Structured Programming Using Processes

Jay Nelson
DuoMark International, Inc.
6523 Colgate Ave, Suite 325
Los Angeles, CA 90048-4410
+1 323 381 0001
jay@duomark.com

Abstract

Structured Programming techniques are applied to a personal accounting software application implemented in erlang as a demonstration of the utility of processes as design constructs. Techniques for enforcing strong encapsulation, partitioning for fault isolation and data flow instrumentation, reusing code, abstracting and adapting interfaces, simulating inputs, managing and distributing resources and creating complex application behavior are described. The concept of *inductive decomposition* is introduced as a method for modularizing code based on the dynamic behavior of the system over time, rather than the static structure of a program. This approach leads to code subsumption, behavior abstraction, automated testing, dynamic data versioning and dynamic code revision all of which contribute to more reliable, fault-tolerant software.

Categories and Subject Descriptors

D.2.2. [**Design Tools and Techniques**]: Modules and interfaces.

General Terms

Design, Reliability, Languages, Theory.

Keywords

Inductive Decomposition, Concurrency Oriented Programming Language, COPL, erlang.

1. Introduction

Structured Programming was formulated in the late 1960s to guide programmers in producing correct, maintainable code. It was based on three fundamental principles: 1) incremental progress, 2) data structure localization and 3) modular decomposition. Together these three principles have constrained and guided the design of programming languages and the systems they implement over the last four decades. The founding principles are less apparent in the modern approach of pattern-based programming techniques, nevertheless the modern programmer should keep them in mind when making software design decisions.

The principle of incremental progress is a result of Dijkstra's letter [2] eschewing the use of the "goto" statement. Subsequently developed languages relied on programming constructs such as if statements, while loops, iterators, case statements and sub-procedures because they led to behavior that could more reliably be proved to progress towards completion. Programs including the arbitrary branching allowed by goto could not be proven to proceed through the code to completion. As a result of his commentary, later languages left out the goto statement.

Dijkstra furthered the structure of programs when he espoused a "unit of program text" which expressed both an abstract data structure and the code which operates on the data [4]. Modern Object-Oriented Programming Languages (OOPLs) are based on this principle. The goal of the approach was to localize the impact of changes to data structures or the code associated with them. He envisioned idealized programs as built from modules, the lowest ones which isolated the hardware and the highest ones which encompassed the program concept, with interfaces between them allowing interchangeability but restricting the combinatorial explosion of code complexity. The OOPL Eiffel extended this concept into formal contracts between objects that explicitly defined the features contained in program units [5].

Parnas described methods for decomposing software systems into modules [6] based on the functionality to be implemented. His approach showed the biases introduced by traditional flowchart methods, and described alternate methods for subdividing software to reduce the impact of requirements changes or coding errors. This approach also focused on isolating data structures.

The basic tenets evolved to encapsulation, abstraction and isolation. Encapsulation provides a protective layer around data structures to reduce the impact of requirements changes. All code which needs direct access to the data is co-located so that a single program unit can be modified to update both the data and the code affected without impacting other program units. Abstraction advertises the capabilities of an encapsulation in a consistent manner that does not advertise the internal implementation, reducing the complexity of code that interacts with the encapsulated data model. Isolation consists of separating modules of code so that the number of interactions is reduced. Interfaces formally define the separation and become the key point of the design: data flows across interfaces and into encapsulated abstraction modules.

OOPLs provide a modern, conventional implementation of these basic elements. Originally a modular decomposition was task oriented, but Parnas' analysis pointed out that data requirements changes could affect all tasks. The approach now favored by

OOPLs hews to his recommendations on “information hiding” -- defining system modules so as to isolate data structures. While these methods have proven useful on single processor applications, the advent of asynchronous, stateless communications over the Internet and multi-CPU computers has led to a need for an alternative approach to decomposing systems. Here we present an unconventional decomposition based on the use of processes as the fundamental organizing elements of a system.

2. Concurrency Oriented Programming

A Concurrency Oriented Programming Language (COPL) is a programming language which makes processes as simple to create, monitor and manage as data structures or objects are in other languages. Because the goal of this discussion is to identify design patterns which leverage processes rather than fault tolerance *per se*, the qualifications for a COPL given here differ from Armstrong [1]. To qualify as a COPL for design purposes, a language must have the ability to:

1. create and destroy truly concurrent processes, either locally or on remote CPUs
2. ensure processes are completely isolated from one another, including fault independence
3. create or destroy thousands of processes per second; manage tens of thousands simultaneously
4. monitor failures and notify other processes when they occur
5. specify transmission and receipt of messages in a clear and concise manner

Processes are the strongest mechanism for encapsulation, abstraction and isolation in a COPL since the Operating System (OS) or Virtual Machine (VM) prevents code from crossing process boundaries. Message passing is the standard mechanism for communicating across interfaces without violating the principle of isolation. The message protocol implements the interface abstraction. Failure monitoring is necessary because processes may reside on separate CPUs and therefore it provides the only method of knowing system status if a remote process can no longer send messages.

Whereas OOPLs lead to system designs emphasizing static data structures and their management, COPLs lead to systems organized around data flows and the dynamic life of processes. The programming patterns employed by the two language families are correspondingly different. Structured analysis in the OOPL case leads to the definition of data structures and the actions that impinge on them, whereas structured analysis in the COPL case points out data transformations and the temporality of data format. To demonstrate these differences, a personal accounting system is described below. *Inductive decomposition* is introduced as a COPL technique which allows code reuse via a series of processes with shared implementations. Where examples are needed to illustrate the design principles, they are provided in the programming language *erlang*.

Erlang was developed by Ericsson to deal with massively concurrent, highly reliable, fault tolerant systems used in telephony products [1]. It provides constructs for creating, monitoring and managing concurrent processes clearly and concisely. It also incorporates a simple but powerful message-passing technique, modern modular units of compilation and write-once variables to avoid data access locking issues. Built from concepts of Prolog and Dijkstra’s original “THE” programming language [3], it utilizes exclusive pattern-matching alternatives embodied in pure functions and has the option of dynamically loading a new version of a code module when a function is called. An introduction to the aspects of the language relevant to the examples below is provided in Appendix A.

A personal accounting application is used as a practical design exercise. It tracks expense and income accounts to produce a transaction record for analyzing budgets, monitoring money flow and graphing comparative results. It is assumed that the application will be delivered standalone on a single CPU, running on multiple CPUs or delivered as a web-based service. Because of the web-based requirement, it is expected that data may coexist in multiple formats as new versions are rolled out and that it should be possible to modify the software without taking the server offline. Likewise, reports may be generated or graphed that display as text, Hyper-Text Markup Language (HTML), scalable vector graphics (SVG) or other formats, potentially in separate windows or on separate CPUs, all viewable simultaneously including requests from multiple users.

