

On Language Choice for Software Projects

Dan Lloyd

Jason Hartline

March 19, 1999

Abstract

We explore the differences in implementing a small project in both a functional language and conventional language in hopes of understanding why functional languages are not utilized as frequently in software projects as their conventional counterparts. A chat server is implemented in Scheme and Java. After the completed implementation, new specifications are given forcing modifications to the original design. We give a contrasting perspective on the design, debug, maintain process of the project for these two languages.

1 Introduction

Functional languages provide many powerful abstractions to programmers. Yet, functional languages are largely ignored as viable solutions to large programming projects such as those found in industry [8].

1.1 Strategy

To research the question of why functional languages are not used more often, we explore one life-cycle of the implementation of a chat server in two languages, Java and Scheme. Java was chosen as a representative for a common conventional language and Scheme as a representative for functional languages. These choices will be discussed more in section 6.

Our goal was to independently develop solutions to the problem of implementing a chat server such that the implementations exploit language features and are natural in each language. We require that both implementations satisfy the same specification

(Appendix A), but we do not require them to use the same algorithm or methodology.

Wadler [8] points out that libraries are an attractive feature of any language. To prevent this research endeavor from becoming a comparison of the Java Class Hierarchy and the R4RS [1] Scheme standard library. We will not be comparing efforts to build up suitable library routines for manipulating fundamental data structures such as queues, stack, lists, and tables.

1.2 Organization

In section 2 we present several related case studies done previously by other researchers. We will discuss the design implementation a basic chat server in both languages in section 3. Debugging issues will be presented in section 4. In section 5 we discuss the modifications that were made to the system and their relative difficulty and their impact on the design. We conclude in section 6 and discuss future work in section 7.

2 Related Case Studies

Before we delve into our own experimentation it is instructive to drag related work by other authors into the picture.

One possible reason why functional languages are not used more often is that implementations in functional languages are less efficient than their C counterparts. Previous work done by P. H. Hartel *et. al.* [3] explored a comparison of functional implementations for the Pseudoknot problem with a benchmark C implementation. The Pseudoknot problem is float-

ing point operation intense and the goal of the investigation was to compare how well the languages could be compiled into efficient code. The Pseudoknot experiment restricted implementations to use identical algorithms. Their comparisons were performance based yielding the best performing functional implementations were 1.4 times slower than the C benchmark. In addition, while the algorithm used was designed for strict languages, the efficiency of solutions in lazy languages was comparable to that of the strict languages and thus the lazy implementations did not suffer from overhead incurred by unexploited laziness. Their work shows that most of the compiled functional languages are only at most four times slower than their C implementation. In a recent paper, Adler [8] corroborates this results and suggests that for this reason performance is not the reason why conventional languages are the choice language for most projects. Java and perl are prime examples of languages that are slower than C and C++ yet are still used widely.

In another experiment Hudak and Jones explore the use of functional languages for software prototyping. In an experiment with nine languages including C++, Ada, Haskell and Lisp, the C++ and Ada solutions had an order of magnitude more code and took more than twice the amount of time¹. In addition, a programmer new to the Haskell language was able to utilize the language to the same degree as that of a veteran Haskell programmer. Hudak and Jones make an interesting point that the reception of such a concise solution to the problem in Haskell by other parties that were more accustomed to conventional languages was first disbelief that it was actually a complete solution and then denial that the use of higher-order functions was broadly applicable. To these attitudes, Hudak and Jones conclude that “sociological and psychological barriers must be overcome” for functional languages to be used [4].

In yet another study by Harrison *et. al.* compared algorithms written in functional language SML and in object-oriented language C++ for a dozen problems taken from the image analysis domain. They devel-

oped a metric for analysis for the SML code based on number of functions called and number of lines of code and a similar metric for the C++ code based on the number of functions declared and the number of lines of code. They found that these numbers were correlated with the complexity of making modifications to the code. They recorded comparable development and modify times from the SML and C++ code, while testing SML took longer. Their metric showed more reuse in the SML code than in the C++ code [2].

These related case studies all suggest that there is something to be gained by using functional programming languages for projects. Wadler suggests many reasons for the lack widespread use of functional languages: compatibility between components of a system; portability to the design platform; packagability as a standalone program; training programmers to use the language; tools to program, understand, and debug; and popularity of the language [8]. Our experimentation does not test these issues, but they are valid points for consideration.

