# Putting an Object-Oriented Face on a Toolbox Manager

***How to Use OOP to Manage Toolbox Complexity***

**by Ralph Krug, P.E., Software Contractor**

## *Abstract:*

*Most Macintosh Toolbox managers are complex units of code. To successfully use a Toolbox Manager, a programmer has to wade through a lot of documentation (Inside Macintosh, Tech Notes, magazine articles) and perform empirical tests. The details of how to use a Toolbox Manager are spread out in an ocean of words and a void of blind tests. Object-oriented programming (OOP) is supposed to provide techniques to encapsulate this kind of complexity. In this paper, I explore the use of OOP to encapsulate a Toolbox Manager.*

Macintosh programming is based on the Macintosh Toolbox. The Toolbox, which resides mostly in ROM, provides sophisticated control of the machine that is Macintosh. The Toolbox is broken into many managers, from biggies like TextEdit and QuickDraw to lesser-knowns like the Deferred Task Manager and the Sound Manager. Eventually, mastering Macintosh programming means mastering the Toolbox Managers.

Most Macintosh Toolbox Managers are complex units of code. To successfully use a Toolbox Manager, a programmer has to wade through lots of documentation and perform empirical tests. The details of how to use a Toolbox Manager are spread out in an ocean of words and a void of blind tests.

Object-oriented programming (OOP) provides techniques to manage this kind of complexity. With OOP, you can package a complex body of procedural code in a way that makes it simpler to use. At the same time, you can access the underlying power through customizing add-ons to the OOP package.

This paper explores the use of OOP to manage complexity in Macintosh programming. In particular, the paper considers how to encapsulate a Toolbox Manager. After describing the motivating context, I propose an approach for the design and implementation of an OOP wrapper for a Toolbox Manager. An OOP wrapper is an object-oriented class structure that hides complexity as much as possible, while retaining the ability to access the full power of the target Toolbox Manager.

> *An aside:* At times in this paper, I'll ramble on about Mac programming in general, how we Mac programmers have a hard lot, and other aspects of Mac programming that you may already know only too well. Please bear with me. I hope to pass on some useful insights on OOP programming that I learned through four years of MacApp programming. Think of the stuff you already know as syntactic sugar or commiseration material.

> *Another aside:* Although my experience has been with MacApp, the complexity-reducing techniques I describe in this paper should apply equally well in other OOP environments on the Mac. Also, the overall technique applies to *any* system of complex procedural code, not just Toolbox Managers.

## The Macintosh Programmer's Context

As a Mac programmer, you have to deal with the Toolbox. Most Toolbox Managers include many constants and functions that interact in precise (and often obscure) ways. You have to understand a lot of details before you can begin to use a Toolbox Manager correctly.

In your attempts to understand a Toolbox Manager, you typically have to haul out many voluminous documents and spend lots of time wading through them. To *really* understand a Toolbox Manager, you have to read and understand the appropriate chapters in Inside Macintosh, any applicable Tech Notes, and the appropriate portions of the Q & A Stack, if any. And then, if you still need answers, you may end up checking on-line conferences and archives, magazine articles, and books.  Inside Macintosh sometimes lacks crucial details on how to use Toolbox code and occasionally leaves gotchas for you to discover the hard way.

After you’ve spent more time than you care to on research, you end up concluding that the available documentation is inadequate. Nevertheless, you need to produce working code, so you bite the bullet and continue the research or (more likely) you begin trying something—anything.  The latter approach amounts to the empirical research phase of Macintosh code development and can consume vast quantities of time.

If you get lucky and *do* find the answers you seek in Inside Macintosh, beware of out-of-date and erroneous information.  You can hope, but the Tech Notes and Q & A Stack do not always make up for Inside Macintosh’s errors and omissions.  If you run into misinformation, you end up in the empirical research phase quicker than you can finish a compile in MPW.

So how do you deal with this complexity?  You can only try to manage it—divide, hide, and conquer.

## Using OOP to Manage Complexity

The main benefit of object-oriented programming (OOP) is its ability to package complex chunks of code in a reusable and extendible way.  An OOP implementation of a Toolbox Manager can hide much of the underlying detail.  At the same time, you can access the full power of the underlying code by overriding the default behavior wherever necessary.  This is how OOP manages complexity—you can access the code at the desired level of abstraction and ignore the lower-level complexities.  (Of course, *somebody* has to worry about the lower-level complexities—more on that later.)

