table of contents

## part one

Choice of user interface items 2

Case statement – ENBF model 4

Case statement – Railroad diagram 4

Data flow diagram 5

Data dictionary 6

## part two

Test data and expected output 7

Discussion of errors 8

Evidence of error checking 9

Evidence of testing 10

Evidence of debugging 13

## part three

Developer documentation 15

Readability of code 16

User documentation 17

part one

# choice of user interface items

Status bar

A feature of many IRC clients is a status bar, which often displays values such as the user and channel modes. Even though the user is able to manually retrieve these through several IRC commands, the decision to include a status bar in r.IRC were due to benefits such as being always available and self-updating. Due to the low minimum requirement specifications for screen size to use r.IRC, the status bar at the bottom was limited to three items; current user modes, current channel modes, and the connectivity of the user. Furthermore, the colour of the text in the connectivity panel varies depending on its status i.e. green when the user is online, red when the user is offline, allowing for greater accessibility, especially by the visually impaired.



Input box

A single line input box was chosen over a larger frame due to the short nature of instant messages – IRC is no exception. By having a smaller input box, there remains more of the limited screen real estate to dedicate to the history panel. Furthermore, a decision was made to not include an actual button to send the message due to a variety of reasons:

* r.IRC is designed to be used with a physical keyboard, not with an on-screen keyboard nor with a touchscreen
* The user is assumed to hold the common knowledge that the ‘Enter’ key sends their message
* The ‘Enter’ key would serve no other purpose, as the input box consists of a single line; line breaks are not possible
* Minimising unnecessary space wastage allows more user input to be seen at once

Multi-window interface

r.IRC was initially designed and storyboarded with a tabbed interface in mind, with design inspiration taken from various web browsers such as Mozilla Firefox and Opera. Due to a combination of factors such as the relative inability for chosen development tools to support such a layout and various ergonomic issues explored whilst debating its suitability, it has been decided to switch from the tabbed interface to a more traditional window-based interface.

The strongest argument for this switch is the tabbed interface’s unsuitability for users with more than a single concurrent chat. A tabbed interface is suitable for web browsers and FTP clients, simply because users can only do a single thing at a time for those software, be it view a web page or upload files. However, this is not the case for an IRC client, with which users can and should be expected to perform actions such as viewing two or more chats side by side, to compare or to communicate information. A tabbed interface would only allow for the viewing of a single chat frame, and thus does not suit this requirement; on the other hand, a multi-window interface allows the user to show and hide chat frames as he/she requires, moving them around to suit the situation.

The infancy of Shoes as a graphical user interface toolkit also encouraged this transition from a tabbed interface to a multi-window interface. Its development state as an alpha release justifies its relative lacking in proposed features; of particular note is the lack of scrolling support in internal frames. Without this feature, a difficult choice had to be made between the removal of extended message history and having persistent screen elements, and the importance of both dictated that a change in approach was the most suitable method to solving this issue.

Therefore, r.IRC now looks remarkably different from what the storyboards in past proposals have depicted. As well as being less clunky and more visually appealing and user encouraging, the new interface design is internally more manageable and less complex in nature. A primary window acts as a main control menu, which lists the currently open chat windows and provides functionality to the user to add more chat windows as well as manipulate settings and profiles. Each chat is now in a separate window, and any IRC commands that change or add an additional connection or chat will do so in a separate window.

# case statement - ebnf model

**case statement =**

CASE (<statement>|<variable>)

{WHEN (<constant>|<variable>)