3 Design and Implementation

The basic chat server supports features such as user login, communication channels, user paging, and chat server state queries (such as channel listings). The overall design in both implementations utilizes user level threads that perform blocking read and write operations on the TCP/IP stream. In each implementation there is a *server thread* that waits for incoming connections and when they arrive, it spawns off *client-handling thread* to handle the new client. The client handling thread firsts prompt the user on the other end of the connection for a login name. Then it enters a read loop reading in commands from the user and executing them on the users behalf. The client handling thread terminates when the user initiates a *quit* or the connection is broken.

3.1 Java Design

The initial Java design included a class for clients, `ChatClient`, a class for channels, `ChatChannel`, and

¹Unfortunately the time for the C++ implementation was not recorded

a class for the server, `ChatServer`.

3.1.1 The `ChatServer` class

The `ChatServer` class could have been replaced by a simple static method controlling the main server loop, but this would have led to a design that assumed a single server thread, and would have precluded augmentation of this design through subclassing. The `ChatServer` works with `ChatChannels` and `ChatClients`, and provides methods for adding and removing channels or clients, retrieving one or all of the channels or clients, and inquiring about whether a server has a specified channel or client.

3.1.2 The `ChatChannel` class

The `ChatChannel` class works with clients, and messages. It provides methods for adding and removing clients, retrieving all of the channel's clients, and inquiring about whether a channel has a specified client. It also provides methods for writing a message in the form of a `String` to the channel – and thus all of the clients of the channel.

3.1.3 The `ChatClient` class

The `ChatClient` class contains the bulk of the code for this system. The `ChatClient` class works with channels, messages and data, and commands. It provides methods for adding and removing channels, retrieving channels, and inquiring if a client has a specified channel. It provides methods for writing messages and data to the client. And though not part of its interface, it also contains methods for writing messages and data to the default channel, for handling the read execute loop, and handling each of the client commands that are available.

3.1.4 Integration

When the `ChatServer` detects a new client connection, a new client thread is created and this `ChatClient` interacts with the user to acquire a unique user name. The client thread then adds itself to the `ChatServer`. If the client creates a new channel, the `ChatClient` requests the `ChatServer` create

this new `ChatChannel`. If a client sends a message on a channel, the `ChatClient` writes the message to the `ChatChannel`, and then the `ChatChannel` writes the message to all of the `ChatClients` which are associated with the `ChatChannel`.

The methods in `ChatClient` that work with `ChatChannels`, and the methods in `ChatChannel` that work with `ChatClients`, enforce consistency, and eliminate these kinds of errors. For example, if a `ChatChannel` is added to a `ChatClient`, then it is automatically ensured that the `ChatClient` is added to the `ChatChannel`, and vice versa.

3.1.5 Reflection on this design

An interesting issue concerning the design of the Java version is that the details of each class's interface migrated during development as the issues concerning the complex interactions of the classes, especially with regard to synchronization, were resolved. Our employment of “incremental prototyping” was due mainly to the fact that the lead designer of the Java version was not familiar with the construction of chat servers, or even many of the components involved. It is likely that an experienced engineer developing a new large software system would experience a similar situation. This, we believe, points out a fundamental challenge in object-oriented design, which is that objects do not work in isolation, they must be designed to work in harmony. How objects should be integrated is not always clear, and the best design for a class may need to consider the design of other classes, and vice versa. It are these cyclic dependencies in the design which caused the most problems since a simple layered uses relationship could not be found. And since a `ChatServer` interacts with `ChatClients` and `ChatChannels`, etc., the uses relationship for this system is the worst case: a clique.

As Java itself has matured, many of the preferred uses of classes found in the supporting libraries have changed as the library developers have come to better understand how these class work together, and thus, how to model the individual classes. Thus, in any object-oriented design, the best way to model something depends on the surrounding context, or really, an understanding of that context, and cannot

be expected to be absolute. But supposing that an interface is absolute is what object-oriented design depends on for many of its merits.

3.2 Scheme Design

The implementation of the basic chat server in scheme consisted of five major parts² a library of useful higher-order functions, implementing threads in scheme, implementing the *client-handling thread*, implementing the *command-table*, and implementing the channels.

