NXApp, Winter 1993 (Volume 1, Issue 1).

Branching Out With Dynamic Loading

written by **Andrew Vyrros**

Dynamic loading is a powerful technique for structuring NEXTSTEP programs. It gives software developers a new set of tools and a greater range of flexibility in creating applications. It also enhances a software vendor's ability to support multiple configurations and frequent updates. There are several ways to apply dynamic loading to typical NEXTSTEP projects, and technical issues to consider with each.

The pressure is on. Your application had been selling well, but a competitor just added a killer new feature. Customers are complaining and threatening to defect. Time to panic? No, you just mail out a small package, customers do a simple drag-and-drop, and you've got the hottest app again.

The stress meter is rising. Your biggest client needs to build a custom version of your program, but you're not too keen on giving them all your sources. The cue to take a long vacation? Nah, you send them a few header files and some instructions, and they're off and running.

These may sound like scenes from a developer's fairy tale, but you can arrive at the same happy endings through the magic of dynamic loading.

PIECING IT ALTOGETHER

Simply put, dynamic loading is the process of adding external modules of code to a running program. That may sound simple, but it has some far-reaching consequences: First, it lets you add functionality to a program without recompiling it. This means that applications easily gain new tools and areas of usability. Second, it lets you change the functionality of a program each time it runs. Since youÐor the userÐcan choose which modules to load, the capabilities of a program can be tailored to a particular need.

When combined with object-oriented programming, dynamic loading really shows its potential. Used with a late-binding language like Objective C, dynamic loading allows you to add to the class hierarchy at run-time. You can subclass existing classes or add altogether new classes. The result is that your program can acquire new objects and entities, bringing extra areas of functionality that you might not have envisioned when you first developed it.

WHAT'S THE POINT?

So dynamic loading is an interesting technique, but what practical value does it add to your NEXTSTEP projects? Because it allows you to choose your program's code modules at run time, dynamic loading can change the way you think about applications and functionality. With a little imagination, you can envision the benefits you'll realize.

Dynamic value

Some of these benefits you've likely already considered. Probably the most prominent benefit is that dynamic loading makes your application customizable. This means that users can extend the functionality of your program to suit their needs; all they have to do is add new modules. For
example, you can supply these modules as an adjunct to the basic set. Or, if you publish the API, customers and third parties can create new modules, adding

value to your application.

Similarly, dynamic loading gives you increased flexibility to manage configuration and packaging. Depending on which modules you put inside the app wrapper, you can instantly create different versions of your application. This makes it easy to create basic, advanced, and custom versions.
It's as simple as dragging and dropping a few files, without writing any new code or recompiling anything.

This flexibility also extends to bug-fixes and upgrades. If new code is restricted to a few modules, then you can distribute just the fixed modules rather than the entire package. Because these modules are much smaller than a whole application, they're easier to fit on a single floppy disk or into an e-mail message. Or, if customers want to upgrade from the basic version of your application to the advanced version, they don't need to re-install the entire packageÐthey simply drag the advanced modules into the app wrapper and restart the app.

A number of performance improvements can result from using dynamic loading, too. Most noticeably, the application should launch more quickly. This is because dynamic loading is usually paired with lazy initialization: The application only loads modules as they are needed. Immediately after the app is double-clicked, only a few modulesÐperhaps noneÐneed to be loaded, so the program launches very quickly. Of course there will be small performance hits as additional modules are needed and loaded, but quick launch speed often has the biggest impact on user perception.

As a side effect of this lazy initialization, there is another benefit: lower memory consumption. Because modules are only loaded as they are needed, the modules that aren't loaded don't consume any memory. Since users typically use a small subset of an application's functionality
in one sitting, the memory consumed is a fraction of what would be needed to hold an entire, non-dynamic version of the program. This means reduced virtual memory consumption, which improves performance for your application and throughout the system.

Programmer pluses

In addition to all the product benefits that result from dynamic loading, there are a number of development boons. These aren't a direct result of dynamic loading, but they can have a great impact on your programming productivity.

