Developing Beginner-Friendly User Interactions for the Clojure Programming Language

Henry Fellows, Aaron Lemmon, Max Magnuson,
Emma Sax, Paul Schliep, and Elena Machkasova
Computer Science Discipline
University of Minnesota Morris
Morris, MN 56267
fello056@morris.umn.edu, lemmo031@morris.umn.edu, magnu401@morris.umn.edu,
saxxx027@morris.umn.edu, schli202@morris.umn.edu, elenam@morris.umn.edu

Abstract

Clojure is a newer functional programming language in the Lisp family of languages that runs on the Java Virtual Machine (JVM). It is structured to use immutable data types and a rich collection of predefined data structures (lists, vectors, hash maps, etc.) which makes it a promising candidate for an introductory Computer Science (CS) class. However, Clojure presents significant challenges to introductory level students. The primary challenge for novice programmers is the confusing error messages that originate in the underlying Java system. Our work is a part of the ClojurEd project which aims to use Clojure to teach an introductory CS course. The goal is to create a programming environment that provides introductory students with understandable error messages and usable project management tools. We discuss our accomplishments, current work in progress, and future directions.

1 Introduction and Background

Felleisen et al [2] have made an excellent case for using a Lisp as a programming language in an introductory CS class: the first language new programmers will use. Lisp offers a simple syntax and introduces students to modularity, abstraction, and data-driven program design while developing good programming practices by being explicit about program design principles. Essential concepts taught in such an introductory class have been shown effective as a strong foundation for students to build upon in later classes that introduce imperative features and the object-oriented paradigm [1].

Clojure is a relatively new language in the Lisp family that is rapidly gaining popularity in industry and in the open source community. Clojure provides a functional easy-to-use approach to developing robust multi-threaded programs.

Clojure offers a variety of built-in datatypes and functions that, when introduced rigorously and systematically in an introductory CS class, would allow students to practice data-focused program development. The abundance of open source libraries and projects allow students to continue development in Clojure after finishing the introductory course, both for their own purposes (such as data processing for other courses) and as contributions to projects of others.

This paper describes our work in progress in making Clojure and its environment usable for introductory level students. Our efforts include transforming Clojure error messages into more beginner-friendly ones and developing tools that would allow students to seamlessly run their program code integrated with our modifications without having to deal with Clojure tools that require more advanced knowledge.

1.1 Overview of Clojure

Clojure, developed in 2007 by Rich Hickey [4], is a dynamic, functional programming language in the Lisp family. Dynamic typing means that Clojure does not associate types with variables, but rather with values that they contain. When a type of a value is inconsistent with one expected in a given context, a runtime error occurs. Functional means that Clojure functions are first-class values that can be passed as arguments and returned as values. Clojure utilizes the Java Virtual Machine (JVM) as a runtime environment. Clojure compiles into Java bytecode and runs on the JVM, which abstracts over the specifics of the hardware. This setup offers easy access to Java frameworks. The Clojure read-eval-print loop (REPL) is another easy way to interact with Clojure. A REPL is an interactive shell program that takes single expressions, evaluates them, and returns the result to the user. The REPL allows users to see immediate results and respond easily and quickly. For any programmer, the use of the REPL is highly advised and can make programming in Clojure much simpler. Use of the REPL allows users to break down their code, test specific pieces, and experiment with new functions and uses [5].

Because Clojure is a functional language, it puts strong emphasis on immutable data types.

An immutable data type means data cannot be changed. When using Clojure, in order to change a data item, an entirely new data item must be made. Immutable data types help prevent side effects in programs. A side effect is when a function alters memory outside of its scope. Side effects can make debugging or resolving errors more difficult. This is because when a function interacts with other parts of a program, when it is not supposed to, any issues in the code can propagate throughout the program. The reduction of side effects makes problems with the code are easier to find and fix. Because of this, immutable data types are practical for novice programmers.

Clojure, like other Lisp languages, uses prefix notation. This means that function calls use parenthesis, followed by the function name, and then any parameters:

```
(<function-name> <argument 1> <argument 2>)
```

An example of this can be seen through addition, which is a built-in function, unlike traditional languages where + is an operation:

```
(+ 5 5)
-> 10
```

Note that -> indicates the result of computations in the Clojure interpreter.

