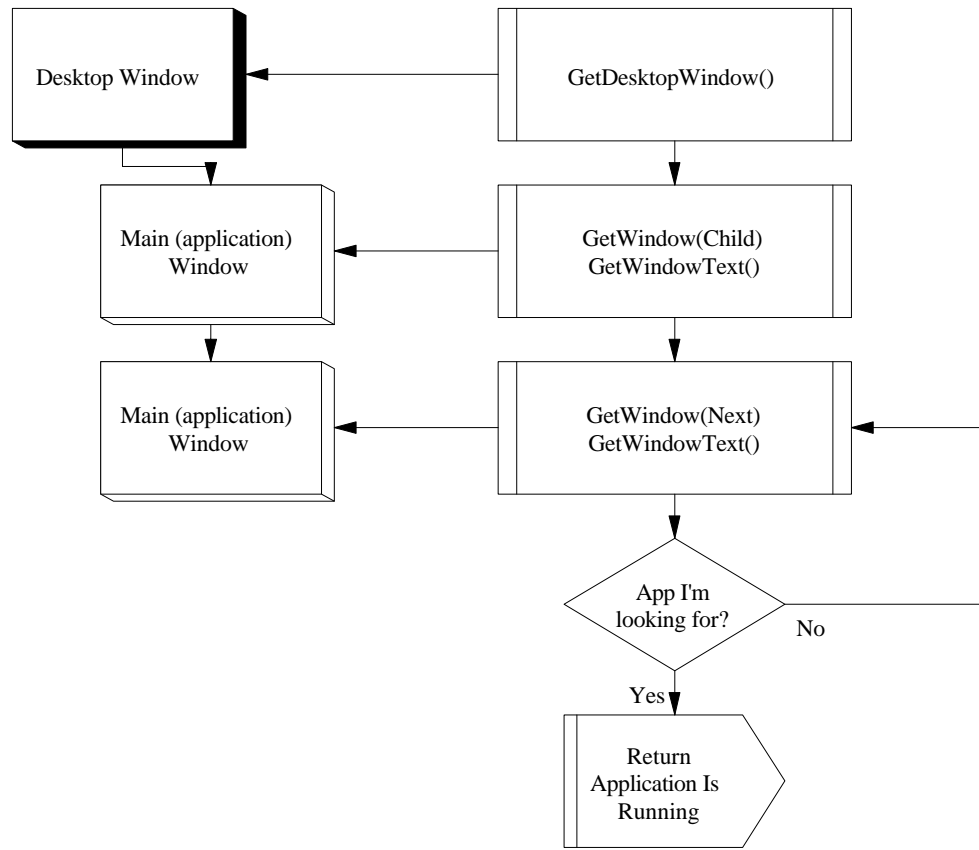


Figure 28 - API Calls to traverse the window hierarchy

Most of us, (including me the first time or seven), will stop here, design the service to accommodate this request, implement the service in our favorite framework, and be done with it. It's unfortunate, because it's very easy to extend this service to be much more powerful.

Extending the Design

When reviewing your design, ask yourself the questions: Is this useful to many applications and what other capability would be useful?

When you ask these questions, we find that we can add to our design. When I revisited this design, I found that I could expand it with the following needs:

- It would be useful if we could check if applications other than our own were running using this same code.

- It would be useful to know how many other occurrences of an application were running.
- It would really be useful if the service could detect if a specific file were open within an application, for example, an Excel spreadsheet or Word document.

With this in mind, we expanded the design from figure 26 to look like that shown in figure 28.

Service to check if application is already running, count # of instances running and detect specified files within specified application

