EXHIBIT 21: SRINIVASAN IN VIEW OF BACH

"Advanced Perl Programming"
O'Reilly & Associates, Inc.
Author: Sriram Srinivasan
Publication Date: August 1997
("*Srinivasan*")

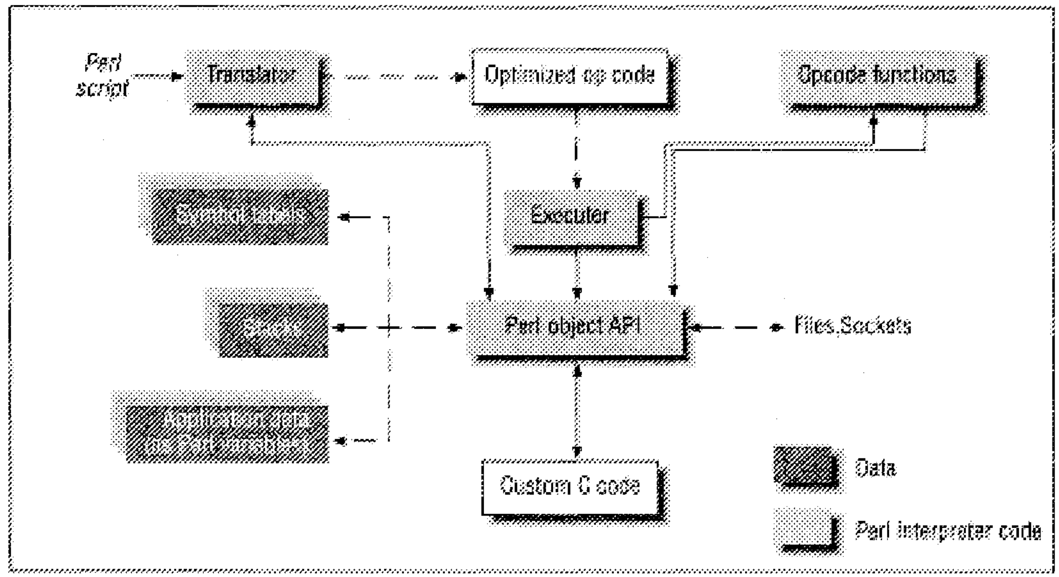
M. J. Bach, The Design of the Unix Operating System, Bell Telephone Labs., Inc. (1986) ("*APA-Bach*")

U.S. Patent No. 7,426,720	*Srinivasan* in view of *APA-Bach*
1. A system for dynamic preloading of classes through memory space cloning of a master runtime system process, comprising:	*Srinivasan* provided a system for dynamic preloading of classes through memory space cloning of a master runtime system process.
A processor; A memory	*Srinivasan* provided a system that ran on a computer with a processor and memory. For example, *Srinivasan* references an "in-memory cache of objects." *Srinivasan* at 178.
a class preloader to obtain a representation of at least one class from a source definition provided as object-oriented program code;	*Srinivasan* provided a class preloader to obtain a representation of at least one class from a source definition provided as object-oriented program code. *Srinivasan* discusses object oriented programming and its usefulness in conjunction with Perl code. *Srinivasan* at 101. *Srinivasan* notes that, in Perl, a "class is a package." *Srinivasan* at 389. *Srinivasan* then discusses two methods for the creation of a package from source definition provided as object oriented program code. *See id.* at 389-390 (discussing creation of the "employee" class). In order to preload the class, *Srinivasan* discloses specific functional code: "Using object package: use Employee; $emp = Employee->new ("Ada", 3 5); $emp->set_salary(1000); *See id.* at 390.

U.S. Patent No. 7,426,720	*Srinivasan* in view of *APA-Bach*
	"Java offers two levels of modularity: packages and classes, where a package is a collection of classes. (We'll learn about the notion of classes in the next chapter.) Perl's package is equivalent to both. Java does not allow one package to mess around with another package's namespace (no export) but allows a package to selectively import the classes it requires. It focuses a considerable amount of attention on security, which hasn't really stopped determined crackers. The Perl world has third-party packages called Safe and Penguin (which depends on Safe) that attempt to provide similar isolation characteristics (and don't offer any security guarantees either). Since the arrival of the Java Beans and the 1.1 version of the Java Development Kit ODK), Java has gained significant reflection capabilities, though nowhere near the amount of publicly available information Perl gives you. There are reasonably good arguments to be made both for providing this information and for not providing it; everything comes down to different models of programming. Men were sent to the moon while FORTRAN and COBOL ruled the roost, which proves that you can get a whole lot done if you don't indulge in language wars. Java allows you to dynamically "dispatch" a function call, by giving the function's name as a string, and to trap an exception if the function doesn't exist; this is like using Perl's symbolic references." *Srinivasan* at 98. "Object orientation (OO) is the latest software methodology to occupy the airwaves, hyped to a point where the term "object-oriented design" seems to automatically imply a good design. In this chapter, we will study what the noise is all about and build objects using Perl. I will leave it to the plethora of 00 literature to convince you that there is a respectable middle-ground and that the object revolution is indeed a good thing." *Srinivasan* at 99. "Objects of a certain type are said to belong to a *class*." *Srinivasan* at 101. "The translator converts a Perl script into a tree of opcodes (explained below). It comprises a hand-coded lexer (*toke.c*), the yacc-based parser (perly.y) , and the code generator (op.c). Regular expressions–which form a distinct sublanguage–are recognized in *toke.c* and compiled to an internal format in *regcomp.c*. Opcodes are similar in concept to machine code; while machine code is executed by hardware, opcodes (sometimes called byte-codes or p-code) are executed by a "virtual machine." The similarity ends there. Modern

U.S. Patent No. 7,426,720	*Srinivasan* in view of *APA-Bach*
	interpreters never emulate the workings of a hardware CPU, for performance reasons. Instead, they create complex structures primed for execution, such that each opcode directly contains a pointer to the next one to execute and a pointer to the data it is expected to work on at run-time. In other words, these opcodes are not mere instruction types; they actually embody the exact unit of work expected at that point in that program. Java and Perl are both examples of such interpreters. While many of java's bytecodes resemble a RISC machine's instruction set, Perl's opcodes represent a much higher level of abstraction. A large number of these opcodes directly correspond to the facilities available at the scripting level, such as regular expression matching and substitution, `chop`, `push`, `index`, `rindex`, `grep`,* and so on, which explains why there are 343 opcodes as of this writing! It also explains why Perl is so fast: instead of spending time in the interpreter, most of the work is done in lovingly hand-optimized e code. You can also see why it is hard to create a Perl-to-Java byte-code translator: there is no correspondence between the two sets." *Srinivasan* at 323-24. "Malcolm has also submitted a Perl compiler extension [5], which is in its early stages as of this writing. It can be asked to translate a script to C code, which can be compiled to form an executable; as it happens, this executable is not much faster than the interpreted script, because most of the action still takes place in opcode functions as they exist now. Static typing hints may usher in some aggressive optimizations. For example, if you say: `my integer $i;` the compiler would use C's native integer type, rather than an SV-this would speed up loops and arithmetic expressions. The compiler can alternatively produce a byte-code file and have the interpreter `eval` it subsequently, similar to the facilities provided by Python and Java. It also supports much better debugging options than those currently provided with *–D*." *Srinivasan* at 370.
a master runtime system process to interpret and to instantiate the representation as a class definition in a memory space of the master runtime	*Srinivasan* provided a master runtime system process to interpret and to instantiate the representation as a class definition in a memory space of the master runtime system process. *Srinivasan* discloses handling multiple clients by "[c]reat[ing] multiple threads of control." *Srinivasan* at 193. In order to create multiple threads of control, *Srinivasan* invokes the

U.S. Patent No. 7,426,720	***Srinivasan* in view of *APA-Bach***
system process;	fork( ) system call to create "process-level parallelism." *See id*. at 194. This creates a new process, called the child process. "The newly created child process meanwhile has a copy of its parent's environment and shares all open file descriptors." *See id.* Thus, *Srinivasan* discloses a "parent" runtime system process, *i.e.*, a master runtime system process. "The data structures described above are normally kept in global C variables. If Perl is compiled with *–DMULTIPLICITY*, it lumps all these global variables into a structure called Perl Interpreter. This allows you to have multiple instances of the interpreter, each with its own "global" space. (Recall from Chapter 19 the API to allocate and construct an object of type Perl Interpreter.) In the absence of this compile-time option, the Perl Interpreter object is a dummy structure, and the internal data structures are truly global, for maximum performance. The API remains the same in either case. You can use multiple interpreters to enforce completely isolated namespaces. Each interpreter has its own "main" package and its own tree of loaded packages. I have not seen this feature used in production Perl applications, but Tel provides a framework called SafeTel for security purposes, which uses a similar feature of multiple interpreter objects. These interpreters can be unrestricted or restricted. The equivalent module in Perl, Safe, uses a different mechanism, though the result (of isolated name spaces) is similar. More on this in the next section." *Srinivasan* at 323.

U.S. Patent No. 7,426,720	***Srinivasan* in view of *APA-Bach***
	Figure 20-1 shows the various components of a running Perl system. Shaded rectangles represent data structures, some of which can have multiple instances in a program. The source code can also be partitioned roughly along these lines.  *Figure 20-1. Snapshot of a running system* *Srinivasan* at 321.
a runtime environment to clone the memory space as a child runtime system process responsive to a process request and to execute the child runtime system process;	*Srinivasan* provided a runtime environment to clone the memory space as a child runtime system process responsive to a process request and to execute the child runtime system process. *Srinivasan* discloses handling multiple clients by "[c]reat[ing] multiple threads of control." *Srinivasan* at 193. In order to create multiple threads of control, *Srinivasan* invokes the fork( ) system call to create "process-level parallelism." *See id*. at 194. This creates a new process, called the child process. "The newly created child process meanwhile has a copy of its parent's environment and shares all open file descriptors." *See id.* Thus, *Srinivasan* discloses a "child" runtime system process, which is in fact a clone of the parent runtime system process.

<table>
<tr><th>U.S. Patent No. 7,426,720</th><th>Srinivasan in view of APA-Bach</th></tr>
<tr><td></td><td>

“Perl doesn't have threads yet (at least not officially*), but on Unix and similarly empowered systems, it supports `fork`, the way to get process-level parallelism. The server process acts as a full-time receptionist: it blocks on `accept`, and when a connection request comes in, it spawns a child process and goes back to `accept`. The newly created child process meanwhile has a copy of its parent's environment and shares all open file descriptors. Hence it is able to read from, and write to, the new socket returned by `accept`. When the child is done with the conversation, it simply exits. Each process is therefore dedicated to its own task and doesn't interfere with the other. The following code shows an example of a forking server:

```
# Forking server
use IO:: Socket;
$SIG{CHLD} sub {wait ()};
$main_sock = new IO::Socket::INET (LocalHost => 'goldengate',
                                   LocalPort => 1200,
                                   Listen => 5,
                                   Proto => 'tcp',
                                   Reuse => 1,
                                  );
die "Socket could not be created. Reason: $!\n" unless ($sock);
while ($new_sock = $main_sock->accept()) {
      $pid = fork();
      die "Cannot fork: $!" unless defined ($pid) ;
      if ($pid == 0) {
            # Child process
            while (defined ($buf <$new_sock»)
                  # do something with $buf ....
                  print $new_sock "You said: $buf\n";
            }
            exit(O); # Child process exits when it is done.
      # else 'tis the parent process, which goes back to accept()
}
close ($main_sock);
```

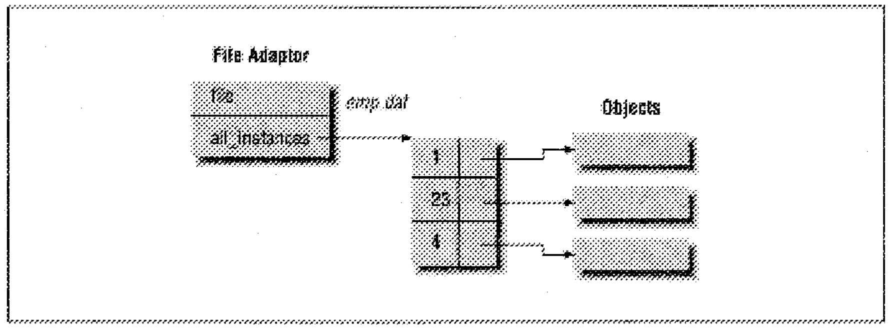
The `fork` call results in two identical processes-the parent and child-starting from the statement following the `fork`. The parent gets a positive return value, the process ID (`$pid`) of the child process. Both processes check this return value and execute their own logic; the

</td></tr>
</table>

U.S. Patent No. 7,426,720	*Srinivasan* in view of *APA-Bach*
	main process goes back to `accept`, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the `CHLD` signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if the child was able to execute `exit()`, or a termination status otherwise), just in case the parent uses `wait` or `waitpid` to enquire about this status. The terminated child process is also known as a *zombie* process, and it is always a good thing to remove it using `wait`; otherwise, the process tables keep filling up with junk. In the preceding code, `wait` doesn't block, because it is called only when we know for sure that a child process has died-the `CHLD` signal arranges that for us. Be sure to read the online documentation for quirks associated with signals in general and `SIGCHLD` in particular. * Malcolm Beattie has a working prototype of a threaded Perl interpreter, which will be incorporated into the mainstream in the Perl 5.005 release." *Srinivasan* at 194-95.
and a copy-on-write process cloning mechanism to instantiate the child runtime system process by copying references to the memory space of the master runtime system process into a separate memory space for the child runtime system process, and to defer copying of the memory space of the master runtime system process until the child runtime system process needs to modify the referenced memory space of the master runtime system process.	*Srinivasan* in view of *APA-Bach* provided the use of *fork* in Unix/Linux, which includes a copy-on-write process cloning mechanism to instantiate the child runtime system process by copying references to the memory space of the master runtime system process into a separate memory space for the child runtime system process, and to defer copying of the memory space of the master runtime system process until the child runtime system process needs to modify the referenced memory space of the master runtime system process. The fork( ) system call disclosed by *Srinivasan* was commonly used in conjunction with the copy-on-write mechanism to further streamline the impact on system memory; this is described in detail in *APA-Bach*. One of ordinary skill in the art at the time of the invention, seeking to reduce the impact of a fork( ) system call on local memory, would look to the disclosure of *APA-Bach* for a description of the copy-on-write bit. "The only way for a user to create a new process in the UNIX operating system is to invoke the *fork* system call." *APA-Bach* at 192. "The *copy-on-write* bit, used in the *fork* system call, indicates that the kernel must create a

