Layered Development with (Unix) Dynamic Libraries

Yannis Smaragdakis
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332
yannis@cc.gatech.edu

Abstract. Layered software development has demonstrably good reuse properties and offers one of the few promising approaches to addressing the *library scalability problem*. In this paper, we show how one can develop layered software using common Unix (Linux/Solaris) dynamic libraries. In particular, we show that, from an object-oriented design standpoint, dynamic libraries are analogous to components in a mixin-based object system. This enables us to use libraries in a layered fashion, mixing and matching different libraries, while ensuring that the result remains consistent. As a proof-of-concept application, we present two libraries implementing file versioning (automatically keeping older versions of files for backup) and application-transparent locking in a Unix system. Both libraries can be used with new, aware applications or completely unaware legacy applications. Further, the libraries are useful both in isolation, and as cooperating units.

1 Introduction

Factored libraries have long been considered one of the most promising approaches to software reuse. Factored libraries are motivated by what Biggerstaff [7] calls the *vertical/horizontal scaling dilemma*. According to this dilemma, libraries should incorporate significant parts of functionality to be worth reusing, but then they become very specific and, hence, less reusable. A factored (or *layered* [3]) library attempts to address the problem by encapsulating a number of distinct components, each implementing a different axis of functionality. These components can then be put together in an exponential number of legal configurations. The selection of components adapts the library functionality to the needs of a specific application.

Several concrete technologies for implementing factored libraries have been proposed in the past. Application generators [2][5], template libraries [21], and binary components are among them. In this paper, we show that standard Unix dynamic libraries are a good architecture for factored or layered development. Using dynamic libraries to represent library components has the advantages of language independence (components can be created from distinct languages), intellectual property protection (dynamic libraries are binary components), and load-time configurability (combining different dynamic libraries is done at application loading time, not at compile time).

The basis of our argument is the observation that a dynamic library model can be viewed as an object system.[1] That is, a dynamic library architecture supports all the

standard properties required to identify a system as object oriented: inheritance, encapsulation, and overriding. Furthermore, as we will show, other common features of object models can be identified in dynamic libraries: a library can choose to call functionality of its parent library ("super" call) and can choose to call functionality in another named library (call through a reference). Indeed, even some not-too-conventional features of object systems are supported by the dynamic library model: "inheritance" is done late (at load time), making the dynamic library model resemble a "mixin-based" [8] object system. Furthermore, different copies of a library can participate in the same inheritance hierarchy.

The conceptual mapping of dynamic libraries to object-like entities is one of the contributions of this paper. Based on this, we translate well-known techniques for layered design in object-oriented languages into analogous techniques for dynamic libraries. The result is a style of dynamic library development that is very useful but unconventional: it requires constant consideration of whether a call should be dispatched "dynamically" (i.e., could be allowed to be overridden by child libraries in the inheritance hierarchy), "statically" (within the current library), or "by chaining" (by forwarding to the parent library in the hierarchy). Dynamic libraries that are designed with the understanding that they can be used in a layered fashion are modular and can be used in many combinations with other libraries. Conventional (*legacy*[2]) Unix dynamic libraries are rarely careful to cooperate with other libraries that may be overriding some of their symbols.

We should point out early on that the term "library" is overloaded in our discussion. When we talk of a "dynamic library", we mean a single file implementing a collection of routines. When we talk of a "layered" or "factored library" we mean a collection of components that are designed to be composed together in many configurations. Under the methodology we are proposing, a single "dynamic library" is just one of the components in a "layered library".