> *A personal example:* In 1987, I came to the Macintosh programming world from a background in minicomputer-based UNIX and Pascal.  I had a terrible time trying to write even a simple program in Lightspeed Pascal.  Lightspeed Pascal seemed great, but the small changes I made to the example applications left me staring at the MacsBug screen.  So I tried MacApp.  MacApp is, in effect, a collection of OOP code that hides the details of the basic Toolbox Managers.  MacApp hid the Toolbox (and other) details well enough to let me accomplish some programming without crashing immediately. In fact, I referred to Inside Macintosh very little in four years of MacApp programming.  Then I got a project that had me delving into the depths of TextEdit.  I learned quickly how devastating (in time and missed deadlines) a Toolbox Manager can be.

Usually, to simplify a complex piece of code, you have to pay the price of reducing its functionality.  This is not the case when using OOP.  Yes, an OOP implementation simplifies by making assumptions.  And yes, those assumptions effectively reduce functionality to certain default cases.  *But*, with OOP, you can override any part of the default code to suit your needs. Overriding does not mean a major rewrite of the existing code—the existing code remains as-is and usable.  Overriding means *adding* chunks of code that effectively customize the existing code to achieve specialized functionality.

In this way, a well-designed set of OOP classes (data and procedure encapsulations) allows a programmer to access underlying functionality at several levels of complexity.  The top level is the simplest, hiding as much complexity as possible by making assumptions and defining defaults wherever feasible. Lower levels are based on the more-general top level, but include customizations to produce useful variations of the default behavior.

## OOP and the Toolbox: No Free Lunch

Although an OOP implementation of a Toolbox Manager can manage its complexity, there is no free lunch.  Before you can reap any benefits, you must create or obtain said OOP implementation of a Toolbox Manager.

If you create the OOP implementation yourself, you have to learn all the details of the target Toolbox Manager.  Of course, this defeats the goal of avoiding Toolbox complexity.  Creating the OOP implementation yourself must be seen as an investment: benefits accrue in the future when you (or your colleagues) can reuse the code.

> The question remains as to whether you can afford such an investment.  On the other hand, can you afford to reinvent the code each time you need it?  Can you afford to maintain several different versions of code that implements very similar functionality?  This conflict amounts to the usual tradeoff between short-term and long-term benefits.

If you can find an already-developed OOP implementation, then you can benefit immediately.  MacApp and the Think Class Library provide OOP environments that cover the main Toolbox Managers that relate to user-interface implementation.  However, there are many Toolbox Managers that these environments do *not* cover.  Beyond what your development environment can provide, you are not likely to find an OOP implementation for a given Toolbox Manager.  Very few reliable, commercial-quality OOP components exist, either for sale or free.  In particular, I know of no stand-alone OOP implementations of Toolbox Managers.

> To use MacApp or the Think Class Library, you have to buy off on a whole programming approach to gain the benefits.  You have to base your software on the generic framework supplied, which means you have to adapt your existing code to the new environment.  Furthermore, in the case of MacApp, the shift means an additional investment in extra memory and fast hardware.  C++-based MacApp 3.0 is tremendously resource hungry (and still very slow to compile).  Nevertheless, MacApp and the Think Class Library are primary means to obtaining usable OOP code.

## Wrapping Your Own

Let's assume you want to take a stab at OOP and create your own OOP wrapper for a Toolbox Manager.  Say you decided that the expected benefits justify an investment of time and effort toward creating the wrapper.  How do you approach using OOP to put a friendlier face on a Toolbox Manager?  The rest of this paper proposes an approach that can help you.

> Whether or not you currently use MacApp or the Think Class Library (TCL), you can employ the wrapper technique described here.  Actually, OOP-based Toolbox wrappers are a good way to get started in OOP without the complete conversion of approach required by MacApp or TCL.

## Do Your Research

Macintosh programming is a complex undertaking, as described earlier.  The first step in any complex undertaking is finding whose shoulders you can stand on.  This means finding all the documentation and example code that applies to your task.

### *Read Inside Macintosh*

Find the chapters of Inside Macintosh that pertain to your target Toolbox Manager.  Inside Mac is something of an ongoing history of the Mac and its Toolbox, so check the earlier volumes for historical background.  Often the complete description of a Toolbox Manager is spread out over two or more volumes.  (This happens when the chapter on a given Toolbox Manager describes only the *changes* to that manager since the previous volume was published.)  Remember that Inside Mac is not perfect, it may not cover some important details and some information may be outdated.