{<statement>}}

END

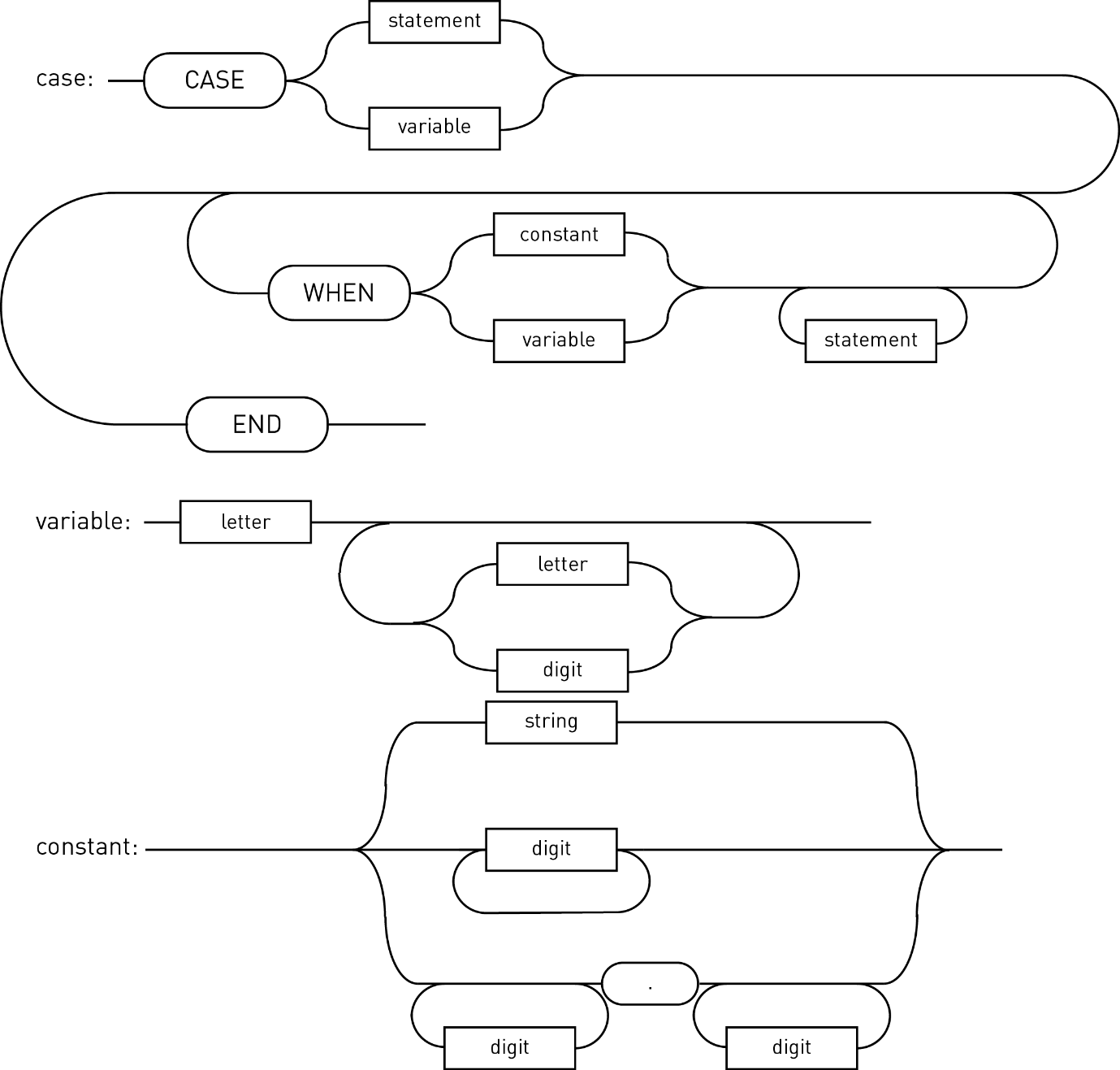
**variable =**

<letter>{<letter>|<digit>}

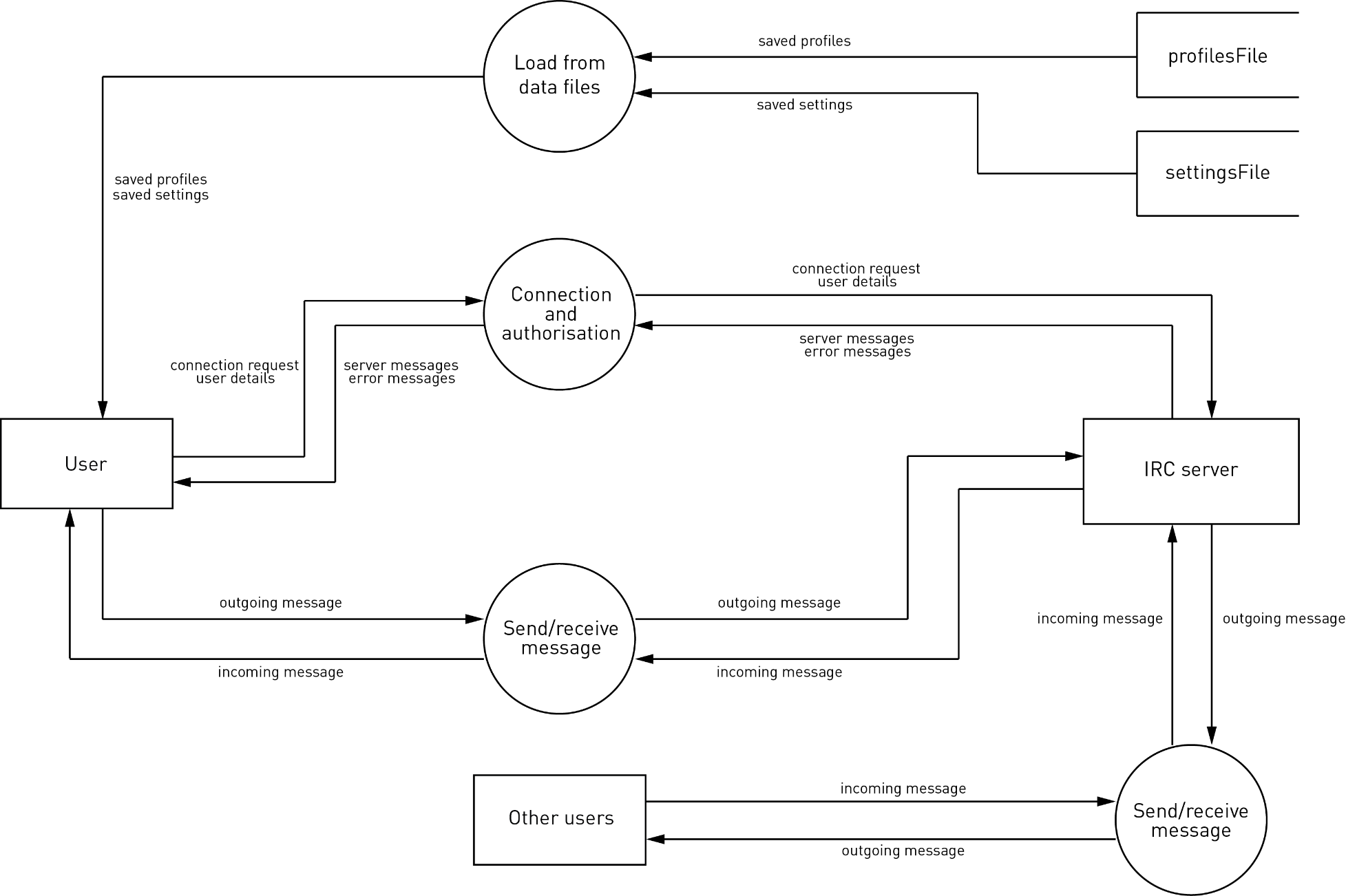
**constant =**

<string>|{<digit>}|({<digit>}.{digit})

# case statement – railroad diagram



# data flow diagram



# data dictionary

Class: Profile

|  |  |  |  |
| --- | --- | --- | --- |
| name | length | type | description |
| profileID | 3 | Integer | Identifies the profile. Unique. |
| profileName | 32 | String | Human-friendly profile name. |
| nickname | 9 | String | Display name. |
| realname | 64 | String | User identifier shown with /whois. |
| username | 64 | String | User identifier shown with /whois. |

Class: Settings

|  |  |  |  |
| --- | --- | --- | --- |
| name | length | type | description |
| defaultProfile | 3 | Integer | Identifier of the default profile. |
| fontSize | 2 | Integer | Base font size for the GUI. |
| maxHistory | 4 | Integer | Maximum number of conversation lines to keep in the displayed chatlog. |