For example, using dynamic loading in your projects can vastly reduce your linking time. The reason is that code for a dynamic module doesn't need to be linked with the main executable. Instead, it's merged into a single relocatable object file. Because this involves only the symbols referenced in the dynamic module, it proceeds much more quickly than linking the entire application. So if the files you are working on are restricted to a dynamic module, your builds finish much more quickly. In addition, since the main executable doesn't change, the debugger doesn't need to reload those symbols. This means that you can immediately restart your application. The bottom line is much faster passage from edit through compile to test and debug.

Using dynamic loading can also improve the architecture of your code. This is because dynamic loading forces you to identify the essential areas of functionality and to distribute them among the main application and the loaded modules. Then you must define clear APIs so that the loaded modules and the main app can interact properly. As a result, you have to spend time aligning your class hierarchy to the natural functional areas that your modules will support. This should bring the rewards of logically designed classes and carefully structured code.

Finally, dynamic loading can help you organize your development efforts. The key is placing each module into a separate Project Builder bundle project. Each bundle project holds all the associated source files, as well as the required resources such as images and interfaces. Because an individual module consists of a small set of functionally related files, these bundle projects become separate, manageable development units. And since each module gets its own subproject directory, you can easily work on it independently, without the distractions of the rest of the project.

There are some important caveats regarding the time and space needs of dynamic programs. These are discussed later on, along with other tradeoffs.

STRUCTURING YOUR APPLICATION

Now that you're sold on the benefits of dynamic loading, you'll need to decide how to incorporate it into your projects. Naturally, the way you use dynamic loading depends mostly on the needs of your application. But there are a few typical architectures and techniques for dynamic programs that you can use as a starting point for application design.

The fundamental distinction between dynamic architectures is the role and complexity of the modules as opposed to that of the main application. The three main prototypes discussed below vary from simple, single-object modules controlled by a large, complex application to elaborate, multi-object modules used by a small, vestigial app.

This article deals primarily with *symmetric module* structure: a group of dynamic modules that have more or less the same role and relationship to the main application. These modules have divergent functionality but share the same end goal.

Some applications use an *asymmetric module* architecture. In this setup, each module is unique and has a specific relationship to the main program. For example, an app might load one module to operate a peripheral, and another to communicate over a network channel. You can apply many of the concepts in this article to building this second kind of program.

Dynamically loaded tools

One likely configuration is an application with dynamically loaded tool modules. The prototype here is Icon BuilderÐsee Figure 1. In this scenario, you have a main application and a set of dynamically loaded tools, one per module. The main application provides the bulk of the program functionality, such as interacting with the user, managing data, opening and saving documents, and updating the display. The tool modules, on the other hand, provide specialized functionality to control or manipulate data in some way.

Figure 1: *A drawing program, like Icon Builder, with loadable tool modules*

This architecture is relatively straightforward to create. When the user chooses a tool, the application loads the corresponding module. (More on the mechanics

later.) This adds a single new entry to the Objective C class hierarchy. Then an instance of this tool class is created. The application maintains a list of these tool objects, and allows the user to select the current one. The current tool then performs whatever functionality is desired, typically by responding to requests from the application or from user interface objects.

Functional groups

An application with loadable functional groups is similar to an application with loadable tools. Typical of this is Interface Builder™ with its palettes, like in Figure 2. In this scenario, the application has functional modules that are loaded at run-time. As before, the main application supplies most of the basic functionality, and the modules provide the customized parts. Instead of just a single tool per module, though, functional groups contain a number of related objects. These objects can act either as tools for manipulating data, or as the building blocks of the data itself.

Δψνα_Φιγ2.επσ ←

Figure 2: *A design application similar to Interface Builder, with Palettes implemented as functional group modules*

Laying out this kind of framework takes a bit more effort. As in the previous case, the main application loads a primary class and creates an instance for each module. Each primary object acts as a kind of entry point for its module. It gives the main application access to the rest of the classes in the module. The primary object can do this either by creating instances and handing them back to the main application, or by returning the class objects directly to the app and letting it create the instances. Some of these subordinate objects might be used to inspect or manipulate data, similar to the tool objects in the first example. These situations will probably only need a single instance, most likely owned by the module's managing object. Other objects may be the sort that represent entities or chunks of data. In this case the application probably creates multiple instances, which it puts into a separate container such as a document object.