> Inside Macintosh is a six-volume set of manuals written by Apple and published by Addison Wesley.  You can find Inside Macintosh in many bookstores—even in national chain bookstores in shopping malls!  You can also obtain Inside Macintosh from several mail-order outfits that advertise in MacWorld and MacUser magazines.

### *Check the Tech Notes*

To minimize discovering problems the hard way, check the Macintosh Developer Technical Notes (Tech Notes) for pertinent information.  The Tech Notes supply clarification of Inside Mac chapters and information that Inside Mac left out.  But, as with Inside Mac, the information may not be complete and may be outdated by the time you read it.  Always be prepared for things to work differently than you're told.

> Macintosh Developer Technical Notes are available from APDA, Apple's mail-order development tools distribution division.  For further information on APDA, call 800-282-2732 (from US) or 800-637-0029 (from Canada) or 408-562-3910 (from other countries).

### *Check the Q & A Stack*

The Q & A Stack is a Hypercard-based compilation of "the most common and informative question DTS receives from developers" (to quote its opening card).  Open up the Q & A Stack and see if there are any pertinent questions and answers regarding your target Toolbox Manager.

> The Q & A Stack is available on the Developer CD Series CD-ROMs from Apple.  To obtain a subscription to the Developer CD-ROMs, you must enroll in Apple's Associates and Partners Program or order the Developer Resource Kit from APDA.

*Look for Pertinent Example Code*

This one is important. You want to use pre-written, pre-tested code if you can. As always, you have to be careful. The code you find can be problematic in many ways:

- The code may not be as well tested as you had hoped: You find bugs, try to fix them, and—because of creeping code revision—end up losing the advantage of starting with existing code.
- The code may include portions that are obsolete: You find problems due to changes in system software or Mac hardware. The revisions required to fix the problems could erase the advantage of using existing code.
- The code may be poorly suited to OOP: You find the code to be so interrelated and non-modular that breaking it up into OOP classes is impractical; you are better off re-designing from scratch.

You can look for example code in several places. Development environments typically come with examples to help you get started. On-line information services (such on America Online, Compuserve, and Internet) can provide source code of many different qualities. However, the most useful example code regarding Toolbox Managers is on Apple's Developer CD-ROMs.

> Example code and lots of other useful developer information and tools (including the Q & A Stack, an electronic version of Inside Mac, and the Tech Notes stack) are available on the Developer CD-ROMs.

*Remember Compatibility*

Keep in mind that different versions of the System Software and different models of the Mac mean different capabilities of the Toolbox. A good Toolbox Manager wrapper encapsulates the details of which resources are available in which combinations of System Software and Mac hardware.

> To behave properly on all Macs under all systems, good code must ensure that all the Toolbox components it needs are available and of an acceptable version. The Gestalt Manager (another Toolbox Manager!) helps with this aspect of Mac programming. (Be careful, the Gestalt Manager may not be available itself!) If, after checking the launch-time Mac environment, an application finds that the current environment is not adequate to run the main code (and there are no alternatives available), it can exit gracefully.

## Create a Design

Once you have done some research and have a good idea of the target Toolbox Manager's capabilities and requirements, you can create a complexity-reducing design. There are several steps in creating the design. Start by reiterating the goals of the project to make clear what the design must achieve. To create a first take at the design, derive a comprehensive structured model of the target Toolbox Manager from the documentation and example code found in the research phase.

Then shift perspective: reshape the structured model into a version suitable for an object-oriented implementation. An object-oriented model defines components in ways that support useful combinations and extensions of the components. From an implementation perspective, the object-oriented model abstracts functionality to maximize code reuse and consistency.

Finally, invest the time and effort required to iterate over these steps until you refine the concepts and plans as much as possible. Making worthwhile changes is easy at this early stage; later, changes are more costly.

The following subsections explain these steps further.

*Identify the Goals*

The start of a development plan is a good time to identify and reiterate the goals of the project. The goals of a Toolbox Manager wrapper are to:

- manage complexity
- maximize ease-of-use
- provide high-level access
- allow for low-level access
- provide extendibility
- maximize reusability
- minimize overhead

Of course, these goals overlap, interrelate, and conflict to varying degrees. You will shape these fuzzy goals into a useful design in later steps. In this step, the idea is to clarify the overall purpose of the wrapper code.