As a proof of concept for our approach, we present two dynamic libraries that can be viewed as different components of a factored library for file operations. The two libraries are designed to be used either together or in isolation. Both libraries perform system-level tasks, that are, however, commonly handled at the application level in Unix. The first library performs transparent file versioning: every time an "interesting" file is modified or erased, its old version is saved for backup. The second library performs file locking, so that inconsistent file editing is prevented. Both libraries can be used either with new applications, or with legacy applications. In the latter case, they can

---

1. For readers familiar with Unix dynamic libraries, a clarification and forward pointer is in order: our arguments are based on the use of the `LD_PRELOAD` technique instead of the more conventional path-based approach to using dynamic libraries.
2. We will use the term "legacy" to denote pre-existing Unix libraries and applications (i.e., binary objects that were not developed following the style of programming described in this paper). We do not assign any negative connotation to the term.

provide backup and locking functionality for existing executables (i.e., without recompilation). Source code for both libraries is available at our web site.

The rest of this paper is structured as follows. Section 2 introduces Unix dynamic libraries and shows how they can be viewed in object-oriented terms. Section 3 discusses our two example applications and their structure as layers of a consistent library. Section 4 presents related work and Section 5 offers our conclusions.

2 Dynamic Libraries

In this section we introduce dynamic linking/loading, with special emphasis to the (not too common) use of the `LD_PRELOAD` environment variable in modern Unix variants. We then show how this technique supports layered software development, by enabling us to simulate a flexible object-oriented programming model.

Our examples have been tested on Linux and Solaris using several different versions of these operating systems over the past 2 years. Hence, reasonable stability can be expected. Most other modern Unix variants (e.g., FreeBSD/OpenBSD/NetBSD, AIX, Mac OS X) support dynamic libraries, but we have not personally tested them.

2.1 Background

Dynamic linking/loading is a common technique in modern operating systems. Under dynamic linking, an executable program can call routines whose code exists in a dynamic library. The dynamic library is loaded at execution time into the address space of the program. Routines from a dynamic library are identified by symbols and it is the responsibility of the dynamic linker to match the symbols referenced in the executable to the symbols exported from a dynamic library. The main advantage of dynamic libraries is that the executable does not need to be burdened by including common library code. This results into smaller executables, thus saving disk space. More importantly, it also enables keeping a single copy of the library code in the system memory during run-time, even though the code may be used by multiple executable files (or even by the OS kernel itself).

Another advantage of dynamic libraries is that they avoid hard-coding dependencies on library code. Instead of statically linking to a certain version of the code, which prevents exploiting future improvements, dynamic linking takes place at program load time. Thus, different libraries (possibly newer and improved or just alternative implementations) can be used. This enables modularity and it has been one of the main reasons why dynamic libraries are the technology of choice for binary object systems, like Microsoft's COM (e.g., see [9]). It is worth mentioning that this flexibility of dynamic libraries has also been the source of problems when library versioning is not managed carefully—the term "DLL hell" has become standard terminology.

Typically the search for libraries is path based: the name of the library exporting a certain symbol is fixed at build time, but by changing the lookup path for the library file,

different dynamic libraries can be linked. For example, in Unix systems, an executable program calling routine `rout`, implemented in library `libroutines.so`, can be built with the `-lroutines` flag to signify that dynamic linking should be done to a file called `libroutines.so`. The user can then influence the process of finding the file by changing the environment variable `LD_LIBRARY_PATH`, which lists directory paths to be searched by the linker in order to find `libroutines.so`.

A very powerful feature of Unix dynamic libraries is the ability to interpose implementations of routines *before* the actual path based symbol lookup takes place. This is done by using the `LD_PRELOAD` environment variable. `LD_PRELOAD` can be set to a list of dynamic library *files* (not directory paths) that are linked before any other dynamic libraries. Effectively, symbols from the libraries in the `LD_PRELOAD` variable take precedence over any normal dynamic library symbols. We show examples of the use of `LD_PRELOAD` in the following section.

2.2 Dynamic Libraries and the Object-Oriented Model

The common attributes of an object-oriented (OO) system are encapsulation, inheritance, and overriding. As we will see, all of these and more are supported by dynamic library technology.[3]

The object model supported by dynamic libraries is not exactly like that of mainstream OO languages like C++ and Java: such languages are class-based, while the object model we discuss here is not. Instead, the dynamic library object model is closer to a delegation-based binary object model, like COM [9]. The use of the `LD_PRELOAD` variable is particularly interesting, as it enables late composition of dynamic modules.