Clojure also has offers accessibility to Java functions and Java interoperability. This means that any Java method can be called just like normal Clojure functions.

Clojure users can also define functions and variables. Using the keyword `def`, we can define variables:

```
(def mystring "Hello World")
mystring
-> "Hello World"
```

In the above example, we define a variable to hold the string to be `"Hello World"`. This way, whenever we reference `mystring`, the string we bound to the variable name will be returned. We can also use `defn` to define functions:

```
(defn increment-number [number] (+ number 1))
(increment-number 2)
-> 3
```

In this example, we defined a function that takes a number and returns that number incremented by 1. In any function, the function name is the word after the `defn`, the pieces in the square brackets after the function name indicate the function's parameters, and what follows is the body, which is the expression returned by the function.

Now let us say that we have a program where we want to use a function, but only once. This means that we do not necessarily need to define it with a name because Clojure supports anonymous functions. Anonymous functions allow programmers to quickly define functions in place when needed. However, since these functions are not stored, the program would not be able to use them more than once. The following example is of the same

increment-number function as above, but implemented anonymously. The `map` function takes in a function and a collection as arguments, and applies the function to the collection:

```
(map (fn [number] (+ number 1)) [0 1 2 3])
-> (1 2 3 4)
```

Clojure also has a variety of different types of data structures, also referred to as collections. All of Clojure's collections are immutable, which means that the data within the structures cannot be modified. The different types of structures Clojure uses are lists (denoted by `()`), sets (denoted by `#{}`), vectors (denoted by `[]`), and hashmaps (denoted by {}). The first three types of data structures mentioned can contain any number of values of any data type.

```
(1 2 "foo" :a 9 "bar")
```

However, hashmaps are unique in the fact that it is a collection of key-value pairs:

```
{key value, key value, key value}
```

Hashmaps commonly use keywords. Keywords are simple names that have a colon in front. An example of a hashmap using keywords follows:

```
{:a 1, :b 2, :c 3}
```

The keywords in the above hashmap are: `:a`, `:b`, and `:c`. Keywords, such as the ones above, are often used as keys within hashmaps. The value is the second piece of a key-value pair, and each value is bound to a key. In the above example, `:a` is bound to `1`, `:b` is bound to `2`, and `:c` is bound to `3`.

Just like all other values, data structures can also be bound to a variable name:

```
(def myhashmap {:a 1 :b 2 :c 3})
myhashmap
-> {:a 1, :b 2, :c 3}
```

Laziness is another common feature of Clojure. Laziness is when the evaluation of an expression is postponed until the return value is needed. An example is the Fibonacci sequence. It would be impossible to store all of the infinite numbers in the Fibonacci sequence. By making a lazy function, Clojure can evaluate only as much of the Fibonacci sequence as necessary. The next example illustrating laziness uses the functions `take` and `range`. `take` takes the first n elements of a collection, both of which are given as arguments. `range` returns an infinite sequence of non-negative integers, beginning at 0:

```
(take 10 (range))
-> (0 1 2 3 4 5 6 7 8 9)
```

Another common example of laziness are `if` statements:

```
(if (< number 10) (+ number 1) (- number 1))
```

In the above example, the `(+ number 1)` or `(- number 1)` only evaluates depending on whether the `(< number 10)` evaluates to true or false.

The opposite of a lazy evaluation is an eager evaluation. Eager evaluations fully evaluate their parameters collection upon runtime.

1.2 Overview of ClojurEd project

ClojureEd is a project at UMM that is devoted to increasing usability of Clojure in an educational setting, in particular in introductory classes. It originated in 2013 as a joint effort of alumni, students, and faculty. Several students have contributed to it as a part of several research efforts, in particular as a part of a two months summer research program in 2014 sponsored by the HHMI (Howard Hughes Medical Institute) grant and UMM MAP (Morris Academic Partnership). Two other students are currently working on the project sponsored by UMN UROP (Undergraduate Research Opportunity Program).