3. Structured Design

3.1. Strong Encapsulation and Code Reuse

The basic approach is to split the system into a set of communicating processes, each of which can be swapped out for a more complete implementation in the future with minimal impact to the rest of the system. For this exercise, the data is stored on disk in a flat text file with one record per line to simplify the explanation. Existing records are updated on disk by marking the original record as replaced and adding a new record at the end of the file. Deleted records are marked and left in place as well. Upgrading to a modern database library is left as an exercise for the reader.

The strategy is to read a disk file into memory, organized for ease of access when adding, editing or updating the existing records, and to then export the information in multiple formats to support the storage, display or graphing of the data. It is assumed that up to 100,000 records will cover the needs of most users, and that this number can be maintained in memory.

When designing an application such as this with OOPLs, a quick sketch of the basic objects is undertaken: accounts, transactions, reports, graphs, user forms and other basic data containers. The idea is to isolate the data elements to a single location with methods to access and manipulate them. With COPLs the initial focus is on the transient nature of the data: flowing from the disk file to memory, from memory back to disk, from memory to reports and graphs. Data is maintained in the format that is most convenient for the task and data flow transformations needed, rather than in a normalized, canonical, non-redundant form. Commonality comes in the form of processes and data manipulation or transformation procedures.

The core of the processing involves maintaining the internal memory database (DB) of transactions. Listing B-1 (found in Appendix B) is the erlang implementation of the *memdb* module which manages, filters, sorts or exports the DB state and notifies the file writing process of changes to the DB state. The memdb module is termed a *behaviour* in COPL parlance. The external interface is defined by the messages that may be accepted and acted upon. The module also relies on two callback modules to create a specific implementation of database records. One module defines the DB organization and the other module defines the format of DB records plus the functions that access their contents. Listing B-2 is the erlang implementation of the *dbtree* module. It defines the in memory DB organization as a general-balanced tree of records that are identified by a unique sequence number. Listing B-4 is the *xact* module which defines the record format for DB records, each representing an account transaction.

Each of these modules is a compilation unit that exposes only its exported functions; together they represent a dynamic process unit that exhibits the behaviour of the message protocol interface. The joining of these three modules is created dynamically as a call to the *start_link* function, passing the module name of the DB organization as well as the module name of the DB record format. This implementation may be replaced at runtime by naming a different module when *start_link* is called. Likewise, the framework can be modified without concern to the implementation or organization of records. Mechanism is completely separate from data structure and data organization.

The process defined above is an example of **strong encapsulation**. The boundaries of the process space cannot be violated, even accidentally, by any other code because the OS will prevent it. To interact with this component of the design, an external process must send a message to either add, update, delete, get a single record, get the number of records, filter the records, export them or stop the server process. All other messages will be ignored.

The filter message is designed to create a clone of the core memory DB, which will contain a subset of the records. This is the technique for obtaining only the entertainment expense records, or, given only the entertainment expenses, all entertainment transactions for a week long vacation. The main DB will have an application controller as its parent process. A subset database will have a main DB process as its parent. A *manage* process can only receive modification messages from its parent process, thus ensuring that messages received which affect the state of the DB can be propagated to the receiver's children and so on in succession, so that all dependent record subsets may be synchronized. For example, one person could be editing the DB while a second person requests a report. The report will be a live clone of a subset of the main DB and will reflect any modifications to the DB subsequent to the initiation of the report.

The design technique illustrated here is termed *inductive decomposition*. It is used when a feature set of a system design can be reduced to a series of identical processes operating on different subsets of the data. Since the report processes **reuse code** already needed by the core DB manager, the design can reduce the amount of code needed and the number of modules implemented, thus increasing the probability of producing a reliable system. Other benefits may result from the code subsumption illustrated: behavior abstraction, automated testing, dynamic data versioning and dynamic code revision, while at the same time allowing parallelism across multiple CPUs. Because all the processes share the same interface, their behavior can be abstracted and adapted by an external management process. The user can control a single process that can direct its messages to any of the dynamically created reports using the same code and mechanism for altering their behavior. If the management process were to read a set of test cases, apply them and verify the results, the abstracted interface could be used for test automation. The strong encapsulation of the process also allows for two core DB managers to be co-resident, each with a different data version module, for example two versions of the *xact* record formats. Each report or record subset created will retain the characteristics and data version of its parent process, while at the same time the behavior abstraction will be consistent so any automated testing tools or user control process can simultaneously interact with either version of data. In the same manner, a particular version of software running in a single process can load a replacement version and continue without disrupting the system, provided the message interface does not change. In Ericsson's full implementation for a non-stop environment [1], this latter restriction is lifted by introducing system messages which notify processes prior to a code change.

3.2. Abstracted and Adapted Interfaces; Transformers

Figure 1 diagrams the full design with the inductively derived core DB and subset DB processes shown in the middle. Consider next the export function provided by the xact module. Instead of generating a report, it formats a record as a binary. The DB servers export records as a list of binaries, sending the list to the calling process. Because strong encapsulation requires the transmission of data from one process to another, each is free to store it in the format most advantageous for the process' own purposes. The OOPL approach is to add interface methods to the xact object to export in the proper format, or adaptors to transform the internal data structure to the desired output format. The COPL approach introduces a transformer process which can be used as an **abstracted interface** to avoid unnecessary code growth in the xact module. It also adds runtime flexibility in the conversion of export data to target formats.

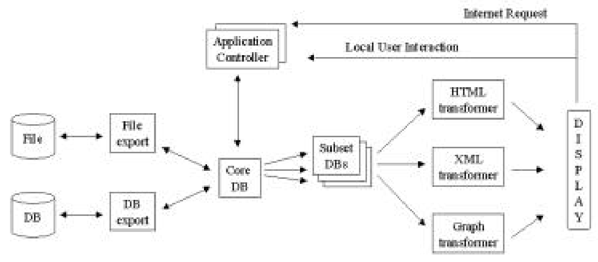

Figure 1. Personal Accounting Application design diagram

One export transformer per DB process can maintain a current reflection of each of the core DB or subset DB processes, or multiple can exist per subset DB if snapshot views prior to recent DB modifications need to be maintained. In any case, the exported data can be **adapted** to output an HTML report, raw ASCII text or even a graph through the introduction of **transformer** processes which generate the desired output data and pass them to the requesting process. A single export process can catch the exported data records and then deliver them to any number of requesting processes by replacing or extracting bytes much as UNIX character stream tools such as awk or sed work [7].