Encapsulation. Dynamic libraries offer encapsulation: they contain the implementations of multiple routines that can be handled as a single unit. Data hiding is supported: routines are distinguished into those that are exported (*external symbols*) and those that are not.

The default resolution of symbols is to routines in the library itself. Thus, if a dynamic library references symbol `sym` and provides an implementation of `sym`, the library's own implementation will be used. Binary code can call the routines encapsulated in a different dynamic library, as long as a "reference" to the library exists. The reference can be obtained using the library pathname. For instance, calling routine `f` of library `lib` is done by explicitly opening the library and looking up the appropriate routine:

```
libhandle = dlopen("lib", RTLD_LAZY);
meth = (methtype *) dlsym(libhandle, "f"); // cache it in meth
meth(arg1); //or whatever is the regular signature of "methname"
```

3. Unix dynamic libraries are commonly also called "shared objects" (hence the `.so` file suffix). This is a fortunate coincidence, since, to our knowledge, the term "object" was not used in the object-oriented sense.

Inheritance. Dynamic libraries support inheritance:[4] a library can behave as if it were automatically "inheriting" all symbols exported by ancestor libraries in a hierarchy. A hierarchy is formed by putting the dynamic libraries in sequence in the value of `LD_PRELOAD`. For instance, consider setting `LD_PRELOAD` to be (`csh` syntax):

```
setenv LD_PRELOAD "$DLIBHOME/C.so $DLIBHOME/B.so $DLIBHOME/A.so"
```

This establishes a linear hierarchy of libraries, each having a notion of a "next" library in the hierarchy. In terms of inheritance, library `C.so` inherits from library `B.so`, which inherits from library `A.so`. All symbols of `A.so` can be referenced by code in `B.so`, etc.

Overriding. A dynamic library automatically overrides symbols from ancestor libraries in the hierarchy. "Dynamic" dispatch (really a misnomer, since the resolution occurs at load time) is effected by looking up a symbol using the `RTLD_DEFAULT` flag in the `dlsym` call:

```
virt_meth = (methtype *) dlsym(RTLD_DEFAULT, "methname");
virt_meth(arg1); // call the most refined method
```

This ensures that the lookup for the symbol proceeds through the dynamic libraries in order, beginning at the final node of the hierarchy. Thus, the overriding method is retrieved, not any overridden versions.

As usual in an OO hierarchy, code in one node can explicitly call code in the next node of the hierarchy (instead of calling its own "overridden" version of the code). Such "parent" calls are effected by looking up the symbol using the `RTLD_NEXT` specifier in the `dlsym` call:

```
super_meth = (methtype *) dlsym(RTLD_NEXT, "methname");
super_meth(arg1); // call the method
```

2.3 Layered Development with Dynamic Libraries

Dynamic libraries form an excellent platform for layered software development. This has already been exploited in limited ways. Windows dynamic libraries are the technology that supports Microsoft's COM. In Unix, there are some applications that extend their capabilities using dynamic loading (e.g., the Apache web server [1]). Nevertheless, to our knowledge, there is no factored library with its components implemented as dynamic libraries. That is, although large, monolithic dynamic libraries have been used successfully, no consistent array of functionality has been implemented as a collection of small dynamic libraries *all designed to cooperate* using load-time inheritance hierarchies.

---

4. The term "aggregation" would perhaps be more appropriate than "inheritance", since the latter is used to describe relationships between classes. Nevertheless, we prefer to use the term "load-time inheritance" or just "inheritance" to appeal to the reader's intuition.