3.2.1 Higher-order functions

Higher-order functions play a key role in the way functional languages are used. They provide a mechanism for gluing modules together in a way that is not available in conventional languages. As Hughes discusses in [5] higher order functions such as *foldr* (Hughes calls this *reduce*), *map*, *compose*, and *curry* allow functions to be combined to create new functions allowing for very reusable code. Our implementation relied very heavily on these and a few other first-order functions.

As an example, the following code defines a function *write-all* which will send a message to all connected clients. Note that the connected clients are in a table *client-table*. This glues together functions from the thread module and the table module.

```
(define (write-all msg)
  (map
    (compose (curry write-line msg) car)
    (table-values client-table))
  )
  'ok
)
```

Here, we map a function which first takes the *car* of a list, then calls *write-line* with the message *msg*. This is called on the all the values in the *client-table* (which incidentally are lists with the first element being the output *port*³. To do the same thing without

²We will not discuss the implementation of the aforementioned *table* and *queue* data structures.

³TCP/TP sockets, as well as the standard input, output, and file input and output in scheme are abstracted into the notion of *ports*.

curry, *compose*, and *map* would have looked like this:

```
(define (write-all msg)
  (write-all-h msg (table-values client-table))
)

(define (write-all-h msg values)
  (if (null? values)
    'ok
    (begin
      (write-line msg (car values))
      (write-all-help msg (cdr values))
    )
  )
)
```

While the actual code is not that difficult to understand in either case, we see real benefit when we do not have to write the recursive helper function *write-all-h* for the dozen or special cases of this same style composition of modules.

3.2.2 Threads in Scheme

The design and implementation of the basic chat server in Scheme involved first using first-class continuations[1] to implement a user level threading system that allows individual threads to make blocking read calls to input *ports*. To do this, threading primitives such as *thread-fork*, *thread-yield*, *client-read-line* and *server-accept* were implemented. The *thread-yield* and *thread-fork* made up a basic non-preemptive threading system and the *client-read-line* and *server-accept*⁴ allowed for thread-blocking reads. With these primitive thread operations written, writing the *server-thread* to call *server-accept* and spawn a new *client-handling thread* as described previously is simple. As well, the client-handling thread can easily be written to repeatedly call *client-read-line*.

As an example of how simple the threaded code was to write, here is the implementation of the *server-thread* which is invoked with a socket that is bound to the TCP/IP port and a function that will be executed as the client-handling thread.

⁴*server-accept* is the thread-blocking wrapper for the Scheme interface to the *accept* system call that returns a TCP/IP stream for a new client connection.

```

(define (server-loop sock client-func)
  (let* ((client (server-accept sock))
        (c-sock (car client))
        (c-info (cdr client))
        )
    (thread-fork (curry
                  client-func
                  c-sock
                  c-info
                  )
                )
    )
  (server-loop sock client-func)
)

```

3.2.3 The client-handling thread

Using lexical scoping and the currying, the first thing the client-handling thread did was create functions `read-line` and `write-line` that read and wrote to the client's TCP/IP stream. With these two functions, the handler first asks the user to login with a unique name, it records this and other client information in a global table accessible to all other client-handling threads, then it enters a tail-recursive loop reading in a line of input from the user and processing it with the *command-table* (see below). When the stream is closed or the client requests to quit, the loop exits and the client-handling thread removes its entry in the global table of clients. The client thread then terminates by returning.

3.2.4 The command-table

The command-table was a table of functions, all lexically scoped inside the client-handling thread with access to the client's login name and other state variables. The table is indexed by string values for the command names. For example, looking up `\who` returns a function that lists all connected clients by login name. Likewise, for the command `\default <channel>`, looking up the command `\default` returns a function that takes one argument, the channel, and sets the default channel for the user to be the given channel.

3.2.5 Channel implementation

Channels were implemented as a global table of channel names, each with an associated list of user names. In the client-handling thread, a function (`write-channel <channel-name> <message>`) was lexically scoped to write a message from this user to the channel specified. It did so by mapping `write-line` composed with a function to lookup the *port* of a user, over the list of users on the specified channel.