The problem description calls for both a file for storage of transaction records and reports or graphs for the display of information. The export format will most closely resemble the disk file format with a list of ASCII binaries each representing a single record. A tab separates fields in the record for ease of parsing. This format can easily be streamed to or from disk, but it can also be used to generate an HTML report by replacing the tab with closing and opening formatting commands, and the line separators with table row format commands. The transformer processes actually resemble the core DB process in that they accept records and reformat them on export to another process. Inductive decomposition allows the transformers to reuse code by implementing the same behaviour as the core and subset DBs. Listing B-6 shows the implementation of the *xactFile* record format for importing and exporting transaction records from a file. By joining the memdb module, the dbtree module and the xactFile module, a new process which maintains a balanced tree of exported records can be created. The notify functions of the xactFile record write the exported format to a file. Not shown, record transformers can share the same internal representation of a xact record (that defined by the export function of the xact module) while creating different export functions for HTML, XML, SVG or other external formats.

3.3. Managing and Distributing Resources; Supervisors

The COPL design as described so far encompasses the core data model. A single instance of the core DB process exists, as do any number of subset DB process and transformer pairings. A separate export process is coupled with the core DB to reflect any permanent changes back to disk storage in one or more locations. Initially, an assumption was made that the entire DB could reside in memory. Now there are multiple processes and multiple copies of portions of the DB, maintained in two different formats. If system resources are limited, this redundancy may preclude running the system without more advanced data handling schemes.

In an OOPL the design would have to consider introducing a caching container, replacement mechanism and cache coherency maintenance. In a COPL design, several options are available. The first is simply **distributing resources** to other CPUs by distributing the processes, thus gaining access to more resources. The second is **managing resources** by recognizing processes as containers which organize data. Introducing a new process is equivalent to introducing a new data structure in single process languages. A caching process can expire individual records or the entire process using timeouts; coherency is achieved using a propagate function as the core DB function did. If the user has too many reports displayed, requesting a new one could cause resources to be freed by eliminating a previously displayed report (hopefully with the user's confirmation) by merely killing the corresponding display process. The export process can also be terminated if it is not needed for other output. Yet another approach is to introduce **supervisor** processes which are designed to independently police worker processes. Inactivity, repeated failure, excessive resource consumption, and other actions can be observed externally and acted on as a way of maintaining a resource efficient, reliable and smoothly running system.

3.4. Data Flow Instrumentation

An export process is used to read from and write to each of the supported external data formats. For the personal accounting application, the implementation supports both a file-based format so that an individual's data may be kept encrypted and in isolation on a webserver, or as a standard SQL database if speed, backup and external tool analysis are desired. One transformer process would be used for each format.

Given that an ISP might want to charge based on access usage or bandwidth, a mechanism for measuring and monitoring activity is needed. Introducing a process between the export process and a file format transformer or Internet delivery allows for **data flow instrumentation** without impacting the design, although it would slow down the data transfer by introducing an extra hop. The instrumenting process can measure volumes, statistics or frequency of usage and report them to other system components concerned with traffic patterns, with or without altering the data it relays.

3.5. Fault Isolation

In a typical single process application, a failure by one component, such as a report display, could cause the entire application to terminate. With a COPL approach, each of the processes independently contribute to the application behavior. The processes provide **fault isolation** because failure of a report will only cause its display process to fail. Processes that are linked to or are monitoring the failed process will notice the failure and can choose to react in an appropriate way individually. COPLs lead to a "fail fast" programming style [1]. Instead of trying to code defensively against errors, the programmer should allow the process to fail as soon as possible after a coding error occurs. Failure prevents propagation and amplification of the error, as well as provides notification to the rest of the system of the error condition. Supervisor processes can be used to restart a failing process in a controlled way. Whenever a failure is more likely, or is expected to cause problems, the best approach is to isolate the case in a separate process. This can also be used as a debugging technique so that it is possible to immediately know when failure occurs and in which piece of code.

3.6. Creating Complex Application Behavior

The design described for the personal accounting application consists of a federation of cooperating processes. It is not obvious from the diagram in Figure 1 whether it is a single user application, a multi-user application or a web-based application. In reality, all of these views can exist simultaneously. For example, the system could be configured so that a user starts a local core DB process which reads the stored DB over the Internet. A second user starts the same application, but the existing core DB process is linked to the display so that both users may edit the same DB of records. Finally, a third user visits a web page to invoke a CGI script on the server to view a report. The report is generated from the export process on the central server that is receiving data from the core DB process on the remote user's workstation. The collection of processes interact in a predetermined way, but they **create complex application behavior** in the way they interact. The dynamic nature of a COPL solution is part of its appeal because it can be applied to a set of problems simultaneously, producing multiple views of the system.

3.7. Automated Testing; Simulating Input

With complex behavior comes difficulty in verification. The many means of interaction makes testing difficult and error prone if repeatable methods aren't available. Because a COPL system is divided into processes with isolated interfaces, an **automated testing** process can systematically try all the interfaces one process at a time by **simulating input**. Standalone testing will isolate and highlight buggy code in each process. Subsequent testing after the processes are corrected can focus on the interprocess communications. If the system is divided into compilation units as when implemented using an OOPL approach, test harnesses and driver programs must be written. With the process separation approach of COPLs, testing tools only need to send messages and record results. Recompilation of running code is not necessary; processes are tested as they would run in the real system without alteration. This same approach can be used in systems where the source data is hard to obtain, whether for financial, physical or environmental reasons.

4. Conclusion

The original principles of structured programming apply as much now as ever before. OOPLs pushed the techniques of information hiding and data encapsulation to the logical limits of single process architectures, but the advent of multiple CPU machines and connected networks of processors necessitates new approaches to architecting systems. COPLs bring an alternative view of design by making highly concurrent systems feasible. The techniques of strong encapsulation, code reuse, abstracting and adapting interfaces, managing and distributing resources, data flow instrumentation, fault isolation and automated testing are all possible through the use of processes as fundamental design elements. Understanding why these are useful techniques and under what situations they apply is the first step in mastering COPLs and going beyond the "one process per real world concurrent activity" stricture [1] often echoed to beginning programmers.

5. Acknowledgements

Thanks to Hal Snyder and Francesco Cesarini for their comments and suggestions. Special thanks to the erlang mailing list participants for discussions which led to the ideas presented here.

6. References

[1] Armstrong, J. *Making reliable distributed systems in the presence of software errors.* Ph.D. Thesis, The Royal Institute of Technology, Stockholm, Sweden , 2003.

[2] Dijkstra, E. W. Go To Statement Considered Harmful. *Comm. ACM*, 11, 3 (Mar. 1968), 147-148.

[3] Dijkstra, E. W. The Structure of the "THE"-Multiprogramming System. *Comm. ACM*, 11, 5 (May 1968), 341-346.