The technology for such a coordinated interaction is already there. Indeed, the object model offered by dynamic libraries is close to a *mixin-based* model—a technology that has been used in layered libraries in the past, most notably in the GenVoca methodology [2]. *Mixins* [8] are classes whose superclass is not specified at mixin implementation time, but is left to be specified at mixin use time. The advantage is that a single mixin can be used as a subclass for multiple other classes. This is similar to what we obtain with dynamic libraries using the `LD_PRELOAD` variable. A single library can refer to "parent" functionality and to "overriding" functionality, but it is not aware of the exact hierarchy in which it participates. The same library can be used in many different hierarchies. The same symbols will be resolved to refer to different code depending on the exact hierarchy.

Consider, for instance, a factored library containing 6 dynamic library components, named `A.so` to `F.so`. Each of these components can encapsulate a different feature, which may be present or absent from a given component composition. All components should be designed with interoperability in mind. Thus, every call to a routine `f` should be carefully thought out to determine whether it should be a call to a routine in the same library (calling known code), a call to the parent library's routine (delegating to the parent), or a call to the overriding version of the routine (allowing the interposition of functionality by all other libraries in the hierarchy). This is the same kind of analysis that goes into the implementation of a mixin-based library.

The advantage of factored libraries is that they can be used to implement a number of combinations that is exponential in the number of components in the library. Each of the combinations is not burdened by unneeded features, yet can be as powerful as needed for the specific application. For instance, a composition of components `A`, `B`, and `E` (henceforth denoted `A[B[E]]`, using GenVoca layer notation [5]) is effected by appropriately setting the `LD_PRELOAD` variable:

```
setenv LD_PRELOAD "$DLIBHOME/A.so $DLIBHOME/B.so $DLIBHOME/E.so"
```

The order of composition could also be important: compositions of the same components in a different order could result into different, but equally valid implementations.

In earlier work [22], we have shown the reuse advantages of layered libraries compared to other object-oriented technologies. Compared to OO application frameworks [15], for instance, layered libraries offer a much more compact representation of feature sets of similar complexity. In our experience, dynamic library technologies can offer full support for layered development. For instance, some important issues in layered development can be handled as follows:

- A major issue in layered libraries is ensuring that a composition is valid. Components commonly have requirements from other components participating in a composition. For instance, in a data structure factored library (like DiSTiL [20]) we can require that a storage policy component be at the root of the component hierarchy. Such requirements are often expressed in a component-centric way: each component exports some boolean flags asserting or negating certain (library-

specific) properties. At the same time, components can enforce requirements on the union of all properties of components above them or below them in a component hierarchy [4]. For instance, component `A` can require that some component above it implement the `storage` property. If component `B` exports the property, then composition `A[B]` is valid.

Dynamic libraries can support automatic checking of properties at load time. By convention, the library can contain a special initialization function called `_init`. This function is called by the dynamic loader to perform library-specific initialization. Properties can be exported by libraries as symbols. For instance, a simple requirement can be expressed as:

```
void _init() {
  assert(dlsym(RTLD_NEXT, "property1"));
}
```

This ensures that a library above the current one in the component hierarchy exports a symbol called "`property1`". Using this technique, a factored library developer can add complex restrictions on what compositions are valid. The restrictions are checked early: at application (and library) load time, and not when the library functionality is called. It is the responsibility of the layered library author to express the dependencies among components as restrictions of the above form.

- A common feature of layered libraries is that layers can be instantiated multiple times in a single composition. At first, this may seem paradoxical: why would the same code be included more than once in a composition? Nevertheless, the code is actually parameterized by all the components above the current one in a component hierarchy. Thus, multiple copies of the same code can be specialized to perform different functions. Consider, for instance, a multi-pass compiler, where one of the passes is implemented as a component called `process_tree`. If the typechecking phase must be completed before reduction to an intermediate language takes place, then a reasonable composition would be:
  `process_tree[typecheck[process_tree[reduce] ] ]`.

