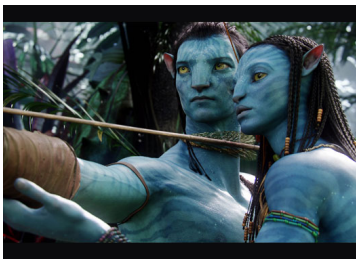


# WRITING PLUGINS WITH RABBITMQ

Noah Gift and Michael Vierling

# Noah's Background



- Film
- Commercial Data Center Automation Software
- Los Angeles, Atlanta, New Zealand, San Francisco
- I like programming in ANY language that is innovative

# How & Why



- ▣ Why AMQP?
- ▣ Why Erlang?
- ▣ Why OTP?
- ▣ Why RabbitMQ?
- ▣ How – Machinery

# Why AMQP

**Microsoft**

**Goldman  
Sachs**



**vmware**

- Industry Message Standard
- Cisco, Microsoft, Red Hat, VMWare, JPMorgan, Chase, Goldman Sachs
- Binary wire protocol

# Why AMQP: Continued



- Key component in Distributed Architecture
- Enables Multi-OS, Multi-Language
- Reliable, Transactional, Flow-Control, Fault-tolerant

# Downsides of AMQP



- ❑ AMQP 1.0 is almost final
- ❑ Support resources
- ❑ Upgrade glitches
- ❑ Monitoring issues
- ❑ Investment in the future

# Why Erlang?



- ▣ Functional Language
- ▣ Immutable Data
- ▣ Scalable
- ▣ Reliable, Fault-tolerant

# Why RabbitMQ



- Built with Erlang
- 10K messages per second – transient mode
- 3-5K messages per second – persistent mode

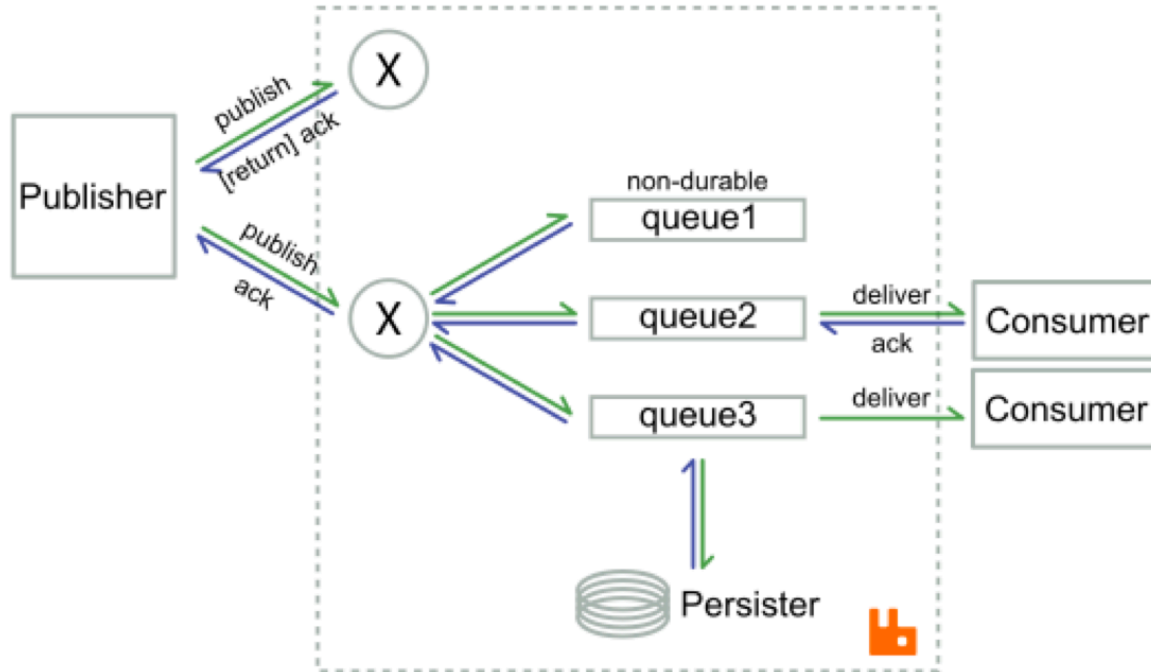


# Why RabbitMQ

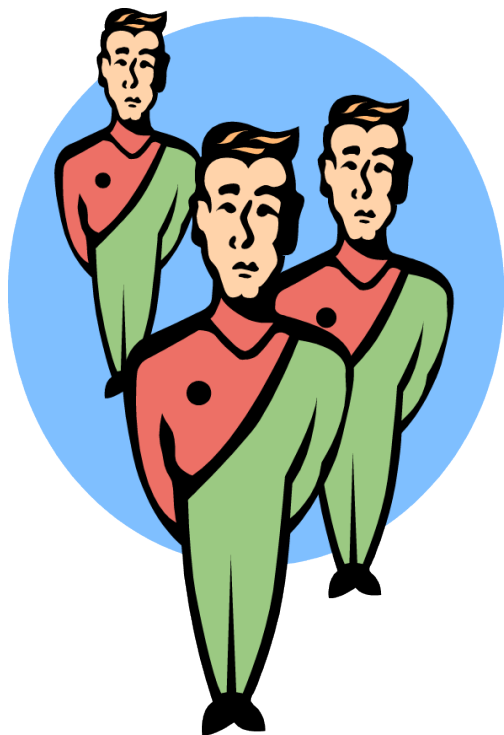


- Transactions
- Open Source
- Maintained by VMWare

# Broker Infrastructure



# Broker Infrastructure



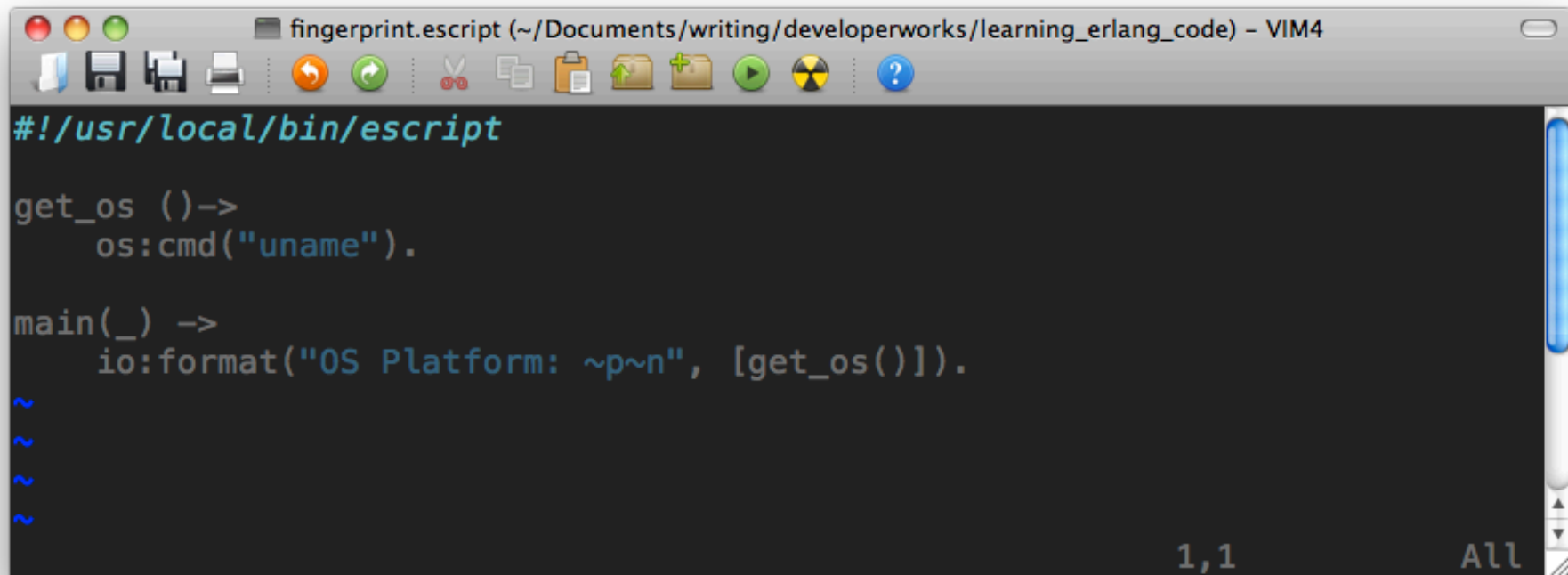
- ▣ Repeatabl Install
- ▣ Reliable storage
- ▣ Redundant hardware

# Broker Infrastructure: Continued



- Cluster of nodes
- Disaster Recovery
- Logging (such as Splunk) and monitoring
- Operations & Support

# Broker Infrastructure: Use Escript



The image shows a VIM4 editor window titled "fingerprint.escript (~/.Documents/writing/developerworks/learning\_erlang\_code) - VIM4". The window contains the following Erlang code:

```
#!/usr/local/bin/escript

get_os ()->
    os:cmd("uname").

main(_) ->
    io:format("OS Platform: ~p~n", [get_os()]).
~
~
~
~
```

The status bar at the bottom right shows "1,1" and "All".

# Broker Infrastructure Future?



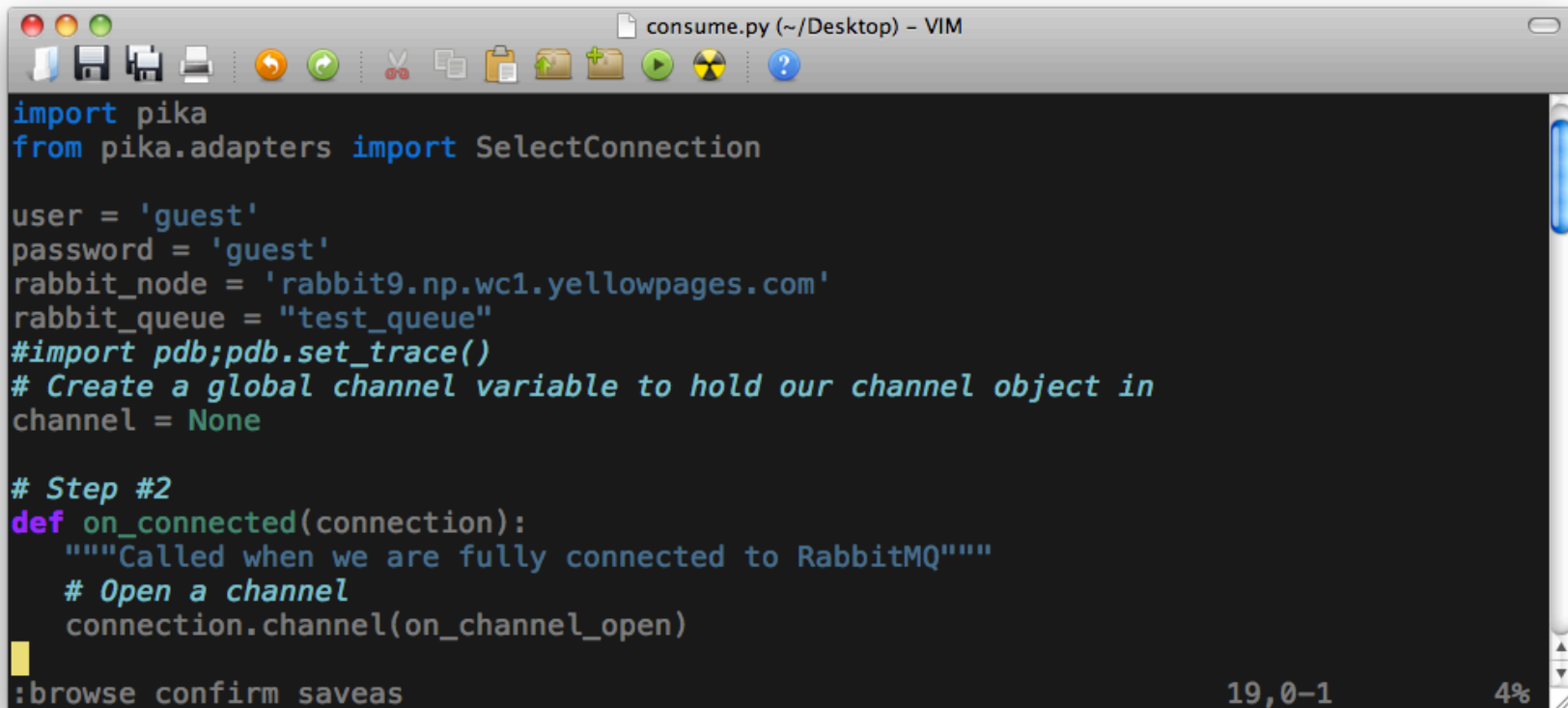
- ❑ Integrate Rebar
- ❑ Continuous Integration Testing
- ❑ QuickCheck

# RabbitMQ Clients



- ▣ Any AMQP Clients (Apache)
- ▣ Ruby
- ▣ Python
- ▣ C#
- ▣ F# (Wrap the C# dll)

# Consume.py Part: A



```
import pika
from pika.adapters import SelectConnection

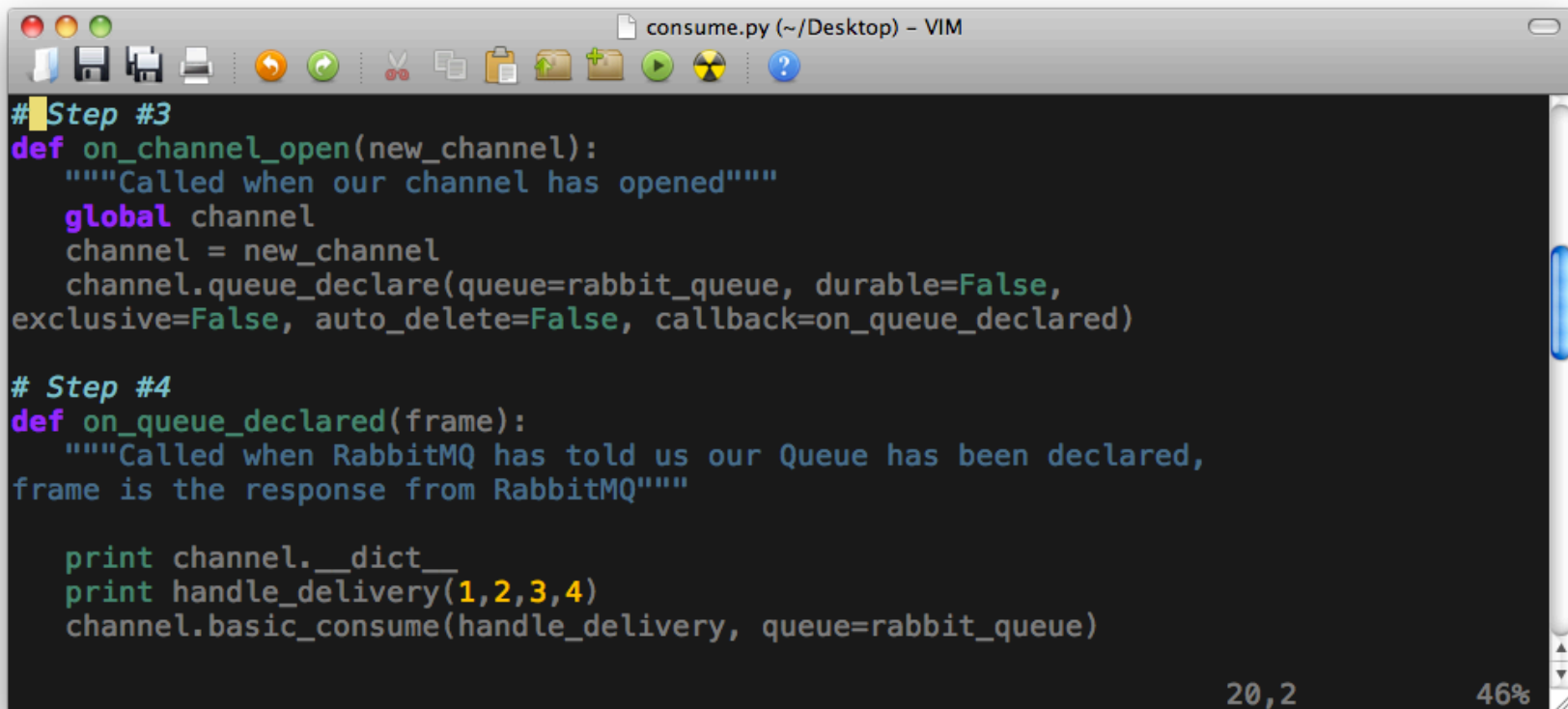
user = 'guest'
password = 'guest'
rabbit_node = 'rabbit9.np.wc1.yellowpages.com'
rabbit_queue = "test_queue"
#import pdb;pdb.set_trace()
# Create a global channel variable to hold our channel object in
channel = None

# Step #2
def on_connected(connection):
    """Called when we are fully connected to RabbitMQ"""
    # Open a channel
    connection.channel(on_channel_open)
```

:browse confirm saves 19,0-1 4%



# Consume.py Part: B



The image shows a VIM editor window titled "consume.py (~/Desktop) - VIM". The code is written in Python and includes comments in English. The code defines two functions: `on_channel_open` and `on_queue_declared`. The `on_channel_open` function declares a queue named `rabbit_queue` with `durable=False`, `exclusive=False`, and `auto_delete=False`. The `on_queue_declared` function prints the channel's dictionary and then calls `channel.basic_consume` with `handle_delivery` as the callback and `rabbit_queue` as the queue name. The code is displayed on a dark background with syntax highlighting. The VIM status bar at the bottom right shows the line and column number "20,2" and the zoom level "46%".