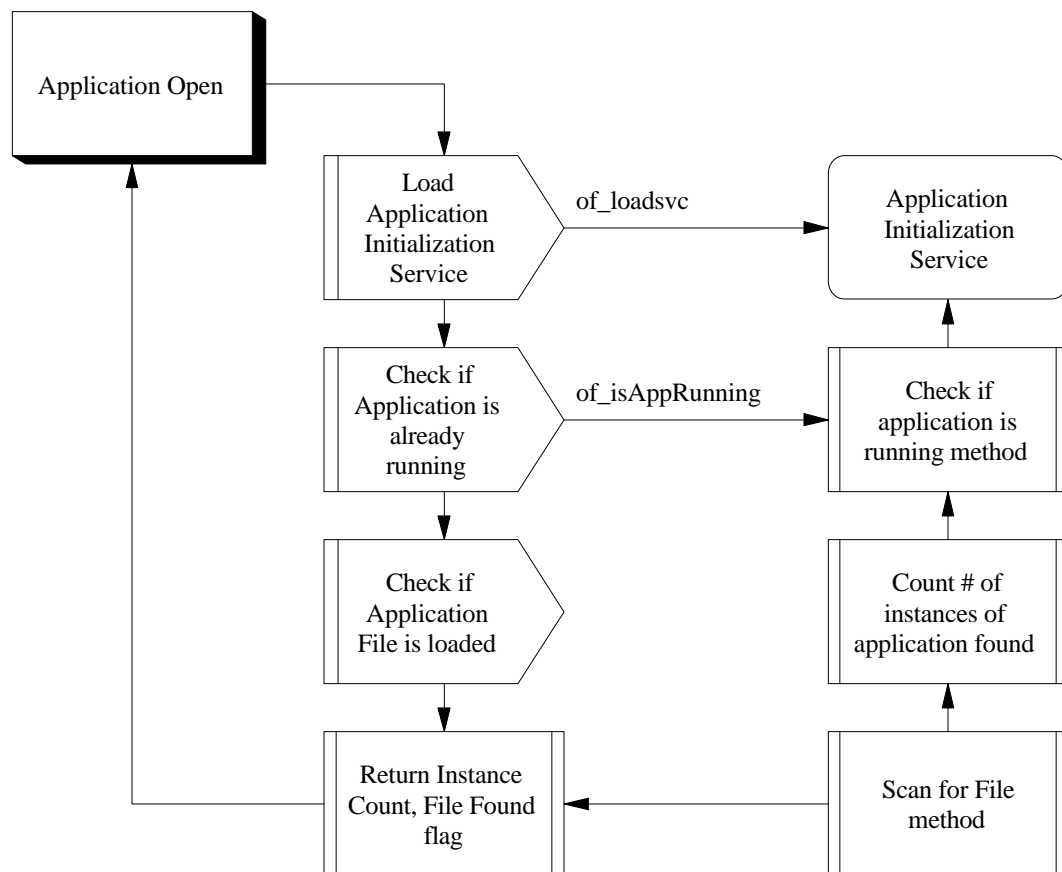


Figure 29 - Expanding the application instance check to detect number of instances and files within applications.

We can do this without any additional API calls too. Once we detect the application, we accumulate an instance count instead of immediately returning. If a File name is also specified, then we enter an additional seek loop to scan through the window hierarchy

of the application to look for the file specified. Figure 29 now shows a more complete design.