Class: Tab

|  |  |  |  |
| --- | --- | --- | --- |
| name | length | type | description |
| id | 3 | Integer | Identifies the tab. Unique. |
| connection | - | TCPSocket | The connection object for the tab’s corresponding server. |
| channel | 50 | String | Name of the channel connected. |
| threads | - | Hash of threads | A record of three threads: one for receive, one for send, one for GUI. |
| messages | - | Array of strings | Stores a number of the past messages received and sent for this particular connection. |
| queue | - | Stack of strings | Enables cross-thread communication between the send and GUI threads. |
| window | - | Hash | Stores various GUI elements |

part two

# test data and expected output

Connection establishment

|  |  |  |  |
| --- | --- | --- | --- |
| test | input | expected  output | real  output |
| Dead server URL | not.a.server | {fail} | Could not reach server: SocketError |
| Invalid server URL | 47eru65 | {fail} | Could not reach server: SocketError |
| Blank server URL | {blank} | {fail} | Could not reach server: Connection refused |
| Invalid connection port | -1 | {fail} | Could not reach server: SocketError |
| Blank connection port | {blank} | {success}1 | {success}1 |

Processing input

|  |  |  |  |
| --- | --- | --- | --- |
| test | input | expected  output | real  output |
| Sending a normal message | Hello World! | PRIVMSG :Hello World! | PRIVMSG :Hello World! |
| Sending a blank message | {blank} | {nothing} | {nothing} |
| Sending a message with escaped forward-slash | //hello world! | PRIVMSG :/hello world! | PRIVMSG :/hello world! |
| Sending a valid command | /nick NewName | NICK NewName | NICK NewName |
| Sending an invalid command2 | /eat food1 | EAT food1 | EAT food1 |

Notes:

1. If no port is specified, r.IRC assumes port 6667, the default IRC port.
2. Invalid commands are still sent and received by the IRC server; it is up to the server to decide how to prompt the client.

# discussion of errors

An error is any unwanted effect of a program that negatively affects its outcome or operation, often fatally. Errors that could occur are categorised into syntax errors, logic errors, and runtime errors.

Syntax errors are the result of incorrect use of properties of programming languages. Effectively, syntax errors occur when the strict rules and guides that a programming language dictates when using it are not followed. As such, syntax errors are generally the result of carelessness rather than that of a flawed algorithm or concept. In this way, syntax errors can occur absolutely anywhere; however, most syntax errors are detected by a compiler or interpreter and thus are usually very easy to resolve.

Logic errors are those that result from a flawed algorithm. Often, the programmer has not thought out the methods behind how the algorithm works thoroughly, and thus any mistake in how the algorithm operates would result in a dramatically different result. Unlike syntax errors, logic errors are rarely detected by a compiler or interpreter, as the computer has no way of knowing that the programmer’s intent was otherwise. Thus, logic errors are usually very difficult to locate and correct, often involving a length debugging process. Logic errors most commonly occur in algorithms where there is a high degree of complexity involved, as the programmer may have had become confused at some point during development and implemented an erroneous algorithm.

Runtime errors, as suggested by their name, occur during runtime. They are most often either the result of unexpected or non-validated input, resulting in illegal operations such as division by zero. Since they only become obvious at runtime, a compiler is normally unable to pre-empt a runtime error, and neither is an interpreter, before the line where the error occurs. Runtime errors can occur as side effects of logic errors and incorrect algorithms. They most often occur when one operation is dependent upon either user input or the output of another operation. Runtime errors are most commonly detected and corrected through the debugging techniques of use of breakpoints and single line stepping; these techniques allow for the checking of the values held within variables at any one point and thus allow the programmer to isolate the error to a particular section of code very quickly.

# evidence of error checking

There was extensive use of debugging output statements in the development of r.IRC. Much of this can be attributed to the lack of a proper IDE, which meant that manually dumping variable contents and outputting statements at certain points was the only way to check for proper flow of execution. The snippet of source code below demonstrates the use of a debugging output statement as a flag to indicate whether a block of code was successful in its execution.

# this makes the connection to the server

b\_newTab(

ta

)

puts “b\_newTab successful”

g\_newTab(

ta.id

)

puts “b\_newTab successful”

dialog.close

r.IRC’s top-down development process necessitated the use of stubs to check for errors throughout its development. Employing stubs enabled an overall module to be created and thoroughly tested before having to create smaller, less significant modules. The following snippet of source code demonstrates the use of a stub to allow testing of the b\_newTab module without having to implement the b\_send module beforehand.

def b\_newTab(

ta

)

...