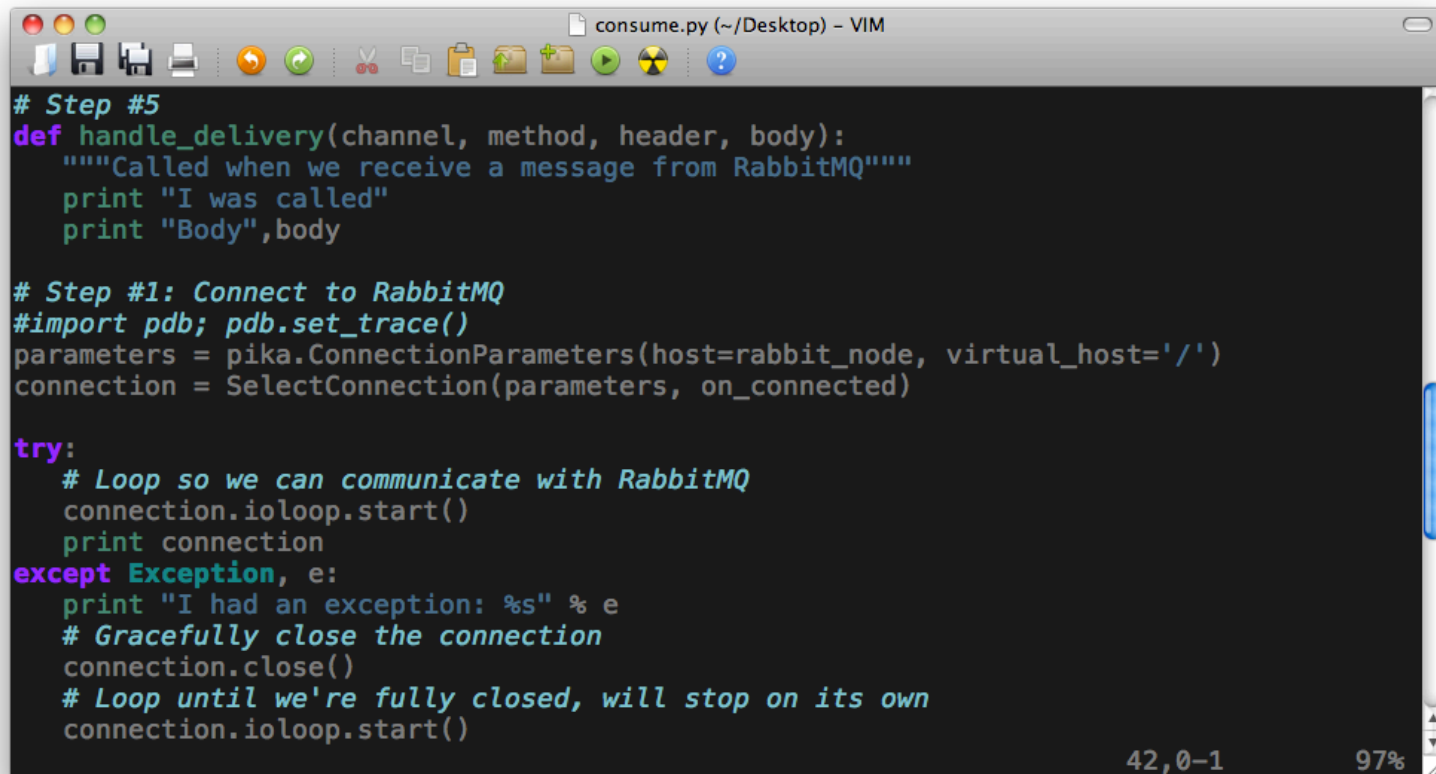
```
# Step #3
def on_channel_open(new_channel):
    """Called when our channel has opened"""
    global channel
    channel = new_channel
    channel.queue_declare(queue=rabbit_queue, durable=False,
exclusive=False, auto_delete=False, callback=on_queue_declared)

# Step #4
def on_queue_declared(frame):
    """Called when RabbitMQ has told us our Queue has been declared,
frame is the response from RabbitMQ"""

    print channel.__dict__
    print handle_delivery(1,2,3,4)
    channel.basic_consume(handle_delivery, queue=rabbit_queue)
```

20,2 46%

# Consume.py Part: C



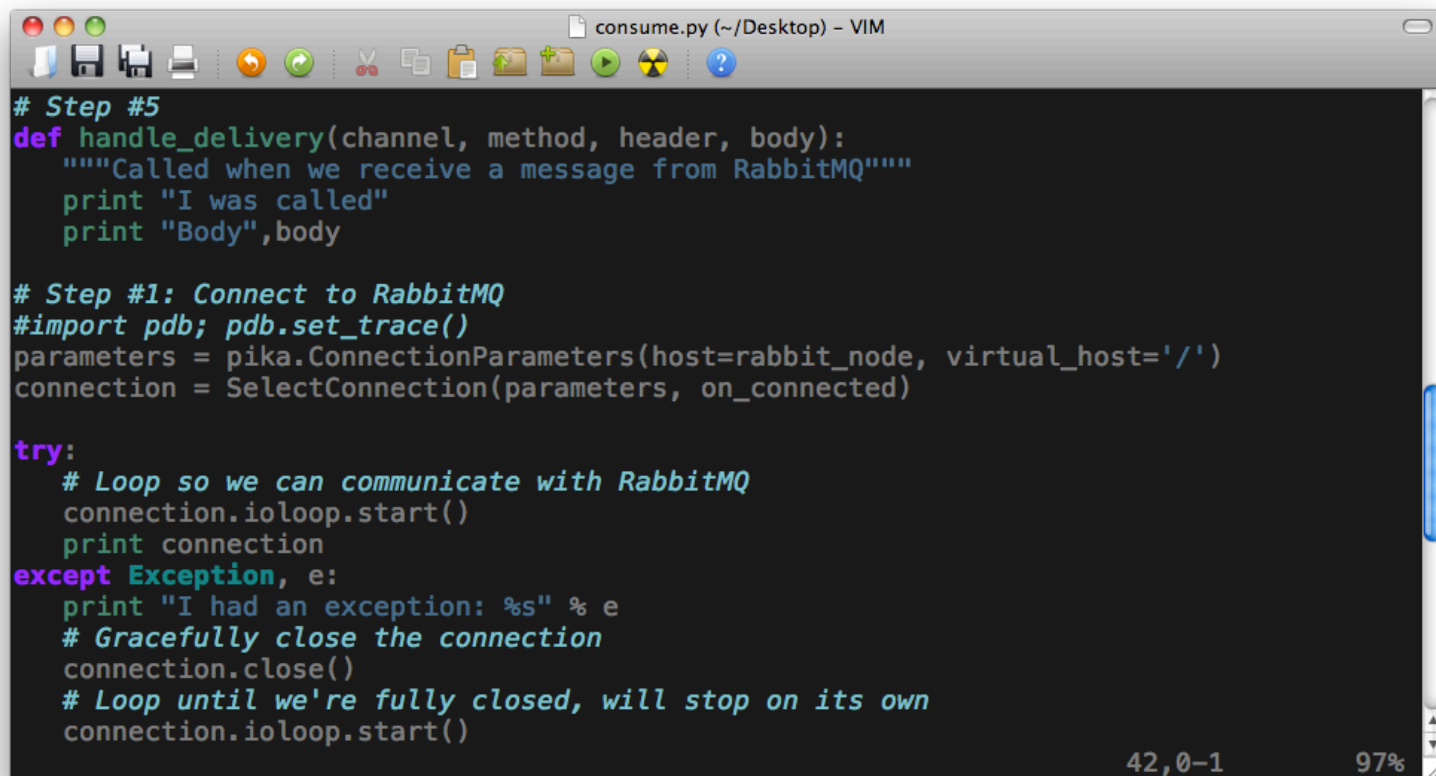
```
# Step #5
def handle_delivery(channel, method, header, body):
    """Called when we receive a message from RabbitMQ"""
    print "I was called"
    print "Body",body

# Step #1: Connect to RabbitMQ
import pdb; pdb.set_trace()
parameters = pika.ConnectionParameters(host=rabbit_node, virtual_host='/')
connection = SelectConnection(parameters, on_connected)

try:
    # Loop so we can communicate with RabbitMQ
    connection.ioloop.start()
    print connection
except Exception, e:
    print "I had an exception: %s" % e
    # Gracefully close the connection
    connection.close()
    # Loop until we're fully closed, will stop on its own
    connection.ioloop.start()
```

42,0-1 97%

# Send.py Part: A



The screenshot shows a VIM editor window titled "consume.py (~/Desktop) - VIM". The code is written in Python and includes comments in a light blue color. The code defines a `handle_delivery` function, connects to RabbitMQ using `pika`, and enters a loop to process messages. It also includes a `try` block to handle exceptions and a `finally` block to gracefully close the connection.

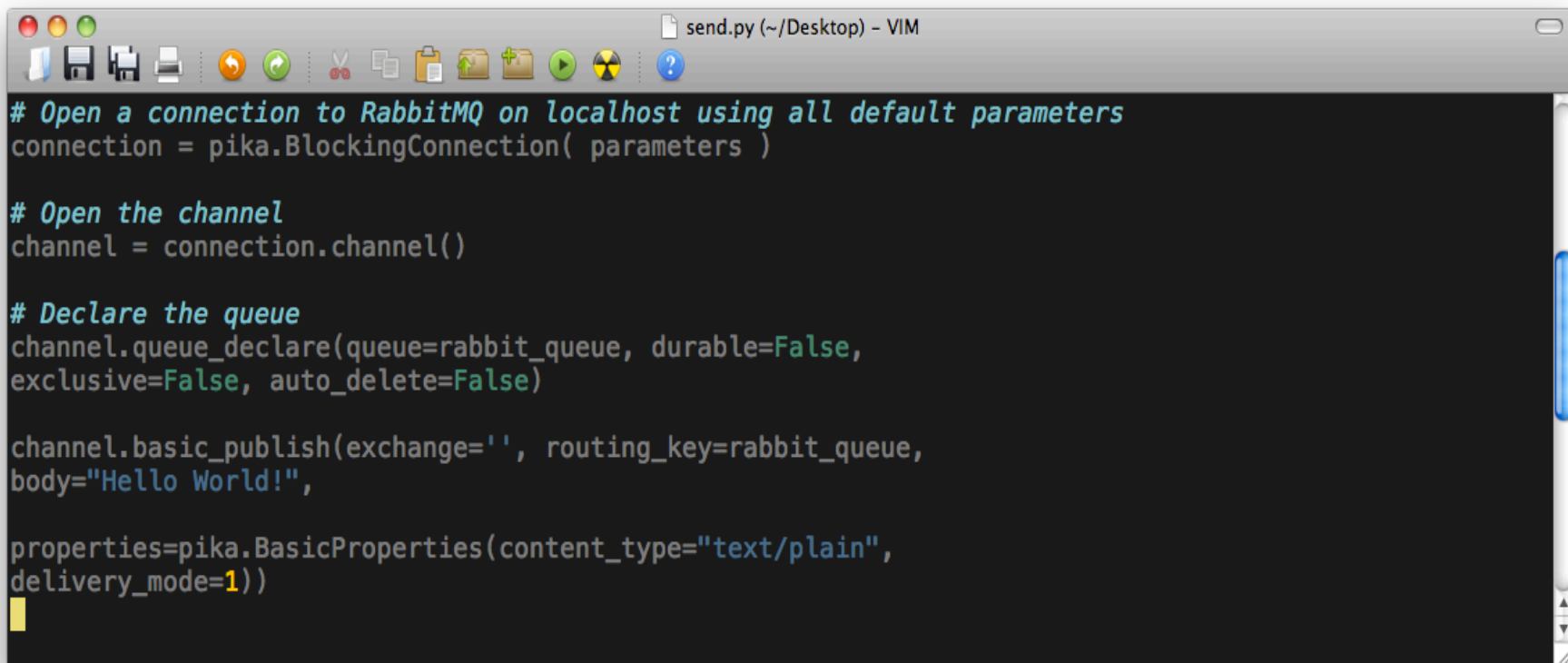
```
# Step #5
def handle_delivery(channel, method, header, body):
    """Called when we receive a message from RabbitMQ"""
    print "I was called"
    print "Body",body

# Step #1: Connect to RabbitMQ
import pdb; pdb.set_trace()
parameters = pika.ConnectionParameters(host=rabbit_node, virtual_host='/')
connection = SelectConnection(parameters, on_connected)

try:
    # Loop so we can communicate with RabbitMQ
    connection.ioloop.start()
    print connection
except Exception, e:
    print "I had an exception: %s" % e
    # Gracefully close the connection
    connection.close()
    # Loop until we're fully closed, will stop on its own
    connection.ioloop.start()
```

42,0-1 97%

# Send.py Part: B



The screenshot shows a VIM editor window titled "send.py (~/Desktop) - VIM". The editor contains the following Python code:

```
# Open a connection to RabbitMQ on localhost using all default parameters
connection = pika.BlockingConnection( parameters )

# Open the channel
channel = connection.channel()

# Declare the queue
channel.queue_declare(queue=rabbit_queue, durable=False,
exclusive=False, auto_delete=False)

channel.basic_publish(exchange='', routing_key=rabbit_queue,
body="Hello World!",

properties=pika.BasicProperties(content_type="text/plain",
delivery_mode=1))
```

The code is written in a dark-themed editor with syntax highlighting. The first line is a comment. The second line creates a blocking connection to RabbitMQ. The third line opens a channel. The fourth line declares a queue. The fifth line publishes a message to the queue. The sixth line sets the message properties.