  Dynamic libraries can handle multiple instances of a library in the same composition. In the worst case a brute-force approach (which we had to use in Solaris) is needed: the dynamic library file needs to be copied manually. In Linux, however, the same library can be used multiple times in an `LD_PRELOAD` hierarchy without problems.

- Layered library development requires a composition mechanism that imposes a low performance penalty for calling code in different layers. Indeed, Unix dynamic libraries have emphasized fast dispatch. A typical Unix loader will resolve symbols at load time and employ binary rewriting techniques to ensure that future invocations are performed at full speed, instead of suffering lookup cost dynamically on every invocation [17]. Although, there is still overhead from employing layering (e.g., routines from different layers cannot be inlined) the overhead is kept reasonably small. Additionally, the expected granularity of com-

ponents developed using dynamic library technology is large: for fine-grained components, a source-code-level technique is more advantageous. Therefore, the overhead of layering using dynamic libraries is negligible.

Based on the above observations, we believe that dynamic libraries are a good technology for implementing layered libraries. The question that arises is why one should prefer dynamic libraries over other layering technologies. Compared to source code component technologies, dynamic libraries have the usual advantages of binary level components. First, dynamic libraries are language-independent: they can be created in many languages and used by code in other languages. Second, dynamic libraries are binary components, offering intellectual property protection. Furthermore, dynamic libraries have a unique feature compared to all other component technologies (binary or source level): their ability for load-time configurability. This ability yields a lot of flexibility in future updates, but also in operation with legacy code. For instance, dynamic libraries interposing on well-known symbols (e.g., from the `libc` library) can be used with completely unsuspecting pre-compiled applications.

3 Example Applications

To demonstrate the potential for layered development using dynamic libraries, we will discuss two libraries that we designed as parts of a transparent "file operations" layered library. We should point out that our code is not yet a mature and feature-rich layered library. In fact, our two libraries are not an ideal example of layered library components, as they are only loosely coupled. Nevertheless, our libraries are actual, useful examples. They serve as a basic proof-of-concept by demonstrating almost all of the techniques described in Section 2. Our source code can be found in:
`http://www.cc.gatech.edu/~yannis/icsrcode.tar.gz`.

3.1 Versioning Library Overview

Typical Unix file systems do not offer automatic backup capabilities. Unix programs commonly resort to application-level solutions when they need to keep older versions of files when these are modified. For instance, the Emacs text editor and the Framemaker word processor both automatically create a backup file storing the previous version of an edited file. Advanced and general solutions have been proposed at the kernel level—for example, see the report on the Elephant file system [18] and its references. Nevertheless, it is relatively easy to come up with a good, quite general, and fairly OS-neutral solution at the user level using dynamic libraries. Our versioning dynamic library interposes its own code to the symbols wrapping common system calls, like `open`, `creat`, `unlink`, and `remove`. By setting `LD_PRELOAD` to point to the library, we can use it with completely unsuspecting legacy applications. The library recognizes "interesting" file suffixes and only acts if the file in question has one of these suffixes. Any attempt to modify (as opposed to just read) a file through one of the calls implemented by the library will result in a backup being created. Thus, unlike the usual "trash can" or "recycle bin" functionality, our library protects both against deletion and against overwriting with new data. Backup versions of files are stored in a "`.ver-`

sion" subdirectory of the directory where the modified file exists. We have put this library to everyday use for source code files (.c, .h, .cpp, .hpp, .cc, .hh, and .java suffixes) text files (.txt), etc.

An interesting issue in versioning functionality is which of the older versions are worth keeping. The Elephant file system [18] allows users to specify policies for keeping older versions. Our library is primitive in this respect: it only keeps a fixed number of the most recent back versions (currently only one, but this can easily change). An interesting future improvement might be to provide versioning policies as other components in our factored library—that is, as dynamic libraries. Then, the user will be able to select the right policy at load time, by composing the versioning library with policy libraries through an LD_PRELOAD component hierarchy.