$tabs[ta.id].threads['s'] = Thread.new do

while outgoing = $tabs[ta.id].queue.pop

b\_addToHistory($tabs[ta.id].messages,outgoing)

b\_send(ta.id,outgoing)

end

end

...

end

def b\_send(

id,

str

)

# do nothing for now

end

# evidence of testing

Modules in question

The tests to be demonstrated have been performed on the b\_send module, as well the b\_checkIfIrcCommand submodule, the b\_checkIfEscapedMessage submodule, and the b\_executeIrcCommand submodule.

def b\_send(

id,

str

)

s = $tabs[id].connection

if b\_checkIfIrcCommand(str)

str = str[1..str.length]

b\_executeIrcCommand(id,str)

elsif b\_checkIfEscapedMessage(str)

str = str[1..str.length]

c\_privmsg(s,$tabs[id].channel,str)

else

c\_privmsg(s,$tabs[id].channel,str)

end

end

def b\_checkIfIrcCommand(

str

)

isCommand = false

if str.length >= 2

if str[0]=="/" && str[1]!="/"

isCommand = true

end

end

return isCommand

end

def b\_checkIfEscapedMessage(

str

)

isEscapedMessage = false

if str.length >= 2

if str[0]=="/" && str[1]=="/"

isEscapedMessage = true

end

end

return isEscapedMessage

end

def b\_executeIrcCommand(

id,

rawCommand

)

s = $tabs[id].connection

command = rawCommand.split(%r{\s+},2)

# separate the command portion from the arguments

case command[0].upcase

when "ADMIN"

c\_admin(s,command[1])

when "AWAY"

(command[1].nil?)? c\_away(s) : c\_away(s,command[1])

# should add the rest of the commands in

else

s.puts rawCommand

end

end

Peer check

A peer check was performed by Sean Batongbacal, who meticulously examined the b\_send module and its submodules for any current or potential errors. The modules successfully passed the peer review, thus showing their state of completion and error-free state. However, comments were made on the relative third-party unfriendliness of the code, specifically the use of meaning in method and function prefixes that were not immediately obvious.

Desk checks

Several desk checks on the b\_send module and its submodules were performed, to ensure its suitability of handling the different expected inputs, especially that of the value of str. The following desk checks cover the below values of str:

* “Hello World!”
* “//a fwdslash”
* “/nick rircUser”
* “/away”
* Using str = “Hello World!” :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **str** | **isCommand** | **isEscapedMessage** | **command** | **Submodule executed:** |
| “Hello World!” | - | - | - | - |
|  | false | - | - | - |
|  |  | false | - | - |
|  |  |  | - | c\_privmsg |

* Using str = “//a fwdslash” :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **str** | **isCommand** | **isEscapedMessage** | **command** | **Submodule executed:** |
| “//a fwdslash” | - | - | - | - |
|  | false | - | - | - |
|  |  | true | - | - |
| “/a fwdslash” |  |  | - | - |
|  |  |  | - | c\_privmsg |

* Using str = “/nick rircUser” :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **str** | **isCommand** | **isEscapedmessage** | **command** | **Submodule executed:** |
| “/nick rircUser” | - | - | - | - |
|  | true | - | - | - |
| “nick rircUser” |  | - | [“nick”,”rircUser”] | - |
|  |  | - | [“NICK”,”rircUser”] | - |
|  |  | - |  | puts |

* Using str = “/away” :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **str** | **isCommand** | **isEscapedMessage** | **command** | **Submodule executed:** |
| /away” | - | - | - | - |
|  | true | - | - | - |
| “away” |  | - | [“away”,””] | - |
|  |  | - | [“AWAY”,””] | - |
|  |  | - |  | c\_away |

# evidence of debugging

Traces

Analysis of program traces and error dumps proved to be extremely useful in the process of debugging. Since Shoes 4 utilises a JRuby base, program error traces for the GUI component were markedly different to those for the non-GUI component – they contained segments relating to Java’s Standard Widget Toolkit which were unfamiliar to the primarily Ruby developer. However, all program traces provided the line source of the problem and a technical error explanation, which despite their obscurity enabled the eventual solving of the error. Of particular note is the rather long program trace which stated that a thread scheduling error had occurred; Java’s necessary separation of the GUI and non-GUI threads and Ruby’s lack of built-in support for a GUI meant that the program trace was especially esoteric, as provided below.