Fully independent modules

A third type of dynamic application is made up of fully independent modules. Here the prototype is PreferencesÐsee Figure 3. This scenario takes functional groups to the extreme; there really is no application without the loaded modules. The main application provides only the most basic of resources, perhaps a menu and a window in which to display the interface for a module. The loaded modules are essentially independent programs that operate within the main app.

Independent modules are similar to functional groups. The main program loads the modules as needed and instantiates a single primary object. But because the modules function as independent programs, they require very little coordination from the main app, other than to select the current module. The module's primary object takes care of everything else, including creating subordinate objects, interacting with the user, and performing its specific functions.


Figure 3: *A program reminiscent of Preferences, with independent modules.*

Mix and match

It's important to realize that these three general architectures are not mutually exclusive. They can be mixed and combined as you see fit. For instance, your app could have both a set of loadable tools and a separate array of loadable functional groups. Or your program could be made up of independent modules, some of which supply their own assortment of loadable tools. The goal is to design an architecture that fits the natural form of the application.

Communication

Once you have chosen the overall structure of your dynamic application, you must make a few more decisions about architecture. Your primary task is to determine how the main application will communicate with the dynamically loaded modules. This is important because the main application shouldn't make any assumptions about which modules will be loaded on any particular launch of the program. It should treat all of the modules as abstract entities that implement their own version of some functionality. Therefore, since it doesn't know anything specific about any of the modules, it must use the same form of

communication with all of them.

There are a couple of ways to work this out. One is to use Objective C protocols. You can define a set of protocols for the objects in the modules and another for the objects in the main application. The two parts of the program then communicate by using methods from the protocols. By seeing that all classes conform to the appropriate protocol, you establish the communication system and ensure that all classes implement the required methods. This approach makes sense when the different modules have little or no functionality in common.

Protocols are a language construct added in NEXTSTEP 3.0 that let you specify the methods a class implements without saying anything about the class's inheritance.

The other option for structuring communication with dynamic modules is to provide abstract superclasses. The abstract classes implement the skeletal functionality that you expect from the objects in the modules. Then the modules create subclasses of these abstract classes to fill in the specific functionality that they want to provide. This way, the classes in the modules are always subclasses of your own classes, so that you define their basic behavior and can communicate with them accordingly. Using abstract superclasses is best when the modules share a lot of common functionality: The overlapping parts are only implemented once, by the abstract superclass in the main app, while the subclasses in the modules only need to provide their specific variations.

SWEATING THE DETAILS

Once you've designed your architecture for dynamic loading, you're ready to start building the application. There are two areas on which you'll be working. First, you handle the mechanics of setting up a Project Builder project for a dynamic application. Once you've created this infrastructure, you can write the code to implement your app.

Setting it up

Dynamic applications have two main conceptual components: the main, static core, and the dynamically loaded modules. Each of these is built from a set of class implementations and other source files. If you've worked out your architecture, you should have a good idea how you want to distribute these files among the components.

The main application is built the same way as a conventional app. You use Project Builder to create a project for the application, then add your source files and other resources to the project. Project Builder manages the resources and massages the Makefiles for you. When you build the project, your sources get compiled into a single executable.

Dynamic modules are a bit different. Each dynamic module has its own loadable code file. This code file is constructed by compiling the classes and other sources for a module and linking them into a single relocatable file. To make this happen, you use Project Builder to create bundle projects for each of your dynamic modules. Then add the sources and other files for a module to its bundle project. When you build the application, the Makefiles build the bundle projects along with the main project. These magic Makefiles compile every module's sources and link them into individual loadable code files, one per module. Then they put the code file and other resources for each module into a separate file package inside the application's file package.