3.2 Locking Library Overview

File locking is another piece of functionality that (although to some extent supported by Unix file systems) is commonly left for the application to provide. (File locking in Unix is a big topic—e.g., see Ch. 2 of the Unix Programming FAQ [11]—and our homegrown implementation is certainly not a general solution.) File locking intends to protect files from concurrent modification and to protect applications from inconsistent file views. Application-specific locking protects against access to a file by different instances of the same application, but does not prohibit access by different applications. The Emacs text editor and the FrameMaker word processor are, again, good examples of applications that provide their own locking implementation.

It should be noted that most text-oriented Unix applications do not operate by keeping files open for long periods of time. Instead, applications processing a file first make a temporary copy of the file, on which all modification takes place. Eventually, the temporary file is copied over the original, to reflect the changes. This upload/download-like approach provides some protection against inconsistent modification, but is not feasible in the case of large files (e.g., multimedia files).

Our file locking library works by overriding file operations like open, close, and creat. Just like our versioning library, the interposed code checks if the file in question is an "interesting" file. The library implements a readers/writers locking policy: multiple open operations are allowed on a file, as long as they are all read-only accesses. Any other concurrent access is prohibited. Thus, our locking is "mandatory" (but only for applications executing with our library in the LD_PRELOAD path) while common Unix locking mechanisms are "advisory" (i.e., they require application participation). Normally, our locking policy would perhaps be too strict. Nevertheless, it only becomes restrictive in the case of large files that are opened "in place". (The only other reasonable alternative in this case would be no locking whatsoever.) For the common case when a temporary copy of the file is created, our locking policy just prevents inconsistent write-backs (interleaved write operations by different processes to different parts of a file).

Locks and shared data (e.g., number of readers/writers) are stored in the file system, as files under a `.lock` subdirectory of the directory where the interesting file is found.

3.3 Implementation and Discussion

The locking and versioning libraries described above employ most of the techniques discussed in Section 2. Although the libraries are loosely coupled, they are designed to cooperate, as they interpose on many of the same symbols. Thus, they can be regarded as components in a simple layered library. The two libraries can be used together or individually on an application.

The main difficulty during library development has to do with identifying which procedure calls should conceptually refer to potentially "overridden" functionality, which should refer to functionality in the same library, and which should just be delegated to the parent library in the component hierarchy (or any other dynamic library through normal, path-based lookup).

To facilitate programming in this way, each library initializes a set of "super" implementations for all the symbols it overrides. For instance, the locking library contains initialization code like:

```
super_open = (Openfn) dlsym(RTLD_NEXT, "open");
super_close = (Closefn) dlsym(RTLD_NEXT, "close");
super_creat = (Creatfn) dlsym(RTLD_NEXT, "creat");
...
```

The `super_open`, etc., function pointers are static global variables, accessible from all the library routines. They are often used when normal, non-layered code would just call `open`, `close`, etc. For instance, the locking library creates a "pre-locking" file using a Unix exclusive file creation operation. The "pre-locking" file serves to ensure that no two processes try to access the locking shared data (i.e., numbers of readers/writers) at the same time. The code for that operation is:

```
lock_fd = super_open(extended_path, O_WRONLY | O_CREAT | O_EXCL,
                                    S_IRUSR | S_IWUSR | S_IXUSR);
```

The most interesting interaction between layers is the one that occurs when a library calls a routine that is potentially overridden. Recall that this is analogous to a "dynamically bound" call in the object-oriented model. A good example of such a use can be found in the finalizer routine of the locking library. Since many processes do not explicitly `close` files before they exit, we tried to approximate the correct functionality by calling `close` on all open files when the library is finalized. This will ensure that the locking library correctly updates its locking information. Nevertheless, the call to `close` does not only concern the locking library, but also any other dynamic libraries loaded in the process. Thus, the call to `close` should be to the overriding method of the `close` routine. A slightly simplified version of our finalizer code is shown here:

```
void _fini() {
  Closefn virt_close = (Closefn) dlsym(RTLD_DEFAULT, "close");
  while (open_files != NULL) {
    open_file_data *next = open_files->next;
    virt_close(open_files->open_fd);
    open_files = next;
  }
}
```

(As can be seen in the above, the locking library has state: it keeps track of what files are open at any point.)