U.S. Patent No. 7,426,720	***Srinivasan* in view of *APA-Bach***
	new copy of the page when a process modifies its contents." *APA-Bach* at 287. "9.2.1.1 Fork in a Paging System As explained in Section 7.1, the kernel duplicates every region of the parent process during the *fork* system call and attaches it to the child process. Traditionally, the kernel of a swapping system makes a physical copy of the parent's address space, usually a wasteful operation, because processes often call *exec* soon after the *fork* call and immediately free the memory just copied. On the System V paging system, the kernel avoids copying the page by manipulating the region tables, page table entries, and pfdata table entries: It simply increments the region reference count of shared regions.... The page can now be referenced through both regions, which share the page until a process writes to it. The kernel then copies the page so that each region has a private version. To do this, the kernel turns on the 'copy on write' bit for every page table entry in private regions of the parent and child processes during *fork*. If either process writes the page, it incurs a protection fault, and in handling the fault, the kernel makes a new copy of the page for the faulting process. The physical copying of the page is thus deferred until a process really needs it." *APA-Bach* at 289–90.
2. A system according to claim 1, further comprising: a cache checker to determine whether the instantiated class definition is available in a local cache associated with the master runtime system process.	*Srinivasan* provided a cache checker to determine whether the instantiated class definition is available in a local cache associated with the master runtime system process. For example, *Srinivasan* discloses the "Adaptor: : File" function, which "converts [a] query expression to an evalable Perl expression and cycles through all objects, matching them against the query specification." *Srinivasan* at 179. In *Srinivasan*, "[o]bjects of a certain type are said to belong to a *class*." *Srinivasan* at 101. Thus, the exemplar disclosure effects a search through the available classes. "This means that Adaptor has to keep an in-memory cache of objects that have been retrieved from disk in previous queries, so that if a database row is reread, the corresponding object is reused." *Srinivasan* at 178. "The op-ppaddr pointer represents the essence of the opcode: it is a pointer to a built-in function-call it an opcode function-that implements the functionality of the opcode. All opcode functions are prefixed with pp (pp-push, pp_grep, and so on) and are distributed over pp.c, pp_ctl.c, pp_sys.c, and pp_hot.c. The last one contains the opcode functions that are

U.S. Patent No. 7,426,720	***Srinivasan* in view of *APA-Bach***
	"hot," or frequently executed, so it is likely to remain within the cache of most RIse systems. Tom Christiansen once mentioned that this feature is also true of the regular expression-matching code, which is why regex matchers written in Java won't come anywhere close in performance. (I'll reevaluate this claim once Sun's Java processors are freely available.) As you will see later on, the opcode functions look strikingly similar to the glue code output by xsubpp/SWJG; this is because they interoperate using the argument stack and obey the same parameter passing protocols." *Srinivasan* at 324. "This returns a list of object references that match the query criteria. Now if you reissue this query, it is not too much to expect it to return an identical list of objects (the same object references, that is). This means that Adaptor has to keep an in-memory cache of objects that have been retrieved from disk in previous queries, so that if a database row is reread, the corresponding object is reused." *Srinivasan* at 178. "The Adaptor: : File module does not have this problem because it maintains a list of all objects given to its store () method (for reasons to be explained in the next section); hence successive identical queries return identical lists. Queries One big reason why object-oriented databases haven't caught on is the lack of a query language (or at least a standard query language). When you have a million objects in the database, it would be a terrible thing to load every single object in memory to see whether it matches your criteria; this is a job best left to the database. Adaptor::DBI simply translates queries to equivalent SQL queries, while Adaptor: : File implements a simple-minded scheme for file based objects: it converts the query expression to an evalable Perl expression and cycles through all objects, matching them against the query specification. Schema Evolution Let us say you have sent your objects' data to a file, and tomorrow, some more attributes are added to the object implementation. The schema is said to have evolved. The framework has to be able to reconcile old data with newer object implementations. Implementation This section explains the implementation of Adaptor::DBI and Adaptor::File. We will cover only the key procedures that perform query processing and file or database I/O. Pay as much attention to the design gotchas and unimplemented features as you do to the code.

U.S. Patent No. 7,426,720	*Srinivasan* in view of *APA-Bach*
	Adaptor::File An Adaptor::File instance represents all objects stored in one file. When this adaptor is created (using new), it reads the entire file and translates the data to in-memory objects. Slurping the entire file into memory avoids the problem of having to implement fancy on-disk schemes for random access to variable-length data; after all, that is the job of DBM and database implementations. For this reason, this approach is not recommended for large numbers of objects (over 1,000, to pick a number). The file adaptor has an attribute called all_instances, a hash table of all objects given to its store method (and indexed by their _id), as shown in Figure 11-2.  *Figure 11-2. Structure of file adaptor* *Srinivasan* at 179-80.
3. A system according to claim 2, further comprising: a class locator to locate the source definition if the instantiated class definition is unavailable in the local cache.	*Srinivasan* provided a class locator to locate the source definition if the instantiated class definition is unavailable in the local cache. For example, *Srinivasan* discloses the "Adaptor: : File" function, which "converts [a] query expression to an evalable Perl expression and cycles through all objects, matching them against the query specification." *Srinivasan* at 179. In *Srinivasan*, "[o]bjects of a certain type are said to belong to a *class*." *Srinivasan* at 101. Thus, the exemplar disclosure effects a search through the available classes. It is inherent that such a search could locate a class definition that is unavailable in the local cache. "This returns a list of object references that match the query criteria. Now if you reissue this query, it is not too much to expect it to return an identical list of objects (the same object references, that is). This means that Adaptor has to keep an in-memory cache of objects that

U.S. Patent No. 7,426,720	*Srinivasan* in view of *APA-Bach*
	have been retrieved from disk in previous queries, so that if a database row is reread, the corresponding object is reused." *Srinivasan* at 178. "The Adaptor: : File module does not have this problem because it maintains a list of all objects given to its store () method (for reasons to be explained in the next section); hence successive identical queries return identical lists. Queries One big reason why object-oriented databases haven't caught on is the lack of a query language (or at least a standard query language). When you have a million objects in the database, it would be a terrible thing to load every single object in memory to see whether it matches your criteria; this is a job best left to the database. Adaptor::DBI simply translates queries to equivalent SQL queries, while Adaptor: : File implements a simple-minded scheme for file based objects: it converts the query expression to an evalable Perl expression and cycles through all objects, matching them against the query specification. Schema Evolution Let us say you have sent your objects' data to a file, and tomorrow, some more attributes are added to the object implementation. The schema is said to have evolved. The framework has to be able to reconcile old data with newer object implementations. Implementation This section explains the implementation of Adaptor::DBI and Adaptor::File. We will cover only the key procedures that perform query processing and file or database I/O. Pay as much attention to the design gotchas and unimplemented features as you do to the code. Adaptor::File An Adaptor::File instance represents all objects stored in one file. When this adaptor is created (using new), it reads the entire file and translates the data to in-memory objects. Slurping the entire file into memory avoids the problem of having to implement fancy on-disk schemes for random access to variable-length data; after all, that is the job of DBM and database implementations. For this reason, this approach is not recommended for large numbers of objects (over 1,000, to pick a number). The file adaptor has an attribute called all_instances, a hash table of all objects given to its store method (and indexed by their _id), as shown in Figure 11-2.

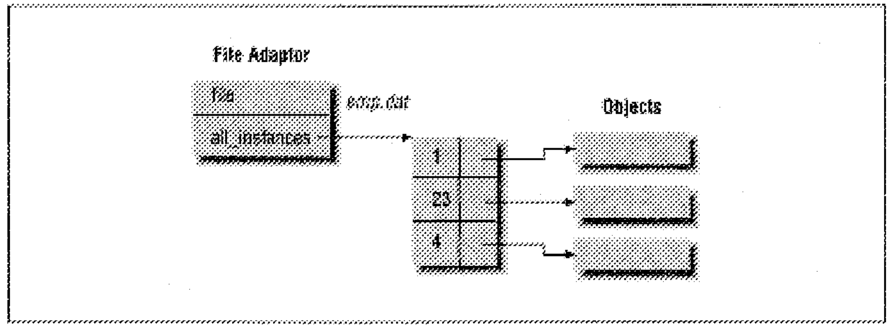
<table>
<tr><th>U.S. Patent No. 7,426,720</th><th>Srinivasan in view of APA-Bach</th></tr>
<tr><td></td><td>

Figure 11-2. Structure of file adaptor

Srinivasan at 179-80.

“Adaptor::DBI is considerably simpler than Adaptor::File. It does not maintain a table of objects in memory; when asked to store an object, it sends it to the database, and when asked to retrieve one or more objects, it simply passes the request along to the database.” Srinivasan at 184.

“retrieve_where creates a select query from the mapping information for that class. As was pointed out earlier, the same query executed twice will get you two different sets of objects, whose data are duplicates of the other…” Srinivasan at 186.</td></tr>
<tr><td>4. A system according to claim 1, further comprising: a class resolver to resolve the class definition.</td><td>Srinivasan provided a class resolver to resolve the class definition. Srinivasan discloses that, in Perl, “[c]lass attributes are simply package global variables, and class methods are ordinary subroutines that don't work on any specific instance. Perl supports polymorphism and run-time binding for these ordinary subroutines (not just instance methods), which can be leveraged to produce a truly flexible design.” Srinivasan at 107-108.

“This returns a list of object references that match the query criteria. Now if you reissue this query, it is not too much to expect it to return an identical list of objects (the same object references, that is). This means that Adaptor has to keep an in-memory cache of objects that have been retrieved from disk in previous queries, so that if a database row is reread, the corresponding object is reused.” Srinivasan at 178.</td></tr>
</table>

U.S. Patent No. 7,426,720	*Srinivasan* in view of *APA-Bach*
	"The Adaptor: : File module does not have this problem because it maintains a list of all objects given to its store () method (for reasons to be explained in the next section); hence successive identical queries return identical lists. Queries One big reason why object-oriented databases haven't caught on is the lack of a query language (or at least a standard query language). When you have a million objects in the database, it would be a terrible thing to load every single object in memory to see whether it matches your criteria; this is a job best left to the database. Adaptor::DBI simply translates queries to equivalent SQL queries, while Adaptor: : File implements a simple-minded scheme for file based objects: it converts the query expression to an evalable Perl expression and cycles through all objects, matching them against the query specification. Schema Evolution Let us say you have sent your objects' data to a file, and tomorrow, some more attributes are added to the object implementation. The schema is said to have evolved. The framework has to be able to reconcile old data with newer object implementations. Implementation This section explains the implementation of Adaptor::DBI and Adaptor::File. We will cover only the key procedures that perform query processing and file or database I/O. Pay as much attention to the design gotchas and unimplemented features as you do to the code. Adaptor::File An Adaptor::File instance represents all objects stored in one file. When this adaptor is created (using new), it reads the entire file and translates the data to in-memory objects. Slurping the entire file into memory avoids the problem of having to implement fancy on-disk schemes for random access to variable-length data; after all, that is the job of DBM and database implementations. For this reason, this approach is not recommended for large numbers of objects (over 1,000, to pick a number). The file adaptor has an attribute called all_instances, a hash table of all objects given to its store method (and indexed by their _id), as shown in Figure 11-2.

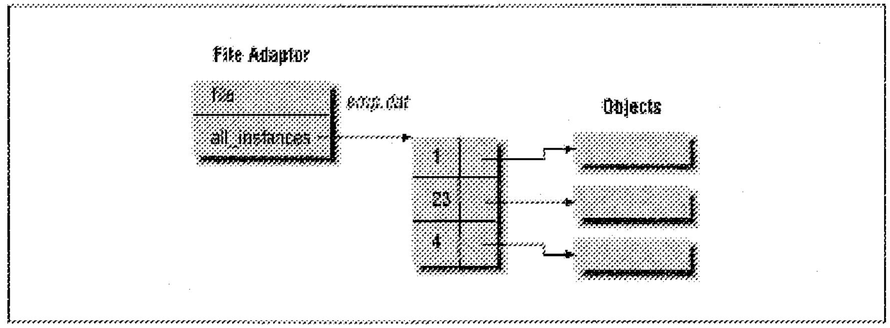
<table>
<tr><th>U.S. Patent No. 7,426,720</th><th>Srinivasan in view of APA-Bach</th></tr>
<tr><td></td><td>

Figure 11-2. Structure of file adaptor

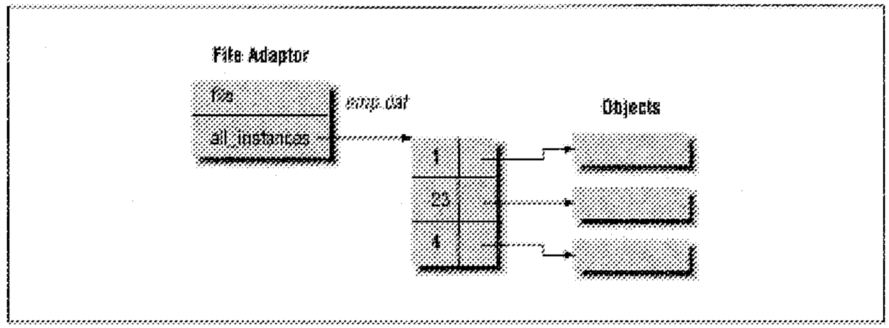
Srinivasan at 179-80.

“Adaptor::DBI is considerably simpler than Adaptor::File. It does not maintain a table of objects in memory; when asked to store an object, it sends it to the database, and when asked to retrieve one or more objects, it simply passes the request along to the database.” Srinivasan at 184.

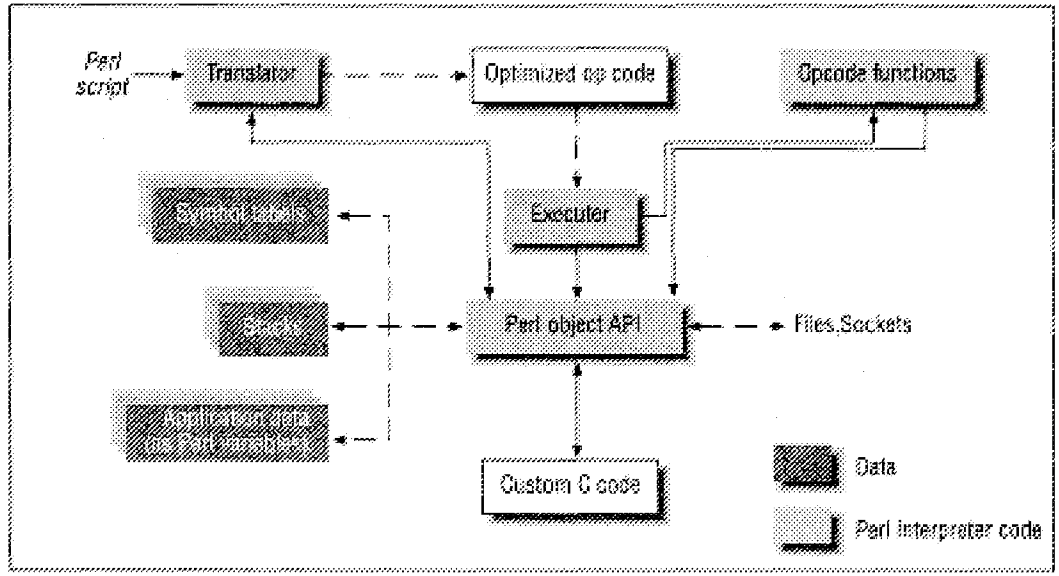
“retrieve_where creates a select query from the mapping information for that class. As was pointed out earlier, the same query executed twice will get you two different sets of objects, whose data are duplicates of the other…” Srinivasan at 186.</td></tr>
<tr><td>5. A system according to claim 1, further comprising: at least one of a local and remote file system to maintain the source definition as a class file.</td><td>Srinivasan provided at least one of a local and remote file system to maintain the source definition as a class file. Srinivasan discloses, for example, that “[i]n a typical client/server system, the server has the "real" objects. But the system is written in such a way that a client can remotely invoke a method of the object, with familiar OO syntax. For example, if a client program wants to invoke a method on a remote bank account, it should be able to say something like this:…” Srinivasan at 134.