Making it run

With the proper setup for a dynamic project established, you're ready to write the code that does the actual loading. The task of dynamic loading can be broken down into three steps: locating the modules in the filesystem, loading the modules into the Objective C class hierarchy, and instantiating the objects.

Fortunately, NEXTSTEP insulates you from most of the nitty-gritty by supplying much of the key functionality with high-level API. The preferred point of access is the NXBundle class. NXBundle is a utility class with methods for retrieving resources within file packages. In particular, NXBundle knows about dynamic modules and how to load them into programs. As long you keep each dynamic module inside a separate file package, NXBundle greatly simplifies the entire

dynamic loading process.

For more in-depth information about the mechanics of dynamic loading and the NXBundle class, see *NEXTSTEP Object-Oriented Programming and the Objective C Language* and the NXBundle class description in *NEXTSTEP General Reference*.

Locating the modules is a simple matter of finding all the file packages with the desired extension, then creating a single NXBundle instance for each package. NXBundle takes the path to the file package as its initialization argument. Then you store the bundles that you create in a List:

```
char                  modulePath[MAXPATHLEN+1];
NXBundle            * newBundle;

while ([self getNextModulePath:modulePath])
{
   newBundle = [[NXBundle alloc] initForDirectory:modulePath];
   [bundleList addObject:newBundle];
}
```

Once you've created a bundle, you've got a handle on the resources in the file package. But NXBundle doesn't actually load any resources until you explicitly request them. To request Objective C class resources, you ask the bundle for its principal class. (NXBundle assumes that the first class in a module is the principal class; this is determined by the first class listed in Project Builder's Classes browser.)

```
NXBundle            * currentBundle;
Class                 primaryClass;

currentBundle = [bundleList objectAt:0];
primaryClass = [currentBundle principalClass];
```

If this is the first time you've asked for the class, NXBundle loads the relocatable file into the Objective C class hierarchy, then gives you the class. You should check the capabilities of the newly loaded class to make sure you got what you

expected. Then use the class object to create an instance. In the case of a module with a single tool class, the process might look like this:

```
Class                 toolClass;
id <ToolMethods>      newTool;

toolClass = [currentBundle principalClass];
if ([toolClass conformsTo:@protocol(ToolMethods)])
{
   newTool = [[toolClass alloc] init];
   [self putToolToWork:newTool];
}
```

For a module with multiple classes, like a palette, it's just slightly more complex. When NXBundle loads the principal class, it actually loads all the classes in the module since they're all in a single file. Once you've created an instance of the module's primary object, you can use it to get at the rest of the classes in the module:

```
Class                  paletteClass;
id <PaletteMethods>    newPalette;

paletteClass = [currentBundle principalClass];
if ([paletteClass conformsTo:@protocol(PaletteMethods)])
{
   newPalette = [[paletteClass alloc] init];
   paletteInspector = [paletteManager inspector];
   [self displayItemView:[newPaletteManager itemView]];
}
```

In the Palette class, you provide the access methods for the contents of the module:

```
- inspector
{
   if (!inspector)
   inspector = [[MyInspector alloc] init];
   return inspector;
```

```
}

- itemView
{
   NXBundle            * myBundle;
   char                  nibPath[MAXPATHLEN+1];

   if (!itemView)       // itemView is an outlet in a nib
   {
      myBundle = [NXBundle bundleForClass:[self class]];
      [myBundle getPath:nibPath forResource:"MyItemView" ofType:"nib"];
      [NXApp loadNibFile:nibPath owner:self withNames:NO];
   }
   return itemView;
}
```

Freedom of choice

Of course, a solitary palette or tool isn't very exciting; the whole point is to have several that the user can choose from. Ordinarily, you'd create all the tools and put them in a List, with each tool's position in the List corresponding to the tags in an interface element like a Matrix or PopUpList.

Unfortunately, this won't work if you're also using lazy initialization and only loading modules as they are needed. In that case, you wouldn't immediately have a tool to put at every position of the List. Instead, you'd need to have some blank placeholder in the tool List to stand in for a tool that hasn't been loaded yet. But since you can't add nil objects to a List, you're stuck.