Finally, we should give a warning. Both the locking and the versioning libraries are based on interposing code on symbols used by existing programs. The disadvantage of this approach is that completeness is hard to guarantee. It is easy to miss a symbol that offers a different way to access the same core functionality. Even in the case of well-defined OS interfaces, there is potential for surprise: our first implementation of the versioning library missed the `open64` symbol, used in Solaris as part of a transitional interface to accessing large files. Executables compiled to use the `open64` symbol circumvented that early version of our library.

4 Discussion and Related Work

There has been a lot of research work presenting advanced techniques for software reuse. This includes work on generators and templates [10], transformation systems [6][16], language-level component technologies [19], module and interconnection languages [12][23], and much more. Our emphasis in this paper was not on proving that an overall approach to software design has good reuse properties. Instead, we adapted the existing approach of scalable libraries and layered designs to a different technology. The benefits of scalable libraries are well established [3][21]. We argued that most of these benefits can be obtained when Unix dynamic libraries are used as the underlying concrete technology.

Despite the emphasis on Unix systems throughout this paper, dynamic libraries are part of all modern operating systems. It may be feasible, for instance, to use some of our ideas in a Windows environment. Nevertheless, our emphasis was on the use of the `LD_PRELOAD` variable, which allows (even a third-party user) to specify compositions simply and concisely. No analogous mechanism exists on Windows systems. The difference between using `LD_PRELOAD` and using a path-based lookup mechanism (not only in Windows, but also in Unix variants) is in convenience and transparency. With path-based lookup, libraries need to have specific names, already known by the precompiled executables. Directories have to be set up appropriately to enforce a search order. Finally, to our knowledge, in Windows systems, there is no way to separate the lookup path for dynamic libraries from the search path for executables.

We should also mention that interposing dynamic libraries through the `LD_PRELOAD` variable raises some security concerns. For instance, there are commonly restrictions

on what libraries can be dynamically linked to set-user-ID or set-group-ID executables. All of these restrictions, however, are orthogonal to the work presented in this paper: they have to do with the general issue of trust of binary programs. Linking a dynamic library is certainly no more dangerous than running an executable program.

Although only tangentially related, we should mention that a lot of work has been done over the years on layered operating system development. The *microkernel* approach is the best known representative of such research, and several object-oriented microkernels (e.g., Spring [14] and recently JX [13]) have been developed. Although conceptually related, the operating systems modularization work deals with completely different concerns (performance and hardware resource management) from this paper.

5 Conclusions

In this paper we argued that Unix dynamic libraries (or "shared objects") are a good platform for implementing layered designs. The basis of our argument is the observation that dynamic libraries offer exactly analogous mechanisms for interaction between libraries in a library hierarchy, as those offered for interactions between classes in an object-oriented inheritance hierarchy. Furthermore, the establishment of a dynamic library hierarchy is done at load time, allowing great configurability.

We believe that the dynamic library technology can form the basis for mature, industrial-strength factored libraries. Although many factored libraries have been produced so far, few are used in practical settings and most could benefit from the unique features of dynamic library technology (e.g., binding with legacy programs without recompiling). Similarly, although many mature dynamic libraries are in use, no consistent collection of cooperating dynamic libraries, allowing mix-and-match configurability, has been developed. Our work makes a first step in this promising direction.

Acknowledgments. This work was partially supported by DARPA/ITO under the PCES program.