*Derive a Structured Model*

To get a good start on the design of a wrapper, develop a structured model of the target Toolbox Manager. By "structured model," I mean a well-organized, hierarchical specification of the capabilities and details of the Toolbox Manager.

The structured model you create should be well organized with respect to the goals identified above. For example, it should help manage complexity by making clear what the Toolbox Manager can do and what it requires to carry out its tasks. It should also maximize ease-of-use through hierarchical ordering. Hierarchical ordering simplifies by abstracting the functionality and details of the Toolbox Manager.

To begin development of a structured model, read the pertinent chapters of Inside Mac and outline them. Using your favorite software outliner, pore over the documentation and make notes as you go.

Periodically, step back and see if the pieces are falling together. Rearrange the notes within the outline to make them fall into a structure that makes sense. Use the hierarchical power of outlining to categorize and abstract the many details of the target Toolbox Manager.

Then, go back to the other sources of information identified in the research phase (the Tech Notes, Q & A Stack, example code, etc.). Mix the details from these sources into your outline. Determine where the new details belong, and correct, update, and expand the outline as you go.

This process should yield a comprehensive model of the target Toolbox Manager. The hierarchical structure of the outline should help you navigate among the sea of details.

*Derive an Object-Oriented Structure*

The next step constitutes a perspective shift. In the previous step, you derived a comprehensive, hierarchical model of the target Toolbox Manager as a whole. In this step, you revise the model as necessary to create an object-oriented design that meets the overall design goals. The revision process consists of dividing, combining, and rearranging parts of the comprehensive model until it achieves the goals to the highest degree possible.

At this point, OOP development turns into an art. There is no well-established, recipe-like approach for creating a good OOP design.

> There are several formalized methods of object-oriented design and analysis (OOD/OOA). My impression—*based on limited reading of journal articles*—is that they do not apply well to Macintosh programming. They appear to concentrate on data flow, providing methods for producing an OOP design to handle the required flow of commands and data. In the Mac approach to software, *event flow* is as important or more important than data flow. The formalized OOD/OOA methods documented in periodicals such as the *Journal of Object Oriented Programming* and *Object Magazine* seem to ignore event flow.

> A new book, "Developing Object-Oriented Software for the Macintosh" by Neal Goldstein and Jeff Alger (part of the Macintosh Inside Out series, published by Addison-Wesley), caused a stir at the latest MADA (MacApp Developers Association) Conference. Judging from a presentation given by Jeff Alger at the previous (1991) MADA Conference, I think the Goldstein-Alger approach (named "Solution-Based Modeling") shows great promise. I look forward to exploring Solution-Based Modeling further.

In lieu of a reliable method, you have to learn by example and by doing. For example, careful study of MacApp's structure, along with experience using MacApp, helps you learn what good OOP is and what to avoid.

A good starting point is to use the Model, View, Controller (MVC) approach. (The MVC approach stems from the Smalltalk development world.) Roughly speaking, the MVC approach categorizes the code of an application into three types: Model, View, and Controller. Models correspond to data structures of the application, Views display information in the application, and Controllers manage the interaction of Models, Views, and the user.

> A weakness of the MVC approach is that it is often difficult to separate code into *one* of the three categories. There is often a combination of Model, View, and Controller aspects in even the smallest chunks of code. Breaking the code into chunks that are strictly Model, View, or Controller would often be impractical due to excessive overhead. Therefore, I suggest that you apply the MVC approach with a grain of salt. Use the MVC approach as a good starting

point, but don’t be a slave to the process of dividing code into Models, Views, and Controllers. Use your judgment to decide when further MVC division serves no useful purpose, then move on.

*Define Models*

Factoring Models out of the structured model is usually straightforward. Models constitute the data structures of the structured model. Go through the structured model outline and define a Model for each coherent collection of data records.

The sometimes difficult part of this step is determining where to draw the lines separating the data components into Models. When in doubt, try to create least-common-denominator Models that can combine in the greatest number of meaningful ways. Remember that you can build larger Models from smaller ones as necessary.

*Define Views*

Should the wrapper provide Views, or should the application that uses the wrapper provide the Views? The answers are yes and yes.

Typically, the Views associated with a Toolbox Manager are quite basic. Also, the target Toolbox Manager may already provide some standard Views. (For example, the Standard File Package supplies standard dialogs for opening and saving documents.)