[4] Dijkstra, E. W. Structured Programming, *Software Engineering Techniques*, NATO Science Committee, Rome, Italy, (Aug 1970), 65-68.

[5] Meyer, B. *Object-Oriented Software Construction, 2nd Edition*. Prentice Hall, Upper Saddle River, NJ, 2000.

[6] Parnas, D.L. On the Criteria To Be Used in Decomposing Systems into Modules. *Comm. ACM*, 15, 12 (Dec. 1972), 1053-1058.

[7] Raymond, E.S. *The Art of Unix Programming*. Addison-Wesley, Boston, MA, 2004.

APPENDIX A. Introduction to erlang

This Appendix is a quick introduction to the programming language *erlang*. The language is a functional language, where functions are organized into compilation *modules*, with only those functions named in an *export* clause callable from other modules or from the interpreter. Functions are declared with a name, a pattern expressed in parentheses, an optional 'when' clause to restrict the patterns which match arguments passed in, followed by '->' to indicate the start of the body of the function. Semi-colons separate alternative function heads with different patterns; a period terminates a set of function alternatives.

```
-module(bank).
-export([open_acct/0, open_acct/1]).

open_acct() -> open_acct(0).
open_acct(StartBal) when StartBal >= 0 ->
  spawn_link(fun()-> acct(StartBal) end).

acct(Balance) ->
  receive
    {From, deposit, X} when X > 0 ->
       NewBal = Balance + X,
       From ! {self(), deposited, NewBal},
       acct(NewBal);
    {From, withdraw, X} ->
       NewBal = checkWithdraw(Balance, X, From),
       acct(NewBal);
    {From, stop} ->
       stopped;
    Other ->
       acct(Balance)
  end.

checkWithdraw(Bal, X, From) when Bal > 0, X > 0, X =< Bal ->
  NewBal = Bal – X,
  From ! {self(), cash, X, NewBal},
  NewBal;
checkWithdraw(Bal, _X, From) ->
  From ! {self(), denied, Bal},
  Bal.
```

Listing A-1. A simple bank account server.

Within a function, commas separate statements that are executed sequentially. The return value of an expression or a function is the return value of the final statement. All variables are single assignment (*i.e.*, once the value of a variable is set it may not be modified) and type determination is dynamic during execution. Built in datatypes are integer (arbitrary length precision), float, character, atom (an explicit term name), list (singly linked variable length collection of elements indicated by *[Elem1, Elem2 | Rest]*), tuple (fixed length collection of elements indicated by *{Elem1, Elem2}*), binary (bitwise or bytewise chunk of memory indicated by << >>) and functor (a first-class function closure). Strings are represented as lists of characters. Listing A-1 shows a module that implements a simple bank account server.

In the example above, the module *bank* exports a single function *open_acct* which takes zero arguments or one argument indicating the opening balance of the account. Variables are identified by the first character, which is either capitalized if the variable may be referred to elsewhere, or an underscore ('_') if the variable is to be matched but is not referenced later. The *acct* function can be referenced by other code within the same module without using the module name as a prefix, but it is not visible outside this module because it is not an element of the list of exported functions.

```
1> Pid = bank:open_acct().
<0.33.0>
2> Pid ! {self(), withdraw, 15.00}, receive Any -> Any end.
{<0.33.0>, denied, 0}
3> Pid ! {self(), deposit, 20.73}, receive Reply -> Reply end.
{<0.33.0>, deposited, 20.73}
4> Pid ! {self(), withdraw, 15.00}, receive Cash -> Cash end.
{<0.33.0>, cash, 15.0000, 5.73000}
```

Listing A-2. Interpreter session with bank acct server.

The function *spawn_link* is a built-in function (BIF) which spawns a new process linked to the calling process. The example here passes a functor as the function to call when the new process starts. The spawn_link BIF returns the process id handle that is used for all future communication with the process. The original caller would keep this id and use the operator '!' to send a message which matches one of the receive clauses in the acct function. For example, typing the sequence shown in Listing A-2 in the interpreter shell would start a new process and deposit $20.73 in it, after denying a $15.00 withdrawal.

Functions are executed by matching the arity (number of arguments) and the value restrictions that may exist for each alternative function head clause. At prompt 1>, the new process is started by calling the function bank:open_acct/0 which has an arity of 0 (*i.e.*, it takes no arguments). This function in turn calls the open_acct function which takes one argument, passing the default opening account balance of $0. The when clause on the function head is a guard that ensures this function clause will only match if the passed in argument is at least 0. If a value less than 0 were passed in, it would not match any function clause and the process would crash with a reason of bad_match.

In this case, a newly spawned process is started running the acct function. It returns the process id of <0.33.0>. Since the function was called with the result being matched to the pattern represented by the variable Pid (which was unbound at the time of the call), Pid is set forevermore to contain the value <0.33.0>. At prompt 2>, a message is sent to the new process which consists of a three-tuple containing the process id of the interpreter, the atom *withdraw* which requests a cash withdrawal, and 15.00 as the amount of the withdrawal.

The server has been waiting at the blocking BIF 'receive' for a message to arrive at the acct process. It receives the message, pattern matches it sequentially against the receive clauses and decides to call the function checkWithdraw. This function has two alternative matching clauses. The first applies to a valid request for a positive amount of money from an account with a balance greater than 0, when the amount requested is no greater than the balance. The second clause applies to all other cases by denying the request and returning the existing balance. In this case, the second clause is executed, denying the withdrawal and returning the existing 0 balance. The acct function receives the result and calls itself with the potentially new (but in this case unchanged) balance. Since tail-recursion such as this is a common approach, the compiler optimizes these calls to make them as efficient as iteration in an imperative language.

At prompt 3> a deposit is successfully made. The reply from the server is captured in a new variable because *Any* is already bound and can only be used to match exactly to the pattern received in at prompt 2>. The acct balance is incremented by 20.73 and the acct function is called with the new balance. This is how a Balance can change without assigning to a previously assigned variable. The recursive call generates a new variable in the new function context. At prompt 4> the new balance allows a with-drawal of $15.00 with a subsequent reduction of the Balance to $5.73.

When calling functions, the patterns described by the function head alternatives are not formal arguments as in other languages, but are actual pattern-matching constructs. For example, in the receive clauses of the bank account server, the tuple patterns defined the exact number of elements in each acceptable message format with some elements required to be atoms, and others matching unbound variables. When a pattern matches an unbound variable, that variable becomes bound with the matching value. Any subsequent use of the variable in an expression amounts to a pattern match request using the value of the variable as the pattern being matched. When the From variable is bound to the sender of the message, the subsequent message send command ('!') is routed to the process id that is now bound to the variable From.