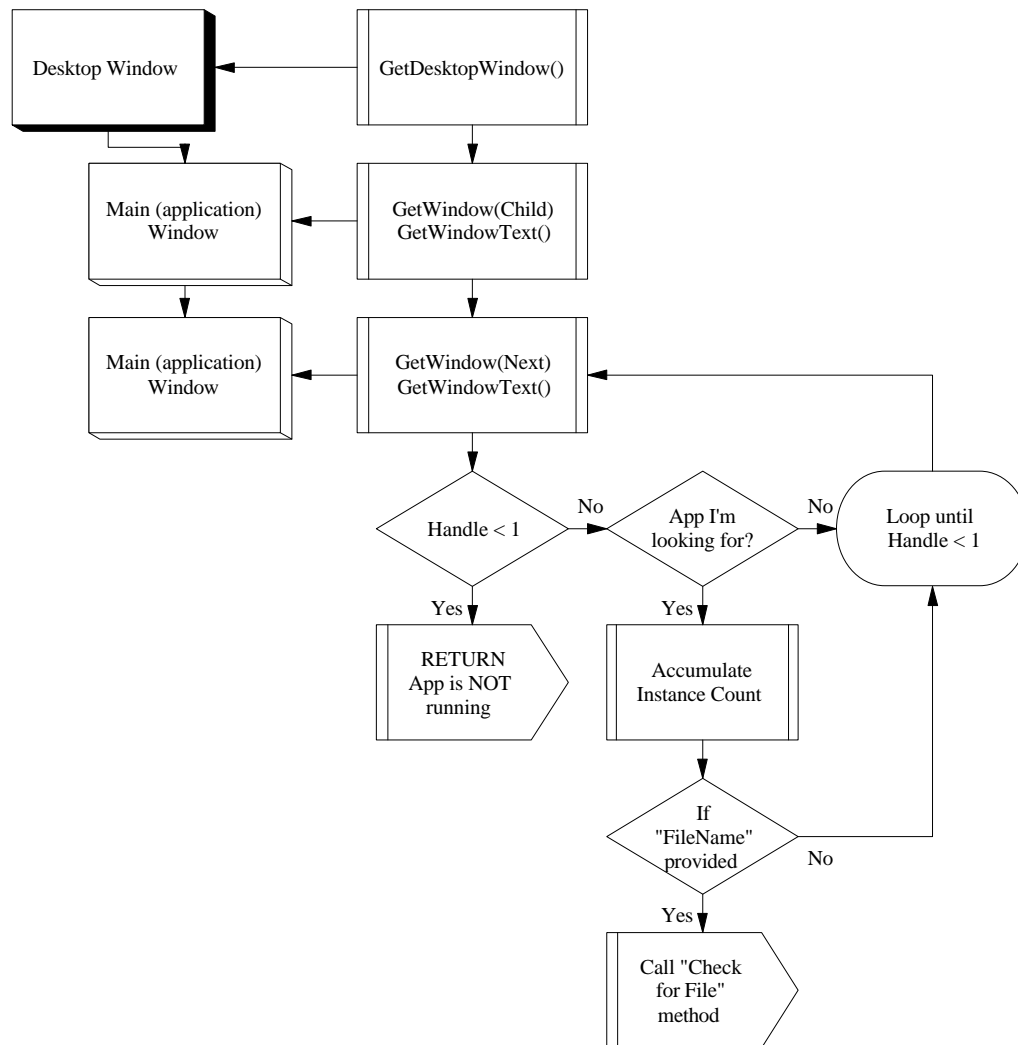


Figure 30 - Application Instance Check service complete

Now we can see that if a filename is provided, (Example: of_isAppRunning(“Excel”, “MySpreadsheet.xls”), then the “Check for File” method would be called to interrogate the child windows within the application to see if the file was loaded. This method is described in figure 30.

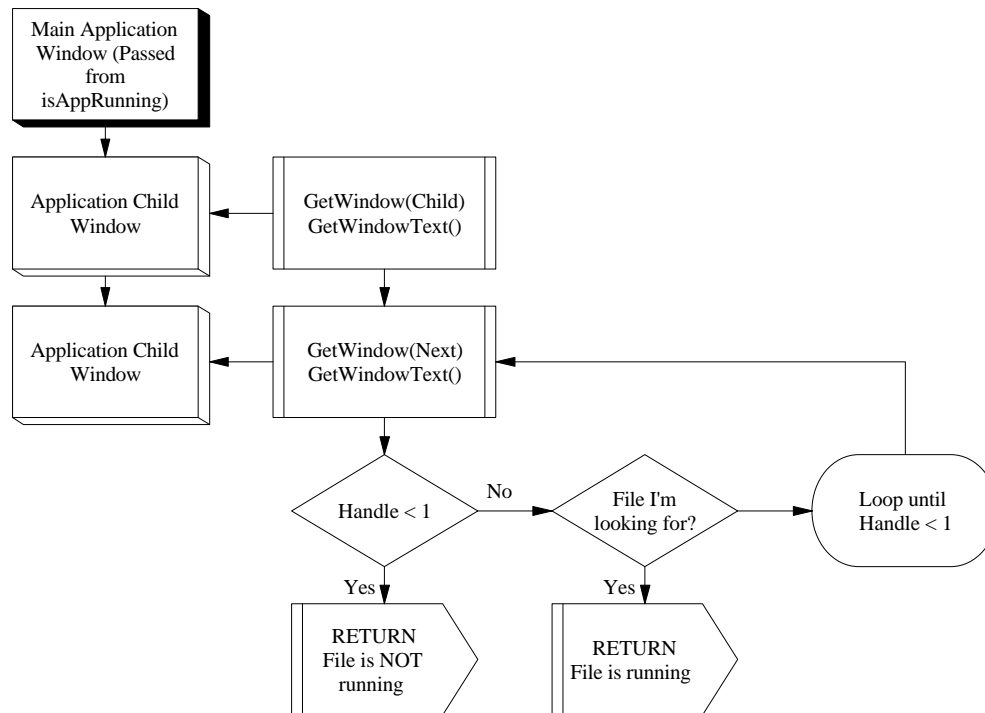


Figure 31 - Extension of the Check Application Running method - Check if File loaded

Of course, this implies that our entry point will be an overloaded function call. This brings up the point, how exactly should you develop an overloaded function call. I follow a simple set of rules.