A Toolbox Manager wrapper should provide Views in the same way that some Toolbox Managers provide standard dialogs. The wrapper can provide standard Views to give the programmer a ready-made user interface to the target Toolbox Manager. The application that uses the wrapper can always ignore, modify, or replace a wrapper-provided View if it is not appropriate.

*Define Controllers*

In a Toolbox Manager wrapper, Controllers provide the higher-level programmer’s interface to the Toolbox Manager. Controllers provide this interface by combining calls to lower-level Toolbox Manager routines with procedural knowledge of how the Toolbox Manager works. In this way, the Controller packages the complexity of the Toolbox Manager.

To define the Controllers for your wrapper, think of how you would like to use the target Toolbox Manager. Create a Controller (or family of Controllers) for each high-level operation you would like to have for the given Toolbox Manager. At this point, the sky is the limit—you can define as much functionality as you want. (Of course, *somebody* still has to implement it…)

*Evolve and Refine the Design*

Software development is never a linear process. Iteration is certain. Embrace the inevitable iteration—rethink your design again and again. With each iteration you understand the details better and see the whole more clearly.

Think of this iteration as a worthwhile investment, for changes and refinements made at this stage are less costly than those made later. Do as much of the design work at this stage as you can stand. Design work in the later stages is also inevitable, but we can at least *try* to minimize it.

## Implement the Design

Once you have a good design, you can begin converting it into code. In general, the OOP classes you create will correspond to the Models, Views, and Controllers in your design.

At this stage, try to maximize code reuse through the creation of abstract classes. You create an abstract class by factoring out code and data common to two or more existing classes and transferring code and data into a new abstract ancestor class. Along the way, the implementation becomes a hierarchy of abstract and derived classes. When you finish, the Models, Views, and Controllers of your design will lie somewhere near the bottom of the hierarchy, descended from abstract classes that encapsulate shared code and data.

If you have many Models, Views, and Controllers to implement, you might want to plan more than one wave of implementation. The idea is to tackle a core group of the Models, Views, and Controllers first, then a more peripheral group, and so on. This approach will yield usable code sooner.

*Convert Models into Classes*

The Models in your design primarily represent data, whereas OOP classes are packages of data members and member functions. Good OOP practice requires that the accessible data of a class be accessed through member functions. This practice allows descendant classes to override the data-access member functions when necessary to customize functionality.

The terms “data member” and “member function” are C++ terminology. The equivalent terms in Object Pascal terminology are “field” and “method,” respectively.

You can also take advantage of access functions to maintain consistency and validity among the data. To do so, write the code to check for consistency and validity with each access of a data member. An inconsistent or invalid value can then call on a Controller object to handle the situation.

*Convert Views into Classes*

The Views of your design convert into classes that maintain the user interface portions of your wrapper.

User interaction with a View causes the View to update itself accordingly. The View can then call the appropriate Controller, if necessary, to propagate the event or change of data.

In general, a View should not change a Model directly. Let a Controller handle the change. This approach concentrates the procedural knowledge of the code in Controllers instead of spreading it out among Views and Models. Confining procedural knowledge to Controllers helps make the resulting code maintainable and flexible.

*Convert Controllers into Classes*

Controllers tie Views and Models together. Controller objects convert a user command or system event into actions that eventually update Views and Models.

Higher-level Controllers can also trigger a combination of lower-level Controllers. You can use this approach to maintain consistency in an application. For example, Controller A always works the same way, whether called from Controller context X, Y, or Z.

*Factor Out the Abstract Classes*

As you write code to implement the Models, Views, and Controllers of your design, you will find classes that have significant similarities. You can consolidate code by combining these similarities into abstract classes. An abstract class serves as an ancestor class, providing data members and member functions needed by two or more subclasses.

## Reiterate Ad Infinitum

In software development, you cannot execute a series of steps once and expect code to work (project plans notwithstanding). Code design and implementation consists of ongoing cycles: you loop through a series of steps numerous times, making a little progress with each repetition (hopefully). As you make progress, you can drop some steps and add some new ones. So it is with OOP, which adds the steps “make the code reusable” and “make the code extendible” to the overall sequence.

During the design stage, you loop within the outline, refining your design while it’s still very easy to do. This is where you decide what you want to see when you are done, in as much detail as possible.

During the implementation stage, you draw up some classes, then divide them and recombine them to form a class hierarchy that meets both the functionality goals and the code-reuse goals.