As a part of the project we have developed the system of modifying Clojure error messages, as described in section 2. The current challenges involve integrating our system with Clojure project management tools and IDEs. The summer project explored some approaches to this problem, but did not result in a solution. Recently we have identified elements of the Clojure project architecture that would allow us to develop plugins to handle student code in a way that would not require any advanced knowledge (such as working with command line) on their part. However, this is still work in progress, and no working solution exists at this point. Section 3 details these efforts and challenges.

The code is available at

`https://github.com/Clojure-Intro-Course/clojure-intro-class`

2 Error Messages

2.1 Current error messages in Clojure

Error messages are an important part of programming since they are the main source of communication between a user and the system when an error occurs in the program. Error messages are especially important for introductory programmers who have had little to no experience with troubleshooting programs. These error messages need to provide helpful and easy-to-understand information that can be used to resolve issues.

Error messages in Clojure are not particularly useful for introductory programmers because they do not provide information that can help guide a student in fixing the error. Also, Clojure error messages are typically complex because they originate from the underlying Java interpreter (JVM), which most novice programmers will be unfamiliar with. Below are some examples of error messages in Clojure that might be unhelpful or unintuitive for an introductory student.

2.1.1 Example 1

Consider the following (erroneous) code fragment:

```
(defn square-this (* input input))
```

In this code sample, the programmer is attempting to create a function `square-this` which will take a number and return the square of its value. However, the programmer forgot to declare that `input` should be a parameter. In Clojure, function declarations always require a vector containing the declared parameter names. The resulting error message is:

```
IllegalArgumentException Parameter declaration * should be
a vector
clojure.core/assert-valid-fdecl (core.clj:6842)
```

It does provide key words that could lead the programmer to fixing the issue such as `Parameter` and `vector`. However, the error message also includes information that a new programmer might find intimidating or confusing such as

```
clojure.core/assert-valid-fdecl (core.clj:6842).
```

The error can be corrected by placing a vector of inputs, in this case containing a single input, right after the function name. Here is an example of the code above after corrections have been made:

```
(defn square-this [input] (* input input))
```

2.1.2 Example 2

In the next example, the programmer is attempting to return a new hashmap with an added key-value pair. The function `assoc` is generally used to make this happen.

```
(assoc :a 3 {:a 5, :b 8, :c 9})
```

In this attempt, the programmer did not put the arguments for `assoc` in the correct order. When using the function `assoc`, the hashmap should go before the new key and value. The error message that results follows:

```
ClassCastException clojure.lang.Keyword
cannot be cast to clojure.lang.Associative
clojure.lang.RT.assoc (RT.java:702)
```

Since Clojure expects the first argument to be a hashmap and it is not, it tries to cast the keyword `:a` into a hashmap. This error message refers to `clojure.lang.Associative`, which is actually a Java interface. Referring to that interface may not be useful for Clojure programmers if they have little to no experience with Java or type hierarchies. Furthermore, the error message refers to typecasting, which is unfamiliar to Clojure users since Clojure is dynamically typed with no explicit type declarations. The error message is inef-

fective at explaining to the programmer that the underlying problem with their code was a misordering of arguments. Here is an example of the code above after corrections:

```
(assoc {:a 5, :b 8, :c 9} :a 3)
-> {:c 9, :b 8, :a 3}
```

2.2 Error message transformations

Since standard Clojure error messages are generally unhelpful, we aim to replace many Clojure error messages with improved ones. A way of accomplishing this is by reading in a user's code and wrapping it in a `try/catch` block. Then any errors thrown by the user's code will be checked against a large collection of regular expressions. If a regular expression matches, important details from the original error message are captured and used to insert details into our replacement message. However, this `try/catch` approach does not work well in all situations. Additional technical challenges arise when dealing with compiler exceptions, the REPL, and lazy sequences. Section 3 discusses the implementation details and current progress in more detail.