“You will not merely dabble with language syntax or the APIs of different modules as you read this book. You will spend just as much time dealing with real-world issues such as avoiding deadlocks during remote procedure calls and switching smoothly between data storage using a flat file or a database.” Srinivasan at xi.</td></tr>
</table>

U.S. Patent No. 7,426,720	***Srinivasan* in view of *APA-Bach***
	"This technique is often employed in the guts of client/server libraries. In a typical client/server system, the server has the "real" objects. But the system is written in such a way that a client can remotely invoke a method of the object, with familiar OO syntax. For example, if a client program wants to invoke a method on a remote bank account, it should be able to say something like this:…" *Srinivasan* at 134. "This returns a list of object references that match the query criteria. Now if you reissue this query, it is not too much to expect it to return an identical list of objects (the same object references, that is). This means that Adaptor has to keep an in-memory cache of objects that have been retrieved from disk in previous queries, so that if a database row is reread, the corresponding object is reused." *Srinivasan* at 178. "The Adaptor: : File module does not have this problem because it maintains a list of all objects given to its store () method (for reasons to be explained in the next section); hence successive identical queries return identical lists. Queries One big reason why object-oriented databases haven't caught on is the lack of a query language (or at least a standard query language). When you have a million objects in the database, it would be a terrible thing to load every single object in memory to see whether it matches your criteria; this is a job best left to the database. Adaptor::DBI simply translates queries to equivalent SQL queries, while Adaptor: : File implements a simple-minded scheme for file based objects: it converts the query expression to an evalable Perl expression and cycles through all objects, matching them against the query specification. Schema Evolution Let us say you have sent your objects' data to a file, and tomorrow, some more attributes are added to the object implementation. The schema is said to have evolved. The framework has to be able to reconcile old data with newer object implementations. Implementation This section explains the implementation of Adaptor::DBI and Adaptor::File. We will cover only the key procedures that perform query processing and file or database I/O. Pay as much attention to the design gotchas and unimplemented features as you do to the code. Adaptor::File

U.S. Patent No. 7,426,720	***Srinivasan* in view of *APA-Bach***
	An Adaptor::File instance represents all objects stored in one file. When this adaptor is created (using new), it reads the entire file and translates the data to in-memory objects. Slurping the entire file into memory avoids the problem of having to implement fancy on-disk schemes for random access to variable-length data; after all, that is the job of DBM and database implementations. For this reason, this approach is not recommended for large numbers of objects (over 1,000, to pick a number). The file adaptor has an attribute called all_instances, a hash table of all objects given to its store method (and indexed by their _id), as shown in Figure 11-2.  *Figure 11-2. Structure of file adaptor* *Srinivasan* at 179-80. "Adaptor::DBI is considerably simpler than Adaptor::File. It does not maintain a table of objects in memory; when asked to store an object, it sends it to the database, and when asked to retrieve one or more objects, it simply passes the request along to the database." *Srinivasan* at 184. "retrieve_where creates a select query from the mapping information for that class. As was pointed out earlier, the same query executed twice will get you two different sets of objects, whose data are duplicates of the other…" *Srinivasan* at 186.
6. A system according to claim 1, further comprising: a process cloning mechanism to instantiate the child runtime system process by copying the	*Srinivasan* provided a process cloning mechanism to instantiate the child runtime system process by copying the memory space of the master runtime system process into a separate memory space for the child runtime system process. *Srinivasan* discloses handling multiple clients by "[c]reat[ing] multiple threads of control." *Srinivasan* at 193. In order

U.S. Patent No. 7,426,720	***Srinivasan* in view of *APA-Bach***
memory space of the master runtime system process into a separate memory space for the child runtime system process.	to create multiple threads of control, *Srinivasan* invokes the fork( ) system call to create "process-level parallelism." *See id*. at 194. This creates a new process, called the child process. "The newly created child process meanwhile has a copy of its parent's environment and shares all open file descriptors." *See id*. Thus, *Srinivasan* discloses a "child" runtime system process, which is in fact a clone of the parent runtime system process. "The data structures described above are normally kept in global C variables. If Perl is compiled with *–DMULTIPLICITY*, it lumps all these global variables into a structure called Perl Interpreter. This allows you to have multiple instances of the interpreter, each with its own "global" space. (Recall from Chapter 19 the API to allocate and construct an object of type Perl Interpreter.) In the absence of this compile-time option, the Perl Interpreter object is a dummy structure, and the internal data structures are truly global, for maximum performance. The API remains the same in either case. You can use multiple interpreters to enforce completely isolated namespaces. Each interpreter has its own "main" package and its own tree of loaded packages. I have not seen this feature used in production Perl applications, but Tel provides a framework called SafeTel for security purposes, which uses a similar feature of multiple interpreter objects. These interpreters can be unrestricted or restricted. The equivalent module in Perl, Safe, uses a different mechanism, though the result (of isolated name spaces) is similar. More on this in the next section." *Srinivasan* at 323.

U.S. Patent No. 7,426,720	***Srinivasan* in view of *APA-Bach***
	Figure 20-1 shows the various components of a running Perl system. Shaded rectangles represent data structures, some of which can have multiple instances in a program. The source code can also be partitioned roughly along these lines.  *Figure 20-1. Snapshot of a running system* *Srinivasan* at 321. “Perl doesn't have threads yet (at least not officially*), but on Unix and similarly empowered systems, it supports `fork`, the way to get process-level parallelism. The server process acts as a full-time receptionist: it blocks on `accept`, and when a connection request comes in, it spawns a child process and goes back to `accept`. The newly created child process meanwhile has a copy of its parent's environment and shares all open file descriptors. Hence it is able to read from, and write to, the new socket returned by `accept`. When the child is done with the conversation, it simply exits. Each process is therefore dedicated to its own task and doesn't interfere with the other. The following code shows an example of a forking server:

U.S. Patent No. 7,426,720	***Srinivasan* in view of *APA-Bach***

```
# Forking server
use IO:: Socket;
$SIG{CHLD} sub {wait ()};
$main_sock = new IO::Socket::INET (LocalHost => 'goldengate',
                                   LocalPort => 1200,
                                   Listen => 5,
                                   Proto => 'tcp',
                                   Reuse => 1,
                                  );
die "Socket could not be created. Reason: $!\n" unless ($sock);
while ($new_sock = $main_sock->accept()) {
      $pid = fork();
      die "Cannot fork: $!" unless defined ($pid) ;
      if ($pid == 0) {
            # Child process
            while (defined ($buf <$new_sock»)
                  # do something with $buf ....
                  print $new_sock "You said: $buf\n";
            }
            exit(O); # Child process exits when it is done.
      # else 'tis the parent process, which goes back to accept()
}
close ($main_sock);
```

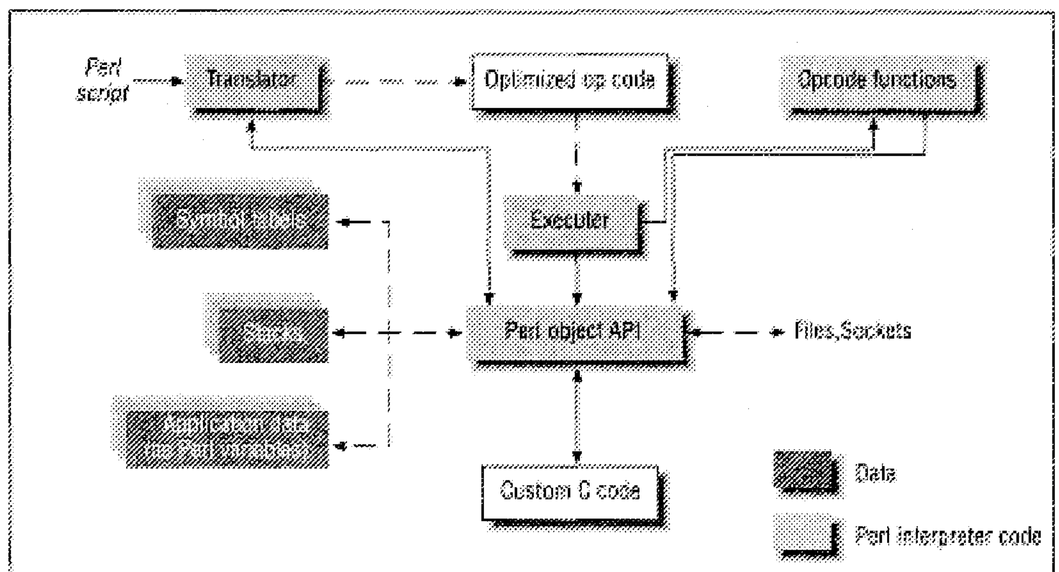
The `fork` call results in two identical processes-the parent and child-starting from the statement following the `fork`. The parent gets a positive return value, the process ID (`$pid`) of the child process. Both processes check this return value and execute their own logic; the main process goes back to `accept`, and the child process reads a line from the socket and echoes it back to the client.

Incidentally, the `CHLD` signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if the child was able to execute `exit()`, or a termination status otherwise), just in case the parent uses `wait` or `waitpid` to enquire about this status. The terminated child process is also known as a *zombie* process, and it is always a good thing to remove it using `wait`; otherwise, the process tables keep filling up with junk. In the preceding code, `wait` doesn't

U.S. Patent No. 7,426,720	*Srinivasan* in view of *APA-Bach*
	block, because it is called only when we know for sure that a child process has died-the `CHLD` signal arranges that for us. Be sure to read the online documentation for quirks associated with signals in general and `SIGCHLD` in particular. * Malcolm Beattie has a working prototype of a threaded Perl interpreter, which will be incorporated into the mainstream in the Perl 5.005 release." *Srinivasan* at 194-95.
7. A system according to claim 1, wherein the master runtime system process is caused to sleep relative to receiving the process request.	*Srinivasan* provided a system wherein the master runtime system process is caused to sleep relative to receiving the process request. For example, *Srinivasan* discloses a persistent process, "also known as a zombie process." *Srinivasan* at 195. "If you want a finer granularity, you can use the Time: : HiRes module available from CPAN, which gives microsecond resolution on Unix systems (gives access to the usleep and ualarm system calls) On Microsoft Windows systems, you can use the Win32 : : Timer call for millisecond-level timing." *Srinivasan* at 141. "The `fork` call results in two identical processes-the parent and child-starting from the statement following the `fork`. The parent gets a positive return value, the process ID (`$pid`) of the child process. Both processes check this return value and execute their own logic; the main process goes back to `accept`, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the `CHLD` signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if the child was able to execute `exit()`, or a termination status otherwise), just in case the parent uses `wait` or `waitpid` to enquire about this status. The terminated child process is also known as a *zombie* process, and it is always a good thing to remove it using `wait`; otherwise, the process tables keep filling up with junk. In the preceding code, `wait` doesn't block, because it is called only when we know for sure that a child process has died-the `CHLD` signal arranges that for us. Be sure to read the online documentation for quirks associated with signals in general and `SIGCHLD` in particular." *Srinivasan* at 195.

U.S. Patent No. 7,426,720	*Srinivasan* in view of *APA-Bach*
8. A system according to claim 1, wherein the object-oriented program code is written in the Java programming language.	*Srinivasan* provided a system wherein the object-oriented program code is written in the Java programming language. For example, *Srinivasan* discloses that "Java offers two levels of modularity: packages and classes, where a package is a collection of classes. (We'll learn about the notion of classes in the next chapter.) Perl's package is equivalent to both." *Srinivasan* at 98. "Java offers two levels of modularity: packages and classes, where a package is a collection of classes. (We'll learn about the notion of classes in the next chapter.) Perl's package is equivalent to both. Java does not allow one package to mess around with another package's namespace (no export) but allows a package to selectively import the classes it requires. It focuses a considerable amount of attention on security, which hasn't really stopped determined crackers. The Perl world has third-party packages called Safe and Penguin (which depends on Safe) that attempt to provide similar isolation characteristics (and don't offer any security guarantees either). Since the arrival of the Java Beans and the 1.1 version of the Java Development Kit ODK), Java has gained significant reflection capabilities, though nowhere near the amount of publicly available information Perl gives you. There are reasonably good arguments to be made both for providing this information and for not providing it; everything comes down to different models of programming. Men were sent to the moon while FORTRAN and COBOL ruled the roost, which proves that you can get a whole lot done if you don't indulge in language wars. Java allows you to dynamically "dispatch" a function call, by giving the function's name as a string, and to trap an exception if the function doesn't exist; this is like using Perl's symbolic references." *Srinivasan* at 98. "The translator converts a Perl script into a tree of opcodes (explained below). It comprises a hand-coded lexer (*toke.c*), the yacc-based parser (perly.y) , and the code generator (op.c). Regular expressions–which form a distinct sublanguage–are recognized in *toke.c* and compiled to an internal format in *regcomp.c*. Opcodes are similar in concept to machine code; while machine code is executed by hardware, opcodes (sometimes called byte-codes or p-code) are executed by a "virtual machine." The similarity ends there. Modern interpreters never emulate the workings of a hardware CPU, for performance reasons. Instead, they create complex structures primed for execution, such that each opcode directly