Then, during the testing stages, you go back and rework the basic design and class structure as new information dictates.

Reiterate over these stages until you are satisfied that the goals have been met. (More likely, you iterate until the deadline can be postponed no more!)

## Conclusion

I hope this paper helps you employ the power of OOP in your software development. As with most things in life, OOP requires that you invest significant time and effort before you can reap its benefits of flexible, reusable code. I hope this paper helps in the process.

Lastly, I want to explain the appendix. Long before I started this paper, I started work on an OOP wrapper for the Sound Manager (mostly as an experiment). The appendix contains the raw outline as I left it many months ago. It is not complete and very rough, but it may help you visualize the concepts presented in this paper. Take a look, but be kind; the outline in the appendix is an unpolished, unfinished work in progress.

## Appendix: An Unfinished Sound Manager Wrapper Outline

I. Sound Manager MVC:
- A. model:
  - 1. square-wave sound sequence
    - a) frequency/rest vs. time specs
  - 2. wave table sound sequence
    - a) frequency/rest vs. time specs
    - b) wave table(s)
  - 3. sampled sound sequence
    - a) play/rest vs. time specs
    - b) sample(s)
    - c) sampling rate(s)
    - d) playback rate(s)
- B. view:
  - 1. the controller window (optional)
    - a) provides graphic feedback that the sound is playing
      - (1) may show a progress/“time remaining” indicator
        - (a) -- shows elapsed/remaining time (in text/numbers)
        - (b) -- shows progress via sideways thermometer
    - b) is optional--the application doesn’t have to initialize and show it
    - c) can optionally include the following indicator sub views:
      - (1) a stop button -- hilites when sound sequence is halted
      - (2) a play button -- hilites when sound sequence is playing
      - (3) the sample’s sampling rate at recording
      - (4) a playback rate--indicates the playback sampling rate
- C. controller:
  - 1. the sound manager
    - a) initializes system software/hardware
    - b) plays sounds (via sound channels and controller window)
      - (1) provides/disposes channels upon request, using the specified channel characteristics
      - (2) passes the sound sequence to the sound channel
      - (3) if so directed, displays a controller window to allow user control of sound playback
      - (4) returns control to caller ASAP or when sound is complete, as directed
    - c) provides error codes to the calling routine and displays user alerts (if so directed) for error/exception conditions
    - d) cleans up when done
  - 2. the sound channel
    - a) handles acquisition of sound data from resources and/or files as necessary
    - b) plays a given sound sequence
    - c) handles user interrupts (CMD-.)
    - d) releases the hardware ASAP (to avoid hardware conflicts/crashes)
    - e) returns error codes to sound manager
  - 3. the controller window (optional)
    - a) halts play when user clicks a stop button (leaving the “sound pointer” at the stopping point)
    - b) initiates/resumes sound sequence play when user clicks a play button
    - c) stops the sound and returns the “sound pointer” to the beginning of the sound sequence when user clicks a reset button
    - d) stops the sound, resets the “sound pointer” to the beginning of the sequence, and plays the sequence again when user clicks a replay button

II. Setting up a sound sequence
- A. TSoundSequence -- abstract class
  - 1. TSquareSound
  - 2. TWaveSound

    3. TSampledSound
  B.
III. Using the Sound Manager
  A. Initialize
    1. TSoundManager.ISoundManager(  ).
  B. Play sound
    1. check if resources are available: If CanPlay(specs, errID) Then ...
      a)  CanPlay(specs:TSoundSequence; errID:Integer) sets up a channel with the given specs and returns true if all is OK/goes OK
      b) if CanPlay finds a problem (e.g., no available channel, CPU overloaded, sound file not available,…), it returns False and puts an appropriate error number in ‘err’.
      c) the calling routine can then call a TSoundManager method to alert the user…
    2. play the sound -- DoPlay() , PlayNotes(),
      a) DoPlay(specs:TSoundSequence ) -- play the sound specified by specs
      b) PlayNotes( noteSequence:TNoteSequence, specs:TSoundSequence) -- play the square wave sounds spec’d by noteSequence
      c)
    3. modify sound/channel characteristics
      a) Modify(specs:TSoundSequence; modList:Xxxx  )
    4. release sound resources/reservations
      a) FinishedPlaying( specs:TSoundSequence )
  C. close sound manager (to reduce memory req’ts...)
    1. TSoundManager.Close
IV. Sound Channel
V.