In the case of predefined functions, such as `map`, we achieve more informative error messages through function substitution. This involves defining functions with the same name as standard Clojure functions. Within these definitions, we can do type checking on arguments passed in. If the type checks find an error, our system displays a customized error message. If no type errors are found, the arguments are passed on to the actual corresponding Clojure function. Although this method can only handle runtime exceptions, it is advantageous because we can capture the actual values of the arguments passed in to the redefined functions and use them to provide a detailed error message. As an example, consider the following erroneous code:

```
(map "add one" [1 2 3])
```

Clojure provides the following error message:

```
ClassCastException java.lang.String cannot be cast to
clojure.lang.IFn  clojure.core/map/fn--4245 (core.clj:2557)
```

Our function substitution error handling displays a more informative message:

```
ERROR: In function map, the first argument "add one" must be
a function but is a string.
```

This message provides information that the problem occurred in the first argument, the offending argument's value was "add one", and that the argument was a string instead of a function. While function substitution is cumbersome because it involves redefining each function individually, it is the only way to provide the user with specific information about which arguments are causing the problem.

Both the `try/catch` and function substitution methods together, unfortunately, do not handle all possible cases. For example, complexities arise when students define their own

functions, especially if they are anonymous functions. We are unable to anticipate the function definitions that users of our system create, so we cannot rely on function substitution for those cases. The `try/catch` approach must handle those scenarios. Replacement error messages are even harder to make for anonymous functions since they have no name by which we can refer to them. Although our system does not handle all cases, we hope that it will be helpful in the majority of cases that a Clojure programmer will encounter.

2.3 User scenarios

In order to better understand what our software users might encounter, we developed user scenarios detailing typical coding problems a new programmer might face. To create these user scenarios, we created solutions to common beginner programming exercises and purposefully introduced errors that a new programmer might make. We then recorded the error messages that these solutions produced. We also took note of the underlying cause of the errors so that we have a better idea of how to improve upon the error messages for our program. We then took the same solution and ran it within our program to compare the error message our system produces against the one Clojure produces.

These user scenarios showed us typical errors a student will be experiencing when learning to program in Clojure for the first time. For example, when a student first starts writing basic operations in Clojure, a typical mistake might be forgetting to put the function in front of the arguments, such as `(5 - 5)`, where it should be `(- 5 5)`. This produces a message that would be unhelpful for introductory programmers:

```
ClassCastException java.lang.Long cannot be cast to
clojure.lang.IFn user/eval769 (core.clj line 675)
```

In this error message, `clojure.lang.IFn` is a Java type representing Clojure functions. So, the error message is actually saying that 5 is a number, not a function, since Clojure expected the function name to be listed first. Our system produces the following error message to replace the one above:

```
ERROR: Attempted to use a number, but a function was
expected. intro.core/-main (core.clj line 675)
```

2.4 Future work: hints

While the error message our system produced above says what the core problem is, it does nothing to suggest to the user to check that the arguments are in the correct order, which is the fundamental issue. We are developing a system of hints that offer additional guidance in figuring out why errors have occurred. Hints are suggestions shown to the user along with an error message. Since there is no sure way of telling exactly what went wrong, we will offer several hints that may be applicable to the situation.

For example, a `ClassCastException` is usually thrown when programmers place arguments in an incorrect order. However, a `ClassCastException` can be thrown in many other cases as well, which can make providing a specific hint challenging. Thus, our system provides several hints in order to cover a variety of situations. Hopefully, one of the suggestions will be helpful in solving the real issue.

Developing user scenarios was a helpful exercise because it allowed us to immediately know the root cause of an error even before seeing the error message. This allowed us to easily record a human interpretation of the problem and relate it to the error. This process gave us a useful methodology for creating hints. As an example, the following user scenario involves a hypothetical attempt to write a program that would print "Hello World":

```
(print Hello World)
```

This results in the Clojure error message:

```
CompilerException java.lang.RuntimeException:
Unable to resolve symbol: Hello in this context
```

Our hint for this may be more informative and educational for a new programmer than the error message above:

```
It looks like Clojure is expecting that Hello is something
named in your program. If you wanted Hello and any following
words to be plain text, try surrounding them with double
quotes. If Hello is referring to something named in your
program, make sure it is spelled correctly.
```

User scenarios provide a useful methodology for generating hints, which we will continue to use to extend the system. We plan to work with Clojure beginners interactively to improve the hints in the system. We also plan to provide links to Clojure documentation when appropriate in the hints. Once beginner programmers are comfortable with the terminology used on the documentation pages, directing them to the documentation will provide them with a more concrete understanding of the language.