# Writing RabbitMQ Plugin: WHY



- Access internal RabbitMQ functionality
- Running in same Erlang VM as broker can increase performance
- Leverage your existing infrastructure

# Writing RabbitMQ Plugin: WHY NOT



- ❑ Developers don't know Erlang
- ❑ Poorly written plugin could crash broker
- ❑ You could lock yourself to internal API

# Stable RabbitMQ Plugin



- ❑ Rabbitmq-management  
(replaced Alice)
- ❑ Rabbitmq-shovel
- ❑ Rabbitmq-stomp
- ❑ Rabbitmq-erlang-client

# Experimental RabbitMQ Plugin Ideas



- ❑ Rabbitmq-auth-backend-ldap
- ❑ Rabbitmq-auth-mechanism-ssl
- ❑ Rabbitmq-jsonrpc-channel
- ❑ Rabbitmq-xmpp



# Crazy RabbitMQ Plugin Ideas



- ❑ Distributed Data Structure
- ❑ Artificial Intelligence
- ❑ Data Mining
- ❑ Create a protocol on top
- ❑ Event Processing

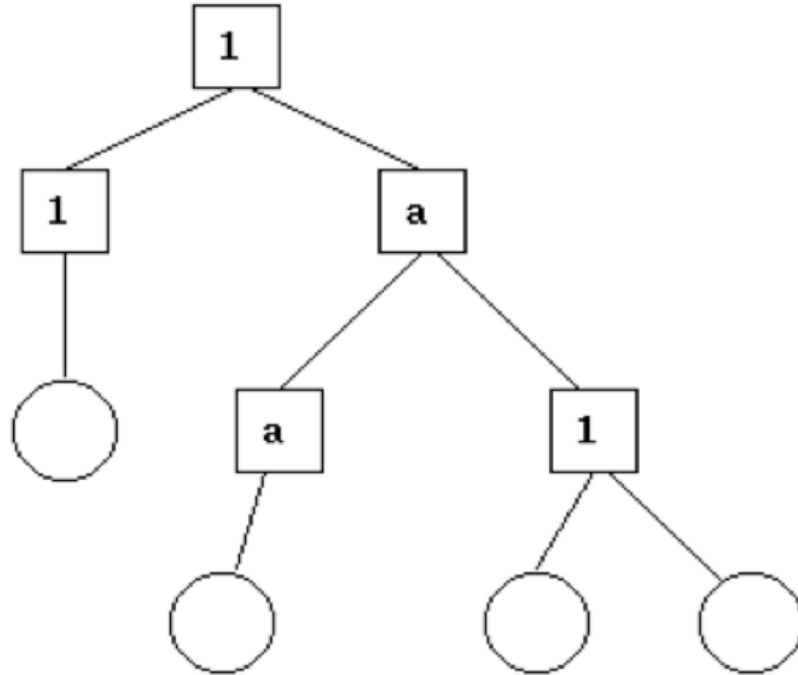
# Michael Background



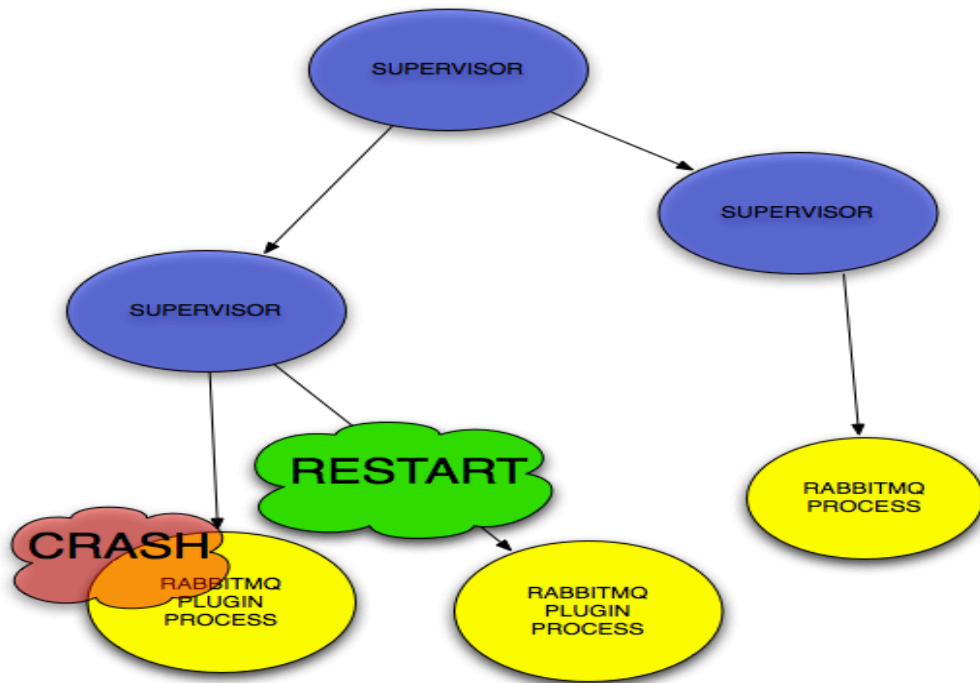
- ▣ Apple 5 Years
- ▣ Ascend/Lucent 4 Years
- ▣ Various Startups



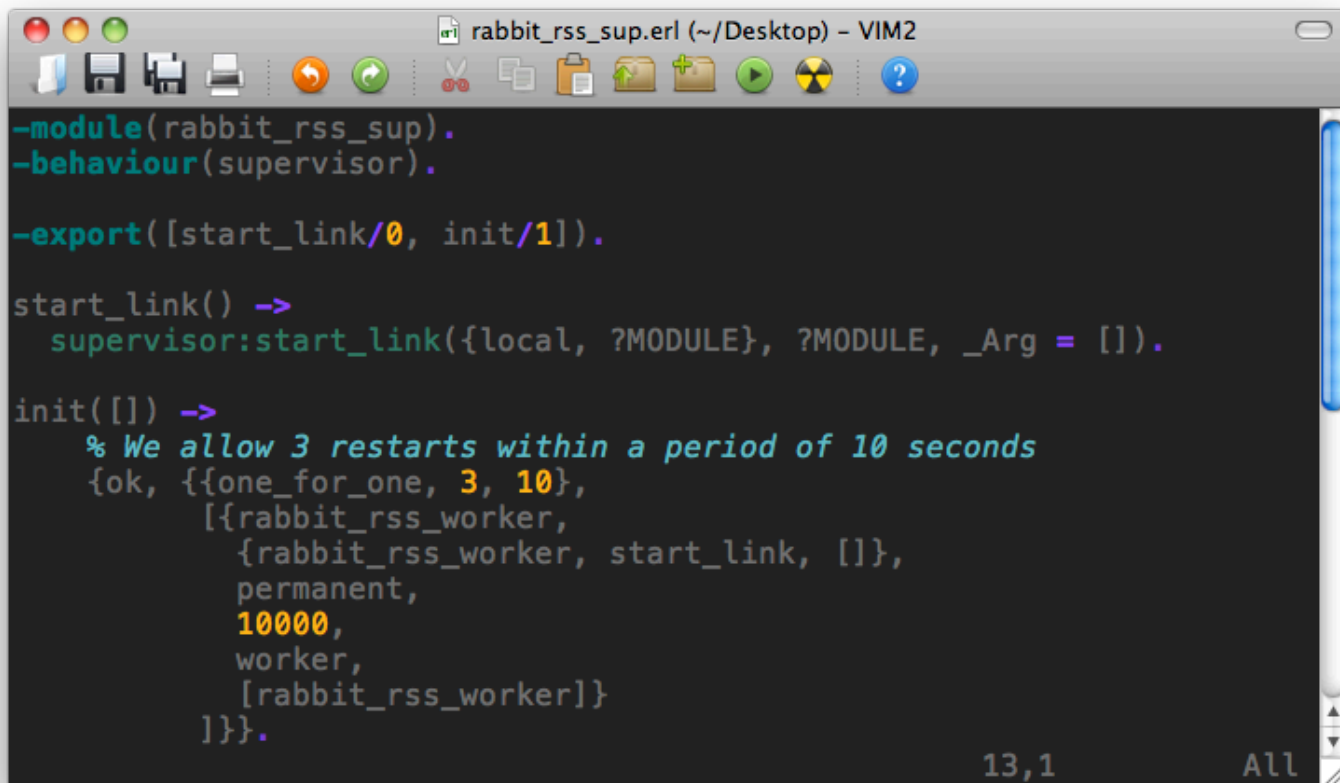
# Supervision Tree



# Plugin Supervisor



# Rabbit RSS Supervisor

A screenshot of a VIM2 editor window titled 'rabbit\_rss\_sup.erl (~/Desktop) - VIM2'. The window displays Erlang code for a supervisor. The code includes module and behaviour declarations, export statements for start\_link and init, and function definitions for start\_link and init. The init function includes a comment about restarts and a list of workers. The status bar at the bottom shows '13,1' and 'All'.

```
rabbit_rss_sup.erl (~/Desktop) - VIM2
- module(rabbit_rss_sup).
- behaviour(supervisor).

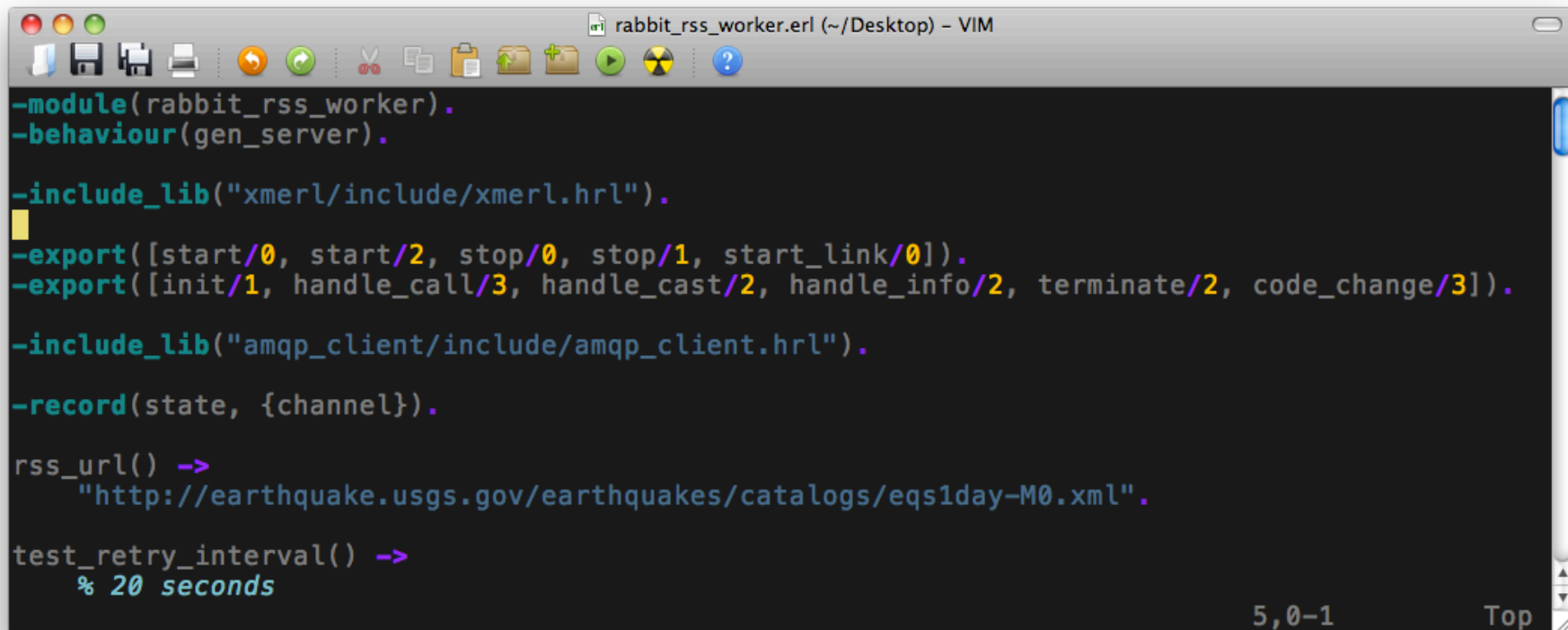
- export([start_link/0, init/1]).

start_link() ->
    supervisor:start_link({local, ?MODULE}, ?MODULE, _Arg = []).

init([]) ->
    % We allow 3 restarts within a period of 10 seconds
    {ok, {{one_for_one, 3, 10},
        [{rabbit_rss_worker,
          {rabbit_rss_worker, start_link, []},
          permanent,
          10000,
          worker,
          [rabbit_rss_worker]}]
        }}}.
```

13,1 All

# Rabbit RSS Worker: Slide A



```
rabbit_rss_worker.erl (~/Desktop) - VIM
- module(rabbit_rss_worker).
- behaviour(gen_server).

- include_lib("xmerl/include/xmerl.hrl").
- export([start/0, start/2, stop/0, stop/1, start_link/0]).
- export([init/1, handle_call/3, handle_cast/2, handle_info/2, terminate/2, code_change/3]).

- include_lib("amqp_client/include/amqp_client.hrl").

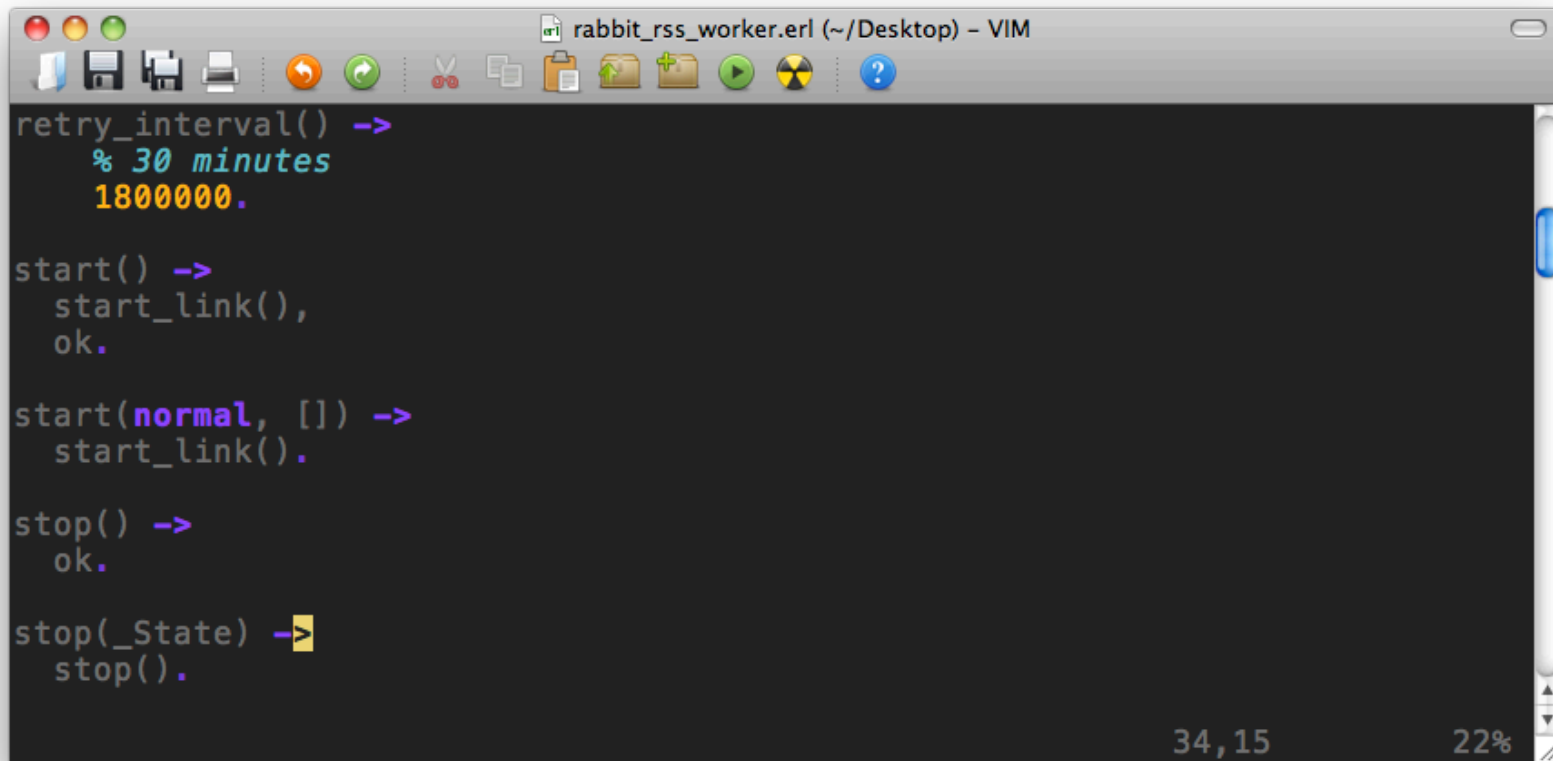
- record(state, {channel}).

rss_url() ->
    "http://earthquake.usgs.gov/earthquakes/catalogs/eqs1day-M0.xml".

test_retry_interval() ->
    % 20 seconds

5,0-1 Top
```