One alternative to using List is to use Storage, which allows null elements. Then you can write a method that returns a tool at a particular position, creating the tool if necessary. The C casting required by Storage may look a bit cryptic, but it does the right thing:

```
- (id <ToolMethods>)toolAtIndex:(int)index
{
   id <ToolMethods>    tool;
   NXBundle            *             bundle;
   Class                 toolClass;
```

```
   tool = *((id <ToolMethods>) *)[toolStorage elementAt:index];
   if (!tool)
   {
      bundle = [bundleList objectAt:index];
      toolClass = [bundle principalClass];
      if ([toolClass conformsTo:@protocol(ToolMethods)])
      {
         tool = [[toolClass alloc] init];
         [toolStorage replaceElementAt:index with:(void *)(&tool)];
      }
   }
   return tool;
}
```

Another solution is to create a subclass of NXBundle that adds an instance variable to store an object created from a loaded class. Then the main application could just use a single List of these special NXBundles both to track the modules that the program might load and to hold on to the primary objects after they are loaded and instantiated:

```
- (id <ToolMethods>)toolAtIndex:(int)index
{
   ToolBundle         *toolBundle;

   toolBundle = [toolBundleList objectAt:index];
   return [toolBundle tool];
}
```

The first time a tool is requested, ToolBundle loads its module and instantiates the primary object:

```
- (id <ToolMethods>)tool
{
  Class  toolClass;

  if (!tool)
  {
    toolClass = [self principalClass];
```

```
    if ([toolClass conformsTo:@protocol(ToolMethods)])
      tool = [[toolClass alloc] init];
  }
  return tool;
}
```

The approach you choose will depend on the needs of the application and your personal preference. But from this point you're in the clear. Once you have a standard technique to access your loaded objects, you've built the structure for a dynamic application. The remainder of your code should be essentially the same as for a traditional app.

KEEPING IT RUNNING

Of course, nothing in life ever comes totally free. Like any other engineering technique, dynamic loading brings a number of tradeoffs and potential pitfalls. If you plan for these in advance, your coding will proceed much more smoothly.

Document dilemmas

One of the key issues of dynamic loading is dealing with documents and archived objects. The difficulty arises when the user tries to open a document that contains instances of a class from a bundle that hasn't been loaded yet.

As an example, imagine an application for designing automobile engines. Say that a user creates an engine using fuel injectors from the fuel system module. When the user saves the document, the injector objects get archived in a file. The next day, when the user tries to re-open the document, if the fuel system module hasn't been loaded yet, there'll be trouble. The application won't be able to instantiate the objects in the file, since it doesn't know about the FuelInjector class.

There are a number of possible solutions to this dilemma. The worst-case is to require your users to make sure that the appropriate modules have been loaded whenever a document is opened. Users might force the loading of a module by selecting its icon from a matrix of available modules, or by explicitly requesting that it be loaded. If a document fails to open because of missing module classes,

you can alert the user to try again after loading the right module. This is a rather unpleasant solution that puts a large burden on your users, but it works.

A more elegant approach is to record the names of the necessary modules in a file inside the document package. (The NXBundleÐand thus the module nameÐfor a particular object can be determined by asking **[NXBundle bundleForClass:[widget class]]**.) The application can consult this file before it attempts to unarchive any of the objects in the document. Then it can load all of the required modules before trying to create any objects from the document. This solution is much more elegant and requires no user intervention.

Unfortunately, this approach relies on the names of modules. If the name of a module changes, or if classes are redistributed into different modulesÐfor instance, if **FuelSystem.automodule** is split into **FuelInjectors.automodule** and **Carburetors.automodule**Ðthen users won't be able to read old documents. If this seems likely in your application, you may want to consider a refinement of this approach. Instead of storing a list of modules, you can write a list of all the names of the specific classes required by a particular document. Then the application can check that those classes exist before trying to read the document. If any are missing, the application will have to find the corresponding module so that it can load it. Rather than using trial-and-error, you can put another list of class names inside each module package, and the application can use these lists to find the appropriate module.