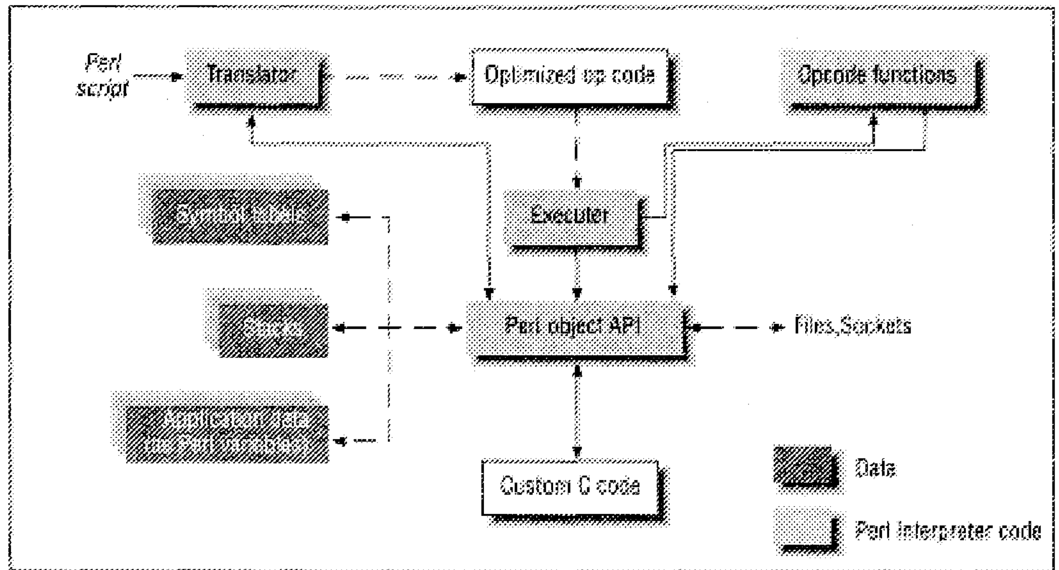
U.S. Patent No. 7,426,720	***Srinivasan* in view of *APA-Bach***
	contains a pointer to the next one to execute and a pointer to the data it is expected to work on at run-time. In other words, these opcodes are not mere instruction types; they actually embody the exact unit of work expected at that point in that program. Java and Perl are both examples of such interpreters. While many of java's bytecodes resemble a RISC machine's instruction set, Perl's opcodes represent a much higher level of abstraction. A large number of these opcodes directly correspond to the facilities available at the scripting level, such as regular expression matching and substitution, `chop`, `push`, `index`, `rindex`, `grep`,* and so on, which explains why there are 343 opcodes as of this writing! It also explains why Perl is so fast: instead of spending time in the interpreter, most of the work is done in lovingly hand-optimized e code. You can also see why it is hard to create a Perl-to-Java byte-code translator: there is no correspondence between the two sets." *Srinivasan* at 323-24.
10. A method for dynamic preloading of classes through memory space cloning of a master runtime system process, comprising:	*Srinivasan* provided a method for dynamic preloading of classes through memory space cloning of a master runtime system process.
executing a master runtime system process;	*Srinivasan* provided a method for executing a master runtime system process. *Srinivasan* discloses handling multiple clients by "[c]reat[ing] multiple threads of control." *Srinivasan* at 193. In order to create multiple threads of control, *Srinivasan* invokes the fork( ) system call to create "process-level parallelism." *See id*. at 194. This creates a new process, called the child process. "The newly created child process meanwhile has a copy of its parent's environment and shares all open file descriptors." *See id.* Thus, *Srinivasan* discloses a "parent" runtime system process, *i.e.*, a master runtime system process. "The data structures described above are normally kept in global C variables. If Perl is compiled with *–DMULTIPLICITY*, it lumps all these global variables into a structure called Perl Interpreter. This allows you to have multiple instances of the interpreter, each with its own "global" space. (Recall from Chapter 19 the API to allocate and construct an object of type Perl Interpreter.) In the absence of this compile-time option, the Perl Interpreter object is a dummy structure, and the internal data structures are truly global, for maximum performance. The API remains the same in either case.

U.S. Patent No. 7,426,720	*Srinivasan* in view of *APA-Bach*
	You can use multiple interpreters to enforce completely isolated namespaces. Each interpreter has its own "main" package and its own tree of loaded packages. I have not seen this feature used in production Perl applications, but Tel provides a framework called SafeTel for security purposes, which uses a similar feature of multiple interpreter objects. These interpreters can be unrestricted or restricted. The equivalent module in Perl, Safe, uses a different mechanism, though the result (of isolated name spaces) is similar. More on this in the next section." *Srinivasan* at 323. Figure 20-1 shows the various components of a running Perl system. Shaded rectangles represent data structures, some of which can have multiple instances in a program. The source code can also be partitioned roughly along these lines.  *Figure 20-1. Snapshot of a running system* *Srinivasan* at 321.
obtaining a representation of at least one	*Srinivasan* provided a method for obtaining a representation of at least one class from a

U.S. Patent No. 7,426,720	*Srinivasan* in view of *APA-Bach*
class from a source definition provided as object-oriented program code;	source definition provided as object-oriented program code. *Srinivasan* discusses object oriented programming and its usefulness in conjunction with Perl code. *Srinivasan* at 101. *Srinivasan* notes that, in Perl, a "class is a package." *Srinivasan* at 389. *Srinivasan* then discusses two methods for the creation of a package from source definition provided as object oriented program code. *See id.* at 389-390 (discussing creation of the "employee" class). In order to preload the class, *Srinivasan* discloses specific functional code: "Using object package: use Employee; $emp = Employee->new ("Ada", 3 5); $emp->set_salary(1000); *See id.* at 390. "Java offers two levels of modularity: packages and classes, where a package is a collection of classes. (We'll learn about the notion of classes in the next chapter.) Perl's package is equivalent to both. Java does not allow one package to mess around with another package's namespace (no export) but allows a package to selectively import the classes it requires. It focuses a considerable amount of attention on security, which hasn't really stopped determined crackers. The Perl world has third-party packages called Safe and Penguin (which depends on Safe) that attempt to provide similar isolation characteristics (and don't offer any security guarantees either). Since the arrival of the Java Beans and the 1.1 version of the Java Development Kit ODK), Java has gained significant reflection capabilities, though nowhere near the amount of publicly available information Perl gives you. There are reasonably good arguments to be made both for providing this information and for not providing it; everything comes down to different models of programming. Men were sent to the moon while FORTRAN and COBOL ruled the roost, which proves that you can get a whole lot done if you don't indulge in language wars. Java allows you to dynamically "dispatch" a function call, by giving the function's name as a string, and to trap an exception if the function doesn't exist; this is like using Perl's symbolic references." *Srinivasan* at 98.

U.S. Patent No. 7,426,720	*Srinivasan* in view of *APA-Bach*
	"Object orientation COO) is the latest software methodology to occupy the airwaves, hyped to a point where the term "object-oriented design" seems to automatically imply a good design. In this chapter, we will study what the noise is all about and build objects using Perl. I will leave it to the plethora of 00 literature to convince you that there is a respectable middle-ground and that the object revolution is indeed a good thing." *Srinivasan* at 99. "Objects of a certain type are said to belong to a *class*." *Srinivasan* at 101. "The translator converts a Perl script into a tree of opcodes (explained below). It comprises a hand-coded lexer (*toke.c*), the yacc-based parser (perly.y) , and the code generator (op.c). Regular expressions–which form a distinct sublanguage–are recognized in *toke.c* and compiled to an internal format in *regcomp.c*. Opcodes are similar in concept to machine code; while machine code is executed by hardware, opcodes (sometimes called byte-codes or p-code) are executed by a "virtual machine." The similarity ends there. Modern interpreters never emulate the workings of a hardware CPU, for performance reasons. Instead, they create complex structures primed for execution, such that each opcode directly contains a pointer to the next one to execute and a pointer to the data it is expected to work on at run-time. In other words, these opcodes are not mere instruction types; they actually embody the exact unit of work expected at that point in that program. Java and Perl are both examples of such interpreters. While many of java's bytecodes resemble a RISC machine's instruction set, Perl's opcodes represent a much higher level of abstraction. A large number of these opcodes directly correspond to the facilities available at the scripting level, such as regular expression matching and substitution, `chop`, `push`, `index`, `rindex`, `grep`,* and so on, which explains why there are 343 opcodes as of this writing! It also explains why Perl is so fast: instead of spending time in the interpreter, most of the work is done in lovingly hand-optimized e code. You can also see why it is hard to create a Perl-to-Java byte-code translator: there is no correspondence between the two sets." *Srinivasan* at 323-24. "Malcolm has also submitted a Perl compiler extension [5], which is in its early stages as of this writing. It can be asked to translate a script to C code, which can be compiled to form an executable; as it happens, this executable is not much faster than the interpreted script,

U.S. Patent No. 7,426,720	***Srinivasan* in view of *APA-Bach***
	because most of the action still takes place in opcode functions as they exist now. Static typing hints may usher in some aggressive optimizations. For example, if you say: `my integer $i;` the compiler would use C's native integer type, rather than an SV-this would speed up loops and arithmetic expressions. The compiler can alternatively produce a byte-code file and have the interpreter `eval` it subsequently, similar to the facilities provided by Python and Java. It also supports much better debugging options than those currently provided with *–D*." *Srinivasan* at 370.
interpreting and instantiating the representation as a class definition in a memory space of the master runtime system process;	*Srinivasan* provided a method for interpreting and instantiating the representation as a class definition in a memory space of the master runtime system process. *Srinivasan* discloses handling multiple clients by "[c]reat[ing] multiple threads of control." *Srinivasan* at 193. In order to create multiple threads of control, *Srinivasan* invokes the fork( ) system call to create "process-level parallelism." *See id*. at 194. This creates a new process, called the child process. "The newly created child process meanwhile has a copy of its parent's environment and shares all open file descriptors." *See id.* Thus, *Srinivasan* discloses a "parent" runtime system process, *i.e.*, a master runtime system process. "The data structures described above are normally kept in global C variables. If Perl is compiled with *–DMULTIPLICITY*, it lumps all these global variables into a structure called Perl Interpreter. This allows you to have multiple instances of the interpreter, each with its own "global" space. (Recall from Chapter 19 the API to allocate and construct an object of type Perl Interpreter.) In the absence of this compile-time option, the Perl Interpreter object is a dummy structure, and the internal data structures are truly global, for maximum performance. The API remains the same in either case. You can use multiple interpreters to enforce completely isolated namespaces. Each interpreter has its own "main" package and its own tree of loaded packages. I have not seen this feature used in production Perl applications, but Tel provides a framework called SafeTel for security purposes, which uses a similar feature of multiple interpreter objects. These interpreters can be unrestricted or restricted. The equivalent module in Perl, Safe, uses a different mechanism, though the result (of isolated name spaces) is similar. More on this in

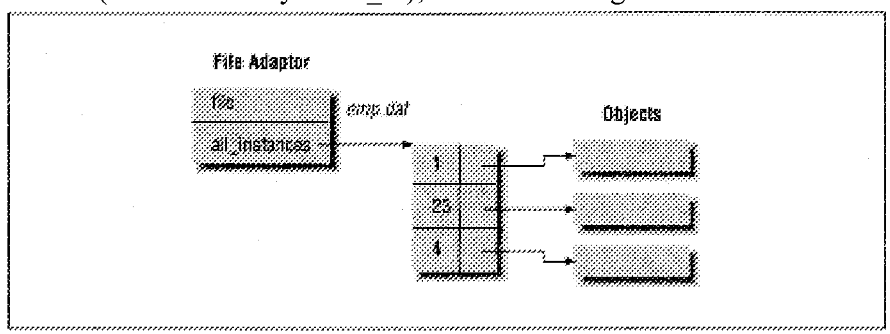
U.S. Patent No. 7,426,720	***Srinivasan* in view of *APA-Bach***
	the next section." *Srinivasan* at 323. Figure 20-1 shows the various components of a running Perl system. Shaded rectangles represent data structures, some of which can have multiple instances in a program. The source code can also be partitioned roughly along these lines.  *Figure 20-1. Snapshot of a running system* *Srinivasan* at 321.
and cloning the memory space as a child runtime system process responsive to a process request and executing the child runtime system process;	*Srinivasan* provided a method for cloning the memory space as a child runtime system process responsive to a process request and executing the child runtime system process. *Srinivasan* discloses handling multiple clients by "[c]reat[ing] multiple threads of control." *Srinivasan* at 193. In order to create multiple threads of control, *Srinivasan* invokes the fork( ) system call to create "process-level parallelism." *See id*. at 194. This creates a new process, called the child process. "The newly created child process meanwhile has a copy of its parent's environment and shares all open file descriptors." *See id.* Thus,

<table>
<tr><th>U.S. Patent No. 7,426,720</th><th>Srinivasan in view of APA-Bach</th></tr>
<tr><td></td><td>
Srinivasan discloses a “child” runtime system process, which is in fact a clone of the parent runtime system process.
“Perl doesn't have threads yet (at least not officially*), but on Unix and similarly empowered systems, it supports <code>fork</code>, the way to get process-level parallelism. The server process acts as a full-time receptionist: it blocks on <code>accept</code>, and when a connection request comes in, it spawns a child process and goes back to <code>accept</code>. The newly created child process meanwhile has a copy of its parent's environment and shares all open file descriptors. Hence it is able to read from, and write to, the new socket returned by <code>accept</code>. When the child is done with the conversation, it simply exits. Each process is therefore dedicated to its own task and doesn't interfere with the other. The following code shows an example of a forking server:
<pre>
Forking server
use IO:: Socket;
$SIG{CHLD} sub {wait ()};
$main_sock = new IO::Socket::INET (LocalHost => 'goldengate',
                                   LocalPort => 1200,
                                   Listen => 5,
                                   Proto => 'tcp',
                                   Reuse => 1,
                                  );
die "Socket could not be created. Reason: $!\n" unless ($sock);
while ($new_sock = $main_sock->accept()) {
      $pid = fork();
      die "Cannot fork: $!" unless defined ($pid) ;
      if ($pid == 0) {
            # Child process
            while (defined ($buf <$new_sock»)
                  # do something with $buf ....
                  print $new_sock "You said: $buf\n";
            }
            exit(O); # Child process exits when it is done.
      # else 'tis the parent process, which goes back to accept()
}
close ($main_sock);
</pre>
</td></tr>
</table>

U.S. Patent No. 7,426,720	***Srinivasan* in view of *APA-Bach***
	The `fork` call results in two identical processes-the parent and child-starting from the statement following the `fork`. The parent gets a positive return value, the process ID (`$pid`) of the child process. Both processes check this return value and execute their own logic; the main process goes back to `accept`, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the `CHLD` signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if the child was able to execute `exit()`, or a termination status otherwise), just in case the parent uses `wait` or `waitpid` to enquire about this status. The terminated child process is also known as a *zombie* process, and it is always a good thing to remove it using `wait`; otherwise, the process tables keep filling up with junk. In the preceding code, `wait` doesn't block, because it is called only when we know for sure that a child process has died-the `CHLD` signal arranges that for us. Be sure to read the online documentation for quirks associated with signals in general and `SIGCHLD` in particular. * Malcolm Beattie has a working prototype of a threaded Perl interpreter, which will be incorporated into the mainstream in the Perl 5.005 release." *Srinivasan* at 194-95.
wherein cloning the memory space as a child runtime system process involves instantiating the child runtime system process by copying references to the memory space of the master runtime system process into a separate memory space for the child runtime system process; and wherein copying references to the memory space of the master runtime system process defers copying of the memory space of the master runtime system process until the child runtime	*Srinivasan* in view of *APA-Bach* provided the use of *fork* in Unix/Linux, which includes a method wherein cloning the memory space as a child runtime system process involves instantiating the child runtime system process by copying references to the memory space of the master runtime system process into a separate memory space for the child runtime system process; and wherein copying references to the memory space of the master runtime system process defers copying of the memory space of the master runtime system process until the child runtime system process needs to modify the referenced memory space of the master runtime system process. The fork( ) system call disclosed by *Srinivasan* was commonly used in conjunction with the copy-on-write mechanism to further streamline the impact on system memory; this is described in detail in *APA-Bach*. One of ordinary skill in the art at the time of the invention, seeking to reduce the impact of a fork( ) system call on local memory, would look to the disclosure of *APA-Bach* for a description of the copy-on-write bit.