# Rabbit RSS Worker: Slide B



The screenshot shows a VIM editor window titled "rabbit\_rss\_worker.erl (~/.Desktop) - VIM". The editor contains the following Erlang code:

```
retry_interval() ->
    % 30 minutes
    1800000.

start() ->
    start_link(),
    ok.

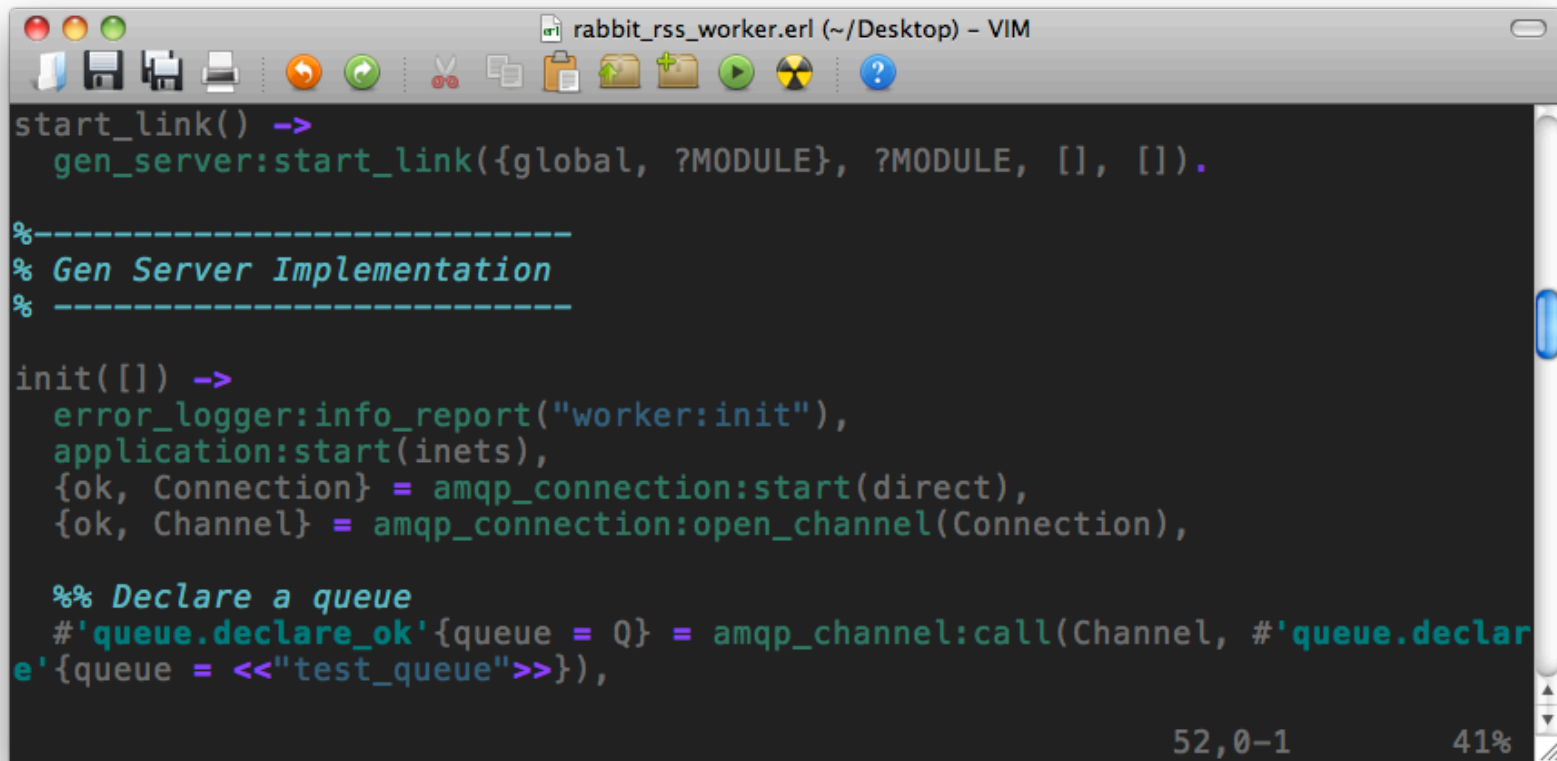
start(normal, []) ->
    start_link().

stop() ->
    ok.

stop(_State) ->
    stop().
```

The bottom right corner of the editor shows the cursor position "34, 15" and the zoom level "22%".

# Rabbit RSS Worker: Slide C



```
rabbit_rss_worker.erl (~/.Desktop) - VIM

start_link() ->
    gen_server:start_link({global, ?MODULE}, ?MODULE, [], []).

%-----
% Gen Server Implementation
% -----

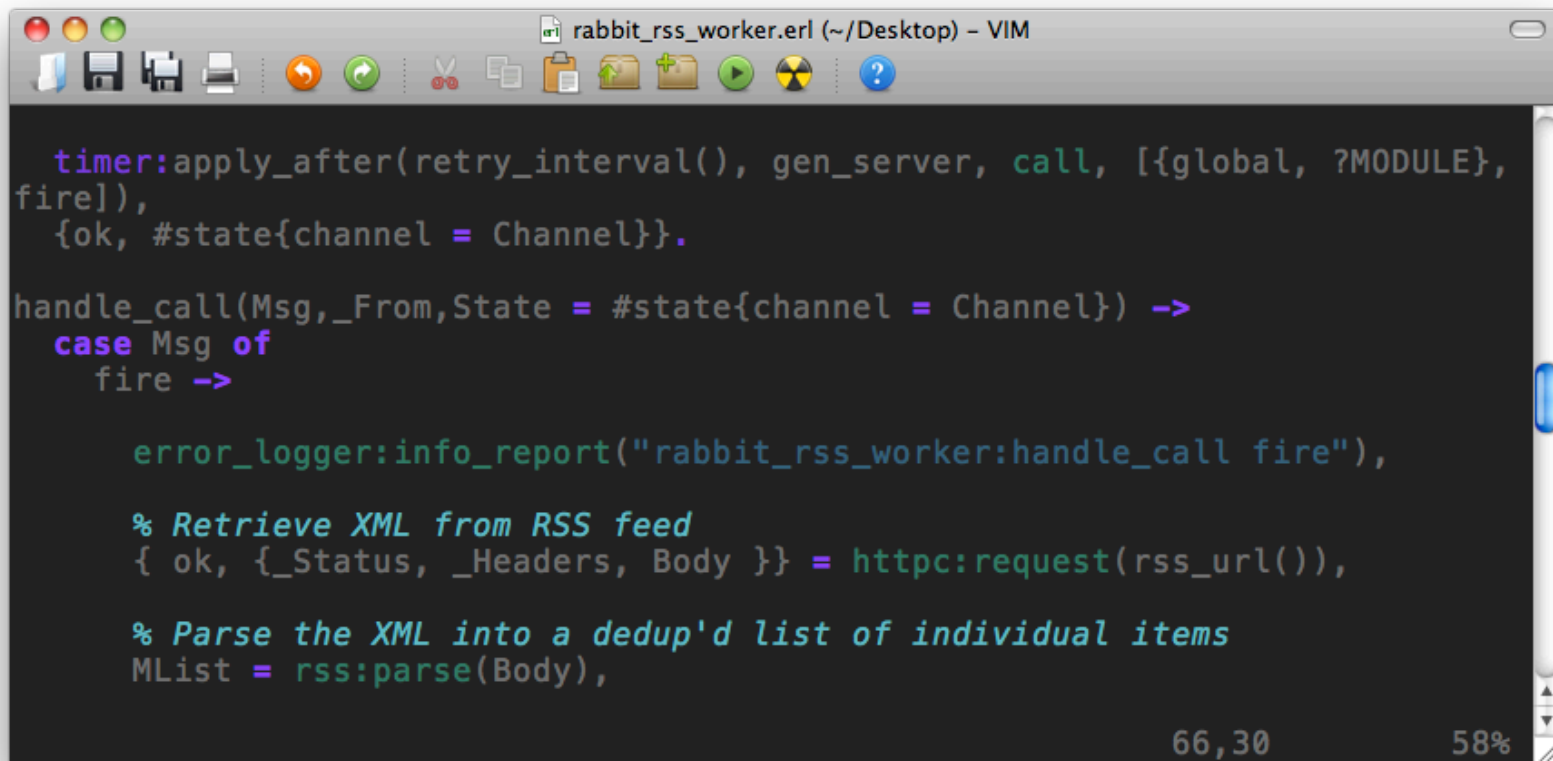
init([]) ->
    error_logger:info_report("worker:init"),
    application:start(inets),
    {ok, Connection} = amqp_connection:start(direct),
    {ok, Channel} = amqp_connection:open_channel(Connection),

    %% Declare a queue
    #'queue.declare_ok'{queue = Q} = amqp_channel:call(Channel, #'queue.declar
e'{queue = <<"test_queue">>}),

52,0-1 41%
```



# Rabbit RSS Worker: Slide D



The image shows a VIM editor window titled "rabbit\_rss\_worker.erl (~/.Desktop) - VIM". The code is written in Erlang and includes a timer-based retry mechanism, a handle\_call function, and logic for retrieving and parsing RSS feed data. The code is as follows:

```
timer:apply_after(retry_interval(), gen_server, call, [{global, ?MODULE},
fire]),
{ok, #state{channel = Channel}}.

handle_call(Msg, _From, State = #state{channel = Channel}) ->
case Msg of
fire ->

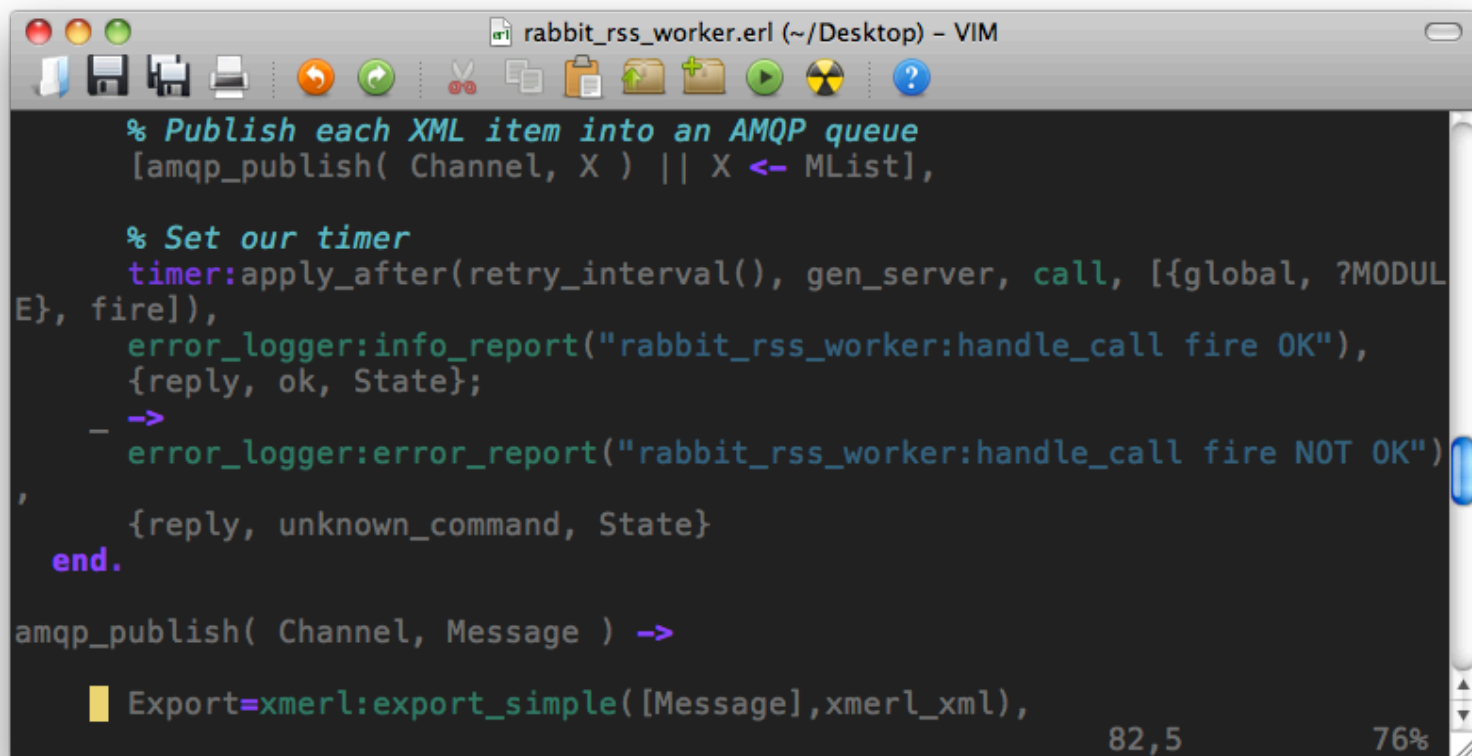
    error_logger:info_report("rabbit_rss_worker:handle_call fire"),

    % Retrieve XML from RSS feed
    { ok, { _Status, _Headers, Body }} = httpc:request(rss_url()),

    % Parse the XML into a dedup'd list of individual items
    MList = rss:parse(Body),
```

The bottom right corner of the editor window displays "66,30" and "58%".