3 Technical Setup

In our prospective error handling, there are a large number of technical issues. First we need to figure out how to catch and process the errors. Then, we need to figure out how to integrate that system with the tools for managing Clojure projects. Throughout this process it is important to be mindful of usability for introductory students. Therefore we would like to synthesize the error handling system and the tools used by the student in a way that is both usable and robust. The system as a whole is a work in progress; the development and choice of tools takes a significant amount of time. We spent our time exploring tools and how to implement them. We decided that we would like to avoid altering the code

of the tools themselves, instead we would like to develop plugins that would provide the functionality that we need.

3.1 On the nature of errors in Clojure

There are two different environments where code can be evaluated in Clojure. A user can evaluate all of their code by running it, or they can evaluate code snippets using the REPL. When developing code, it is common practice for a user to use REPL to test pieces of their code. This is practice is described in depth in section 1.1. Due to the importance of REPL interactions for code development and debugging, it is essential for the errors in both environments to be consistent. Therefore, we need to make sure that our error handling system is capable of addressing the errors from compilation, runtime and REPL.

In Clojure, error messages are divided into two different categories, compilation time and runtime. As their names imply, compilation errors occur when the code is compiled, and runtime errors occur when the code is run. Compilation errors cannot be handled by a simple `try/catch` because they originate in the compiler before the code is even evaluated. Therefore, we need to catch the errors from the compiler so that we can handle them. The compiler errors common in Java shift to runtime errors in Clojure, because the types in Clojure are dynamic and are thus checked at runtime. Handling runtime errors is made relatively simple by using a `try/catch` with the exception of handling errors that involve laziness. Since lazy sequences are evaluated when they are needed, an error in a lazy sequence will not occur until the sequence needs to be evaluated. This means that the error will often not occur where it originated. When this is the case, the user will be provided with unhelpful or often misleading error messages about the nature of the problem. These errors can be very frustrating to troubleshoot especially for introductory students.

None of these problems have not been resolved, but we have determined approaches that should solve these problems. Finding the origin of errors caused by laziness has not been solved, but we have a potential solution that involves forcing evaluation of lazy sequences in student code. Compilation errors are also problematic, but we hope to resolve this by intercepting error messages as they are created, which is detailed in section 3.4.

3.2 Leiningen

Leiningen is an open source project under active development that is the primary tool for project management in the Clojure community[3]. Leiningen is a fully featured project manager that cleanly deals with dependencies as well as provides useful tools for developing projects. When a project is run in Leiningen, Leiningen will resolve all dependencies. For example, if your project requires a plugin, Leiningen will automatically retrieve, install, and apply that plugin to your code. After it has resolved the dependencies, Leiningen uses other tools to build, compile, and run the project. Leiningen makes running Clojure projects extremely simple for the user. With Leiningen, it only takes one command to begin

a new project, and it only takes one command to run a project. Because of the level of abstraction that Leiningen offers, we have decided that it is the best option for an introductory course.

While Leiningen is a very helpful tool, there are still some aspects that are challenging for introductory students. The most difficult of which is the interface. Leiningen uses the command line as its primary interface. The command line is both a distraction and a barrier to teaching the material in an introductory course. Therefore, we would like to abstract over the command line by using a graphical interface, ideally integrated with the IDE used by students. This graphical interface would extend the simplicity of Leiningen while still providing the same functionality needed by students.

3.3 IDEs

Choosing the right IDE is important, especially for an introductory class, because it is the environment in which students will develop their code. We have looked into many of the IDEs available for Clojure, and we have decided that the most promising choices are LightTable and Nightcode. Both IDEs offer intuitive interfaces, syntax checking, built-in REPLs, and easy project management. They each use Leiningen to build projects, but neither of them provides a complete integration with Leiningen. Users still need to use the command line for some functionality. They are also both maintained by open source communities and are under active development. This allows us to take advantage of new features and modifications developed by others.