U.S. Patent No. 7,426,720	***Srinivasan*** **in view of *APA-Bach***
system process needs to modify the referenced memory space of the master runtime system process.	"The only way for a user to create a new process in the UNIX operating system is to invoke the *fork* system call." *APA-Bach* at 192. "The *copy-on-write* bit, used in the *fork* system call, indicates that the kernel must create a new copy of the page when a process modifies its contents." *APA-Bach* at 287. "9.2.1.1 Fork in a Paging System As explained in Section 7.1, the kernel duplicates every region of the parent process during the *fork* system call and attaches it to the child process. Traditionally, the kernel of a swapping system makes a physical copy of the parent's address space, usually a wasteful operation, because processes often call *exec* soon after the *fork* call and immediately free the memory just copied. On the System V paging system, the kernel avoids copying the page by manipulating the region tables, page table entries, and pfdata table entries: It simply increments the region reference count of shared regions.... The page can now be referenced through both regions, which share the page until a process writes to it. The kernel then copies the page so that each region has a private version. To do this, the kernel turns on the 'copy on write' bit for every page table entry in private regions of the parent and child processes during *fork*. If either process writes the page, it incurs a protection fault, and in handling the fault, the kernel makes a new copy of the page for the faulting process. The physical copying of the page is thus deferred until a process really needs it." *APA-Bach* at 289–90.
11. A method according to claim 10, further comprising: determining whether the instantiated class definition is available in a local cache associated with the master runtime system process.	*Srinivasan* provided a method for determining whether the instantiated class definition is available in a local cache associated with the master runtime system process. For example, *Srinivasan* discloses the "Adaptor: : File" function, which "converts [a] query expression to an evalable Perl expression and cycles through all objects, matching them against the query specification." *Srinivasan* at 179. In *Srinivasan*, "[o]bjects of a certain type are said to belong to a *class*." *Srinivasan* at 101. Thus, the exemplar disclosure effects a search through the available classes. "This means that Adaptor has to keep an in-memory cache of objects that have been retrieved from disk in previous queries, so that if a database row is reread, the corresponding object is reused." *Srinivasan* at 178.

U.S. Patent No. 7,426,720	*Srinivasan* in view of ***APA-Bach***
	"The op-ppaddr pointer represents the essence of the opcode: it is a pointer to a built-in function-call it an opcode function-that implements the functionality of the opcode. All opcode functions are prefixed with pp (pp-push, pp_grep, and so on) and are distributed over pp.c, pp_ctl.c, pp_sys.c, and pp_hot.c. The last one contains the opcode functions that are "hot," or frequently executed, so it is likely to remain within the cache of most RIse systems. Tom Christiansen once mentioned that this feature is also true of the regular expression-matching code, which is why regex matchers written in Java won't come anywhere close in performance. (I'll reevaluate this claim once Sun's Java processors are freely available.) As you will see later on, the opcode functions look strikingly similar to the glue code output by xsubpp/SWJG; this is because they interoperate using the argument stack and obey the same parameter passing protocols." *Srinivasan* at 324. "This returns a list of object references that match the query criteria. Now if you reissue this query, it is not too much to expect it to return an identical list of objects (the same object references, that is). This means that Adaptor has to keep an in-memory cache of objects that have been retrieved from disk in previous queries, so that if a database row is reread, the corresponding object is reused." *Srinivasan* at 178. "The Adaptor: : File module does not have this problem because it maintains a list of all objects given to its store () method (for reasons to be explained in the next section); hence successive identical queries return identical lists. Queries One big reason why object-oriented databases haven't caught on is the lack of a query language (or at least a standard query language). When you have a million objects in the database, it would be a terrible thing to load every single object in memory to see whether it matches your criteria; this is a job best left to the database. Adaptor::DBI simply translates queries to equivalent SQL queries, while Adaptor: : File implements a simple-minded scheme for file based objects: it converts the query expression to an evalable Perl expression and cycles through all objects, matching them against the query specification. Schema Evolution Let us say you have sent your objects' data to a file, and tomorrow, some more attributes are added to the object implementation. The schema is said to have evolved. The framework has

U.S. Patent No. 7,426,720	***Srinivasan* in view of *APA-Bach***
	to be able to reconcile old data with newer object implementations. Implementation This section explains the implementation of Adaptor::DBI and Adaptor::File. We will cover only the key procedures that perform query processing and file or database I/O. Pay as much attention to the design gotchas and unimplemented features as you do to the code. Adaptor::File An Adaptor::File instance represents all objects stored in one file. When this adaptor is created (using new), it reads the entire file and translates the data to in-memory objects. Slurping the entire file into memory avoids the problem of having to implement fancy on-disk schemes for random access to variable-length data; after all, that is the job of DBM and database implementations. For this reason, this approach is not recommended for large numbers of objects (over 1,000, to pick a number). The file adaptor has an attribute called all_instances, a hash table of all objects given to its store method (and indexed by their _id), as shown in Figure 11-2.  *Figure 11-2. Structure of file adaptor* *Srinivasan* at 179-80.
12. A method according to claim 11, further comprising: locating the source definition if the instantiated class definition is unavailable in the local cache.	*Srinivasan* provided a method for locating the source definition if the instantiated class definition is unavailable in the local cache. For example, *Srinivasan* discloses the "Adaptor: : File" function, which "converts [a] query expression to an evalable Perl expression and cycles through all objects, matching them against the query specification." *Srinivasan* at 179. In *Srinivasan*, "[o]bjects of a certain type are said to belong to a *class*." *Srinivasan* at 101. Thus, the exemplar disclosure effects a search through the available

U.S. Patent No. 7,426,720	*Srinivasan* in view of *APA-Bach*
	classes. It is inherent that such a search could locate a class definition that is unavailable in the local cache. "This returns a list of object references that match the query criteria. Now if you reissue this query, it is not too much to expect it to return an identical list of objects (the same object references, that is). This means that Adaptor has to keep an in-memory cache of objects that have been retrieved from disk in previous queries, so that if a database row is reread, the corresponding object is reused." *Srinivasan* at 178. "The Adaptor: : File module does not have this problem because it maintains a list of all objects given to its store () method (for reasons to be explained in the next section); hence successive identical queries return identical lists. Queries One big reason why object-oriented databases haven't caught on is the lack of a query language (or at least a standard query language). When you have a million objects in the database, it would be a terrible thing to load every single object in memory to see whether it matches your criteria; this is a job best left to the database. Adaptor::DBI simply translates queries to equivalent SQL queries, while Adaptor: : File implements a simple-minded scheme for file based objects: it converts the query expression to an evalable Perl expression and cycles through all objects, matching them against the query specification. Schema Evolution Let us say you have sent your objects' data to a file, and tomorrow, some more attributes are added to the object implementation. The schema is said to have evolved. The framework has to be able to reconcile old data with newer object implementations. Implementation This section explains the implementation of Adaptor::DBI and Adaptor::File. We will cover only the key procedures that perform query processing and file or database I/O. Pay as much attention to the design gotchas and unimplemented features as you do to the code. Adaptor::File An Adaptor::File instance represents all objects stored in one file. When this adaptor is created (using new), it reads the entire file and translates the data to in-memory objects. Slurping the entire file into memory avoids the problem of having to implement fancy

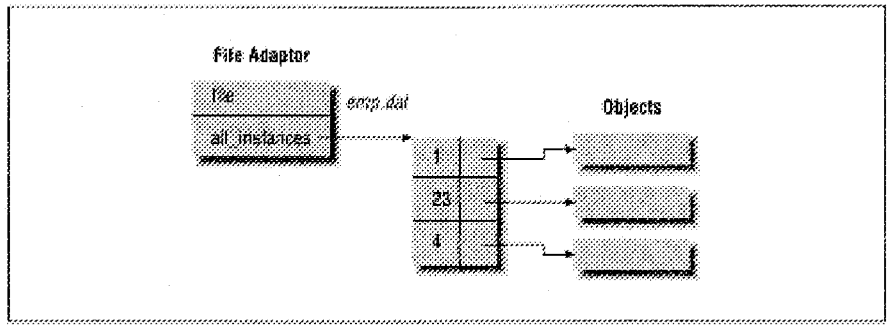
<table>
<tr><th>U.S. Patent No. 7,426,720</th><th>Srinivasan in view of APA-Bach</th></tr>
<tr><td></td><td>on-disk schemes for random access to variable-length data; after all, that is the job of DBM and database implementations. For this reason, this approach is not recommended for large numbers of objects (over 1,000, to pick a number).
The file adaptor has an attribute called all_instances, a hash table of all objects given to its store method (and indexed by their _id), as shown in Figure 11-2.


Figure 11-2. Structure of file adaptor

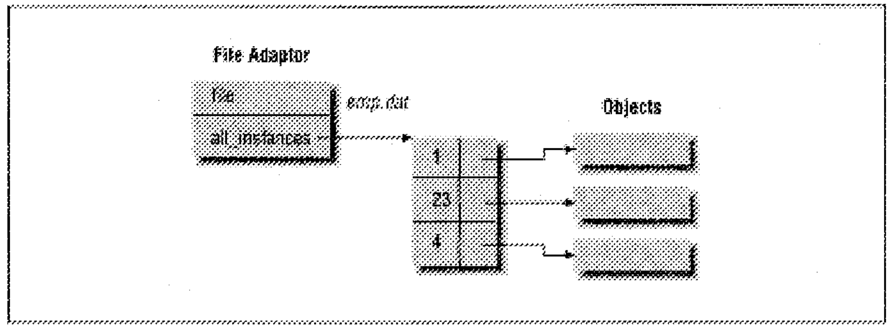
Srinivasan at 179-80.

“Adaptor::DBI is considerably simpler than Adaptor::File. It does not maintain a table of objects in memory; when asked to store an object, it sends it to the database, and when asked to retrieve one or more objects, it simply passes the request along to the database.” Srinivasan at 184.

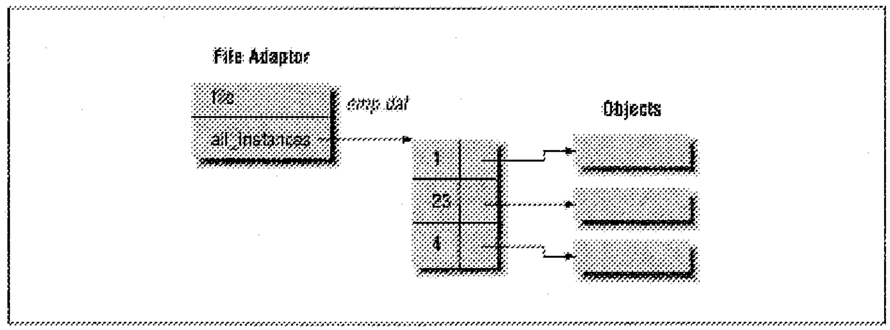
“retrieve_where creates a select query from the mapping information for that class. As was pointed out earlier, the same query executed twice will get you two different sets of objects, whose data are duplicates of the other…” Srinivasan at 186.</td></tr>
<tr><td>13. A method according to claim 10, further comprising: resolving the class definition.</td><td>Srinivasan provided a method for resolving the class definition. Srinivasan discloses that, in Perl, “[c]lass attributes are simply package global variables, and class methods are ordinary subroutines that don't work on any specific instance. Perl supports polymorphism and run-time binding for these ordinary subroutines (not just instance methods), which can be leveraged to produce a truly flexible design.” Srinivasan at 107-108.

“This returns a list of object references that match the query criteria. Now if you reissue this</td></tr>
</table>

U.S. Patent No. 7,426,720	*Srinivasan* in view of *APA-Bach*
	query, it is not too much to expect it to return an identical list of objects (the same object references, that is). This means that Adaptor has to keep an in-memory cache of objects that have been retrieved from disk in previous queries, so that if a database row is reread, the corresponding object is reused." *Srinivasan* at 178. "The Adaptor: : File module does not have this problem because it maintains a list of all objects given to its store () method (for reasons to be explained in the next section); hence successive identical queries return identical lists. Queries One big reason why object-oriented databases haven't caught on is the lack of a query language (or at least a standard query language). When you have a million objects in the database, it would be a terrible thing to load every single object in memory to see whether it matches your criteria; this is a job best left to the database. Adaptor::DBI simply translates queries to equivalent SQL queries, while Adaptor: : File implements a simple-minded scheme for file based objects: it converts the query expression to an evalable Perl expression and cycles through all objects, matching them against the query specification. Schema Evolution Let us say you have sent your objects' data to a file, and tomorrow, some more attributes are added to the object implementation. The schema is said to have evolved. The framework has to be able to reconcile old data with newer object implementations. Implementation This section explains the implementation of Adaptor::DBI and Adaptor::File. We will cover only the key procedures that perform query processing and file or database I/O. Pay as much attention to the design gotchas and unimplemented features as you do to the code. Adaptor::File An Adaptor::File instance represents all objects stored in one file. When this adaptor is created (using new), it reads the entire file and translates the data to in-memory objects. Slurping the entire file into memory avoids the problem of having to implement fancy on-disk schemes for random access to variable-length data; after all, that is the job of DBM and database implementations. For this reason, this approach is not recommended for large numbers of objects (over 1,000, to pick a number). The file adaptor has an attribute called all_instances, a hash table of all objects given to its store

U.S. Patent No. 7,426,720	*Srinivasan* in view of *APA-Bach*
	method (and indexed by their _id), as shown in Figure 11-2.  Figure 11-2. Structure of file adaptor *Srinivasan* at 179-80. "Adaptor::DBI is considerably simpler than Adaptor::File. It does not maintain a table of objects in memory; when asked to store an object, it sends it to the database, and when asked to retrieve one or more objects, it simply passes the request along to the database." *Srinivasan* at 184. "retrieve_where creates a select query from the mapping information for that class. As was pointed out earlier, the same query executed twice will get you two different sets of objects, whose data are duplicates of the other…" *Srinivasan* at 186.
14. A method according to claim 10, further comprising: maintaining the source definition as a class file on at least one of a local and remote file system.	*Srinivasan* provided a method for maintaining the source definition as a class file on at least one of a local and remote file system. *Srinivasan* discloses, for example, that "[i]n a typical client/server system, the server has the "real" objects. But the system is written in such a way that a client can remotely invoke a method of the object, with familiar OO syntax. For example, if a client program wants to invoke a method on a remote bank account, it should be able to say something like this:…" *Srinivasan* at 134. "You will not merely dabble with language syntax or the APIs of different modules as you read this book. You will spend just as much time dealing with real-world issues such as avoiding deadlocks during remote procedure calls and switching smoothly between data storage using a flat file or a database." *Srinivasan* at *xi*.