Of course, none of these techniques can solve the problem of missing modules. For instance, suppose a user creates an engine design using some special pistons from the new third-party **Cosworth.automodule**, then sends it to a colleague for review. If the colleague doesn't have the new module, he won't be able to read the document. Unfortunately, there's little you can do to prevent this situation. Your only recourse is to notify the user of the situation and try to provide as much information as possible about the missing module so that he will be able to track it down.

What's in a name

The problem of conflicting class names is also a concern. Because there is only a single name-space for Objective C classes, dynamically loaded modules musn't contain duplicate class names. If your application attempts to load a module containing a class name already in use, then the loading of the entire moduleÐnot just the offending classÐfails.

If you produce all the modules to be used with your application, then you can control the naming of classes to avoid these clashes. But if you allow third parties to provide modules, then you must be aware of this possibility. The instructions with your API should mention this issue and instruct developers to be meticulous in creating meaningful names with unique prefixes, to ensure that their class names don't collide with others. As an additional precaution, you can use the technique given earlier of listing class names in a separate file inside the module package. Then your application can check these lists for name collisions and choose the desired module to load.

Symbolically speaking

Dynamic loading involves an additional complication when it comes to symbol tables. The
problem is that loadable code files, unlike regular executables, can't be linked to shared libraries. Instead, the external references in loaded modules are resolved through the symbol table of the main executable. This means that dynamic modules can't use any symbols unless those symbols appear in the main program's symbol table. To put it more explicitly, it means that dynamic modules can't create instances or define subclasses of any classesÐyour classes or Kit classesÐunless the symbols for those classes are included in the main application's symbol table. Likewise, modules can't call any functionsÐin your code or the shared librariesÐunless the symbols for those functions appear in the main symbol table.

To reduce file size, applications usually are stripped of their symbol tables, making dynamic loading essentially impossible. Fortunately, project Makefiles take into account the special symbol needs of dynamic projects. If your project includes bundle projects, the main executable retains its symbol table. By

default, this symbol table includes entries for all classes defined in the main application, plus all classes in the libraries you link against. For functions, the table has entries for all functions in the main application and any library functions that the main application calls; it also has entries for *all other* functions defined in *any* library member that includes a class definition or function that the main application calls. This means that dynamic modules can use all of your classes and all Kit classes, and can call your functions and most of the common library functions.

Unfortunately, it's difficult to anticipate exactly which functions a dynamic module will need to call. For instance, suppose you have a musical composition app. As originally designed, this
application might not use many mathematical functions, so its symbol table wouldn't include entries for functions like **lgamma()** and **j0()**. But if you later wanted to supply a loadable module that analyzed musical waveforms, your module might need these functions. To ensure that modules can call any library functions, you can use the linker flag **-all_load** to force inclusion of all library symbols in your main app's symbol table.

To use **-all_load**, add the macro **OTHER_LDFLAGS=-all_load** to the **Makefile.preamble** in your main project.

The main issue here is the API you want to provide to loadable modules. If you leave a complete symbol table, then modules will have access to all classes and functions used by the main program. This may be undesirable if you have some private classes that you don't want modules to use, or if there are library functions that you want to prevent modules from calling. If this is the case, assemble a list of symbols that loaded modules should be able to access. Then use the **-s** option of **strip** to strip your main executable so that it provides the desired API.

A secondary issue is the effect of the symbol table on the file size of your main executable. The default symbol table mentioned above will increase the size of your program roughly 110 kilobytes per architecture for the library symbols, plus a variable amount for the program's own symbols.

If you link against additional libraries and load all library symbols, the size penalty could easily double. But it's important to realize that this is the program's static file size as it sits on the disk; the amount of virtual memory it consumes when running is affected by a number of additional factors, discussed later. However, if small program file size is a critical requirement of your application, you might consider trimming the symbol table.

Traveling time and space

As noted earlier, dynamic loading can improve your application's launch speed and reduce its memory consumption. But there is a subtle interaction at work regarding the mechanics of dynamically linking code files. When a program is launched, only the pages of code that are actually executed get swapped into memory. This is very fast and efficient, because it limits memory consumption to the minimum set of code a user needs on a particular occasion.