Both IDEs have a lot of functionality that we are looking for, but they do have some notable differences. For instance, LightTable requires a separate installation of Leiningen. This adds another layer of difficulty for the student since installing Leiningen is non-trivial. Nightcode comes packaged with its own distribution of Leiningen. This makes Nightcode easier to install and use. The REPL differs between each IDE. In LightTable, the user opens up REPL using a context menu, and the REPL constantly evaluates each line of code in it. This ultimately provides more information, but it is very different from the traditional REPL in ways that may be a little confusing for introductory students. In Nightcode, the REPL will always up in a separate frame, and each line is only evaluated when the user hits enter. LightTable uses hot keys and a context menu to do a lot of the tasks in LightTable such as running the project or starting a REPL. This is less familiar to introductory students, and it may be a little daunting at first. LightTable does provide a nice easy reference for searching for the appropriate hot keys to lessen the learning curve. Alternatively, Nightcode uses a lot of clearly labeled buttons and tabs that are intuitive for the user to perform tasks. Since it is unclear which IDE will be more common in the Clojure community in the future, we plan to support both options.

3.4 Approaches

Our first approach was to develop an IDE-specific plugin to intercept the error messages going to the console of the IDE. We discovered that it was not the IDE that was directly interacting with the compiler. Our next approach was to use a Leiningen plugin to try and intercept error messages. We searched through the Leiningen source code, and attempted various methods of interacting with Leiningen. None of these approaches allowed us to intercept the error messages that were being printed to the console. We have determined that Leiningen uses nREPL, which is a Clojure interpreter, to handle compilation and error messages.

The nREPL interpreter is a client-server system for evaluating user code. nREPL middleware can intercept messages from the nREPL system and apply functions to those messages. nREPL middleware uses handlers for interacting with the messages. These handlers look for operations with certain keywords and only are applied to those specific operations. For example, an nREPL middleware can specify a handler for evaluation operations. The middleware will inspect every operation that passes through nREPL. If it is an evaluation operation, then the functions specified in the middleware will be applied to the operation. If the operation is not an evaluation function, then nothing will be changed.

Now, our planned approach is to write an nREPL middleware to intercept the error messages handled by nREPL and clean them up. This approach is also preferable to our initial IDE specific approach because it allows for our system to be IDE independent.

3.5 Implementing error handling

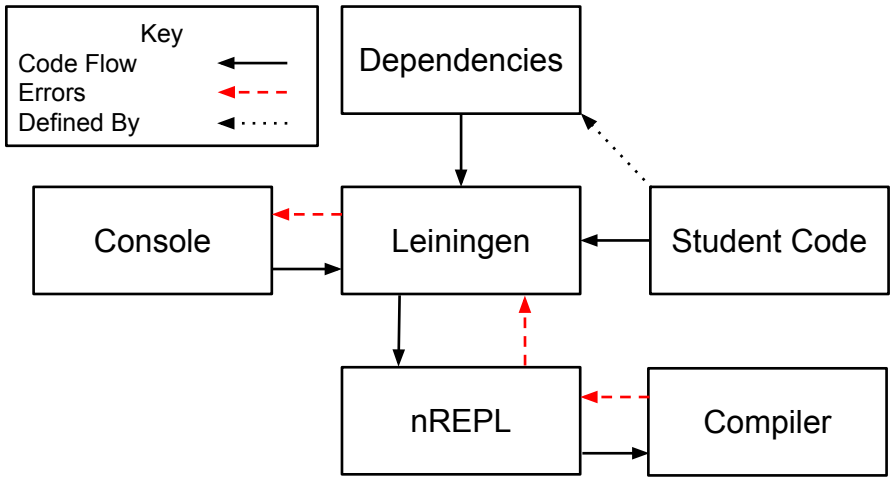

Figure 1: Workflow diagram of running a Clojure project in Leiningen

The current error handling system is described by figure 1.First, the student runs their project through Leiningen using the console by calling `lein run`. Then, Leiningen will take the dependencies defined in the student code and resolve them. After the dependencies

have been resolved, Leiningen then sends the code to nREPL to be compiled and evaluated. Any errors are passed back to the current output device, which is the terminal by default.

In our approach, we would like to catch the errors from nREPL by using nREPL middleware. Once we catch those errors we will transform them as described in section 2. After being cleaned up, the error will then be rethrown to Leiningen, so that it can be printed out for the student to see. We believe that nREPL handles all errors thrown by the compiler, runtime, and the REPL. The middleware will then handle all of these cases in a consistent manner.