U.S. Patent No. 7,426,720	*Srinivasan* in view of *APA-Bach*
	"This technique is often employed in the guts of client/server libraries. In a typical client/server system, the server has the "real" objects. But the system is written in such a way that a client can remotely invoke a method of the object, with familiar OO syntax. For example, if a client program wants to invoke a method on a remote bank account, it should be able to say something like this:…" *Srinivasan* at 134. "This returns a list of object references that match the query criteria. Now if you reissue this query, it is not too much to expect it to return an identical list of objects (the same object references, that is). This means that Adaptor has to keep an in-memory cache of objects that have been retrieved from disk in previous queries, so that if a database row is reread, the corresponding object is reused." *Srinivasan* at 178. "The Adaptor: : File module does not have this problem because it maintains a list of all objects given to its store () method (for reasons to be explained in the next section); hence successive identical queries return identical lists. Queries One big reason why object-oriented databases haven't caught on is the lack of a query language (or at least a standard query language). When you have a million objects in the database, it would be a terrible thing to load every single object in memory to see whether it matches your criteria; this is a job best left to the database. Adaptor::DBI simply translates queries to equivalent SQL queries, while Adaptor: : File implements a simple-minded scheme for file based objects: it converts the query expression to an evalable Perl expression and cycles through all objects, matching them against the query specification. Schema Evolution Let us say you have sent your objects' data to a file, and tomorrow, some more attributes are added to the object implementation. The schema is said to have evolved. The framework has to be able to reconcile old data with newer object implementations. Implementation This section explains the implementation of Adaptor::DBI and Adaptor::File. We will cover only the key procedures that perform query processing and file or database I/O. Pay as much attention to the design gotchas and unimplemented features as you do to the code.

U.S. Patent No. 7,426,720	*Srinivasan* in view of *APA-Bach*
	Adaptor::File An Adaptor::File instance represents all objects stored in one file. When this adaptor is created (using new), it reads the entire file and translates the data to in-memory objects. Slurping the entire file into memory avoids the problem of having to implement fancy on-disk schemes for random access to variable-length data; after all, that is the job of DBM and database implementations. For this reason, this approach is not recommended for large numbers of objects (over 1,000, to pick a number). The file adaptor has an attribute called all_instances, a hash table of all objects given to its store method (and indexed by their _id), as shown in Figure 11-2.  *Figure 11-2. Structure of file adaptor* *Srinivasan* at 179-80. "Adaptor::DBI is considerably simpler than Adaptor::File. It does not maintain a table of objects in memory; when asked to store an object, it sends it to the database, and when asked to retrieve one or more objects, it simply passes the request along to the database." *Srinivasan* at 184. "retrieve_where creates a select query from the mapping information for that class. As was pointed out earlier, the same query executed twice will get you two different sets of objects, whose data are duplicates of the other…" *Srinivasan* at 186.
15. A method according to claim 10, further comprising: instantiating the child runtime system process by	*Srinivasan* provided the use of *fork* in Unix/Linux, which includes a method for instantiating the child runtime system process by copying the memory space of the master runtime system process into a separate memory space for the child runtime system process. *Srinivasan*

U.S. Patent No. 7,426,720	***Srinivasan* in view of *APA-Bach***
copying the memory space of the master runtime system process into a separate memory space for the child runtime system process.	discloses handling multiple clients by "[c]reat[ing] multiple threads of control." *Srinivasan* at 193. In order to create multiple threads of control, *Srinivasan* invokes the fork( ) system call to create "process-level parallelism." *See id*. at 194. This creates a new process, called the child process. "The newly created child process meanwhile has a copy of its parent's environment and shares all open file descriptors." *See id.* Thus, *Srinivasan* discloses a "child" runtime system process, which is in fact a clone of the parent runtime system process. "Perl doesn't have threads yet (at least not officially*), but on Unix and similarly empowered systems, it supports `fork`, the way to get process-level parallelism. The server process acts as a full-time receptionist: it blocks on `accept`, and when a connection request comes in, it spawns a child process and goes back to `accept`. The newly created child process meanwhile has a copy of its parent's environment and shares all open file descriptors. Hence it is able to read from, and write to, the new socket returned by `accept`. When the child is done with the conversation, it simply exits. Each process is therefore dedicated to its own task and doesn't interfere with the other. The following code shows an example of a forking server:

```
# Forking server
use IO:: Socket;
$SIG{CHLD} sub {wait ()};
$main_sock = new IO::Socket::INET (LocalHost => 'goldengate',
                                   LocalPort => 1200,
                                   Listen => 5,
                                   Proto => 'tcp',
                                   Reuse => 1,
                                  );
die "Socket could not be created. Reason: $!\n" unless ($sock);
while ($new_sock = $main_sock->accept()) {
      $pid = fork();
      die "Cannot fork: $!" unless defined ($pid) ;
      if ($pid == 0) {
            # Child process
            while (defined ($buf <$new_sock»)
                  # do something with $buf ....
                  print $new_sock "You said: $buf\n";
```

U.S. Patent No. 7,426,720	*Srinivasan* in view of *APA-Bach*

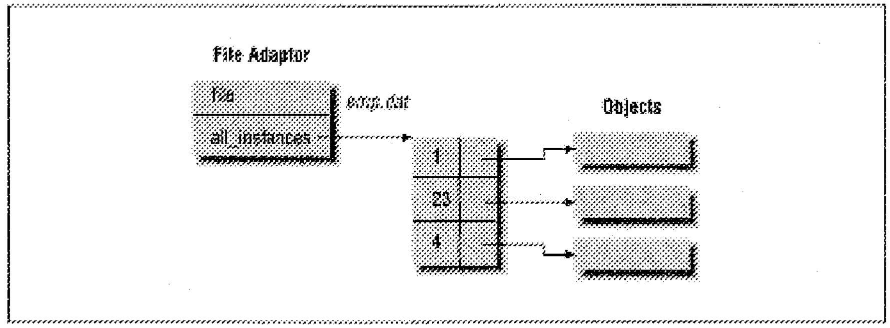
	`}`
	`exit(0); # Child process exits when it is done.`
	`# else 'tis the parent process, which goes back to accept()`
	`}`
	`close ($main_sock);`
	
	The `fork` call results in two identical processes-the parent and child-starting from the statement following the `fork`. The parent gets a positive return value, the process ID (`$pid`) of the child process. Both processes check this return value and execute their own logic; the main process goes back to `accept`, and the child process reads a line from the socket and echoes it back to the client.
	Incidentally, the `CHLD` signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if the child was able to execute `exit()`, or a termination status otherwise), just in case the parent uses `wait` or `waitpid` to enquire about this status. The terminated child process is also known as a *zombie* process, and it is always a good thing to remove it using `wait`; otherwise, the process tables keep filling up with junk. In the preceding code, `wait` doesn't block, because it is called only when we know for sure that a child process has died-the `CHLD` signal arranges that for us. Be sure to read the online documentation for quirks associated with signals in general and `SIGCHLD` in particular.
	
	* Malcolm Beattie has a working prototype of a threaded Perl interpreter, which will be incorporated into the mainstream in the Perl 5.005 release." *Srinivasan* at 194-95.
	
	This limitation is also disclosed by *APA-Bach*, as below:
	
	"The only way for a user to create a new process in the UNIX operating system is to invoke the *fork* system call." *APA-Bach* at 192.
	
	"The *copy-on-write* bit, used in the *fork* system call, indicates that the kernel must create a new copy of the page when a process modifies its contents." *APA-Bach* at 287.

U.S. Patent No. 7,426,720	*Srinivasan* in view of *APA-Bach*
	"9.2.1.1 Fork in a Paging System As explained in Section 7.1, the kernel duplicates every region of the parent process during the *fork* system call and attaches it to the child process. Traditionally, the kernel of a swapping system makes a physical copy of the parent's address space, usually a wasteful operation, because processes often call *exec* soon after the *fork* call and immediately free the memory just copied. On the System V paging system, the kernel avoids copying the page by manipulating the region tables, page table entries, and pfdata table entries: It simply increments the region reference count of shared regions.... The page can now be referenced through both regions, which share the page until a process writes to it. The kernel then copies the page so that each region has a private version. To do this, the kernel turns on the 'copy on write' bit for every page table entry in private regions of the parent and child processes during *fork*. If either process writes the page, it incurs a protection fault, and in handling the fault, the kernel makes a new copy of the page for the faulting process. The physical copying of the page is thus deferred until a process really needs it." *APA-Bach* at 289–90.
16. A method according to claim 10, further comprising: causing the master runtime system process to sleep relative to receiving the process request.	*Srinivasan* provided a method for causing the master runtime system process to sleep relative to receiving the process request. For example, *Srinivasan* discloses a persistent process, "also known as a zombie process." *Srinivasan* at 195. "If you want a finer granularity, you can use the Time: : HiRes module available from CPAN, which gives microsecond resolution on Unix systems (gives access to the usleep and ualarm system calls) On Microsoft Windows systems, you can use the Win32 : : Timer call for millisecond-level timing." *Srinivasan* at 141. "The `fork` call results in two identical processes-the parent and child-starting from the statement following the `fork`. The parent gets a positive return value, the process ID (`$pid`) of the child process. Both processes check this return value and execute their own logic; the main process goes back to `accept`, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the `CHLD` signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other

U.S. Patent No. 7,426,720	***Srinivasan* in view of *APA-Bach***
	resources associated with it. But it retains a small amount of information (the exit status if the child was able to execute `exit()`, or a termination status otherwise), just in case the parent uses `wait` or `waitpid` to enquire about this status. The terminated child process is also known as a *zombie* process, and it is always a good thing to remove it using `wait`; otherwise, the process tables keep filling up with junk. In the preceding code, `wait` doesn't block, because it is called only when we know for sure that a child process has died-the `CHLD` signal arranges that for us. Be sure to read the online documentation for quirks associated with signals in general and `SIGCHLD` in particular." *Srinivasan* at 195.
17. A method according to claim 10, wherein the object-oriented program code is written in the Java programming language.	*Srinivasan* provided a method wherein the object-oriented program code is written in the Java programming language. For example, *Srinivasan* discloses that "Java offers two levels of modularity: packages and classes, where a package is a collection of classes. (We'll learn about the notion of classes in the next chapter.) Perl's package is equivalent to both." *Srinivasan* at 98. "Java offers two levels of modularity: packages and classes, where a package is a collection of classes. (We'll learn about the notion of classes in the next chapter.) Perl's package is equivalent to both. Java does not allow one package to mess around with another package's namespace (no export) but allows a package to selectively import the classes it requires. It focuses a considerable amount of attention on security, which hasn't really stopped determined crackers. The Perl world has third-party packages called Safe and Penguin (which depends on Safe) that attempt to provide similar isolation characteristics (and don't offer any security guarantees either). Since the arrival of the Java Beans and the 1.1 version of the Java Development Kit ODK), Java has gained significant reflection capabilities, though nowhere near the amount of publicly available information Perl gives you. There are reasonably good arguments to be made both for providing this information and for not providing it; everything comes down to different models of programming. Men were sent to the moon while FORTRAN and COBOL ruled the roost, which proves that you can get a whole lot done if you don't indulge in language wars. Java allows you to dynamically "dispatch" a function call, by giving the function's name as a string, and to trap an exception if the function doesn't exist; this is like using Perl's symbolic references." *Srinivasan* at 98.

U.S. Patent No. 7,426,720	*Srinivasan* in view of ***APA-Bach***
	"The translator converts a Perl script into a tree of opcodes (explained below). It comprises a hand-coded lexer (*toke.c*), the yacc-based parser (perly.y) , and the code generator (op.c). Regular expressions–which form a distinct sublanguage–are recognized in *toke.c* and compiled to an internal format in *regcomp.c*. Opcodes are similar in concept to machine code; while machine code is executed by hardware, opcodes (sometimes called byte-codes or p-code) are executed by a "virtual machine." The similarity ends there. Modern interpreters never emulate the workings of a hardware CPU, for performance reasons. Instead, they create complex structures primed for execution, such that each opcode directly contains a pointer to the next one to execute and a pointer to the data it is expected to work on at run-time. In other words, these opcodes are not mere instruction types; they actually embody the exact unit of work expected at that point in that program. Java and Perl are both examples of such interpreters. While many of java's bytecodes resemble a RISC machine's instruction set, Perl's opcodes represent a much higher level of abstraction. A large number of these opcodes directly correspond to the facilities available at the scripting level, such as regular expression matching and substitution, `chop`, `push`, `index`, `rindex`, `grep`,* and so on, which explains why there are 343 opcodes as of this writing! It also explains why Perl is so fast: instead of spending time in the interpreter, most of the work is done in lovingly hand-optimized e code. You can also see why it is hard to create a Perl-to-Java byte-code translator: there is no correspondence between the two sets." *Srinivasan* at 323-24.
19. A computer-readable storage medium holding code for performing the method according to claim 10.	*Srinivasan* provided a computer-readable storage medium holding code for performing the method according to claim 10. For example, *Srinivasan* references an "in-memory cache of objects." *Srinivasan* at 178. "This returns a list of object references that match the query criteria. Now if you reissue this query, it is not too much to expect it to return an identical list of objects (the same object references, that is). This means that Adaptor has to keep an in-memory cache of objects that have been retrieved from disk in previous queries, so that if a database row is reread, the corresponding object is reused." *Srinivasan* at 178.