# Rabbit RSS Worker: Slide E



```

% Publish each XML item into an AMQP queue
[amqp_publish( Channel, X ) || X <- MList],

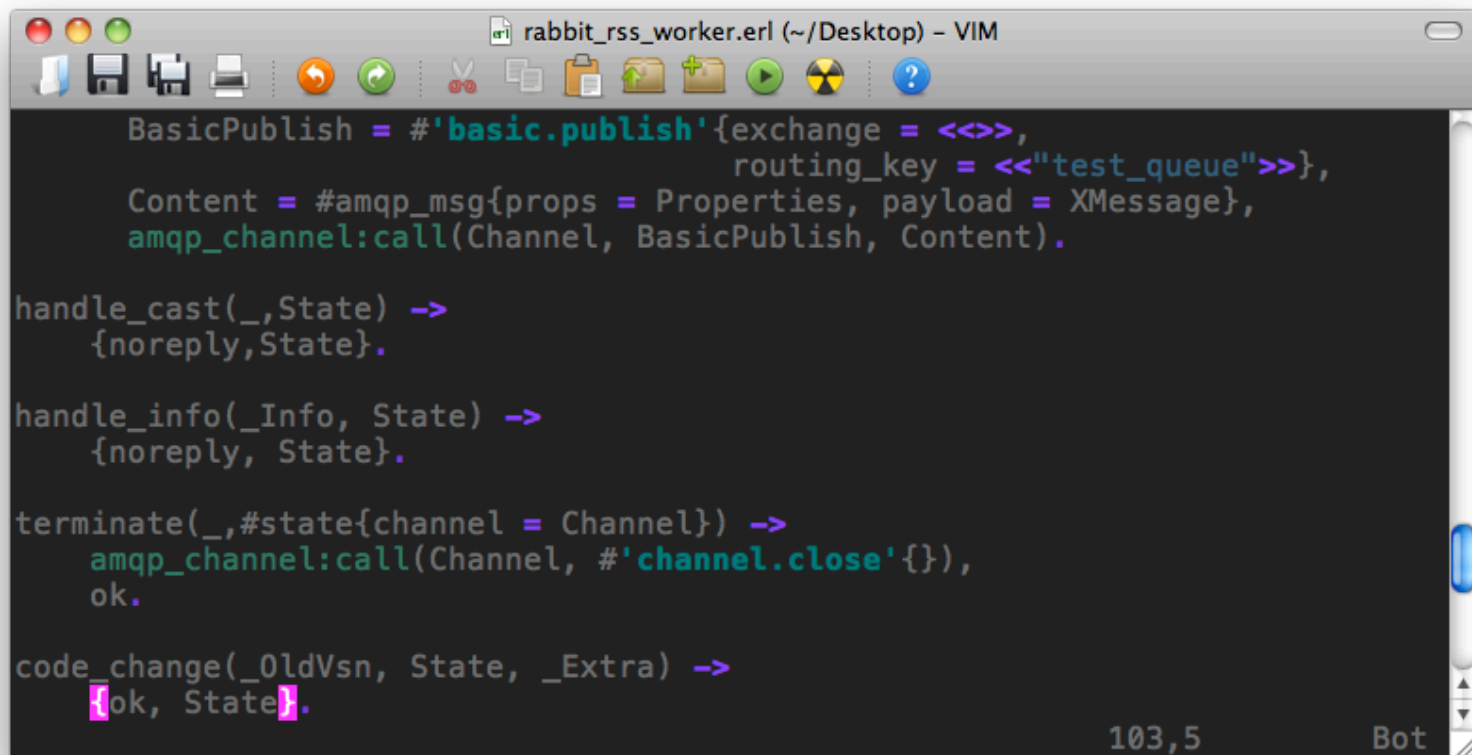
% Set our timer
timer:apply_after(retry_interval(), gen_server, call, [{global, ?MODULE,
fire}, fire]),
error_logger:info_report("rabbit_rss_worker:handle_call fire OK"),
{reply, ok, State};
- ->
error_logger:error_report("rabbit_rss_worker:handle_call fire NOT OK")
,
{reply, unknown_command, State}
end.

amqp_publish( Channel, Message ) ->
    Export=xmerl:export_simple( [Message], xmerl_xml),

```

82,5 76%

# Rabbit RSS Worker: Slide F



```
BasicPublish = #'basic.publish'{exchange = <<>>,
                                routing_key = <<"test_queue">>},
Content = #amqp_msg{props = Properties, payload = XMessage},
amqp_channel:call(Channel, BasicPublish, Content).

handle_cast(_,State) ->
    {noreply,State}.

handle_info(_Info, State) ->
    {noreply, State}.

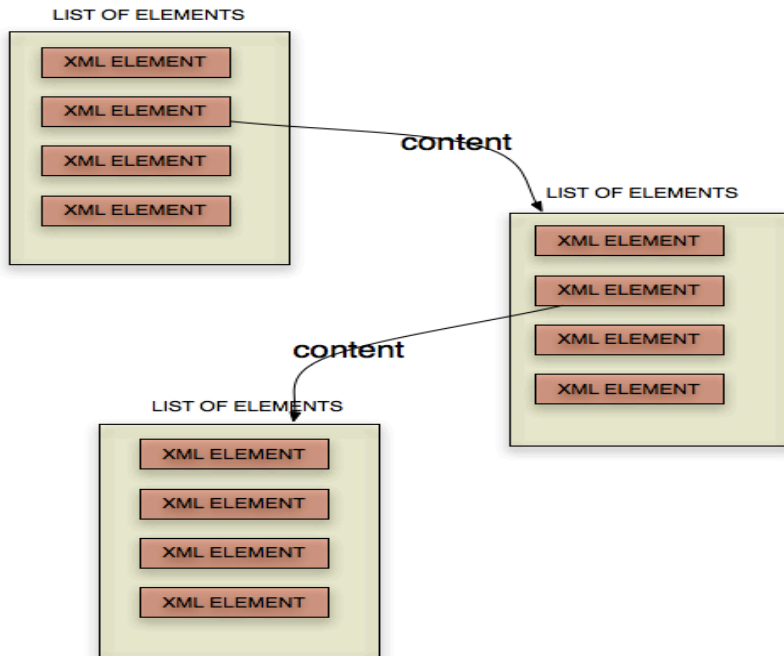
terminate(_,#state{channel = Channel}) ->
    amqp_channel:call(Channel, #'channel.close'{}),
    ok.

code_change(_OldVsn, State, _Extra) ->
    {ok, State}.
```

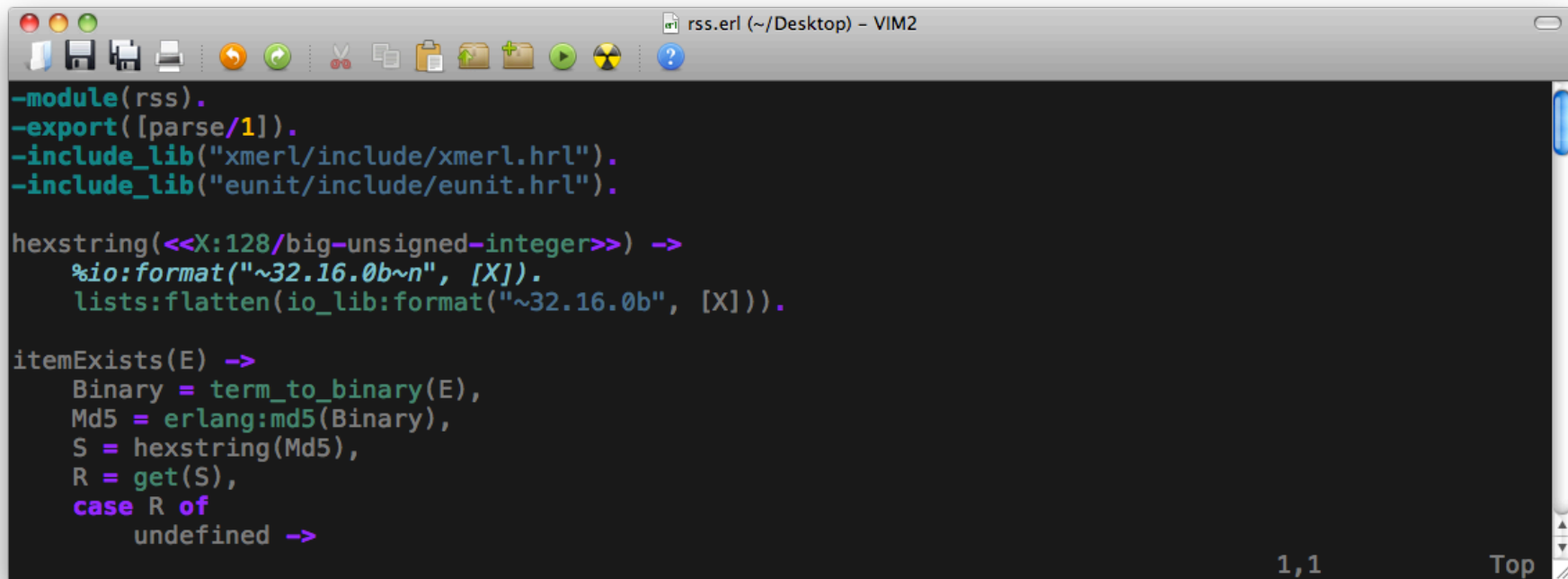
103,5 Bot

# XML Parsing

## RECURSIVE XML STRUCTURE RETURNED BY XMERL LIBRARY



# RSS.ERL: Slide A

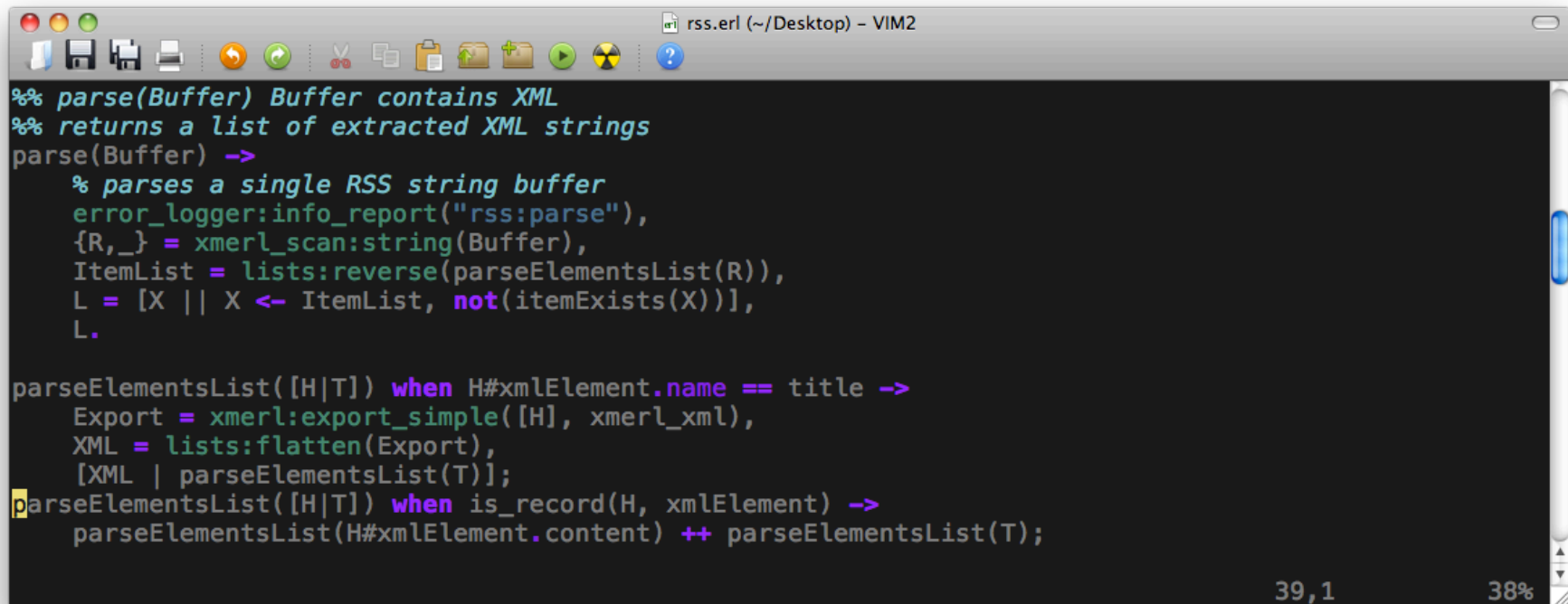


The image shows a VIM2 editor window titled "rss.erl (~/Desktop) - VIM2". The window has a standard macOS-style title bar with red, yellow, and green window control buttons. Below the title bar is a toolbar with various icons for file operations (save, open, print, etc.) and editing (undo, redo, cut, copy, paste, etc.). The main editing area has a dark background with syntax-highlighted Erlang code. The code defines a module named 'rss' and exports a function 'parse/1'. It includes libraries 'xmerl' and 'eunit'. The 'hexstring' function converts a binary to a hexadecimal string. The 'itemExists' function checks if an item exists by converting it to a binary, calculating its MD5 hash, and then checking if the hash is in a list.

```
-module(rss).  
-export([parse/1]).  
-include_lib("xmerl/include/xmerl.hrl").  
-include_lib("eunit/include/eunit.hrl").  
  
hexstring(<<X:128/big-unsigned-integer>>) ->  
    %io:format("~32.16.0b~n", [X]).  
    lists:flatten(io_lib:format("~32.16.0b", [X])).  
  
itemExists(E) ->  
    Binary = term_to_binary(E),  
    Md5 = erlang:md5(Binary),  
    S = hexstring(Md5),  
    R = get(S),  
    case R of  
        undefined ->
```

1,1 Top

# RSS.ERL: Slide B

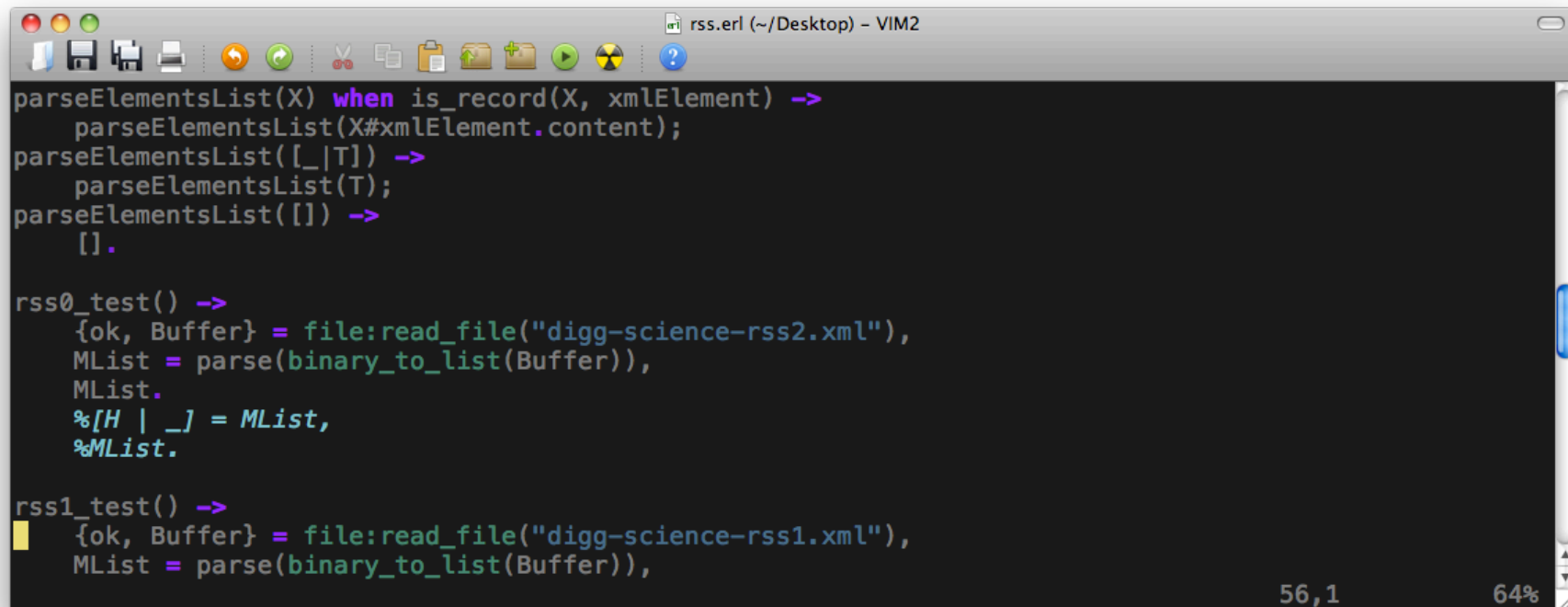


```
rss.erl (~/Desktop) - VIM2
%% parse(Buffer) Buffer contains XML
%% returns a list of extracted XML strings
parse(Buffer) ->
    % parses a single RSS string buffer
    error_logger:info_report("rss:parse"),
    {R,_} = xmerl_scan:string(Buffer),
    ItemList = lists:reverse(parseElementsList(R)),
    L = [X || X <- ItemList, not(itemExists(X))],
    L.

parseElementsList([H|T]) when H#xmlElement.name == title ->
    Export = xmerl:export_simple([H], xmerl_xml),
    XML = lists:flatten(Export),
    [XML | parseElementsList(T)];
parseElementsList([H|T]) when is_record(H, xmlElement) ->
    parseElementsList(H#xmlElement.content) ++ parseElementsList(T);
```

39,1 38%

# RSS.ERL: Slide C



The screenshot shows a VIM2 editor window titled "rss.erl (~/Desktop) - VIM2". The code is written in Erlang and defines a function `parseElementsList` and two test functions, `rss0_test` and `rss1_test`. The `parseElementsList` function uses a `when` clause to handle different input types. The test functions read XML files and parse them into lists.