3.2.6 Reflection on this design

This design heavily used higher-order functions and an applicative programming style that is natural in Scheme and other functional languages. One annoyance was that in order to get functions lexically scoped inside the client-handling thread, all of these functions had to be defined inside the thread function. Thus, the thread function consisted mostly of function definitions with a few calls to these functions in the body of the function. From a readability standpoint, it would have been convenient to have been able to declare these functions elsewhere and yet have them scoped inside the client-handling thread.

We did not try to write the Scheme code in a pure functional style (without side-effects). While there are many advantages to this style [5, ?], it is not the most natural style for Scheme. Since Scheme is a strict language and does not take advantage of referential transparency that is present in pure functional language it seemed reasonable not to adhere to the side-effect free programming style.

4 Debugging

4.1 Java Debugging

The debugging phase of the Java version was initially difficult, and then quite simple. Initially, the standard Java runtime environment was used (the `java` command) to execute the server, but this runtime system would not report all of the information that was encoded in the bytecodes when an exception was thrown: for example, instead of listing the

associated line numbers on the stack trace when an exception was thrown, it would just list `compiled code`. A debugger was then used, `jdb`, and it displayed this symbolic information. The debugger was also used instead of `print` statements as a means of tracking down errors. None of the bugs encountered took more than a few minutes to correct, and in total, learning the functionality of the debugger took a similar amount of time as debugging this project.

4.2 Scheme Debugging

It is relatively easy for an interpreted language such as Scheme to give good debugging information such as stack-traces and accurate information line numbers of errors. The Scheme interpreter that was used for this project, `guile`, sometimes gave really helpful information about errors. However, more often than not, the error messages were as obtuse as

```
ERROR: bad bindings
```

without a clue as to where in the code this happened. Incidentally, `bad bindings` is a result of a malformed `let` expression. If the scheme code were to be compiled instead of interpreted, this would be a compile-time error. Thus, the `guile` interpreter did not make the debug process as easy as it could.

Scheme, like many other functional languages, is blessed with the quality that once the programmer gets the code passed the compiler, it works on the first try [8]. While this was not universally true in the implementation of the debugging of the chat server, it was evident. Originally we were planning on using a version of scheme called `RScheme`[6]; however, we switched to `guile` early in the implementation stage for technical reasons. Once the appropriate modifications were made to call `guile`'s networking primitives, the code worked the first time. It never worked in `RScheme`.

5 Modifications to the Problem

We introduced two “unexpected” changes into the specifications of the chat server. Each change was purposely chosen to test the flexibility of each of the

designs, and thus each change needed to be a reasonable, though perhaps not anticipated, change.

Both the Java implementation and the Scheme implementation rely heavily on the fact that people use the chat server for conversing in plain text in a line by line fashion. For this reason, we chose to add a new requirement to the system: that it handle binary data. This would allow clients to exchange binary data with each other over channels.

The second modification was to allow a user to specify a “user block” which blocks all messages from a specified user or users. With this feature, a user can easily ignore the comments of any annoying user. We chose to implement this change because it also seemed non trivial given the design of both implementations thus far.

There were many other proposed modifications that were not implemented because the changes that would have been involved with implementing the modification would not flex the design of either of our projects, and thus, would not highlight the language issues that we were trying to expose.

5.1 Java Modifications

5.1.1 Binary data transfer modification

To implement binary data transfer, the code to parse the corresponding command and the method which processed the command needed to be added. The code that needed to be added for parsing a new command is trivial, and the code to execute the command itself leveraged new functionality added to the classes, and thus was also trivial. However, both `ChatClients` and `ChatChannels` were only designed to read and write `Strings`. Methods which are very similar to those for handling `Strings` were added to both the `ChatClient` and `ChatChannel` classes. Thus, the interface for both of these classes changed with the addition of this feature, however, they should not have needed to if a more general design was adapted originally: if messages were objectified, then a message object could have encased a `String`, an array of bytes, or any other kind of data representation that may have been requested in a future modification. Also, the system was not altered

into this more general design, even though the modification pointed out the problem with the current design. This corner was cut for the purely practical reasons of time. And in practice, it is unlikely that even if a modification helps point out a design flaw that the system will be redesigned, unless this redesign is seen as largely beneficial in the long run and can be afforded at the time.