AfterDo::CallbackError: A Java::OrgEclipseSwt::SWTException: Invalid thread access was raised during an after\_do callback block for method 'displace\_left=' on the instance Shoes::Dimensions with the following arguments: defined in the file /home/john/.rvm/gems/jruby-1.7.13/gems/shoes-4.0.0.pre1/lib/shoes/swt/redrawing\_aspect.rb in line 84. This is the backtrace of the Java::OrgEclipseSwt::SWTException:

java.lang.reflect.Method.invoke(java/lang/reflect/Method.java:606)

Shoes::Swt::App.redraw(/home/john/.rvm/gems/jruby-1.7.13/gems/shoes-4.0.0.pre1/lib/shoes/swt/app.rb:66)

Shoes::Swt::App.redraw(/home/john/.rvm/gems/jruby-1.7.13/gems/shoes-4.0.0.pre1/lib/shoes/swt/app.rb:66)

Shoes::Swt::RedrawingAspect.redraw\_area(/home/john/.rvm/gems/jruby-1.7.13/gems/shoes-4.0.0.pre1/lib/shoes/swt/redrawing\_aspect.rb:104)

Shoes::Swt::RedrawingAspect.redraw\_area(/home/john/.rvm/gems/jruby-1.7.13/gems/shoes-4.0.0.pre1/lib/shoes/swt/redrawing\_aspect.rb:104)

Shoes::Swt::RedrawingAspect.redraw\_element(/home/john/.rvm/gems/jruby-1.7.13/gems/shoes-4.0.0.pre1/lib/shoes/swt/redrawing\_aspect.rb:97)

Shoes::Swt::RedrawingAspect.redraw\_element(/home/john/.rvm/gems/jruby-1.7.13/gems/shoes-4.0.0.pre1/lib/shoes/swt/redrawing\_aspect.rb:97)

RUBY.add\_redraws(/home/john/.rvm/gems/jruby-1.7.13/gems/shoes-4.0.0.pre1/lib/shoes/swt/redrawing\_aspect.rb:85)

org.jruby.RubyProc.call(org/jruby/RubyProc.java:271)

AfterDo.\_after\_do\_execute\_callback(/home/john/.rvm/gems/jruby-1.7.13/gems/after\_do-0.3.1/lib/after\_do.rb:113)

AfterDo.\_after\_do\_execute\_callback(/home/john/.rvm/gems/jruby-1.7.13/gems/after\_do-0.3.1/lib/after\_do.rb:113)

AfterDo.\_after\_do\_execute\_callbacks(/home/john/.rvm/gems/jruby-1.7.13/gems/after\_do-0.3.1/lib/after\_do.rb:107)

AfterDo.\_after\_do\_execute\_callbacks(/home/john/.rvm/gems/jruby-1.7.13/gems/after\_do-0.3.1/lib/after\_do.rb:107)

org.jruby.RubyArray.each(org/jruby/RubyArray.java:1613)

AfterDo.\_after\_do\_execute\_callbacks(/home/john/.rvm/gems/jruby-1.7.13/gems/after\_do-0.3.1/lib/after\_do.rb:106)

AfterDo.\_after\_do\_execute\_callbacks(/home/john/.rvm/gems/jruby-1.7.13/gems/after\_do-0.3.1/lib/after\_do.rb:106)

RUBY.displace\_left=(/home/john/.rvm/gems/jruby-1.7.13/gems/after\_do-0.3.1/lib/after\_do.rb:98)

RUBY.init\_with\_arguments(/home/john/.rvm/gems/jruby-1.7.13/gems/shoes-4.0.0.pre1/lib/shoes/dimensions.rb:131)

RUBY.init\_with\_hash(/home/john/.rvm/gems/jruby-1.7.13/gems/shoes-4.0.0.pre1/lib/shoes/dimensions.rb:122)