Dynamic loading throws a monkey wrench into the works. When the dynamic loading system loads a module, it needs the symbol table of the main program to resolve any external references in the loadable code file. So, the first time you load a module, the system builds the memory image of the symbol table. To do this, it must map the entire main executable into memory. This is inefficient, because it swaps in all pages of code of the main application, regardless of whether the user needs them. It also reduces performance while the system is occupied constructing the table. The more symbols involved, the longer it takes.

You can approach this problem from a couple of directions. One option is to make the main program as small as possible by moving code to loadable modules. This will minimize the number of pages in the main executable that the system is forced to swap in when it builds the symbol table before the first load. However, that may not be feasible for all applications.

The other option is to reduce the number of entries in the main program's symbol table. This reduces the number of symbols that the system needs to process as it constructs the table. But it's important to remember that these are the tradeoffs you make after you choose dynamic loading.

Because of these caveats, you probably shouldn't use dynamic loading solely for the potential performance enhancements. The compelling motivations for dynamic loading are the flexibility, extensibility, and ease of maintenance it brings to your projects. Performance wins are just a nice possible side effect.

Odds and ends

There are a few other simple mistakes to watch out for the first time you try dynamic loading. Plan your interface layout so that it can accommodate an arbitrary number of loaded modules, not just the few that you expect to ship. If you give your modules an extension other than the default **.bundle**, add that extension to the list of extensions your application owns, so that the directories will look like file packages in Workspace Manager. Make sure that your primary class is the first one in each module's loadable file. If your app has multi-architecture binaries, be aware that under NEXTSTEP 3.0 the main program will run, but the multi-architecture modules won't loadÐonly modules created for NeXT computers will load under 3.0.

CROSS-EYED AND PAINFUL?

If all this dynamic mumbo jumbo makes your head spin, relax. Working with dynamic loading requires a bit of a conceptual shift, so it may take some time before you are comfortable using it in your code. To help you get started, an example project accompanies this issueÐa simple graphical display application called DynaDoodle. It displays various dynamically loaded doodle modules. The user can choose which module to display and adjust graphical characteristics of the doodle.

In addition, we'll revisit dynamic loading in a future issue of *NXApp*. At that time, we'll further probe the subtle technical issues and investigate some advanced techniques. Until then, have fun exploring dynamic loading in your programming projects.

Andrew Vyrros is Director of Development at Codeworks, an independent NEXTSTEP consulting firm in San Francisco. His recent projects include a dynamically loaded data visualization

engine. You can reach him by e-mail at **av@codeworks.com** or by phone at (415) 626-7144.

DYNAMIC LOADING TERMINOLOGY

Some of the terms used in dynamic loading have multiple, overlapping meanings. To simplify matters, this article gives a single, specific meaning to each term.

bundle An overloaded term that can mean loadable module, file package, or an instance of the NXBundle class. In this article, a bundle always means an NXBundle instance.

bundle project In Project Builder, a kind of subproject that manages the source files and other resources that are used to make a loadable module.

dynamic module See *loadable module*.

file package A directory that packages together a set of related resource files. In Workspace Manager, file package directories appear as simple files. Also called a *bundle*.

loadable code file A kind of executable file that contains some compiled code, typically Objective C class definitions. It's different from a normal executable file in that it's relocatable, which means it includes the extra symbol information needed to join it with another executable. Also called a *loadable object file* or a *relocatable object file*.

loadable module A chunk of resources that gets loaded into a program at run-time. Each loadable module generally consists of two primary components: a loadable object file, and the accompanying resources such as images and interfaces. Also called a *dynamic module*. Sometimes called a bundle, although a bundle can contain any kind of resources, not just loadable code.

NXBundle A NEXTSTEP Common class used to access the resources inside a file package. NXBundle knows about loadable module packages and how to load code files into applications.*-AV*

WHERE TO STORE AND FIND MODULES

Before you can load any dynamic modules you'll need to find them in the file system. At the very least, you

should look for modules inside of the application's **.app** file package. To make it easy for users to add new modules to the system, you should also establish some other standard locations where you'll look for modules. There are no hard rules, but a de facto standard has emerged that you'll probably want to adhere to.