The personal accounting system examples also use the 'record' construct. A record is a data structure with named fields that may be accessed by matching only some of the elements of the record. The '-record' directive creates a new type of record. The first argument to the directive indicates the record type name. The second is a tuple of the field names that refer to each of the elements in the record. A record match pattern can take the form of '*#recordtype{field=value}*' where the record type indicates that a matching value can only be a record of the named type, with the element *field* already set equal to *value*. If this record were bound to the variable 'Rec', the expression '*Rec#recordtype.field*' would match to the current value of the *field* element of the record.

The last feature introduced here is called a *list comprehension*. It allows the conversion of a list to a new list using the shorthand of *[convert(Elem) || Elem <- List, Elem > 0]*, which should be read as "create a new list by calling convert on Elem where Elem comes from List and Elem is greater than 0". Each element is taken in order from the list, filtered by the guard clause (Elem > 0) and assembled into a new list. The guard clause(s) are optional.

See Armstrong [1] for a more thorough introduction to the language and an in-depth discussion of concurrency.

APPENDIX B. Code Listings

Partial erlang code listings for the personal accounting application are provided to help explain the techniques used in designing the system. Space restrictions prevent the inclusion of a full working application. Listing B-1 is the core DB server process that maintains transaction records in an internal format. DBOrgMod refers to the caller's selected DB organization code module; RecFmtMod refers to the caller's selected DB record format module. Together these three elements define the mechanism, organization and format of the core database of records.

```
-module(memdb).
-vsn("1.0.0").
-export([start_link/4, start_link/5]).

start_link(Parent, DBOrgMod, InitArgs, RecFmtMod) ->
  InitState = DBOrgMod:init(InitArgs),
  spawn_link(fun() -> manage(DBOrgMod, InitState, RecFmtMod,
                             none, none, Parent, [])
             end).

start_link(Parent, DBOrgMod, InitArgs, RecFmtMod, Filename) ->
  InitState = DBOrgMod:init(InitArgs),
  spawn_link(fun() -> manage(DBOrgMod, InitState, RecFmtMod,
                             none, Filename, Parent, [])
             end).

manage(DBOrgMod, State, RecFmtMod, FileNotify, Filename, Parent,
       Children) ->
  receive
    % Only one file can be loaded per CoreDB process.
    {Parent, load, FileModule, Permissions, SortFn,ValidStatus} ->
      {NewState, FilePid} =
        load(DBOrgMod, State, FileModule, Filename, RecFmtMod,
             Permissions, SortFn, ValidStatus, Parent),
      case FilePid of
        none -> manage(DBOrgMod, NewState, RecFmtMod, FileNotify,
                       Filename, Parent, Children);
        FilePid -> manage(DBOrgMod, NewState, RecFmtMod, FilePid,
                          Filename, Parent, Children)
      end;

    {Parent, dbsize, OnlyActive} ->
      _Size = dbsize(DBOrgMod, State, Parent, OnlyActive),
      manage(DBOrgMod, State, RecFmtMod, FileNotify, Filename,
             Parent, Children);

    {Parent, addlist, RecList} ->
      NewState = addlist(DBOrgMod, State, RecFmtMod, RecList,
                         Parent, Children, FileNotify),
      manage(DBOrgMod, NewState, RecFmtMod, FileNotify, Filename,
             Parent, Children);

    {Parent, add, Rec} ->
      NewState = add(DBOrgMod, State, RecFmtMod, Rec,
                     Parent, Children, FileNotify),
      manage(DBOrgMod, NewState, RecFmtMod, FileNotify, Filename,
             Parent, Children);

    {Parent, update, Id, Rec} ->
      NewState = update(DBOrgMod, State, RecFmtMod, Id, Rec,
                        Parent, Children, FileNotify),
      manage(DBOrgMod, NewState, RecFmtMod, FileNotify, Filename,
             Parent, Children);

    {Parent, delete, Id} ->
      NewState = delete(DBOrgMod, State, Id, Parent, Children,
                        FileNotify),
      manage(DBOrgMod, NewState, RecFmtMod, FileNotify, Filename,
             Parent, Children);
    {Parent, getrec, Id} ->
      NewState = getrec(DBOrgMod, State, Id, Parent),
      manage(DBOrgMod, NewState, RecFmtMod, FileNotify, Filename,
             Parent, Children);

    {Parent, filter, FilterFn, FiltParent, ValidStatus} ->
      Pid = filter(DBOrgMod, State, RecFmtMod, FilterFn, Parent,
                   FiltParent, ValidStatus),
      manage(DBOrgMod, State, RecFmtMod, FileNotify, Filename,
             Parent, [{Pid, FilterFn} | Children]);

    {Parent, export, SortFn, ValidStatus} ->
      _BinText = sort(DBOrgMod, State, RecFmtMod, SortFn, Parent,
                      ValidStatus),
      manage(DBOrgMod, State, RecFmtMod, FileNotify, Filename,
             Parent, Children);

    {Parent, notify_newlist, LastId, Records} ->
      NewState = notify_newlist(Filename,DBOrgMod,State,RecFmtMod,
                                LastId, Records),
      manage(DBOrgMod, NewState, RecFmtMod, FileNotify, Filename,
             Parent,Children);

    {Parent, notify_new, NewId, Record} ->
      NewState = notify_new(Filename, DBOrgMod, State, RecFmtMod,
                            NewId, Record),
      manage(DBOrgMod, NewState, RecFmtMod, FileNotify, Filename,
             Parent, Children);

    {Parent, notify_upd, ReplacedId, NewId, Record} ->
      NewState = notify_upd(Filename, DBOrgMod, State, RecFmtMod,
                            ReplacedId, NewId, Record),
      manage(DBOrgMod, NewState, RecFmtMod, FileNotify, Filename,
             Parent, Children);

    {Parent, notify_del, DeletedId} ->
      NewState = notify_del(Filename, DBOrgMod, State, RecFmtMod,
                            DeletedId),
      manage(DBOrgMod, NewState, RecFmtMod, FileNotify, Filename,
             Parent, Children);

    {Parent, stop} ->
      [Child ! {self(), stop} || Child <- Children],
      case FileNotify of
        none -> ok;
        _Else -> FileNotify ! {self(), stop}
      end,
      io:format("Manage process ~w is quitting.~n", [self()]),
      Parent ! {self(), stopped};

    Unknown ->
      io:format("Manage process ~w is ignoring request ~s.~n",
                [self(), Unknown]),
      manage(DBOrgMod, State, RecFmtMod, FileNotify, Filename,
             Parent, Children)
    %% Web server shutdown if user not coming back.
    after
      30000 -> timeout
    end.
```

Listing B-1. Server core DB process module.