RUBY.initialize(/home/john/.rvm/gems/jruby-1.7.13/gems/shoes-4.0.0.pre1/lib/shoes/dimensions.rb:58)

RUBY.initialize(/home/john/.rvm/gems/jruby-1.7.13/gems/shoes-4.0.0.pre1/lib/shoes/dimensions.rb:164)

RUBY.set\_attributes\_from\_options(/home/john/.rvm/gems/jruby-1.7.13/gems/shoes-4.0.0.pre1/lib/shoes/app.rb:301)

RUBY.initialize(/home/john/.rvm/gems/jruby-1.7.13/gems/shoes-4.0.0.pre1/lib/shoes/app.rb:136)

RUBY.initialize(/home/john/.rvm/gems/jruby-1.7.13/gems/shoes-4.0.0.pre1/lib/shoes/app.rb:57)

RUBY.app(/home/john/.rvm/gems/jruby-1.7.13/gems/shoes-4.0.0.pre1/lib/shoes/app.rb:29)

RUBY.g\_newTab(/home/john/Projects/r.irc/common/methods\_rirc.rb:341)

java.lang.Thread.run(java/lang/Thread.java:744)

\_after\_do\_execute\_callback at /home/john/.rvm/gems/jruby-1.7.13/gems/after\_do-0.3.1/lib/after\_do.rb:115

\_after\_do\_execute\_callback at /home/john/.rvm/gems/jruby-1.7.13/gems/after\_do-0.3.1/lib/after\_do.rb:113

\_after\_do\_execute\_callbacks at /home/john/.rvm/gems/jruby-1.7.13/gems/after\_do-0.3.1/lib/after\_do.rb:107

each at org/jruby/RubyArray.java:1613

\_after\_do\_execute\_callbacks at /home/john/.rvm/gems/jruby-1.7.13/gems/after\_do-0.3.1/lib/after\_do.rb:106

displace\_left= at /home/john/.rvm/gems/jruby-1.7.13/gems/after\_do-0.3.1/lib/after\_do.rb:98

init\_with\_arguments at /home/john/.rvm/gems/jruby-1.7.13/gems/shoes-4.0.0.pre1/lib/shoes/dimensions.rb:131

init\_with\_hash at /home/john/.rvm/gems/jruby-1.7.13/gems/shoes-4.0.0.pre1/lib/shoes/dimensions.rb:122

initialize at /home/john/.rvm/gems/jruby-1.7.13/gems/shoes-4.0.0.pre1/lib/shoes/dimensions.rb:58

initialize at /home/john/.rvm/gems/jruby-1.7.13/gems/shoes-4.0.0.pre1/lib/shoes/dimensions.rb:164

set\_attributes\_from\_options at /home/john/.rvm/gems/jruby-1.7.13/gems/shoes-4.0.0.pre1/lib/shoes/app.rb:301

initialize at /home/john/.rvm/gems/jruby-1.7.13/gems/shoes-4.0.0.pre1/lib/shoes/app.rb:136

initialize at /home/john/.rvm/gems/jruby-1.7.13/gems/shoes-4.0.0.pre1/lib/shoes/app.rb:57

app at /home/john/.rvm/gems/jruby-1.7.13/gems/shoes-4.0.0.pre1/lib/shoes/app.rb:29

g\_newTab at /home/john/Projects/r.irc/common/methods\_rirc.rb:341

Breakpoints

Due to a combination of the lack of an IDE as well as technical issues in using the official Ruby debugger with JRuby 1.7.13, regular and frequent use of breakpoints in the debugging process was limited. Rather, an alternative to breakpoints, to manually dump the values of key variables and pause execution of the program within the source code itself, was employed, to high success. Following is a snippet of code that demonstrates this in practice.

newTabConfirmButton = button “go” do

channelArray = []

channelArray << channelLine.text

# breakpoint, dumping contents of channelArray

puts channelArray

gets

# end breakpoint

...

end

Single line stepping

Similarly with breakpoints, the lack of an IDE and several technical issues limited the use of the Ruby debugger in the development process of r.IRC. Success was not achieved when attempting to single line step using the Ruby debugger as the several thousand lines of code loaded by the Shoes toolkit at execution contained exceptions that stopped strict execution and were simply too numerous to single line step through manually. Instead, a similar makeshift system was employed, using the gets function as a pause point in execution that allowed for each individual non-debug line to run.