- Define the main function with the maximum number of arguments. Place all your main code in this function.
- Overload the function with other argument combinations, filling in the missing pieces. (This is NOT polymorphism, by the way, simply function overloading).

My function prototypes would then look like the following:

```

Long FUNCTION of _isAppRunning(string Appname, string filename, ref
AppInstanceCount, integer arg_Style)
Long FUNCTION of _isAppRunning(string AppName, ref AppInstanceCount, ,
integer arg_Style)
Long FUNCTION of _isAppRunning(Application AppHandle, ref AppInstanceCount, ,
integer arg_Style)
  
```

This allows me to call the function passing either the application handle (THIS in the application object), an Application name ("My Application"), or an Application name and File name ("Excel", "Spreadsheet1.xls"). The second two functions simply call the first one without a filename, and in the case of the third, obtains the application name

for you. The `arg_style` argument is a style indicating what the service should do once it detects any of the specified conditions, so that the processing can be imbedded within the service class. For example, we might want to “Warn the user, but continue loading”, or “Warn the user and quit loading and swap to the already running application”. The style variable is a combination of individual instructions to instruct the service how to react.

Building the Objects

As much care as an architect takes to design something, the builder should be equally aware of their responsibility to the completed product.

Building a service class is a relatively painless process once you have a good design specification. Typically, I would take the design discussed earlier to the next level and formally spec out the class, but for the purpose of this article, we’ll make the assumption that this has been done, and move on to building the class. Based on the standards proposed in the previous chapter, we have three levels of service classes, Instance or Local service class, Managed service class and Brokered service class. Figure 31 illustrates the SBA standard hierarchy.

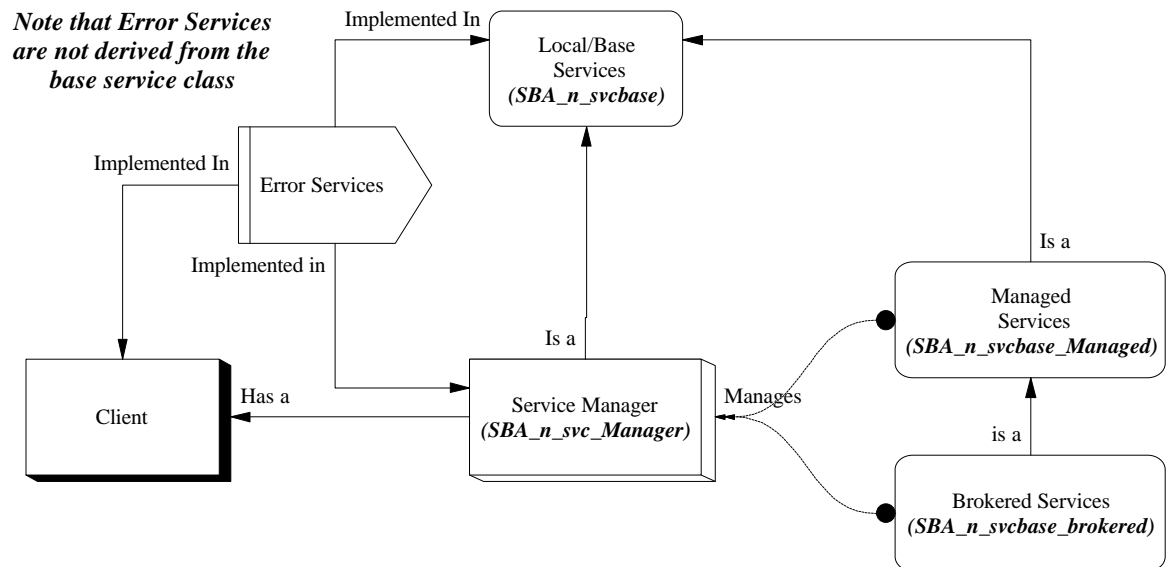


Figure 32 - SBA Service Class Hierarchy and implementation

Based on the description of these classes, I chose Managed Service as my implementation choice. Why? I usually try to avoid Local services as much as possible as I prefer to have the services managed for me automatically, and brokered services

seem overkill for this implementation. In order to utilize the Managed Service, I need to implement the SBA Service Manager.

So now, in order to build the service, I will create a new service class inherited from my Managed Service class, and code the methods that I have designed⁶. I decided to call the service class “Application Scanner” as it can now look for any application.