The notify clauses are needed for the disk storage of data because file location information and ASCII formatted export records are used in the disk transformer process rather than internal xact records. Listing B-2 provides the standard implementation of a balanced record tree used for the in memory DB organization.

```
-module(dbtree).
-export([init/1, addlist/2, add/2, update/3, delete/2, getrec/2,
         dbsize/2, filter/4, exp/4]).
-record(rec, {id, status, data}).
-define(ACTIVE, $A).
-define(REPLACED, $R).
-define(DELETED, $X).

init([]) -> gb_trees:empty().

%% Kick off adding a list of records.
addlist(RecordTree, RecordList) when is_list(RecordList) ->
  addlist(RecordTree, RecordList, 0).
%% Add a list of records (internal function).
addlist(RecordTree, [], LastRec) -> LastRec;
addlist(RecordTree, [Next | Rest], _LastRec) ->
  LastRec = add(RecordTree, Next),
  {new, NewTree, Id} = LastRec,
  addlist(NewTree, Rest, LastRec).

%% Add a single new record, assigning the next available id.
add(RecordTree, Record) ->
  Id = case gb_trees:is_empty(RecordTree) of
    true -> 1;
    false ->
      {Count, _Rec} = gb_trees:largest(RecordTree), Count + 1
    end,
  NewRec = #rec{id=Id, status=?ACTIVE, data=Record},
  NewTree = gb_trees:insert(Id, NewRec, RecordTree),
  {new, NewTree, Id}.

%% Mark the old record as replaced, then add a new record.
update(RecordTree, Id, Record) ->
  case gb_trees:lookup(Id, RecordTree) of
    {value, #rec{status=?ACTIVE} = OldRec} ->
      MarkedAsReplacedRec = OldRec#rec{status=?REPLACED},
      NewTree =gb_trees:update(Id,MarkedAsReplacedRec,RecordTree),
      {new, FinalTree, NewId} = add(NewTree, Record),
      {upd, FinalTree, NewId};
    _Other -> {record_not_found, RecordTree}
  end.

delete(RecordTree, Id) ->
  case gb_trees:lookup(Id, RecordTree) of
    {value, #rec{status=?ACTIVE} = OldRec} ->
      MarkedAsDeletedRec = OldRec#rec{status=?DELETED},
      NewTree = gb_trees:update(Id,MarkedAsDeletedRec,RecordTree),
      {del, NewTree};
    _Other -> {record_not_found, RecordTree}
  end.

getrec(RecordTree, Id) ->
  case gb_trees:lookup(Id, RecordTree) of
    {value, #rec{status=?ACTIVE, data=DesiredRecord}} ->
      {got, RecordTree, DesiredRecord};
    _Other -> {record_not_found, RecordTree}
  end.

dbsize(RecordTree, OnlyActive) ->
  Iterator = gb_trees:iterator(RecordTree),
  dbsize(gb_trees:next(Iterator), OnlyActive, 0).

% Internal
dbsize(none, _OnlyActive, Count) -> Count;
dbsize({_Id, #rec{status=?ACTIVE}, Iterator}, true, Count) ->
  dbsize(gb_trees:next(Iterator), true, Count+1);
dbsize({_Id, #rec{}, Iterator}, true, Count) ->
  dbsize(gb_trees:next(Iterator), true, Count);
dbsize({_Id, #rec{}, Iterator}, false, Count) ->
  dbsize(gb_trees:next(Iterator), false, Count+1).
filter(RecordTree, RecFmtMod, FilterFn, ValidStatus) ->
  Records = to_list(ValidStatus, RecordTree),
  FilteredRecs = RecFmtMod:FilterFn(Records).
exp(RecordTree, RecFmtMod, SortFn, ValidStatus) ->
  Records = to_list(ValidStatus, RecordTree),
  SortedRecs = lists:sort(fun(E1,E2) -> RecFmtMod:SortFn(E1,E2)
                          end, Records),
  RecFmtMod:explist(SortedRecs).
to_list(active, RecordTree) ->
  Iterator = gb_trees:iterator(RecordTree),
  activeRecords(gb_trees:next(Iterator), []);
to_list(replaced, RecordTree) ->
  Iterator = gb_trees:iterator(RecordTree),
  replacedRecords(gb_trees:next(Iterator), []);
to_list(deleted, RecordTree) ->
  Iterator = gb_trees:iterator(RecordTree),
  deletedRecords(gb_trees:next(Iterator), []);
to_list(all, RecordTree) ->
  [Record#rec.data || Record <- gb_trees:values(RecordTree)].

activeRecords(none, RecordList) -> RecordList;
activeRecords({Id, #rec{status=?ACTIVE, data=Record}, Iterator},
              RecordList) ->
  activeRecords(gb_trees:next(Iterator), [Record | RecordList]);
activeRecords({Id, #rec{}, Iterator}, RecordList) ->
  activeRecords(gb_trees:next(Iterator), RecordList).
replacedRecords(none, RecordList) -> RecordList;
replacedRecords({Id,#rec{status=?REPLACED,data=Record}, Iterator},
                RecordList) ->
  replacedRecords(gb_trees:next(Iterator), [Record | RecordList]);
replacedRecords({Id, #rec{}, Iterator}, RecordList) ->
  replacedRecords(gb_trees:next(Iterator), RecordList).
deletedRecords(none, RecordList) -> RecordList;
deletedRecords({Id,#rec{status=?DELETED, data=Record}, Iterator},
               RecordList) ->
  deletedRecords(gb_trees:next(Iterator), [Record | RecordList]);
deletedRecords({Id, #rec{}, Iterator}, RecordList) ->
  deletedRecords(gb_trees:next(Iterator), RecordList).
```

Listing B-2. DB organization callback module.

```
-module(xactApp).
-vsn("1.0.0").
-export([start/2]).
start(Filename, ClientType) ->
  % Create a coreDB process and load records to it...
  CoreDB = memdb:start_link(self(), dbtree, [], xact, Filename),
  CoreDB ! {self(), load, xactFile, read_write, locSort, all},
  % Add records using the shell...
  xactShell:start(CoreDB),
  % Stop the CoreDB process.
  CoreDB ! {self(), stop},
  receive {CoreDB, stopped} -> stopped after 3000 -> timeout end.
```

Listing B-3. Application controller process.