6 References

[1] Apache HTTP Server Documentation Project, "Version 2.0: Dynamic Shared Object (DSO) Support", available at `http://httpd.apache.org/docs-2.0/dso.html`.

[2] D. Batory and S. O'Malley, "The Design and Implementation of Hierarchical Software Systems with Reusable Components", *ACM TOSEM*, October 1992.

[3] D. Batory, V. Singhal, M. Sirkin, and J. Thomas, "Scalable Software Libraries", *ACM SIGSOFT* 1993.

[4] D. Batory and B.J. Geraci, "Component Validation and Subjectivity in GenVoca Generators", *IEEE Trans. on Softw. Eng.*, February 1997, 67-82.

[5] D. Batory, "Intelligent Components and Software Generators", *Software Quality Institute Symposium on Software Reliability*, Austin, Texas, April, 1997.

[6] I.D. Baxter, “Design maintenance systems”, *Communications of the ACM* 35(4): 73-89, April 1992.

[7] T. Biggerstaff, “The Library Scaling Problem and the Limits of Concrete Component Reuse”, *1994 International Conference on Software Reuse*.

[8] G. Bracha and W. Cook, “Mixin-Based Inheritance”, *ECOOP/OOPSLA 1990*, 303-311.

[9] K. Brockschmidt, Inside OLE (2nd. ed.), Microsoft Press, 1995.

[10] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Techniques, and Applications*. Addison-Wesley, 2000.

[11] A. Gierth (ed.), *Unix Programming FAQ*, available at `http://www.erlenstar.demon.co.uk/unix/faq_toc.html`.

[12] J. Goguen, “Reusing and interconnecting software components”, *IEEE Computer*, February 1986, 16-28.

[13] M. Golm, J. Kleinoeder, F. Bellosa, “Beyond Address Spaces - Flexibility, Performance, Protection, and Resource Management in the Type-Safe JX Operating System”, *8th Workshop on Hot Topics in OS* (HotOS-VIII), 2001.

[14] G. Hamilton, P. Kougiouris, “The Spring Nucleus: A Microkernel for Objects”, *Sun Microsystems Laboratories Tech. Report*, TR-93-14.

[15] R. Johnson and B. Foote, “Designing Reusable Classes”, *Journal of Object-Oriented Programming*, 1(2): June/July 1988, 22-35.

[16] J. Neighbors, “Draco: a method for engineering reusable software components”, in T.J. Biggerstaff and A. Perlis (eds.), *Software Reusability*, Addison-Wesley/ ACM Press, 1989.

[17] C. Phoenix, “Windows vs. Unix: Linking dynamic load modules”, available at: `http://www.best.com/~cphoenix/winvunix.html`.

[18] D. Santry, M. Feeley, N. Hutchinson, A. Veitch, R. Carton, and J. Ofir, “Deciding when to forget in the Elephant file system”, *17th ACM Symposium on Operating Systems Principles* (SOSP’99).

[19] M. Sitaraman and B.W. Weide, editors, “Special Feature: Component-Based Software Using RESOLVE”, *ACM Softw. Eng. Notes*, October 1994, 21-67.

[20] Y. Smaragdakis and D. Batory, “DiSTiL: a Transformation Library for Data Structures”, *USENIX Conference on Domain-Specific Languages (DSL 97)*.

[21] Y. Smaragdakis and D. Batory, “Implementing Reusable Object-Oriented Components”, *5th Int. Conf. on Softw. Reuse (ICSR ‘98)*, IEEE Computer Society Press, 1998.

[22] Y. Smaragdakis, *Implementing Large Scale Object-Oriented Components*, Ph.D. Dissertation, University of Texas at Austin, December 1999.

[23] W. Tracz, “LILEANNA: A Parameterized Programming Language”, in Ruben Prieto-Diaz and William B. Frakes, editors, *Advances in Software Reuse: Selected Papers from the Secomd Int. Work. on Softw. Reusability*, 1993, IEEE Computer Society Press, 66-78.