def f\_getsLine(

file

)

line = file.gets.chomp

gets # step

while line.strip.nil? == true

|| line == “\n”

|| line[0] == ‘#’

|| line == “”

line = file.gets.chomp

gets # step

end

gets # step

return line

end

part three

# developer documentation

Developer documentation provided includes a log book, previously shown data flow diagrams and data dictionaries, as well as internal documentation in the form of intrinsic documentation.

Internal documentation within the source code is evidenced by the abundance of code comments as well as self-describing code. The former aims to explain the purpose of certain code blocks and functions, as well as to minimise any confusion created by complex algorithms. The latter aims to reduce the complexity of the code to any other parties and thus reduce the necessity of code comments, through the use of meaningful identifiers and consistent indentation of code.

def initialize(

connection,

channel

)

@id = @@tabId

@connection = connection

@channel = channel

@threads = Hash.new

# three threads per tab

# `r` => receive

# `s` => send

# `g` => gui

@messages = []

@queue = Queue.new

# queue is for outgoing messages

# enables cross thread data sharing

@window = Hash.new

# a way to store the gui elements to allow

# for manipulation by back end functions

@@tabId += 1

end

The snippet of source code provided above demonstrates the usefulness of code comments in documenting the purpose of certain elements in the Tab class. Consistent indenting of the initial values of the class elements allows for clearer code, and the use of descriptive element names removes the need for explanatory comments.

# readability of code

A large effort has been placed into attempting to make the source code as readable and clear as possible. Internal documentation in the form of code comments and intrinsic documentation make it simple for any third party to obtain a grasp on how the application works and how the different modules interact.

A multitude of custom classes have been defined to be used in r.IRC. These include Profile, Settings, and Tab, and all these classes have clear and meaningful method and element names to aid in understanding the purpose of the code; the attributes and class methods of Tab are listed below.

class Tab

attr\_reader :id, :connection, :channel

attr\_accessor :threads, :messages, :queue, :window

def initialize; end

def self.create; end

def self.count; end

end

A consistent style guide has been followed throughout the source code of r.IRC, including the use of camelCase and preference for do/end over braces in containing code blocks. Functions and methods have been classified into several main groups, and each of these has its unique prefix to show approximately what it is responsible for in the entire program. Examples include:

* back end – b\_send
* GUI – g\_newTab
* file – f\_getsLine
* command – c\_user

Furthermore, the segregation of different parts of the application is manifested in the separation of these functions into appropriate files containing related methods. For example, methods\_io.rb contains the bulk of the functions dealing with input and output in relation to the file system, and classes.rb holds all the declarations for the classes used in r.IRC. All these files are loaded alongside required system classes, providing for a cleaner and simpler main source file, increasing readability and accessibility.

# user documentation

A range of user-aimed documentation is to be provided with the completion of r.IRC’s development process, including installation documentation as well as user technical documentation.

Installation documentation

Installing r.IRC is a relatively simple process. A document with step-by-step instructions will be provided to the user in both plain text and PDF format, guiding them through the process of installing the prerequisites of JRuby and Shoes, both which are included with the r.IRC package.

User technical documentation

User technical documentation will consist of two main sections:

* Basic documentation
* Advanced documentation

This segregation is due to the wide range of technical expertise and background possessed by IRC users. Those who have a great amount of knowledge on the IRC protocol and have major experience with it may find the basic documentation shallow and uninformative, whereas new users will find the advanced documentation overly complex and unnecessarily detailed, as well as being irrelevant to their direct interests.

The basic documentation will include instruction on how to operate the core functionality of r.IRC, such as setting up profiles, connecting to servers and joining channels, and the most basic and necessary IRC commands. On the other hand, the advanced documentation will detail how r.IRC performs its functionality and how power users can manipulate r.IRC to operate to their liking, including details on all official IRC commands as well as several unofficial but popular ones, as well as instructions on how to manually edit settings in r.IRC’s configuration files to modify how the application looks and runs.