The AppScanner class (SBA_n_svc_AppScanner)

We’ve already defined the methods we want on this object, namely an overloaded of_isAppRunning() function and an of_isFileLoaded() function. What we still need to do is to define the external functions that will be used. I do this using a technique I call Integrated Platform Services, which is a technique that differs slightly from the standard method of implementing platform independent services. The method is discussed in detail in chapter 5, so I will simply state that I created a Windows API service class, (SBA_n_Svc_Winapi), which in turn implements the platform specific external function declaration containers, SBA_n_api_win16 and SBA_n_api_win32. In these objects we coded the external declarations as shown below.

SBA_n_api_win16

```
FUNCTION uint GetDesktopWindow() LIBRARY "user.exe"  
FUNCTION uint GetWindow(uint hwnd, int wflags) LIBRARY "user.exe"  
FUNCTION uint GetWindowText(uint hwnd, ref string wintext, int textlength)  
LIBRARY "user.exe"
```

SBA_n_api_win32

```
FUNCTION uint GetDesktopWindow() LIBRARY "user32"  
FUNCTION uint GetWindow(uint hwnd, int wflags) LIBRARY "user32"  
FUNCTION uint GetWindowText(uint hwnd, ref string wintext, int textlength)  
LIBRARY "user32" Alias for "GetWindowTextA"
```

Notice that the GetWindowtext() function uses an alias. This function is actually already coded in the win32 platform services as GetWindowTextA(), meaning that you have to write a wrapper so that the wrapper can call the correct function prototype. Using an Alias eliminates that need. Regardless of which platform you’re running on, you will still call GetWindowText(). The 32-bit function will reroute the function internally to GetWindowTextA() without our interference.

In the ancestor to the API services, I add a wrapper function for each of these functions which simply translates my preferred function calling method to the method supported by the function. (I prefer that a function does not return anything other than a function return code and that any data to be returned be done using a reference argument variable).

⁶ The completed service class, along with the service manager and other SBA base classes, may be downloaded from my web-site at <http://www.cris.com/~bgcastle>

PFC Code

To do this in the PFC we have to extend the existing PFC platform services in order to add the Windows API external functions. Again, whether or not you use a Corporate Extension layer inserted between PFC and PFE, will decide where you place your extended code. I utilize the Corporate Extension concept and therefore will place my code in my PFD layer. For the sake of discussion here however, we'll simply put the code in the PFE layer.

N_cst_platformwin16

First we'll add the external function declarations to the platform services. We'll start with the 16-bit version. The external declarations we'll need are:

```
FUNCTION uint GetDesktopWindow() LIBRARY "user.exe"  
FUNCTION uint GetWindow(uint hwnd, int wflags) LIBRARY "user.exe"  
FUNCTION uint GetWindowText(uint hwnd, ref string wintext, int textlength)  
LIBRARY "user.exe"
```

We're done here. We could write a wrapper function to call the external function, which is typical, but not necessary. I choose not to.

N_cst_platformwin32

Now we add the external function declarations for the 32-bit calls. These are:

```
FUNCTION uint GetDesktopWindow() LIBRARY "user32"  
FUNCTION uint GetWindow(uint hwnd, int wflags) LIBRARY "user32"  
FUNCTION uint GetWindowText(uint hwnd, ref string wintext, int textlength)  
LIBRARY "user32" Alias for "GetWindowTextA"
```

Notice that the GetWindowtext() function uses an alias. This function is actually already coded in the win32 platform services as GetWindowTextA(), meaning that you have to write a wrapper so that the wrapper can call the correct function prototype. Using an Alias eliminates that need. Regardless of which platform you're running on, you will still call GetWindowText(). The 32-bit function will reroute the function internally to GetWindowTextA() without our interference.

SBA_n_svc_appScanner

We create our service class by inheriting from SBA_n_svcbase_managed. This ensures that Service Management features are automatically activated.

Next we code the main function call, of_isAppRunning(string AppName, string filename). In this function we place the following code.

```
IF of_LocateApplication(a_appname, iu_apphandle) = 0 THEN RETURN 0  
IF of_CountApplicationInstances(a_appname, ii_instancecount) = 0 THEN RETURN 0  
IF of_LocateFileinApplication(a_appname, a_filename, iu_apphandle) = 0 THEN RETURN 0
```

The first thing you'll see is that the isAppRunning() is actually a function which makes calls to other functions. IsAppRunning() is provided as an overloaded wrapper function that makes your application developer code simpler and easier to understand. So you can call isAppRunning() and get an answer indicating yes or no (by the returned AppInstanceCount variable), but internally, it works a little differently.