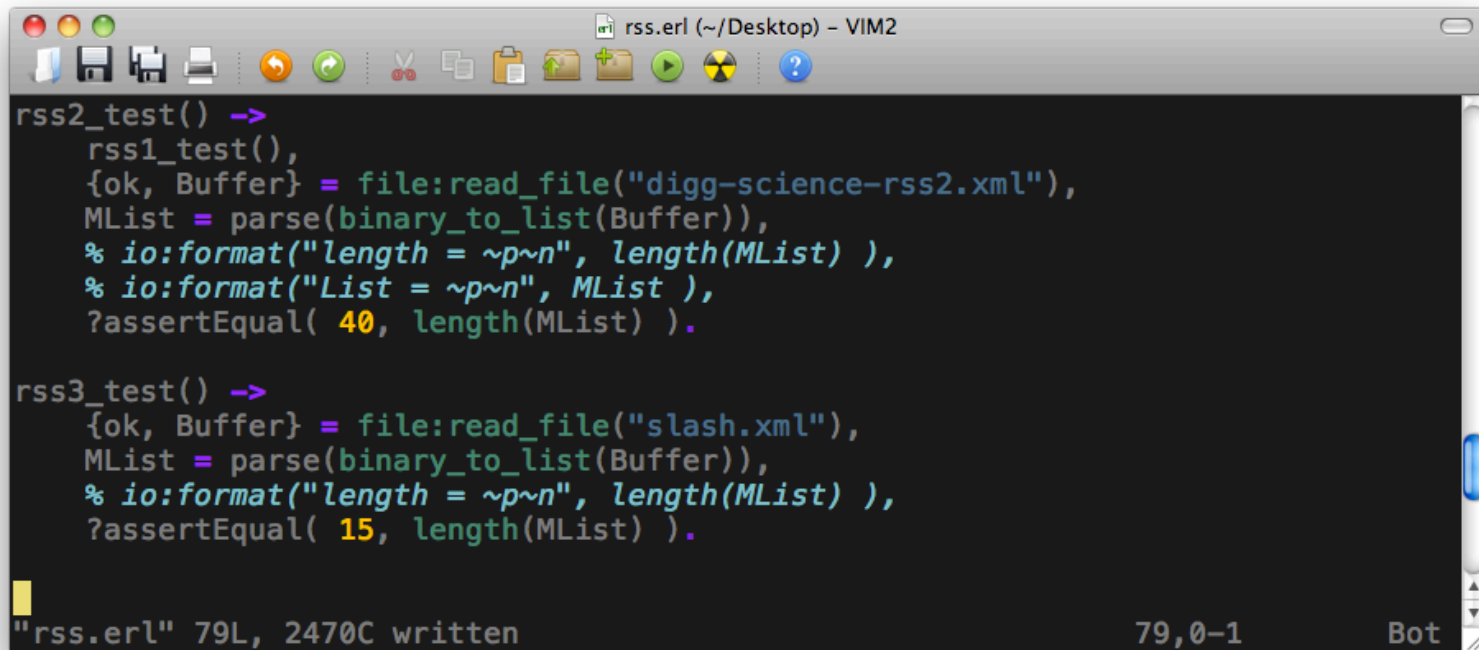
```
parseElementsList(X) when is_record(X, xmlElement) ->
    parseElementsList(X#xmlElement.content);
parseElementsList([_|T]) ->
    parseElementsList(T);
parseElementsList([]) ->
    [].

rss0_test() ->
    {ok, Buffer} = file:read_file("digg-science-rss2.xml"),
    MList = parse(binary_to_list(Buffer)),
    MList.
    %[H | _] = MList,
    %MList.

rss1_test() ->
    {ok, Buffer} = file:read_file("digg-science-rss1.xml"),
    MList = parse(binary_to_list(Buffer)),
```

56,1 64%

# RSS.ERL: Slide D



The image shows a VIM2 editor window titled "rss.erl (~/Desktop) - VIM2". The editor contains Erlang code for two test functions, `rss2_test()` and `rss3_test()`. The code uses `file:read_file()` to read XML files, `binary_to_list()` to convert the binary data to a list, `parse()` to parse the XML, and `io:format()` to print the results. Assertions are used to verify the length of the parsed lists.

```
rss2_test() ->
    rss1_test(),
    {ok, Buffer} = file:read_file("digg-science-rss2.xml"),
    MList = parse(binary_to_list(Buffer)),
    % io:format("length = ~p~n", length(MList) ),
    % io:format("List = ~p~n", MList ),
    ?assertEqual( 40, length(MList) ).

rss3_test() ->
    {ok, Buffer} = file:read_file("slash.xml"),
    MList = parse(binary_to_list(Buffer)),
    % io:format("length = ~p~n", length(MList) ),
    ?assertEqual( 15, length(MList) ).
```

The status bar at the bottom of the window shows "rss.erl" 79L, 2470C written, 79,0-1, and Bot.



# Checkout source: HG clone



```
Terminal — zsh — ttys002 — 63x8
lion% hg clone http://hg.rabbitmq.com/rabbitmq-public-umbrella
```

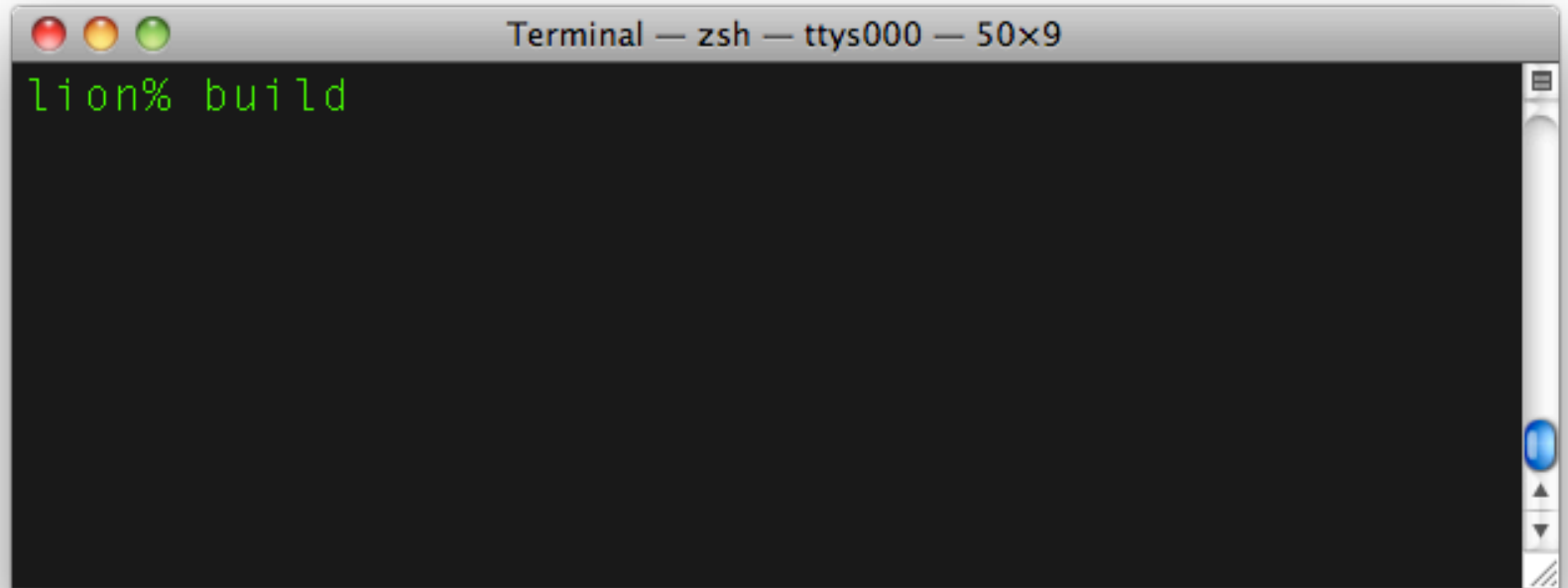
A terminal window with a grey title bar containing three window control buttons (red, yellow, green) on the left and the text "Terminal — zsh — ttys002 — 63x8" on the right. The terminal area has a black background with green text. The command "lion% hg clone http://hg.rabbitmq.com/rabbitmq-public-umbrella" is entered, followed by a green cursor. A vertical scrollbar is visible on the right side of the terminal window.

# Make: Be Patient....

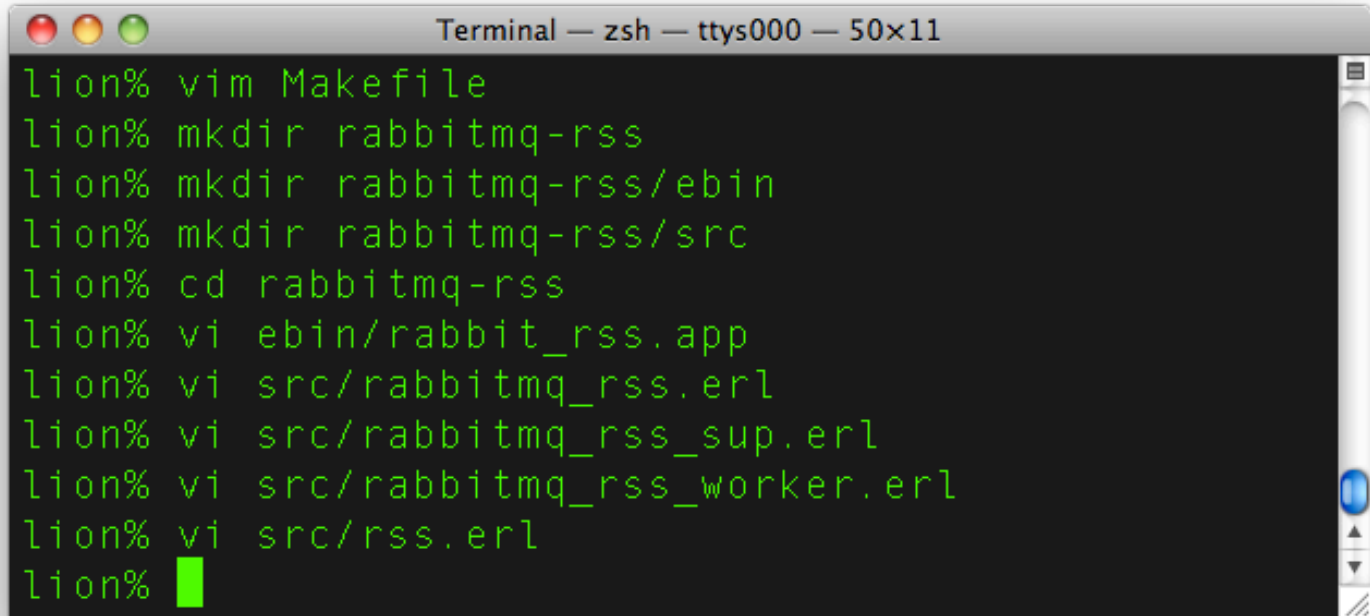
A screenshot of a macOS Terminal window. The title bar at the top reads "Terminal — zsh — ttys000 — 50x9". The window has three colored window control buttons (red, yellow, green) on the left. The main area is black with green text. The text shows a user at a prompt "lion%" changing the directory to "rabbitmq-public-umbrella" and then typing "make co" followed by a green cursor block.

```
lion% cd rabbitmq-public-umbrella
lion% make co█
```

# Build

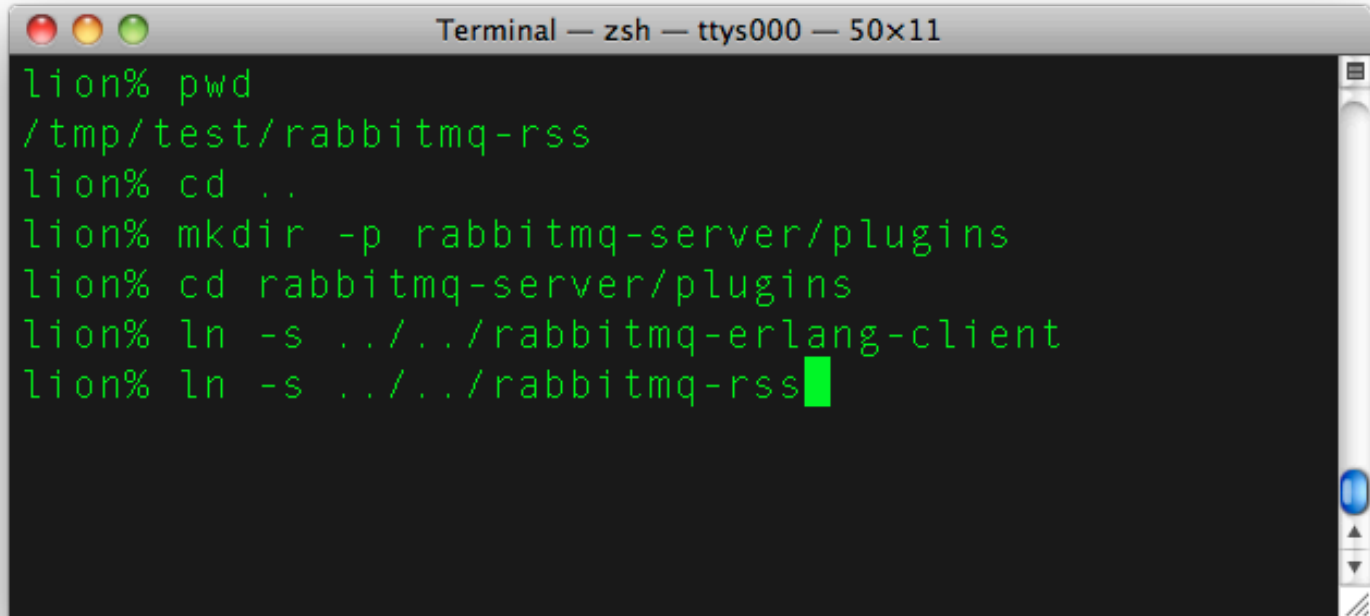


# Skeleton



```
Terminal — zsh — ttys000 — 50x11
lion% vim Makefile
lion% mkdir rabbitmq-rss
lion% mkdir rabbitmq-rss/sbin
lion% mkdir rabbitmq-rss/src
lion% cd rabbitmq-rss
lion% vi ebin/rabbit_rss.app
lion% vi src/rabbitmq_rss.erl
lion% vi src/rabbitmq_rss_sup.erl
lion% vi src/rabbitmq_rss_worker.erl
lion% vi src/rss.erl
lion% █
```