5.1.2 User blocking modification

In the original design, when a client wrote a message to its default channel, the message data was the only data that was passed to the channels and ultimately back to the appropriate clients. For this modification, all of the corresponding methods needed to be modified so that the extra piece of information, the message originator, could be included. Again, this modification pointed out the same deficiency in the original design as the first modification: messages should have been objects. A message object could have been easily augmented to hold the originator of the message, and then a method would have been added to its interface to retrieve the message originator. This design would have made this modification quite simple, and would not have affected parts of the code which didn't concern the change.

5.2 Scheme Modifications

5.2.1 Binary data transfer modification

The first of the modifications to the scheme code, namely adding the ability to send binary involved two steps. First the homespun thread system was only written to allow for a thread-blocking `read-line` method and was not designed for doing a thread-blocking read of an arbitrary amount of data. To change this, a thread-blocking `read-bytes` function was written and added to the underlying thread library. The implementation of this was very similar to that of the original `read-line` function; however, none of the code was reused. Note that in a system with builtin threads, the input and output calls should automatically perform thread blocking correctly. Since we wrote our own thread system we

have to individually modify all output calls so as to block just a single thread and not the entire process.

The second stage of adding the ability to send binary was to modify the chat server to use the `read-bytes` function. This modification was simple. The *key-command* `#<num>` was added, to read in `num` bytes of data from the client and broadcast this to the default channel. This was just a simple call to `read-bytes`. This change to the server was easily isolated from other features of the server and did not make the actual server code more complicated or cumbersome.

5.2.2 User blocking modification

The second modification to the scheme code was adding the ability for a client to request that messages from a certain set of other clients be filtered out at the server end. This modification consisted of two stages as well. The first stage involved making the message sending system more general. Instead of allowing channels to write to a scheme *port* for sending a message to a client, we require that channels instead invoke that clients `receive-from` or `raw-receive-from` function. These functions are kept in the global client table. This would allow each client to specify exactly how it wanted messages sent to it. It would also allow it to ignore certain messages. The `receive-from` functions take as arguments the channel that the message is being sent on, and the name of the client that is sending the message. To completely make this modification, the existing `channel-write` method was changed to use the `receive-from` for the clients on that channel. In addition a `channel-write-raw` function was added to do more low-level channel writes. Existing code called only `channel-write` all channel writes so this code had to be modified to handle the new `channel-write` interface.

With these changes in place, adding the ability for a user to block messages from other users was simple. A list of blocked users was associated with the user. The `raw-receive-from` function was modified to only write out messages that were sent from users that are not in the *block-list*. The commands `\block <user>` and `\unblock <user>` were added to

the *command-table*.

With the changes made to allow for the ability to block messages, the system was made more general. Again, the added code did not make the system more complicated or cumbersome to work with.

6 Conclusions

While initially it seemed reasonable to choose Java for our prototypical conventional language Wadler [7] points out that Java is strongly typed, garbage collected, and “higher order” in the sense that objects which are first-class contain methods which can be used to give much of the same effect as first-class functions.

Although Java could have been used in this way, it was not. One way that the Java design could have benefited from this, however, would have been by objectifying the client commands. This would have easily allowed special commands to be added to a specific user or channel, or more generally, the commands available to any user or channel to be easily regulated. Recall that the Scheme version defined the client commands as a table of functions, and thus could have incorporated a change of this kind rather easily, while the Java version would need an extensive revision.

It is also evident that having user level *thread* support either in the language as in Java, or having the language expressive enough so as to allow threads to be implemented by hand as in Scheme, makes handling the clients connected to the chat server much easier. Each client gets a thread of control that has read/write access to a TCP/IP stream. These threads can do blocking read and writes to the stream without affecting other clients. Thus, the clients state is stored in the call stack of the thread. Implementations of a chat server in a language such as C or C++ that does not support user level threads that block themselves on system calls but do not block the entire process (i.e. kernel threads) would be forced to model client state as a state machine. A change such as adding support for binary would then involve a new state that signifies that binary is being read, where as in the threaded Java and Scheme implementations, a

simple function call to the `read-binary` function.