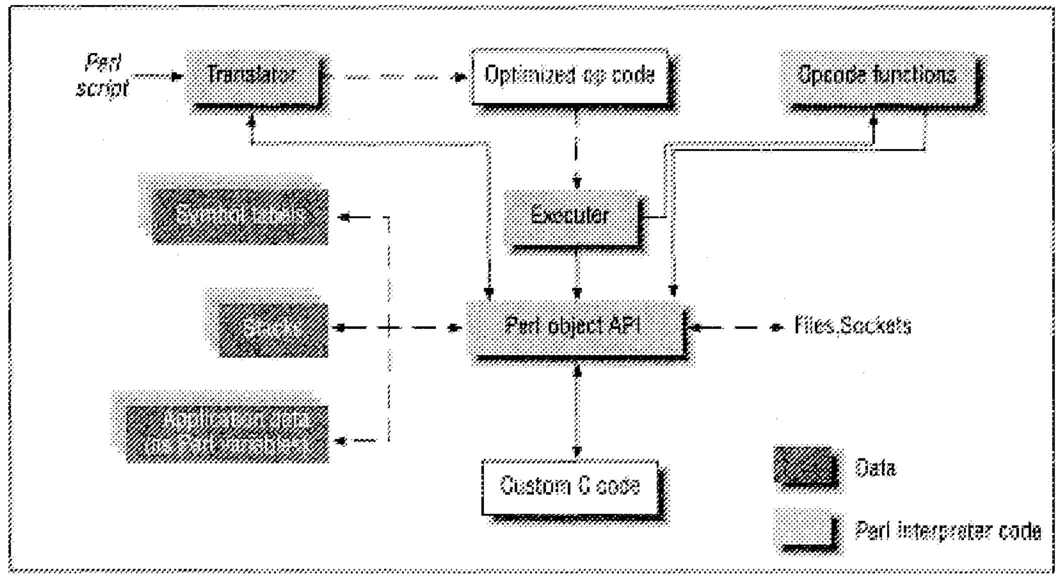
U.S. Patent No. 7,426,720	*Srinivasan* in view of *APA-Bach*
	"The Adaptor: : File module does not have this problem because it maintains a list of all objects given to its store () method (for reasons to be explained in the next section); hence successive identical queries return identical lists. Queries One big reason why object-oriented databases haven't caught on is the lack of a query language (or at least a standard query language). When you have a million objects in the database, it would be a terrible thing to load every single object in memory to see whether it matches your criteria; this is a job best left to the database. Adaptor::DBI simply translates queries to equivalent SQL queries, while Adaptor: : File implements a simple-minded scheme for file based objects: it converts the query expression to an evalable Perl expression and cycles through all objects, matching them against the query specification. Schema Evolution Let us say you have sent your objects' data to a file, and tomorrow, some more attributes are added to the object implementation. The schema is said to have evolved. The framework has to be able to reconcile old data with newer object implementations. Implementation This section explains the implementation of Adaptor::DBI and Adaptor::File. We will cover only the key procedures that perform query processing and file or database I/O. Pay as much attention to the design gotchas and unimplemented features as you do to the code. Adaptor::File An Adaptor::File instance represents all objects stored in one file. When this adaptor is created (using new), it reads the entire file and translates the data to in-memory objects. Slurping the entire file into memory avoids the problem of having to implement fancy on-disk schemes for random access to variable-length data; after all, that is the job of DBM and database implementations. For this reason, this approach is not recommended for large numbers of objects (over 1,000, to pick a number). The file adaptor has an attribute called all_instances, a hash table of all objects given to its store method (and indexed by their _id), as shown in Figure 11-2.

U.S. Patent No. 7,426,720	*Srinivasan* in view of *APA-Bach*
	 *Figure 11-2. Structure of file adaptor* *Srinivasan* at 179-80.
20. An apparatus for dynamic preloading of classes through memory space cloning of a master runtime system process, comprising:	*Srinivasan* provided an apparatus for dynamic preloading of classes through memory space cloning of a master runtime system process.
A processor; A memory means for executing a master runtime system process;	*Srinivasan* provided an apparatus with a processor and a memory means for executing a master runtime system process. For example, *Srinivasan* references an "in-memory cache of objects." *Srinivasan* at 178.
means for obtaining a representation of at least one class from a source definition provided as object-oriented program code;	*Srinivasan* provided an apparatus with a means for obtaining a representation of at least one class from a source definition provided as object-oriented program code. *Srinivasan* discusses object oriented programming and its usefulness in conjunction with Perl code. *Srinivasan* at 101. *Srinivasan* notes that, in Perl, a "class is a package." *Srinivasan* at 389. *Srinivasan* then discusses two methods for the creation of a package from source definition provided as object oriented program code. *See id.* at 389-390 (discussing creation of the "employee" class). In order to preload the class, *Srinivasan* discloses specific functional code: "Using object package: use Employee;

U.S. Patent No. 7,426,720	***Srinivasan* in view of *APA-Bach***
	$emp = Employee->new ("Ada", 3 5); $emp->set_salary(1000); *See id.* at 390. "Java offers two levels of modularity: packages and classes, where a package is a collection of classes. (We'll learn about the notion of classes in the next chapter.) Perl's package is equivalent to both. Java does not allow one package to mess around with another package's namespace (no export) but allows a package to selectively import the classes it requires. It focuses a considerable amount of attention on security, which hasn't really stopped determined crackers. The Perl world has third-party packages called Safe and Penguin (which depends on Safe) that attempt to provide similar isolation characteristics (and don't offer any security guarantees either). Since the arrival of the Java Beans and the 1.1 version of the Java Development Kit ODK), Java has gained significant reflection capabilities, though nowhere near the amount of publicly available information Perl gives you. There are reasonably good arguments to be made both for providing this information and for not providing it; everything comes down to different models of programming. Men were sent to the moon while FORTRAN and COBOL ruled the roost, which proves that you can get a whole lot done if you don't indulge in language wars. Java allows you to dynamically "dispatch" a function call, by giving the function's name as a string, and to trap an exception if the function doesn't exist; this is like using Perl's symbolic references." *Srinivasan* at 98. "Object orientation (OO) is the latest software methodology to occupy the airwaves, hyped to a point where the term "object-oriented design" seems to automatically imply a good design. In this chapter, we will study what the noise is all about and build objects using Perl. I will leave it to the plethora of 00 literature to convince you that there is a respectable middle-ground and that the object revolution is indeed a good thing." *Srinivasan* at 99. "Objects of a certain type are said to belong to a *class*." *Srinivasan* at 101. "The translator converts a Perl script into a tree of opcodes (explained below). It comprises a

U.S. Patent No. 7,426,720	*Srinivasan* in view of ***APA-Bach***
	hand-coded lexer (*toke.c*), the yacc-based parser (perly.y) , and the code generator (op.c). Regular expressions–which form a distinct sublanguage–are recognized in *toke.c* and compiled to an internal format in *regcomp.c*. Opcodes are similar in concept to machine code; while machine code is executed by hardware, opcodes (sometimes called byte-codes or p-code) are executed by a "virtual machine." The similarity ends there. Modern interpreters never emulate the workings of a hardware CPU, for performance reasons. Instead, they create complex structures primed for execution, such that each opcode directly contains a pointer to the next one to execute and a pointer to the data it is expected to work on at run-time. In other words, these opcodes are not mere instruction types; they actually embody the exact unit of work expected at that point in that program. Java and Perl are both examples of such interpreters. While many of java's bytecodes resemble a RISC machine's instruction set, Perl's opcodes represent a much higher level of abstraction. A large number of these opcodes directly correspond to the facilities available at the scripting level, such as regular expression matching and substitution, `chop`, `push`, `index`, `rindex`, `grep`,* and so on, which explains why there are 343 opcodes as of this writing! It also explains why Perl is so fast: instead of spending time in the interpreter, most of the work is done in lovingly hand-optimized e code. You can also see why it is hard to create a Perl-to-Java byte-code translator: there is no correspondence between the two sets." *Srinivasan* at 323-24. "Malcolm has also submitted a Perl compiler extension [5], which is in its early stages as of this writing. It can be asked to translate a script to C code, which can be compiled to form an executable; as it happens, this executable is not much faster than the interpreted script, because most of the action still takes place in opcode functions as they exist now. Static typing hints may usher in some aggressive optimizations. For example, if you say: `my integer $i;` the compiler would use C's native integer type, rather than an SV-this would speed up loops and arithmetic expressions. The compiler can alternatively produce a byte-code file and have the interpreter `eval` it subsequently, similar to the facilities provided by Python and Java. It also supports much better debugging options than those currently provided with *–D*." *Srinivasan* at 370.

U.S. Patent No. 7,426,720	*Srinivasan* in view of *APA-Bach*
means for interpreting and means for instantiating the representation as a class definition in a memory space of the master runtime system process;	*Srinivasan* provided an apparatus with a means for interpreting and means for instantiating the representation as a class definition in a memory space of the master runtime system process. *Srinivasan* discloses that, in Perl, "[c]lass attributes are simply package global variables, and class methods are ordinary subroutines that don't work on any specific instance. Perl supports polymorphism and run-time binding for these ordinary subroutines (not just instance methods), which can be leveraged to produce a truly flexible design." *Srinivasan* at 107-108. "The data structures described above are normally kept in global C variables. If Perl is compiled with *–DMULTIPLICITY*, it lumps all these global variables into a structure called Perl Interpreter. This allows you to have multiple instances of the interpreter, each with its own "global" space. (Recall from Chapter 19 the API to allocate and construct an object of type Perl Interpreter.) In the absence of this compile-time option, the Perl Interpreter object is a dummy structure, and the internal data structures are truly global, for maximum performance. The API remains the same in either case. You can use multiple interpreters to enforce completely isolated namespaces. Each interpreter has its own "main" package and its own tree of loaded packages. I have not seen this feature used in production Perl applications, but Tel provides a framework called SafeTel for security purposes, which uses a similar feature of multiple interpreter objects. These interpreters can be unrestricted or restricted. The equivalent module in Perl, Safe, uses a different mechanism, though the result (of isolated name spaces) is similar. More on this in the next section." *Srinivasan* at 323.

U.S. Patent No. 7,426,720	*Srinivasan* in view of *APA-Bach*
	Figure 20-1 shows the various components of a running Perl system. Shaded rectangles represent data structures, some of which can have multiple instances in a program. The source code can also be partitioned roughly along these lines.  *Figure 20-1. Snapshot of a running system* *Srinivasan* at 321.
and means for cloning the memory space as a child runtime system process responsive to a process request and means for executing the child runtime system process;	*Srinivasan* provided an apparatus with a means for cloning the memory space as a child runtime system process responsive to a process request and means for executing the child runtime system process. *Srinivasan* discloses handling multiple clients by "[c]reat[ing] multiple threads of control." *Srinivasan* at 193. In order to create multiple threads of control, *Srinivasan* invokes the fork( ) system call to create "process-level parallelism." *See id*. at 194. This creates a new process, called the child process. "The newly created child process meanwhile has a copy of its parent's environment and shares all open file descriptors." *See id.* Thus, *Srinivasan* discloses a "child" runtime system process, which is in fact a clone of the parent runtime system process.

U.S. Patent No. 7,426,720	*Srinivasan* in view of *APA-Bach*

"Perl doesn't have threads yet (at least not officially*), but on Unix and similarly empowered systems, it supports `fork`, the way to get process-level parallelism. The server process acts as a full-time receptionist: it blocks on `accept`, and when a connection request comes in, it spawns a child process and goes back to `accept`. The newly created child process meanwhile has a copy of its parent's environment and shares all open file descriptors. Hence it is able to read from, and write to, the new socket returned by `accept`. When the child is done with the conversation, it simply exits. Each process is therefore dedicated to its own task and doesn't interfere with the other. The following code shows an example of a forking server:

```
# Forking server
use IO:: Socket;
$SIG{CHLD} sub {wait ()};
$main_sock = new IO::Socket::INET (LocalHost => 'goldengate',
                                   LocalPort => 1200,
                                   Listen => 5,
                                   Proto => 'tcp',
                                   Reuse => 1,
                                  );
die "Socket could not be created. Reason: $!\n" unless ($sock);
while ($new_sock = $main_sock->accept()) {
      $pid = fork();
      die "Cannot fork: $!" unless defined ($pid) ;
      if ($pid == 0) {
            # Child process
            while (defined ($buf <$new_sock»)
                  # do something with $buf ....
                  print $new_sock "You said: $buf\n";
            }
            exit(O); # Child process exits when it is done.
      # else 'tis the parent process, which goes back to accept()
}
close ($main_sock);
```

The `fork` call results in two identical processes-the parent and child-starting from the statement following the `fork`. The parent gets a positive return value, the process ID (`$pid`)

U.S. Patent No. 7,426,720	***Srinivasan* in view of *APA-Bach***
	of the child process. Both processes check this return value and execute their own logic; the main process goes back to `accept`, and the child process reads a line from the socket and echoes it back to the client. Incidentally, the `CHLD` signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if the child was able to execute `exit()`, or a termination status otherwise), just in case the parent uses `wait` or `waitpid` to enquire about this status. The terminated child process is also known as a *zombie* process, and it is always a good thing to remove it using `wait`; otherwise, the process tables keep filling up with junk. In the preceding code, `wait` doesn't block, because it is called only when we know for sure that a child process has died-the `CHLD` signal arranges that for us. Be sure to read the online documentation for quirks associated with signals in general and `SIGCHLD` in particular. * Malcolm Beattie has a working prototype of a threaded Perl interpreter, which will be incorporated into the mainstream in the Perl 5.005 release." *Srinivasan* at 194-95.
wherein the means for cloning the memory space is configured to clone the memory space of a child runtime system process using a copy-on-write process cloning mechanism that instantiates the child runtime system process by copying references to the memory space of the master runtime system process into a separate memory space for the child runtime system process and that defers copying of the memory space of the master runtime system process until the child runtime system process needs to modify the referenced memory space of the master runtime system process.	*Srinivasan* in view of *APA-Bach* provided the use of *fork* in Unix/Linux, which includes an apparatus wherein the means for cloning the memory space is configured to clone the memory space of a child runtime system process using a copy-on-write process cloning mechanism that instantiates the child runtime system process by copying references to the memory space of the master runtime system process into a separate memory space for the child runtime system process and that defers copying of the memory space of the master runtime system process until the child runtime system process needs to modify the referenced memory space of the master runtime system process. The fork( ) system call disclosed by *Srinivasan* was commonly used in conjunction with the copy-on-write mechanism to further streamline the impact on system memory; this is described in detail in *APA-Bach*. One of ordinary skill in the art at the time of the invention, seeking to reduce the impact of a fork( ) system call on local memory, would look to the disclosure of *APA-Bach* for a description of the copy-on-write bit. "The only way for a user to create a new process in the UNIX operating system is to invoke

U.S. Patent No. 7,426,720	***Srinivasan* in view of *APA-Bach***
	the *fork* system call." *APA-Bach* at 192. "The *copy-on-write* bit, used in the *fork* system call, indicates that the kernel must create a new copy of the page when a process modifies its contents." *APA-Bach* at 287. "9.2.1.1 Fork in a Paging System As explained in Section 7.1, the kernel duplicates every region of the parent process during the *fork* system call and attaches it to the child process. Traditionally, the kernel of a swapping system makes a physical copy of the parent's address space, usually a wasteful operation, because processes often call *exec* soon after the *fork* call and immediately free the memory just copied. On the System V paging system, the kernel avoids copying the page by manipulating the region tables, page table entries, and pfdata table entries: It simply increments the region reference count of shared regions.... The page can now be referenced through both regions, which share the page until a process writes to it. The kernel then copies the page so that each region has a private version. To do this, the kernel turns on the 'copy on write' bit for every page table entry in private regions of the parent and child processes during *fork*. If either process writes the page, it incurs a protection fault, and in handling the fault, the kernel makes a new copy of the page for the faulting process. The physical copying of the page is thus deferred until a process really needs it." *APA-Bach* at 289–90.
21. A system according to claim 1, further comprising: a resource controller to set operating system level resource management parameters on the child runtime system process.	*Srinivasan* provided an apparatus with a resource controller to set operating system level resource management parameters on the child runtime system process. For example, "[o]n Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it." *Srinivasan* at 195. "Perl doesn't have threads yet (at least not officially*), but on Unix and similarly empowered systems, it supports `fork`, the way to get process-level parallelism. The server process acts as a full-time receptionist: it blocks on `accept`, and when a connection request comes in, it spawns a child process and goes back to `accept`. The newly created child process meanwhile has a copy of its parent's environment and shares all open file descriptors. Hence it is able to read from, and write to, the new socket returned by `accept`. When the child is