# Symlink Plugin

A terminal window titled "Terminal — zsh — ttys000 — 50x11" with a dark background and green text. The window shows a series of commands being executed in a shell. The commands are: 'pwd' showing the current directory as '/tmp/test/rabbitmq-rss', 'cd ..' moving to the parent directory, 'mkdir -p rabbitmq-server/plugins' creating a new directory, 'cd rabbitmq-server/plugins' moving into the new directory, and two 'ln -s' commands creating symbolic links from the parent directory to the current directory. The first link is for 'rabbitmq-erlang-client' and the second is for 'rabbitmq-rss'. The terminal window has standard macOS window controls (red, yellow, green buttons) at the top left and a scroll bar on the right side.

```
Terminal — zsh — ttys000 — 50x11
lion% pwd
/tmp/test/rabbitmq-rss
lion% cd ..
lion% mkdir -p rabbitmq-server/plugins
lion% cd rabbitmq-server/plugins
lion% ln -s ../../rabbitmq-erlang-client
lion% ln -s ../../rabbitmq-rss
```

# Run Broker

```
$ make run
```

```
(rabbit@rabbit9)1>
```

```
(rabbit@rabbit9)1>application:which_applications().
```

```
{rabbit_rss,"Embedded Rabbit RSS Reader","0.01"},
```

# Test Broker

```
$ mkdir test
```

```
$ vi test/rabbit_rss_tests.erl
```

```
$ vi Makefile
```

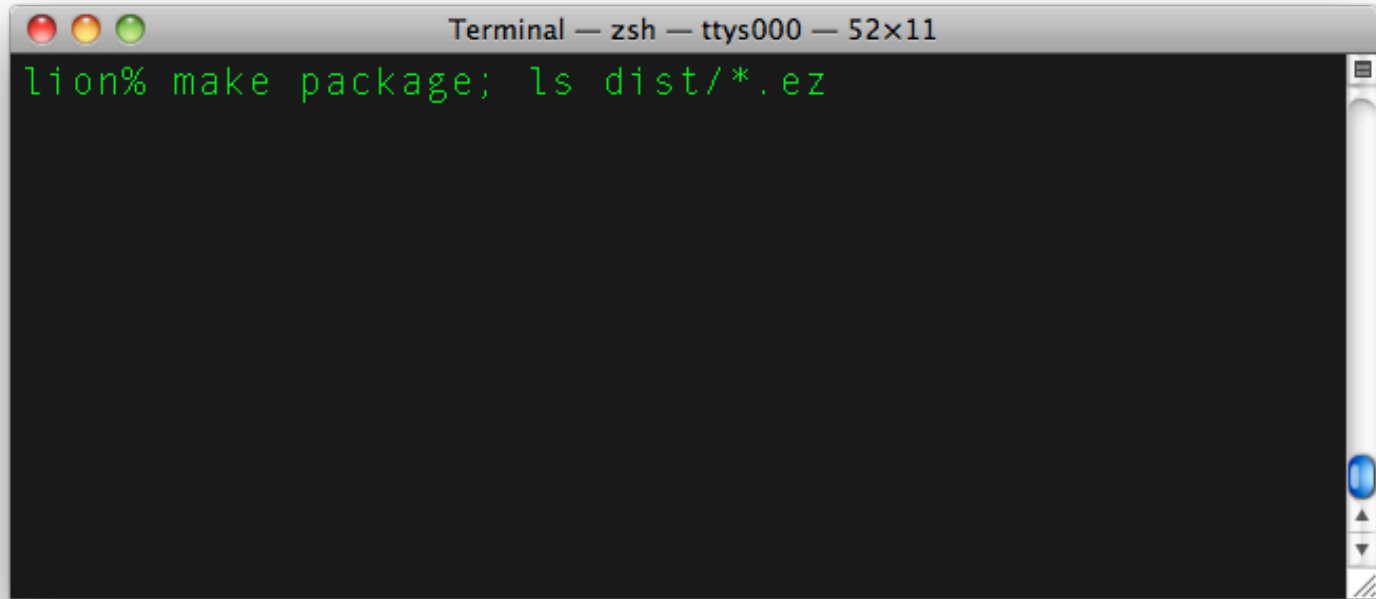
```
TEST_APPS=amqp_client rabbit_rss
```

```
TEST_COMMANDS=rabbit_rss_tests:test()
```

```
START_RABBIT_IN_TESTS=true
```

```
$ make test
```

# Package Plugin

A screenshot of a macOS Terminal window. The title bar at the top reads "Terminal — zsh — ttys000 — 52x11". The window has standard macOS window controls (red, yellow, green buttons) on the top left. The main area is black with green text. The command "lion% make package; ls dist/\*.ez" is entered and displayed. The right side of the window shows a vertical scrollbar and a status bar with a blue button and navigation arrows.

```
lion% make package; ls dist/*.ez
```



# Plugin Supporting Machinery



- ▣ Unit Tests
- ▣ Source Code
- ▣ Logger
- ▣ Printf
- ▣ AMQP Clients

# How To Build a Plugin



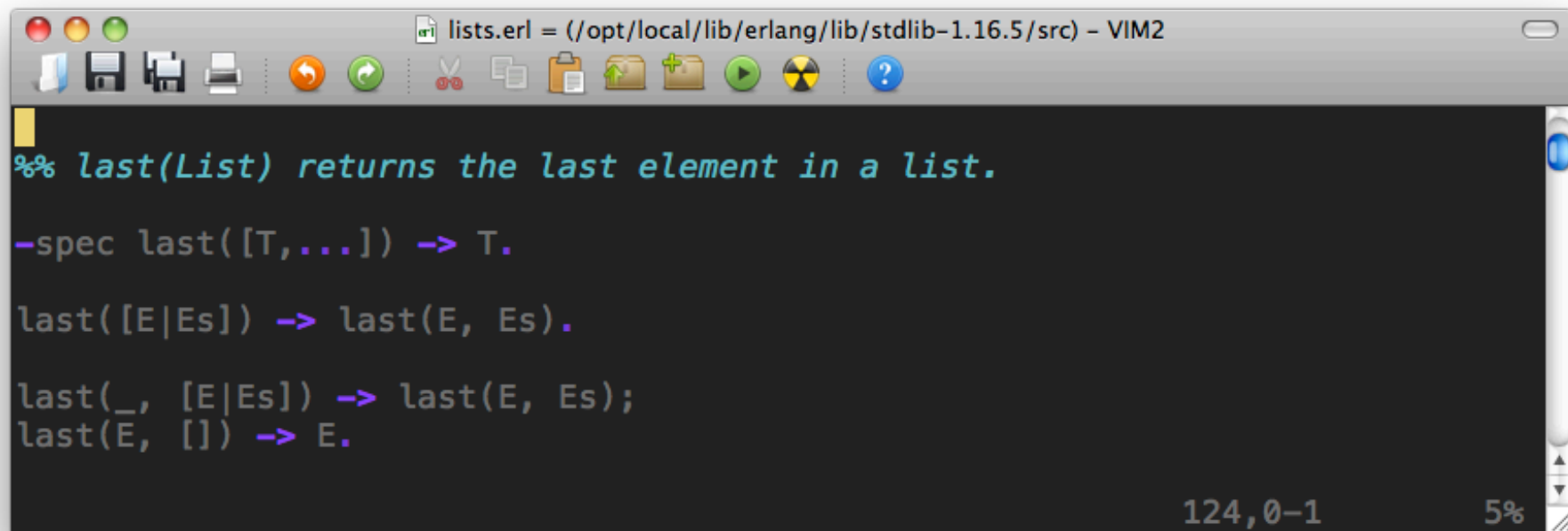
- ▣ RabbitMQ docs
- ▣ Source
- ▣ Examples

# Use the Source!



- ❑ Documentation is hit/miss
- ❑ Google “erlang lists”
- ❑ <http://www.erlang.org/doc/apps/stdlib/>
- ❑ Or skip the docs and go directly to source
- ❑ `./lib/stdlib/src/lists.erl`

# Lists.erl



The screenshot shows a VIM2 editor window with the title bar "lists.erl = (/opt/local/lib/erlang/lib/stdlib-1.16.5/src) - VIM2". The editor contains the following Erlang code:

```
%% last(List) returns the last element in a list.  
  
-spec last([T,...]) -> T.  
  
last([E|Es]) -> last(E, Es).  
  
last(_, [E|Es]) -> last(E, Es);  
last(E, []) -> E.
```

The bottom right corner of the editor window displays "124,0-1" and "5%".

# Logger



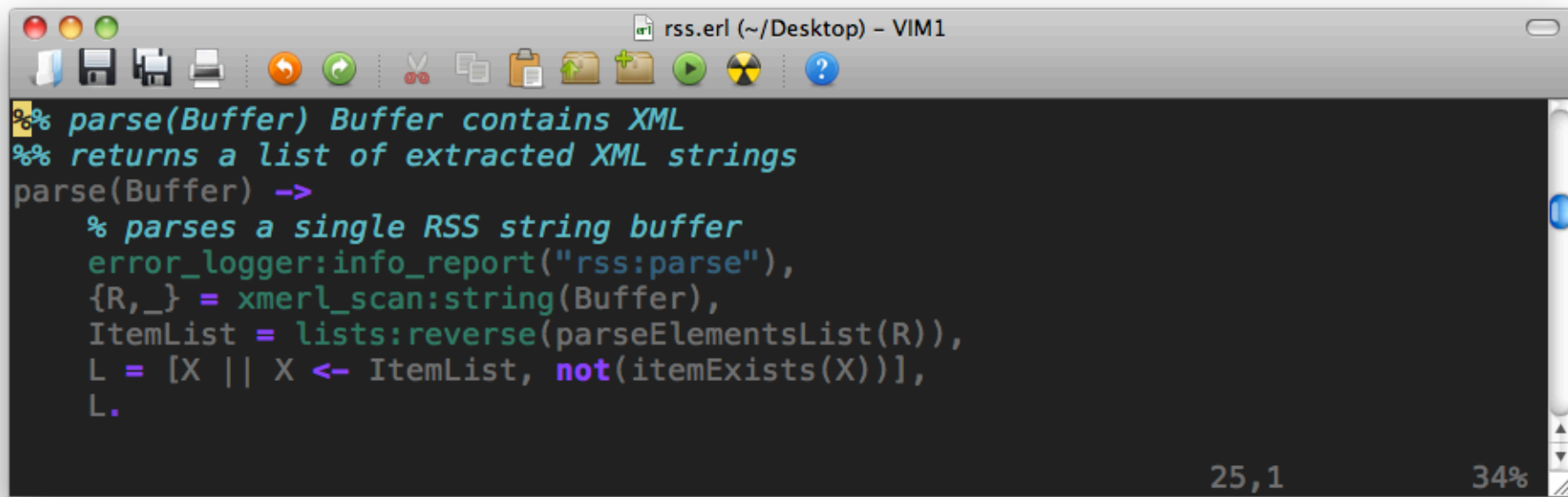
- ❑ Plugin executes in its own process
- ❑ No plugin stdout
- ❑ No plugin execution debugger

# Logger: Continued



- ❑ Solution: Erlang's logger
- ❑ Output to `/var/log/rabbitmq/<hostname`
- ❑ Nothing to configure

# RSS.erl



```
% parse(Buffer) Buffer contains XML
%% returns a list of extracted XML strings
parse(Buffer) ->
    % parses a single RSS string buffer
    error_logger:info_report("rss:parse"),
    {R,_} = xmerl_scan:string(Buffer),
    ItemList = lists:reverse(parseElementsList(R)),
    L = [X || X <- ItemList, not(itemExists(X))],
    L.
```

25,1 34%

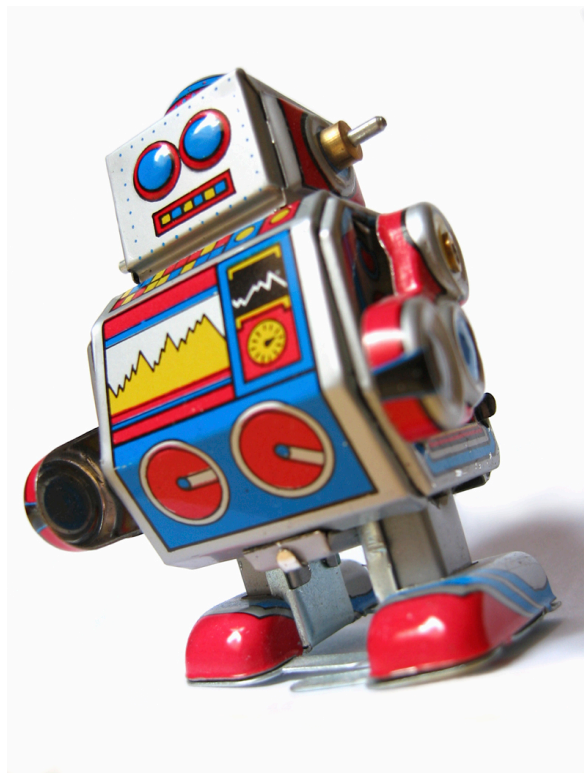
# OTP Design Principles



- ▣ Supervision Tree
- ▣ Behaviors
- ▣ Applications
- ▣ Releases
- ▣ Release Handling



# Machine Learning



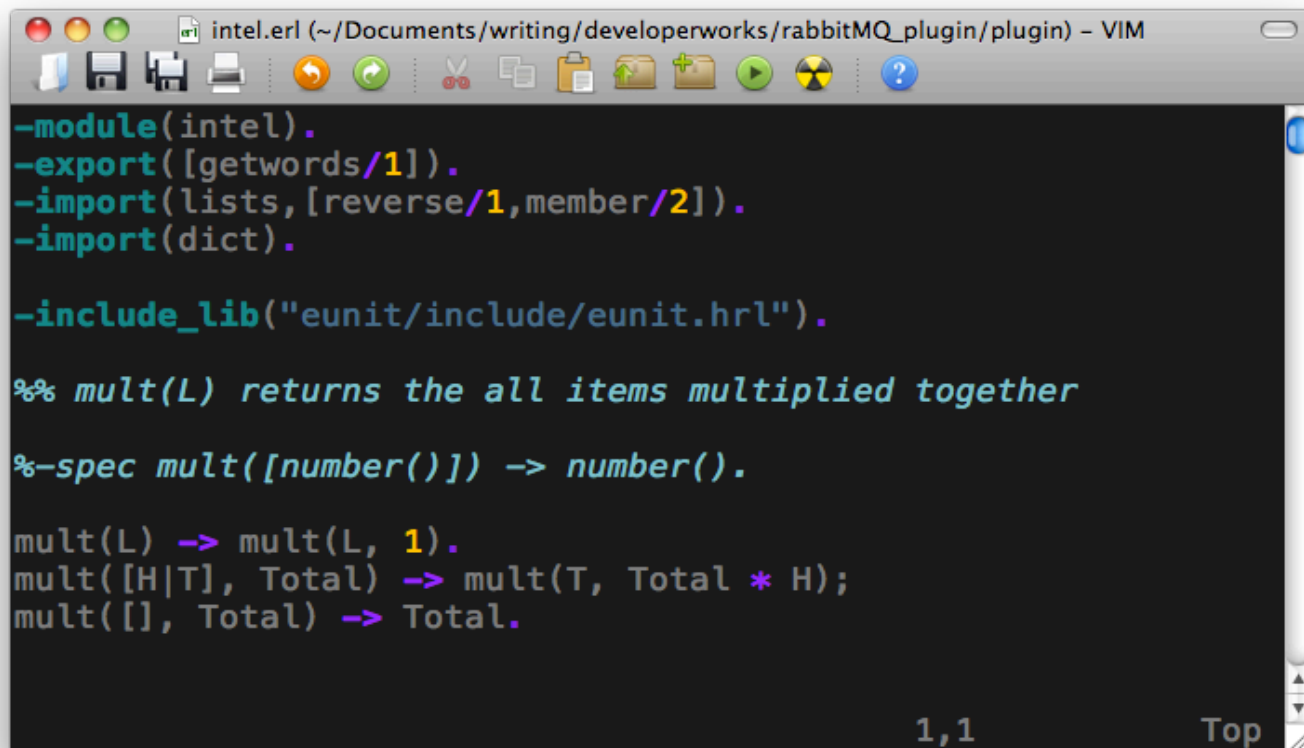
- The next step for the RSS plugin.
- A potential use case for RabbitMQ plugins.