Listing B-3 above shows a simple application controller client process that loads a transaction database, allows the user to edit the records using a command-line shell and then shuts down the processes. Listing B-4 below defines the structure of xact records in the internal memory database.

```
-module(xact).
-vsn("1.0.0").
-export([make_record/1, split_record/1, explist/1, amtSort/2,
         acctSort/2]).
-record(xact, {status, date, type, desc, amt, acct, subacct,
               class, memo}).

make_record(FieldVals) ->
  [Status, Date, Type, Desc, Amt, Acct, SubAcct, Class, Memo]
    = convert_types(FieldVals),
  #xact{status=Status, date=Date, type=Type, desc=Desc,
        amt=Amt,acct=Acct,subacct=SubAcct,class=Class,memo=Memo}.

split_record(#xact{status=Status, date=Date, type=Type, desc=Desc,
                   amt=Amt,acct=Acct,subacct=SubAcct,class=Class,
                   memo=Memo}) ->
  [Status, integer_to_list(Date), Type, Desc, float_to_list(Amt),
   Acct, SubAcct, Class, Memo].

explist(Records) -> [exp(Elem) || Elem <- Records].

amtSort(#xact{amt=Amt1}, #xact{amt=Amt2}) -> Amt2 > Amt1.

acctSort(#xact{date=Date1, acct=Acct1, subacct=Sub1},
         #xact{date=Date2, acct=Acct2, subacct=Sub2}) ->
  if Acct1 > Acct2 -> false;
     Acct1 == Acct2 ->
       if Sub1 > Sub2 -> false;
          Sub1 == Sub2 ->
            if
               Date1 > Date2 -> false;
               true -> true
            end;
          true->true
       end;
    true -> true
  end.

convert_types([Status, Date, Type, Desc, Amt, Acct, SubAcct,
               Class, Memo]) ->
  DateNum = list_to_integer(Date),  AmtNum = list_to_float(Amt),
  [Status, DateNum, Type, Desc, AmtNum, Acct, SubAcct,Class,Memo].

exp(#xact{status=Status, date=Date, type=Type, desc=Desc, amt=Amt,
          acct=Acct, subacct=SubAcct, class=Class, memo=Memo}) ->

list_to_binary(
           io_lib:format("~s\t~w\t~s\t~s\t~.2f\t~s\t~s\t~s\t~s\t",
                             [Status, Date, Type, Desc, Amt, Acct,
                                SubAcct, Class, Memo])).
```

Listing B-4. DB xact record implementation callback module.

The functions *make_record*, *split_record* and *explist* are required callbacks used by the memdb module. They provide the code needed to convert from disk process export format to internal erlang record format, and back again. The *amtSort* and *acctSort* functions are examples of how to supply a sort routine for the core DB to use when exporting records.

Listing B-5 is a very simple shell used to add, edit or delete records from a core DB. It uses a looping mechanism to prompt the user until the user wishes to quit.

```
-module(xactShell).
-export([start/1]).

start(CoreDB) ->
  io:format("~nXact V1.0  -- Press 'q' to quit~n~n", []),
  loop(CoreDB, none, ["cash", "gmcard"]).

loop(CoreDB, LastReg, LegalRegs) ->
  io:format("~nLast register was (~s).~nPress <enter> for 
same...~n",
            [LastReg]),
  Reg = prompt('Register: '),
  EditReg = case Reg of
              "q" -> quit;
              "" -> continue(CoreDB, LastReg), LastReg;
              Other -> case lists:member(Reg, LegalRegs) of
                         true -> continue(CoreDB, Reg), Reg;
                         false -> LastReg
                       end
            end,
  case EditReg of
    quit -> ok;
    _Else -> loop(CoreDB, EditReg, LegalRegs)
  end.

continue(CoreDB, Reg) ->
  EditType = prompt('(A)dd, (U)pdate, or (D)elete: '),
  case EditType of
    "a" -> add(CoreDB, Reg);
    "d" -> del(CoreDB, Reg);
    "u" -> upd(CoreDB, Reg)
  end.


del(CoreDB, Reg) ->
  IdString = prompt('Id: '),
  Id = list_to_integer(IdString),
  CoreDB ! {self(), delete, Id},
  receive
    {CoreDB, deleted} -> io:format("Record ~w deleted~n", [Id]);
    {CoreDB,delerr,not_found}-> io:format("Record not found~n",[])
  after
    3000 -> timeout
  end.

upd(CoreDB, Reg) ->
  IdString = prompt('Id: '),
  Id = list_to_integer(IdString),
  CoreDB ! {self(), getrec, Id},


  receive
    {CoreDB, gotrec, Id, Record} ->
      io:format("Got ~w~n", [Record]),
      updData(CoreDB, Reg, Id, xact:split_record(Record));

    {CoreDB,geterr,not_found}-> io:format("Record not found~n",[])

  after 3000 -> timeout
  end.

updData(CoreDB, Reg, OldId, [Status, Date, Type, Desc, Amt, Acct,
                             SubAcct, Class, Memo] = Defaults) ->
  DateStr = prompt("Date (" ++ Date ++ ") :"),
  TypeStr = prompt(“Type (“ ++ Type ++ “) : “),
  DescStr = prompt(“Desc (“ ++ Desc ++ “) : “),
  AmtStr = prompt(“Amt (“ ++ Amt ++ “) : “),
  AcctStr = prompt(“Acct (“ ++ Acct ++ “) : “),
  SubAcctStr = prompt(“SubAcct (“ ++ SubAcct ++ “) : “),
  ClassStr = prompt(“Class (“ ++ Class ++ “) : “),
  MemoStr = prompt(“Memo (“ ++ Memo ++ “) : “),
  Record = replaceDefaults(Defaults,
                           [“A”, DateStr, TypeStr, DescStr,AmtStr,
                            AcctStr,SubAcctStr,ClassStr,MemoStr]),
  io:format(“~nRegister: ~s  Record: ~s~n”,
            [Reg, lists:flatten(Record)]),
  CoreDB ! {self(), update, OldId, Record},
  receive {CoreDB, updated, LastId} -> LastId after 3000 -> timeout
end.

replaceDefaults(Defaults, Overrides) ->
  replaceDefaults(Defaults, Overrides, []).

replaceDefaults([], [], Vals) ->
  lists:reverse(Vals);
replaceDefaults([Default | DefMore], [““ | OverMore], Vals) ->
  replaceDefaults(DefMore, OverMore, [Default | Vals]);
replaceDefaults([_Default | DefMore], [Override|OverMore],Vals) ->
  replaceDefaults(DefMore, OverMore, [Override | Vals]).

add(CoreDB, Reg) ->
  Date = prompt(‘Date: ‘),
  Type = prompt(‘Type: ‘),
  Desc = prompt(‘Desc: ‘),
  Amt = prompt(‘Amt: ‘),
  Acct = prompt(‘Acct: ‘),
  SubAcct = prompt(‘SubAcct: ‘),
  Class = prompt(‘Class: ‘),
  Memo = prompt(‘Memo: ‘),
  Record = [“A”, Date, Type, Desc, Amt, Acct, SubAcct,Class,Memo],
  io:format(“~nRegister: ~s  Record: ~s~n”,
            [Reg, lists:flatten(Record)]),
  CoreDB ! {self(), add, Record},
  receive {CoreDB, new, LastId} -> LastId after 3000 -> timeout
end.

prompt(Token) when is_list(Token) ->
  prompt(list_to_atom(Token));
prompt(Token) when is_atom(Token) ->
  Result = string:strip(io:get_line(Token), right, $\n),
  string:strip(Result, both).
```

Listing B-5. Example shell module xactShell.