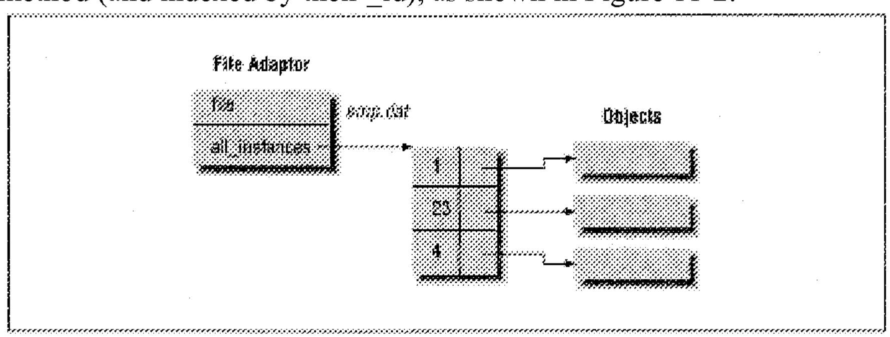
U.S. Patent No. 7,426,720	*Srinivasan* in view of *APA-Bach*
	done with the conversation, it simply exits. Each process is therefore dedicated to its own task and doesn't interfere with the other. The following code shows an example of a forking server:

```
# Forking server
use IO:: Socket;
$SIG{CHLD} sub {wait ()};
$main_sock = new IO::Socket::INET (LocalHost => 'goldengate',
                                   LocalPort => 1200,
                                   Listen => 5,
                                   Proto => 'tcp',
                                   Reuse => 1,
                                  );
die "Socket could not be created. Reason: $!\n" unless ($sock);
while ($new_sock = $main_sock->accept()) {
      $pid = fork();
      die "Cannot fork: $!" unless defined ($pid) ;
      if ($pid == 0) {
            # Child process
            while (defined ($buf <$new_sock»)
                  # do something with $buf ....
                  print $new_sock "You said: $buf\n";
            }
            exit(O); # Child process exits when it is done.
      # else 'tis the parent process, which goes back to accept()
}
close ($main_sock);
```

The `fork` call results in two identical processes-the parent and child-starting from the statement following the `fork`. The parent gets a positive return value, the process ID (`$pid`) of the child process. Both processes check this return value and execute their own logic; the main process goes back to `accept`, and the child process reads a line from the socket and echoes it back to the client.

Incidentally, the `CHLD` signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if the child was able to execute `exit()`, or a termination status otherwise), just in case the

U.S. Patent No. 7,426,720	***Srinivasan* in view of *APA-Bach***
	parent uses `wait` or `waitpid` to enquire about this status. The terminated child process is also known as a *zombie* process, and it is always a good thing to remove it using `wait`; otherwise, the process tables keep filling up with junk. In the preceding code, `wait` doesn't block, because it is called only when we know for sure that a child process has died-the `CHLD` signal arranges that for us. Be sure to read the online documentation for quirks associated with signals in general and `SIGCHLD` in particular. * Malcolm Beattie has a working prototype of a threaded Perl interpreter, which will be incorporated into the mainstream in the Perl 5.005 release." *Srinivasan* at 194-95. "The op-ppaddr pointer represents the essence of the opcode: it is a pointer to a built-in function-call it an opcode function-that implements the functionality of the opcode. All opcode functions are prefixed with pp (pp-push, pp_grep, and so on) and are distributed over pp.c, pp_ctl.c, pp_sys.c, and pp_hot.c. The last one contains the opcode functions that are "hot," or frequently executed, so it is likely to remain within the cache of most RIse systems. Tom Christiansen once mentioned that this feature is also true of the regular expression-matching code, which is why regex matchers written in Java won't come anywhere close in performance. (I'll reevaluate this claim once Sun's Java processors are freely available.) As you will see later on, the opcode functions look strikingly similar to the glue code output by xsubpp/SWJG; this is because they interoperate using the argument stack and obey the same parameter passing protocols." *Srinivasan* at 324. "This returns a list of object references that match the query criteria. Now if you reissue this query, it is not too much to expect it to return an identical list of objects (the same object references, that is). This means that Adaptor has to keep an in-memory cache of objects that have been retrieved from disk in previous queries, so that if a database row is reread, the corresponding object is reused." *Srinivasan* at 178. "The Adaptor: : File module does not have this problem because it maintains a list of all objects given to its store () method (for reasons to be explained in the next section); hence successive identical queries return identical lists. Queries One big reason why object-oriented databases haven't caught on is the lack of a query

U.S. Patent No. 7,426,720	***Srinivasan*** **in view of** ***APA-Bach***
	language (or at least a standard query language). When you have a million objects in the database, it would be a terrible thing to load every single object in memory to see whether it matches your criteria; this is a job best left to the database. Adaptor::DBI simply translates queries to equivalent SQL queries, while Adaptor: : File implements a simple-minded scheme for file based objects: it converts the query expression to an evalable Perl expression and cycles through all objects, matching them against the query specification. Schema Evolution Let us say you have sent your objects' data to a file, and tomorrow, some more attributes are added to the object implementation. The schema is said to have evolved. The framework has to be able to reconcile old data with newer object implementations. Implementation This section explains the implementation of Adaptor::DBI and Adaptor::File. We will cover only the key procedures that perform query processing and file or database I/O. Pay as much attention to the design gotchas and unimplemented features as you do to the code. Adaptor::File An Adaptor::File instance represents all objects stored in one file. When this adaptor is created (using new), it reads the entire file and translates the data to in-memory objects. Slurping the entire file into memory avoids the problem of having to implement fancy on-disk schemes for random access to variable-length data; after all, that is the job of DBM and database implementations. For this reason, this approach is not recommended for large numbers of objects (over 1,000, to pick a number). The file adaptor has an attribute called all_instances, a hash table of all objects given to its store method (and indexed by their _id), as shown in Figure 11-2.  *Figure 11-2. Structure of file adaptor*

<table>
<tr><th>U.S. Patent No. 7,426,720</th><th>Srinivasan in view of APA-Bach</th></tr>
<tr><td></td><td>Srinivasan at 179-80.

“Perl automatically garbage collects a data structure when its reference count drops to zero. If a data structure has been blessed into a module, Perl allows that module to perform some clean-up before it destroys the object, by calling a special procedure in that module called DESTROY and passing it the reference to the object to be destroyed:

<code>package Employee;</code>
<code>sub DESTROY {</code>
<code>   my ($emp) = @_;</code>
<code>   print "Alas, ", $emp->{"name"}, "is now no longer with us \n";</code>
<code>}</code>

This is similar to C++'s destructor or the finalize () method in Java in that Perl does the memory management automatically, but you get a chance to do something before the object is reclaimed. (Unlike Java's finalize, Perl's garbage collection is deterministic; DESTROY is called as soon as the object is not being referred to any more.)” Srinivasan at 112.</td></tr>
<tr><td>22. A method according to claim 10, further comprising: setting operating system level resource management parameters on the child runtime system process.</td><td>Srinivasan provided an apparatus with a setting operating system level resource management parameters on the child runtime system process. For example, “[o]n Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it.” Srinivasan at 195.

“Perl doesn't have threads yet (at least not officially*), but on Unix and similarly empowered systems, it supports <code>fork</code>, the way to get process-level parallelism. The server process acts as a full-time receptionist: it blocks on <code>accept</code>, and when a connection request comes in, it spawns a child process and goes back to <code>accept</code>. The newly created child process meanwhile has a copy of its parent's environment and shares all open file descriptors. Hence it is able to read from, and write to, the new socket returned by <code>accept</code>. When the child is done with the conversation, it simply exits. Each process is therefore dedicated to its own task and doesn't interfere with the other. The following code shows an example of a forking server:</td></tr>
</table>

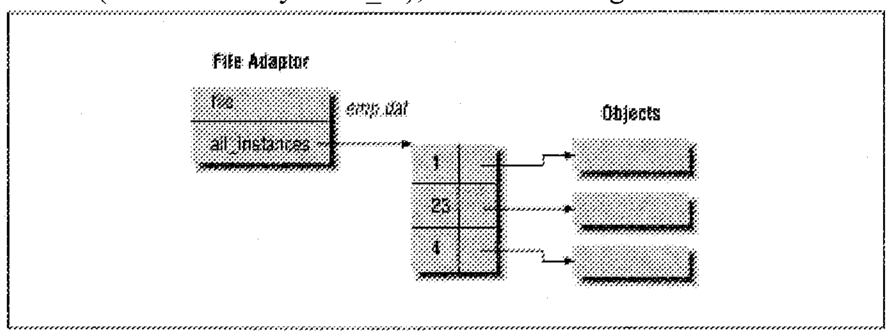
U.S. Patent No. 7,426,720	*Srinivasan* in view of ***APA-Bach***

```
# Forking server
use IO:: Socket;
$SIG{CHLD} sub {wait ()};
$main_sock = new IO::Socket::INET (LocalHost => 'goldengate',
                                   LocalPort => 1200,
                                   Listen => 5,
                                   Proto => 'tcp',
                                   Reuse => 1,
                                  );
die "Socket could not be created. Reason: $!\n" unless ($sock);
while ($new_sock = $main_sock->accept()) {
      $pid = fork();
      die "Cannot fork: $!" unless defined ($pid) ;
      if ($pid == 0) {
            # Child process
            while (defined ($buf <$new_sock»)
                  # do something with $buf ....
                  print $new_sock "You said: $buf\n";
            }
            exit(O); # Child process exits when it is done.
      # else 'tis the parent process, which goes back to accept()
}
close ($main_sock);
```

The `fork` call results in two identical processes-the parent and child-starting from the statement following the `fork`. The parent gets a positive return value, the process ID (`$pid`) of the child process. Both processes check this return value and execute their own logic; the main process goes back to `accept`, and the child process reads a line from the socket and echoes it back to the client.

Incidentally, the `CHLD` signal has nothing to do with IPC per se. On Unix, when a child process exits (or terminates abnormally), the system gets rid of the memory, files, and other resources associated with it. But it retains a small amount of information (the exit status if the child was able to execute `exit()`, or a termination status otherwise), just in case the parent uses `wait` or `waitpid` to enquire about this status. The terminated child process is also known as a *zombie* process, and it is always a good thing to remove it using `wait`; otherwise, the process tables keep filling up with junk. In the preceding code, `wait` doesn't

U.S. Patent No. 7,426,720	*Srinivasan* in view of *APA-Bach*
	block, because it is called only when we know for sure that a child process has died-the `CHLD` signal arranges that for us. Be sure to read the online documentation for quirks associated with signals in general and `SIGCHLD` in particular. * Malcolm Beattie has a working prototype of a threaded Perl interpreter, which will be incorporated into the mainstream in the Perl 5.005 release." *Srinivasan* at 194-95. "The op-ppaddr pointer represents the essence of the opcode: it is a pointer to a built-in function-call it an opcode function-that implements the functionality of the opcode. All opcode functions are prefixed with pp (pp-push, pp_grep, and so on) and are distributed over pp.c, pp_ctl.c, pp_sys.c, and pp_hot.c. The last one contains the opcode functions that are "hot," or frequently executed, so it is likely to remain within the cache of most RIse systems. Tom Christiansen once mentioned that this feature is also true of the regular expression-matching code, which is why regex matchers written in Java won't come anywhere close in performance. (I'll reevaluate this claim once Sun's Java processors are freely available.) As you will see later on, the opcode functions look strikingly similar to the glue code output by xsubpp/SWJG; this is because they interoperate using the argument stack and obey the same parameter passing protocols." *Srinivasan* at 324. "This returns a list of object references that match the query criteria. Now if you reissue this query, it is not too much to expect it to return an identical list of objects (the same object references, that is). This means that Adaptor has to keep an in-memory cache of objects that have been retrieved from disk in previous queries, so that if a database row is reread, the corresponding object is reused." *Srinivasan* at 178. "The Adaptor: : File module does not have this problem because it maintains a list of all objects given to its store () method (for reasons to be explained in the next section); hence successive identical queries return identical lists. Queries One big reason why object-oriented databases haven't caught on is the lack of a query language (or at least a standard query language). When you have a million objects in the database, it would be a terrible thing to load every single object in memory to see whether it matches your criteria; this is a job best left to the database. Adaptor::DBI simply translates

U.S. Patent No. 7,426,720	***Srinivasan* in view of *APA-Bach***
	queries to equivalent SQL queries, while Adaptor:: File implements a simple-minded scheme for file based objects: it converts the query expression to an evalable Perl expression and cycles through all objects, matching them against the query specification. Schema Evolution Let us say you have sent your objects' data to a file, and tomorrow, some more attributes are added to the object implementation. The schema is said to have evolved. The framework has to be able to reconcile old data with newer object implementations. Implementation This section explains the implementation of Adaptor::DBI and Adaptor::File. We will cover only the key procedures that perform query processing and file or database I/O. Pay as much attention to the design gotchas and unimplemented features as you do to the code. Adaptor::File An Adaptor::File instance represents all objects stored in one file. When this adaptor is created (using new), it reads the entire file and translates the data to in-memory objects. Slurping the entire file into memory avoids the problem of having to implement fancy on-disk schemes for random access to variable-length data; after all, that is the job of DBM and database implementations. For this reason, this approach is not recommended for large numbers of objects (over 1,000, to pick a number). The file adaptor has an attribute called all_instances, a hash table of all objects given to its store method (and indexed by their _id), as shown in Figure 11-2.  *Figure 11-2. Structure of file adaptor* *Srinivasan* at 179-80.

U.S. Patent No. 7,426,720	***Srinivasan* in view of *APA-Bach***

	“Perl automatically garbage collects a data structure when its reference count drops to zero. If a data structure has been blessed into a module, Perl allows that module to perform some clean-up before it destroys the object, by calling a special procedure in that module called DESTROY and passing it the reference to the object to be destroyed:
	
	`package Employee;`
	`sub DESTROY {`
	`      my ($emp) = @_;`
	`      print "Alas, ", $emp->{"name"}, "is now no longer with us \n";`
	`}`
	
	This is similar to C++'s destructor or the finalize () method in Java in that Perl does the memory management automatically, but you get a chance to do something before the object is reclaimed. (Unlike Java's finalize, Perl's garbage collection is deterministic; DESTROY is called as soon as the object is not being referred to any more.)” *Srinivasan* at 112.