Depending on which overloaded function you call, you might call the internal functions of `_LocateApplication`, `of_CountApplicationInstances` and/or `of_locateFileinApplication`. `Of_LocateApplication()` performs the first level of our checking. It will do the following steps.

1. Check that the winapi service is loaded, and if not, load it.
2. Obtains the handle to the desktop
3. Obtain the handle to the first child window (child to the desktop)
4. Check if this first child is the application window we are looking for.
(Remember that the application name may be a portion of the application window name, so the check must be length specific).
5. If not, enter a loop to get the next child window and check it's title
6. When the loop ends (either handle returned = 0 or we find the application we are looking for), we enter the next phase of the method.
7. If the handle = 0, we've discovered that the application is not running and we can simply return an application instance count of zero to the caller.
8. If we've found the application we can enter the function `of_CountApplicationInstances` where the loop above is continued, counting each time we find the application.
9. If a filename is supplied, the caller is looking for a combination of application and filename. We will call the `of_LocateFileinApplication()` function where we will do two steps. The first will be to check that the current window we have access to does not contain the filename in it's title anywhere. If not, we enter a loop scanning all of the child windows for the particular application looking for the filename in the title of the child window. If we find it, the file is loaded. If not, the file is not loaded for this particular application and we can continue. This process is repeated for each instance of the application that is found until we can determine whether or not the application/filename combination is running or not.

One thing we have not discussed is what to do is the application is found. I've left it up to you to make this decision. When you look at the sample code supplied for this in the SBA Demo application, you'll see how I approached the problem.

Now that the object classes are developed, we're ready to integrate the service into our application.

Integrating the Services

Integration of services is like selling real estate. The most important three elements are location, location, location.

Because this is an Application service, the choice is quite simple. If you are using the PFC, we implement the service into the Application Manager. If you've adopted the inserted corporate layer technique, then this new service will be implemented in the corporate layer. If not, you would implement it in whatever you have designated the corporate extension layer to be, or, if you're not using the PFC at all, I choose to implement the service in the application object open event. In either case, we need to follow a few steps.

- Add an instance variable reference pointer to the application manager for the Service Manager. (PFC uses variable name `Invo_ServiceName`. I recommend you follow this practice when working with the PFC. Use the convention that meets your standards). Remember to implement the service manager at the extension level. (`n_svc_Manager` rather than `SBA_n_svc_Manager`) so that you can extend the service manager for yourself. The Service Manager is an auto-instantiated service, so create/destroy is handled for you.
- Add an Instance variable reference pointer in the application manager for the new service. (I called it `SBA_n_svc_AppScanner`, therefore I code the instance variable as `SBA_n_svc_AppScanner invo_AppScanner`). Note that we do NOT declare this service at the extension level. There is no extension level because the Service Manager allows us to define the class being instantiated at runtime. To extend this service, inherit from it, add your code, and then change the name of the service being loaded in step 3 to the class you create.

| |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| NOTE: In order to use the service, you must do 2 things. Load the service, and call the service method. Typically, I add the service load to the Application Manager Constructor event. I need to code the following: |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

- `invo_SBAMgr.of_LoadSvc(invo_AppScanner, SBA_n_svc_const.Svc_AppScanner) //Loads the application initialization service.`

Note that I use a constant to send the service name. Using constants this way allows you to set up your service classes in one location, and, because the constants are defined as `CONSTANT String`
`svc_AppScanner="SBA_n_svc_AppScanner"`, you do not have to physically load the constants object anywhere. At compile time (when you save the object), the constant reference is replaced with the actual value.

In future, any new services I add will only need steps 2 and 3. The service manager takes care of destroying the service once out of scope.

Finally, in the PFC_Open event in the application manager, I will add the code: `invo_AppScanner(GetApplication().AppName, li_AppCount)`. Of course you would replace the `invo_` declaration to match the reference pointer you defined.

With this implementation, we're looking for an instance of our application, so we're using the simple form of looking for the application only. The `li_AppCount` variable, a local integer, will contain the number of instances of the application that is running and you can therefore choose to do what you want to based on the knowledge you now have.

In other areas within our application, we might want to look for an instance of an Excel spreadsheet. We would simply call that function using the Application manager reference pointer as:

`gnv_App.invo_AppScanner("Excel", "MySpreadsheet.xls", ii_AppInstanceCount)`
which would tell us if excel were running and file `MySpreadsheet.xls` were open. Of course, to look simply to see if excel were running, you would omit the filename argument.