Though the Java and Scheme implementation had many dissimilarities, there were more similarities, and it was difficult finding modifications that would lead to one implementation exhibiting better design than the other. The number of non-blank lines of code in the Java version was 724 compared to 729 lines of code in the Scheme version⁵. Additionally, the time for the implementation of both systems was similar, as was testing and debugging time. While it is difficult to generalize the results that we obtained in this endeavor to other software projects, potentially of much larger size, it is clear that functional languages, and the features they offer, have more worth than is conventionally recognized.

7 Future Work

There were several issues that were not fully explored by the work presented in this study. We leave these as excersizes for future research. First, there is evidence that Java is not a prototypical conventional language. It is garbage collected, strongly typed, and can be considered to be higher-order [7]. Future work should include comparisions to implementations in C and C++.

As mentioned in section 6 a crucial issue was support for threads with thread-blocking input and output operations. This study was not ment to be an arguement for or against a certain variety of user level threads. It might be that any conventional language with good support for threads would be a good language for this project. Likewise, any functional language that either has support for threads or allows the programmer to create a homespun threads system would have the same advantage. Threads are a powerful abstraction that allows the programer to avoid state-machine style archetectures which can be cumbersom to deal with. In Java threads are first-class objects. Synchronization primitives allow for fine-grained control of threads much as we have fine grained control over functions in a functional languages. It seems like adding first-class threads

⁵The Scheme figure also counts lines of code which are in its thread library.

with synchronization primitives to a conventional language might be the sort of glue needed to allow modules in these languages to be combined in ways similar to those of functional languages.

The Scheme implementation, while it did exercise several powerful features of functional languages, it did not explore the use of pure functional, lazy languages that take advantage of referential transparency as applauded in [5]. Studying this same problem from a pure functional setting is left for future work.

A Chat Server Documentation

A.1 Original Commands

- To send a message `<message>` on the channel specified by `<channel-name>`

```
\channel <channel-name> <message>
@<channel-name> <message>
```

- This creates a new channel named `<channel-name>` and adds the user to it.

```
\newchannel <channel-name>
```

- This prints out a list of all the active channels.

```
\channels
```

- This sends a `<message>` to `<user>`.

```
\user <user-name> <message>
'<user-name> <message>
```

- This reports on the user's status, it prints out the the user's current list of channels and the default channel.

```
\status
```

- This prints out a list of all the users that are currently connected to the chat server.

```
\who
```

- This quits.

```
\quit
```

- This prints out help information with a list of commands.

```
\help
```

- This adds the user to channel, `<channel-name>`.

```
\join <channel-name>
+<channel-name>
```

- This removes the user from channel, `<channel-name>`.

```
\leave <channel-name>
-<channel-name>
```

- This sets the user's default channel to `<channel-name>`

```
\default <channel-name>
```

A.2 Added Commands

- This allows the user to send binary to the current channel. This is a two line command, the first line specifies how much binary there is and the second "line" contains the binary.

```
#<num>
<num bytes of binary>
```

- This blocks all messages from `<user>`

```
\block <user>
```

- This allows message from currently blocked user, `<user>`.

```
\unblock <user>
```

References

- [1] William Clinger, Jonathan Rees, et al. The Revised⁴ Report on the Algorithmic Language Scheme. *ACM Lisp Pointers*, 4(3):1–55, 1991.
- [2] R. Harrison, L. G. Samaraweera, M.R. Dobie, and P. H. Lewis. An Empirical Evaluation of Functional and Object-Oriented Languages. Technical report, University of Southampton, 1994.

- [3] P. H. Hartel et al. Pseudoknot: A Float-Intensive benchmark for functional compilers. pages 13.1–13.34. School of Information Systems, University of East Anglia, Norwich, UK, Sept 1994.
- [4] Paul Hudak and Mark P. Jones. Haskell vs. Ada vs. C++ vs. Awk vs. ... An Experiment in Software Prototyping Productivity. Technical Report 1049, Department of Computer Science, Yale University, New Haven, CT 06518, July 1994.
- [5] J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.
- [6] Donovan Kolbly, Paul Wilson, and Robert Strandh. RScheme – Design and Implementation. unpublished, 1997.
- [7] Philip Wadler. Why no one uses functional languages. *ACM SIGPLAN Notices*, 7(2):127–128, 1997.
- [8] Philip Wadler. Why no one uses functional languages. *ACM SIGPLAN Notices*, to appear.