Listing B-6 is the alternate structure used for managing exported xact records as well as for importing them to the core DB process when the external file is first loaded. It only needs to store the file offset location of each record since records are written to disk immediately, but in this implementation the actual exported xact record is maintained as well for debugging purposes.

```
-module(xactFile).
-export([start_link/2, load/3]).
-export([make_record/1, explist/1, recOnly/1, locOnly/1,locSort/2,
          notify_add/2, notify_upd/3, notify_del/2]).

-record(xactFile, {loc, data}).
-define(REPLACED, $R).
-define(DELETED, $X).
-define(FIELD_SEPARATOR, <<“\t”>>).

start_link(Filename, Parent) ->
  {ok, Pid} = load(Filename, xact, Parent),
  Pid.
%% Returns {ok, Pid} or {timeout, Pid}
load(Filename, Type, Parent) ->
  {Vsn, Records} = load_file(Filename, Type),
  launch_vsn(Records, Type, Vsn, Filename).

make_record([Loc, Record]) -> #xactFile{loc=Loc, data=Record}.
explist(Records) -> [exp(Elem) || Elem <- Records].
exp(#xactFile{loc=Loc, data=Record}) ->
  RawBinFields =
    [Rec || {_Pos,Rec}<-bin_utils:split(Record,?FIELD_SEPARATOR)],
  RawFields = [binary_to_list(B) || B <- RawBinFields].

recOnly(Records) when is_list(Records) ->
          [recOnly(R) || R <- Records];
recOnly(#xactFile{data=Record}) ->
  RawBinFields =
    [Rec || {_Pos,Rec}<-bin_utils:split(Record,?FIELD_SEPARATOR)],
  RawFields = [binary_to_list(B) || B <- RawBinFields].

locOnly(Records) when is_list(Records) ->
          [locOnly(R) || R <- Records];
locOnly(#xactFile{loc=Loc}) -> Loc.
locSort(Loc1, Loc2) -> Loc1 < Loc2.

notify_add(Filename, Record) ->
  {ok, Handle} = file:open(Filename, [read, write, raw, binary]),
  {ok, Pos} = file:position(Handle, eof),
  io:format(“Loc: ~w notify_add ~s : ~s~n”,
          [Pos, Filename, binary_to_list(hd(Record))]),
  ok = file:write(Handle, Record),
  ok = file:write(Handle, io_lib:nl()),
  ok = file:close(Handle),
  make_record([Pos, Record]).

notify_upd(Filename, #xactFile{loc=OldRecLoc}, Record) ->
  {ok, Handle} = file:open(Filename, [read, write, raw, binary]),
  {ok, Pos} = file:position(Handle, {bof, OldRecLoc}),
  NewRecord = if (Pos == OldRecLoc) ->
                  ok = file:write(Handle, ?REPLACED),
                  {ok, NewPos} = file:position(Handle, eof),
                  io:format(“Loc: ~w notify_upd ~s : ~s~n”,
                  [NewPos,Filename,binary_to_list(hd(Record))]),
                  ok = file:write(Handle, Record),
                  ok = file:write(Handle, io_lib:nl()),
                  make_record([Pos, Record]);
              true ->
                io:format(“Pos: ~w not equal to OldRecLoc: ~w~n”,
                            [Pos, OldRecLoc]), []
             end,
  file:close(Handle),
  NewRecord.

notify_del(Filename, #xactFile{loc=Loc}) ->
  {ok, Handle} = file:open(Filename, [read, write, raw, binary]),
  {ok, Pos} = file:position(Handle, {bof, Loc}),
  if (Pos == Loc) -> ok = file:write(Handle, ?DELETED) end,
  file:close(Handle).

sendAddList(Pid, Records) ->
  % Records = [Rec#xactFile.data || Rec <- FileRecords],
  Pid ! {self(), addlist, Records},
  receive
    {Pid, newlist, LastId} ->
       io:format("Got ~w as last id.~n", [LastId]), ok
  after
    3000 -> timeout
  end.

launch_vsn(Records, xact, v1_0, Filename) ->
  launch_vsn_1_0(Records, xact, Filename).

launch_vsn_1_0(Records, xact, Filename) ->
  Pid = memdb:start_link(self(), dbtree, [], xactFile, Filename),
  {sendAddList(Pid, Records), Pid}.


load_file(Filename, Type) ->
  io:format("File: ~s~n", [Filename]),
  {ok, Binary} = file:read_file(Filename),
  Lines = bin_utils:split_lines(Binary),
  {Vsn, Records} = load_vsn(Lines, Type).

load_vsn([{Loc1,<<"#$# Xact v1.0 #$#">>}, {Loc2, <<>>} | Rest],
         xact) ->
  Records = make_vsn_1_0(Rest, xact, []),
  {v1_0, Records};
load_vsn([{Loc1, <<"#$# Xact v", Rest/binary>>} | More], xact) ->
  unrecognized_xact_vsn;
load_vsn([{Loc1, Other} | More], xact) ->
  io:format("~s~n", [Other]),
  unknown_file_fmt;
load_vsn([], xact) ->
  empty_or_missing_file.
```

```
make_vsn_1_0([], xact, Records) ->
  lists:reverse(Records);
make_vsn_1_0([{Loc, <<>>} | Rest], xact, Records) ->
  make_vsn_1_0(Rest, xact, Records);
make_vsn_1_0([{{Start, _End}, Line} | Rest], xact, Records) ->
  NewRecord = [Start, Line],
  make_vsn_1_0(Rest, xact, [NewRecord | Records]).
```

Listing B-6. xactFile record implementation callback module.

This module is the file transformer. It is a combination of two behaviours: file loading and change notification exporting. When the application controller starts up it calls on the *xactFile* module to read the DB source file, determine the data format version and spawn an import/export memdb process to hold the records. This process uses a dbtree organization, but a xactFile record format as opposed to the xact record format used by the core DB process. A xactFile record contains the disk offset to the beginning of the xact record and the actual disk representation of the record (which happens to match the export format defined in the xact module).

Once the source transaction file is loaded, the entire set of records is exported to the core DB process so that the two processes have a matching view of the DB state, however, the core DB is operating on xact records rather than xactFile records. Whenever a record is added, modified or deleted in the core DB process (as the xactShell module allows), the information is relayed via the notify functions. This causes the transformer process to update its in memory hierarchy of xactFile records and to write the information to disk.