Summary

This concludes the discussion on how to design, develop and implement a service. The key elements emphasized are to set a goal for the service, expand that goal by thinking beyond the scope of your application, design the service following a framework of object classes, build the classes following your design, and devise an implementation strategy to implement the service in any application. Next we will take a look at how to go about extending a service class beyond it's original scope.

Chapter 18 - Service Class Families - The Transaction Object

The transaction connection service in the PFC (PFC_n_tr and extension n_tr) does not include the necessary code to connect to various DBMS's. How should you go about extending this class to allow this generic connection capability?

Goal

Goal Statement:

The goal here is quite simple. Extend the existing n_tr class to allow access to various DBMS's.

Design

There are two schools of thought when it comes to extending the capabilities of client classes. The first that comes to mind is the traditional inheritance approach, which, when you think about it, almost makes sense. The transaction object is a nonvisual component which provides a service to PowerBuilder applications. The way to extend the capabilities of a service is to use inheritance. So you end up building a hierarchy of the transaction object that looks like that shown in figure 33.

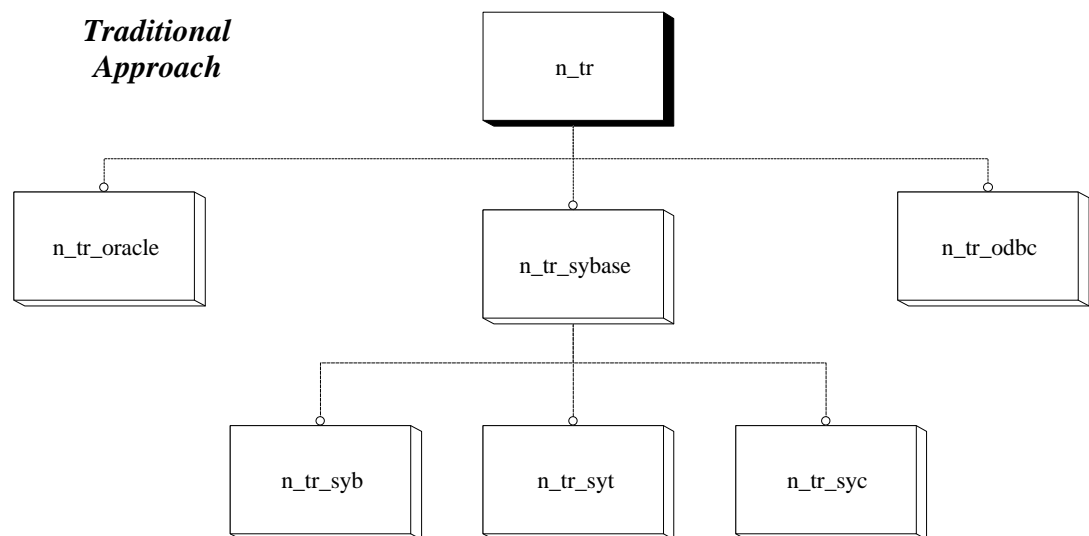


Figure 33 - The traditional approach to extending the n_tr class

When you look a little closer however, you will find that this is perhaps not the best approach. For example, suppose you want to create a Sybase transaction object for your specific environment, so you want to specialize the transaction class and create your own object, `n_tr_mytrans`. Which of the Sybase descendants do you use? OK, so we decide this will be system 11 only, straight CTlib access, so we decide on `n_tr_syc`. We've now locked our design into a single approach.

So, how do we improve upon this approach? First, let's change one myth. Not all Nonvisualobjects are services, and not all services are nonvisualobjects. Now, having said that, we can rethink our position on the transaction object and realize that is in fact, a client class, not a service class. Being a client class we can further surmise that to extend the class, we should use a) Extend by Insertion, or b) Extend by Delegation. These are two of the possible extension methods available using SBA. This first dictates that you "insert" a new layer between the actual object you wish to use, and it's predefined extension layer descendant. Extension by delegation is the process of delegating required behavior to a service class.

The best approach here is extension by delegation. By this I mean that we will be best served by implementing the DBMS specific behavioral requirements in a Service Class that can then be implemented into the `n_tr` transaction class. This is better represented by figure 34.