The system is to specify an identifying filename extension for your modules and identify a subdirectory within the standard library directories where you expect modules to be placed. Typically this subdirectory has the name of the application. You search the library directories in this order: the user's home library, **~/Library**; the site's library, **/LocalLibrary**; and the NeXT-supplied library, **/NextLibrary**. Finally, after the library directories, you look in the application file package for the modules that come with the application.

For instance, say you have an app called Engine Builder that loads modules of auto parts. Using this system, you would search for all the modules with extension **.automodule** in the following directories: **~/Library/EngineBuilder**, **/LocalLibrary/EngineBuilder**, **/NextLibrary/EngineBuilder**, and **EngineBuilder.app**. This system is analogous to the one NEXTSTEP uses for fonts and other resources. It allows sites to install modules that enhance or override the default configuration, and it lets individual users provide their own, private modules.

Depending on your application's needs, you may want to use a slight variation on this plan. In this situation, you split your resources into further subdirectories within your application's directories. As an example, you might have Engine Builder look for **.autotool** files inside of the **EngineBuilder/AutoTools** directories in the user, site, and NeXT-supplied Library directories, and load **.automodule** files from the **EngineBuilder/AutoModules** directories also in the set of Library directories. This approach can be helpful when your application has a lot of resources to manage.*ĐAV*

LAYING OUT MODULES WITH PROJECT BUILDER

Dynamic modules often require accompanying files in addition to the loadable code file: resources like images, interfaces, and string tables that are needed once the modules are loaded. To keep your modules organized, group the files of each module into a file package. If you lay out your modules this way, you can use the NXBundle class to do your loading, and Project Builder for your file management.

NXBundle expects dynamic modules to look like this: The file package should have a name that includes an identifying extension. Inside the package should be the loadable code file, with the same name as the file package but without the extension. For instance, if your module is named **Engine.automodule**, the code file should be named **Engine.automodule/Engine**. Additional resources like images and class name lists can

also go inside the file package. Localized resources such as interfaces and string tables should be in **.lproj** subdirectories within the package.

Project Builder automatically sets up your modules just this way. The technique is to create a separate bundle project for each dynamic module. Then add the classes and resources that you want to be part of a particular module to the corresponding bundle project. The generated Makefiles do the rest of the work.

To add a new bundle project to your application project, in the Files view choose New Subproject from the Project menu. Specify the name of the new module without an extension and select the type Bundle. This adds a new bundle project as a subproject of your main project. To add resources to a bundle project, select it in the Files view and drag-and-drop it.

Project Builder can also control which class in inserted first by the linker into the module's loadable code file. This determines which class NXBundle identifies as the principal class, the entry point to the rest of the module. To make a class the first class, control-drag it to the top of the list in Project Builder's Classes browser.

When you build the project, the Makefiles perform all the tasks necessary to create a properly configured module. All files in the Classes and Other Sources lists are compiled and linked with the appropriate flags into a single loadable code file with the proper name and location. Localized resources are copied to the appropriate **.lproj** directories and other resources are copied to the file package. The package itself is inserted as a subdirectory inside the main application package.

By default, the Makefiles give module packages the extension **.bundle**. Directories with this extension look like generic module files in Workspace Manager™. If you want a unique look for your modules, you can supply your own extension. Starting in 3.2, Project Builder allows you to set the extension for a module. In 3.0 or 3.1, you'll have to use some custom rules in your Makefile.preamble to make the change. If you do this, add the new extension and icon to the application's list of file types to make your module packages appear as custom module files in Workspace Manager.*ĐAV*

Next Article NeXTanswer #1504 **An Informal Approach to Object-Oriented Design**
Previous article NeXTanswer #1499 **Automated Testing of NEXTSTEP Applications**
Table of contents
http://www.next.com/HotNews/Journal/NXapp/Winter1994/ContentsWinter1994.html