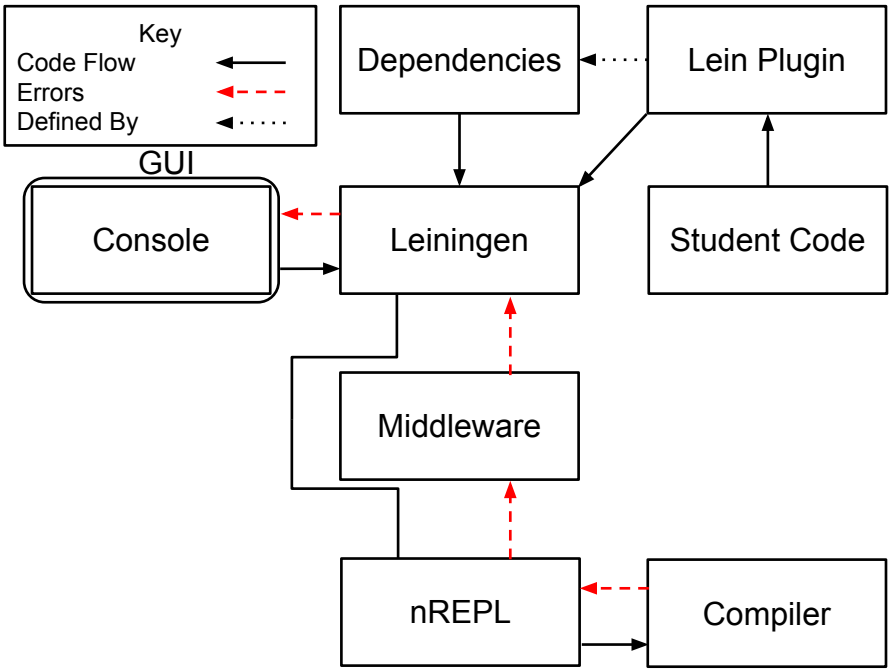

Figure 2: Planned workflow for our error handling system

Our plan for error handling is laid out in figure 2. First, the student will use our graphical user interface(GUI) to run their project. The GUI abstracts over the command line interactions by providing a way of interacting that is more familiar to students. Next, the Leiningen plugin will inject any dependencies into the student's code required by our system. This includes our library of functions for the student or a graphical library that the student may need. After being processed by the Leiningen plugin, the student code is then given to nREPL by Leiningen to be evaluated. nREPL then uses the Clojure compiler to compile the code. If any errors are thrown they will be given to the middleware. The middleware will then clean up those error messages before they are sent back to Leiningen to be displayed by the GUI.

4 Conclusions

Currently we have a framework for making Clojure error messages more accessible for an introductory student. However, we still need to integrate this framework with IDEs and tools for managing students' Clojure code. We have determined an approach to perform such an integration, and are currently implementing it.

Further work would involve adjusting the error messaging framework based on the actual users' feedback, developing an environment that would make it easy for students to run their code and manage their files, and performing usability studies to test that framework and system.

References

[1] BIENIUSA, A., DEGEN, M., HEIDEGGER, P., THIEMANN, P., WEHR, S., GASBICHLER, M., SPERBER, M., CRESTANI, M., KLAEREN, H., AND KNAUEL, E. Htdp and dmda in the battlefield: A case study in first-year programming instruction. In *Proceedings of the 2008 International Workshop on Functional and Declarative Programming in Education* (New York, NY, USA, 2008), FDPE '08, ACM, pp. 1–12.

[2] FELLEISEN, M., FINDLER, R. B., FLATT, M., AND KRISHNAMURTHI, S. The structure and interpretation of the computer science curriculum. *J. Funct. Program. 14*, 4 (July 2004), 365–378.

[3] HAGELBERG, P., OSBORNE, A., LARKIN, D., AND CONTRIBUTORS. Leiningen readme. https://github.com/technomancy/leiningen, 2015.

[4] HICKEY, R. The clojure programming language. In *Proceedings of the 2008 symposium on Dynamic languages* (New York, NY, USA, 2008), DLS '08, ACM, pp. 1:1–1:1.

[5] O'BRIEN, T. Clojure's advantage: Immediate feedback with repl. *Radar* (May 2012).