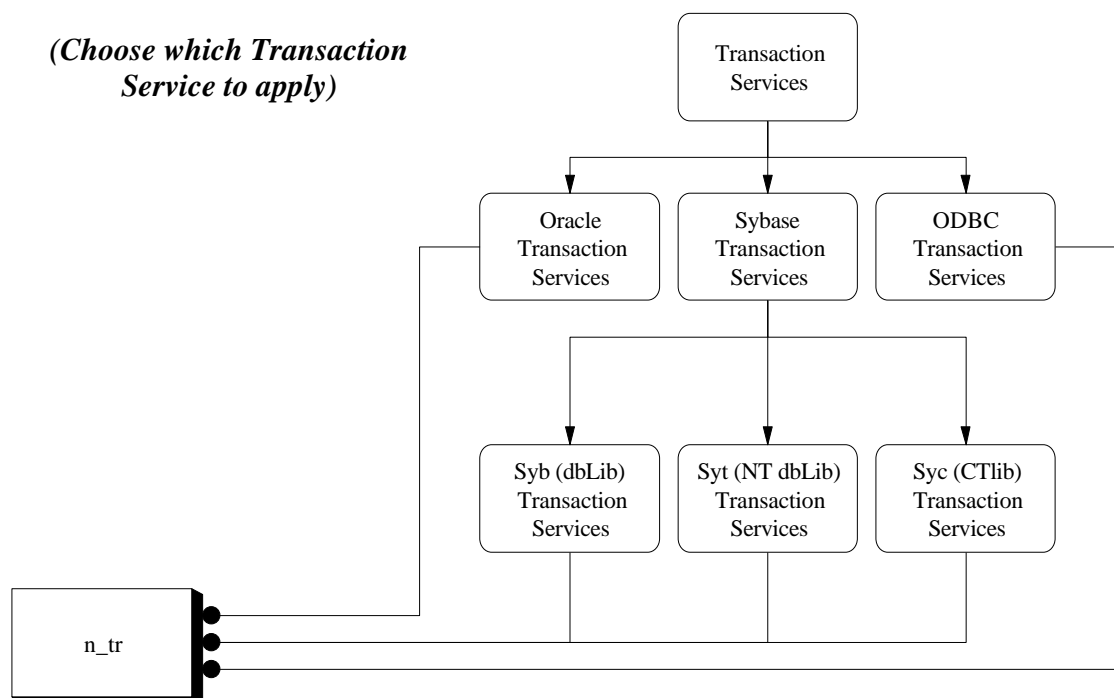


Figure 34 - Transaction Services implemented into the `n_tr` class

Now, the code which instantiates the service class within `n_tr` simply has to make a sensible decision as to which service class should be instantiated at runtime.

Chapter 19 - The DataWindow Button (Coming Soon)

The DataWindow has never allowed us to put buttons on it. Various workarounds include putting buttons on the container window, positioned to appear as part of the datawindow, to drawing a button on the datawindow using the rectangle and line objects. This approach differs a little in that it provides the look-and-feel of a button, but adds a service to automatically process the buttons “clicked” event.

The Goal

The Design

Expanding the Design

Building the Objects

Integrating the Objects

Future Attractions

Brokered Service example

Automated Business Rules

Appendix A

This appendix will house the naming conventions and general service rules/guidelines that are used in this document

Naming Conventions

Service Class Naming

<prefix>_<class type identifier>_classification_name

example: SBA_n_svc_error

Example PFC: PFC_n_cst_dwsrv_linkage

prefix - a prefix assigned to identify the source of the object. For example, I used *SBA* throughout this document to identify the SBA defined classes. The PFC uses *PFC* to identify it's base classes. Prefix is often omitted to indicate an extension class.

Class Type identifier - an indicator of the type of class being defined. Most typically the class is a nonvisualobject, and the identifier is *n_*

Classification - a classification of the type of class this is. The PFC specifies *cst* for Custom Class for most service classes, with datawindow services further customized with *dwsrv*. I prefer to identify the type of class more using:

| | |
|-------|-------------------------------------------|
| svc | a Service class |
| const | a Constants container class |
| api | an API external functions container class |

Name - The name of the service

Development Guidelines

Developing a service class following a standard framework results in more adaptable and integrated services. This means that a few guidelines must be established, and adhered to, if services are to truly become “plug-n-play”. The guidelines established and followed in this document are listed as a checklist below to allow you to measure your service class “compliance”:

- Do not maintain information pertaining to the client within a service class. Pass client information to the service with each called method.

*** This section will continue to evolve ***