# Fisher Classifier For RSS Feeds



- ▣ Data mining
- ▣ Classification

# Fisher Classifier: Slide A



```
intel.erl (~/.Documents/writing/developerworks/rabbitMQ_plugin/plugin) - VIM

-module(intel).
-export([getwords/1]).
-import(lists,[reverse/1,member/2]).
-import(dict).

-include_lib("eunit/include/eunit.hrl").

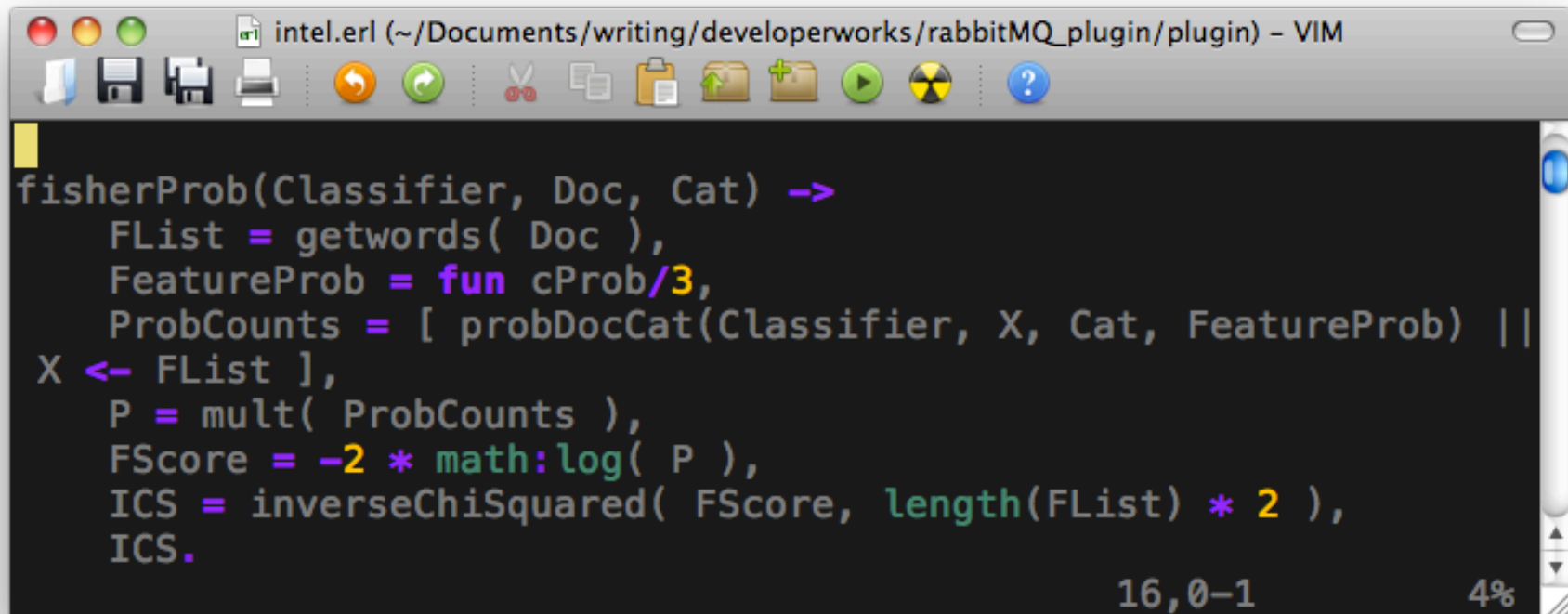
%% mult(L) returns the all items multiplied together

%-spec mult([number()]) -> number().

mult(L) -> mult(L, 1).
mult([H|T], Total) -> mult(T, Total * H);
mult([], Total) -> Total.
```

1,1 Top

# Fisher Classifier: Slide B

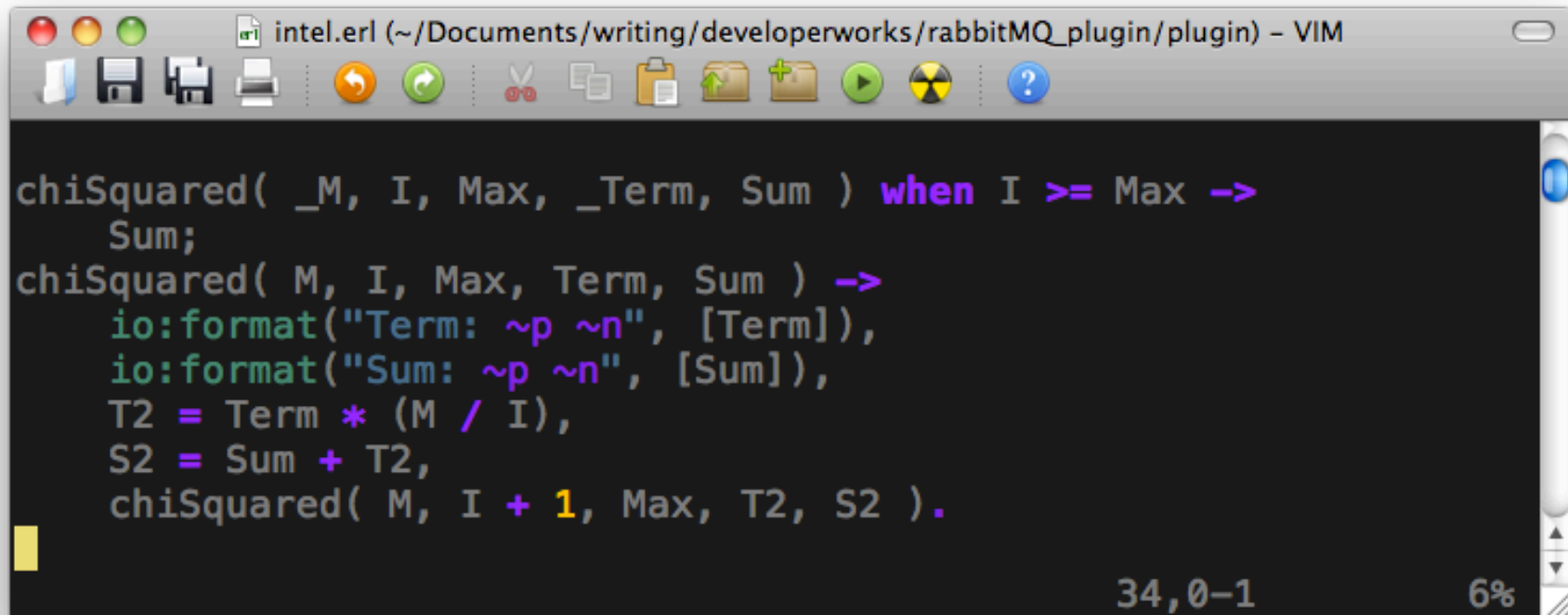


The image shows a VIM editor window with a title bar that reads "intel.erl (~/Documents/writing/developerworks/rabbitMQ\_plugin/plugin) - VIM". The editor has a standard toolbar with icons for file operations and editing. The code is written in Erlang and defines a function `fisherProb` that takes three arguments: `Classifier`, `Doc`, and `Cat`. The function calculates a Fisher score for a document given a classifier and a category. It uses `getwords` to extract words from the document, then iterates over each word to calculate its probability using `probDocCat`. These probabilities are multiplied together to get `P`, which is then used to calculate the Fisher score `FScore` using `math:log`. Finally, `inverseChiSquared` is used to convert the Fisher score into an Inverse Chi-Squared Score (ICS). The code is as follows:

```
fisherProb(Classifier, Doc, Cat) ->
    FList = getwords( Doc ),
    FeatureProb = fun cProb/3,
    ProbCounts = [ probDocCat(Classifier, X, Cat, FeatureProb) ||
X <- FList ],
    P = mult( ProbCounts ),
    FScore = -2 * math:log( P ),
    ICS = inverseChiSquared( FScore, length(FList) * 2 ),
    ICS.
```

The status bar at the bottom right of the editor shows the cursor position "16,0-1" and the percentage of the file loaded "4%".

# Fisher Classifier: Slide C

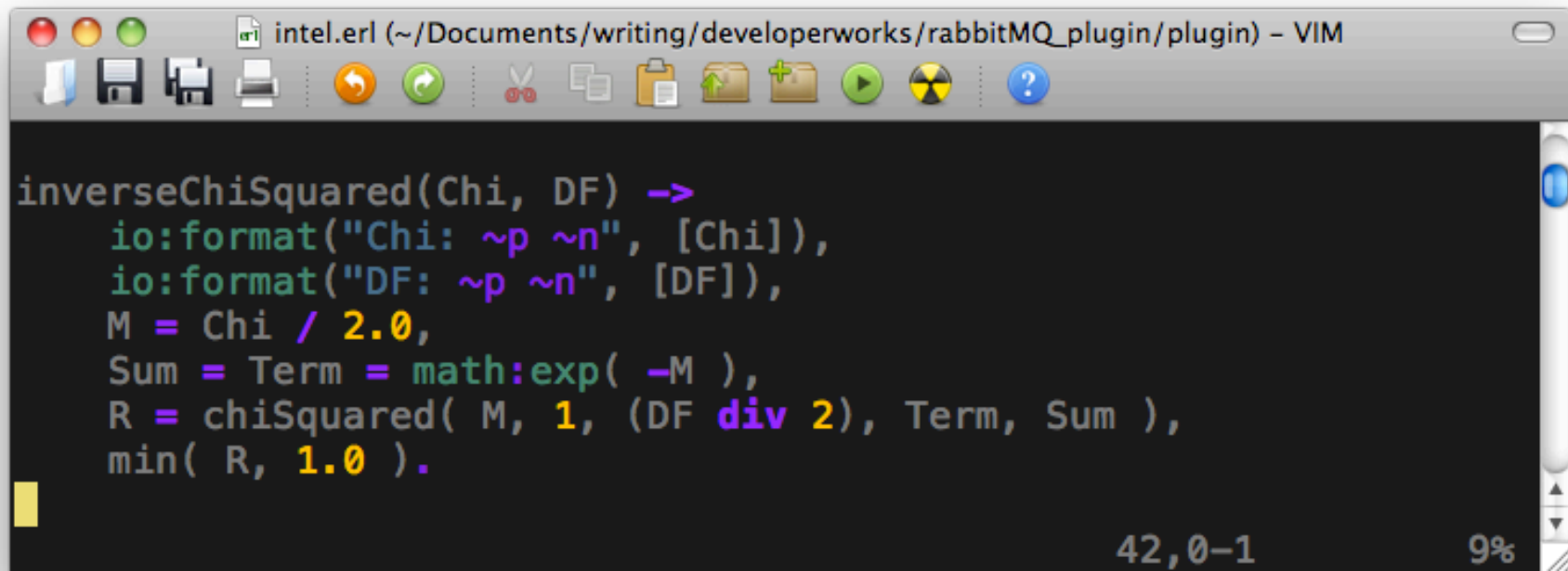


```
intel.erl (~/Documents/writing/developerworks/rabbitMQ_plugin/plugin) - VIM

chiSquared( _M, I, Max, _Term, Sum ) when I >= Max ->
    Sum;
chiSquared( M, I, Max, Term, Sum ) ->
    io:format("Term: ~p ~n", [Term]),
    io:format("Sum: ~p ~n", [Sum]),
    T2 = Term * (M / I),
    S2 = Sum + T2,
    chiSquared( M, I + 1, Max, T2, S2 ).

34,0-1 6%
```

# Fisher Classifier: Slide D

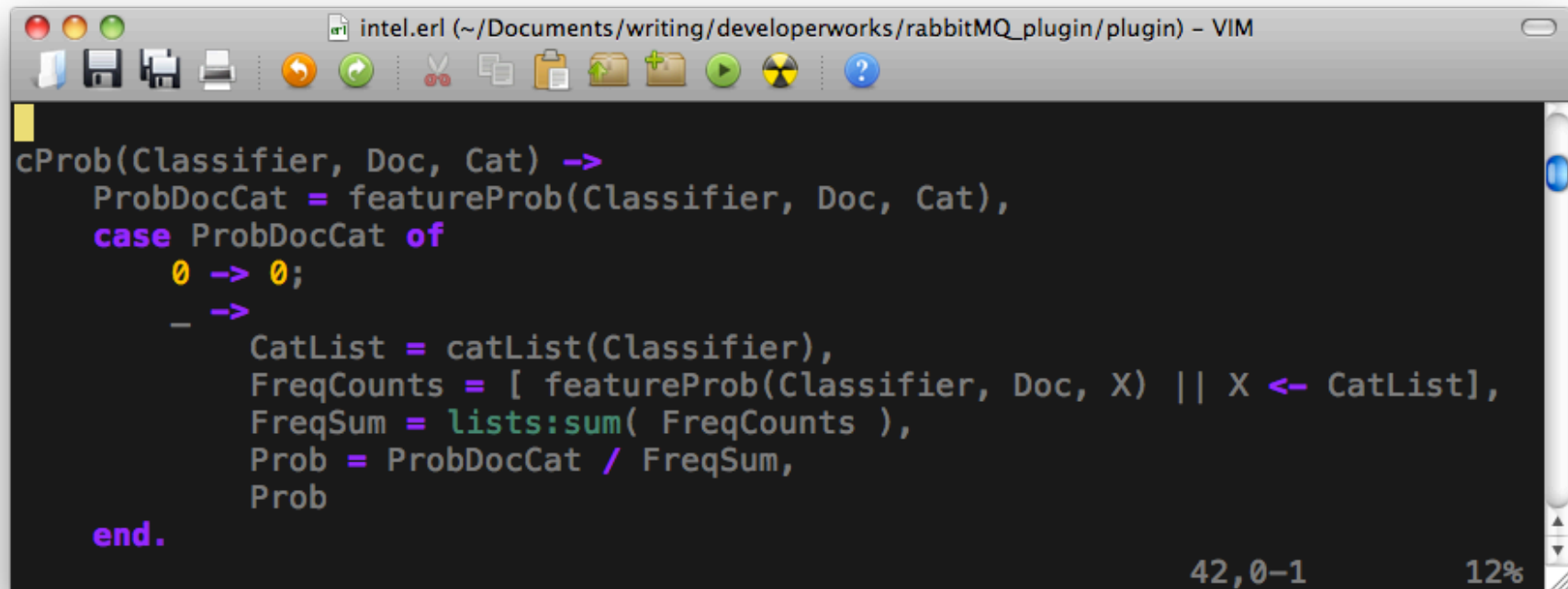


The image shows a VIM editor window with a macOS-style title bar. The title bar text is "intel.erl (~/.Documents/writing/developerworks/rabbitMQ\_plugin/plugin) - VIM". Below the title bar is a toolbar with icons for file operations (save, open, print, etc.) and editing (undo, redo, cut, copy, paste, etc.). The main editing area has a dark background and displays the following Erlang code:

```
inverseChiSquared(Chi, DF) ->
    io:format("Chi: ~p ~n", [Chi]),
    io:format("DF: ~p ~n", [DF]),
    M = Chi / 2.0,
    Sum = Term = math:exp( -M ),
    R = chiSquared( M, 1, (DF div 2), Term, Sum ),
    min( R, 1.0 ).
```

At the bottom right of the editor window, the text "42,0-1" and "9%" are visible, indicating the current line and column position and the zoom level.

# Fisher Classifier: Slide E



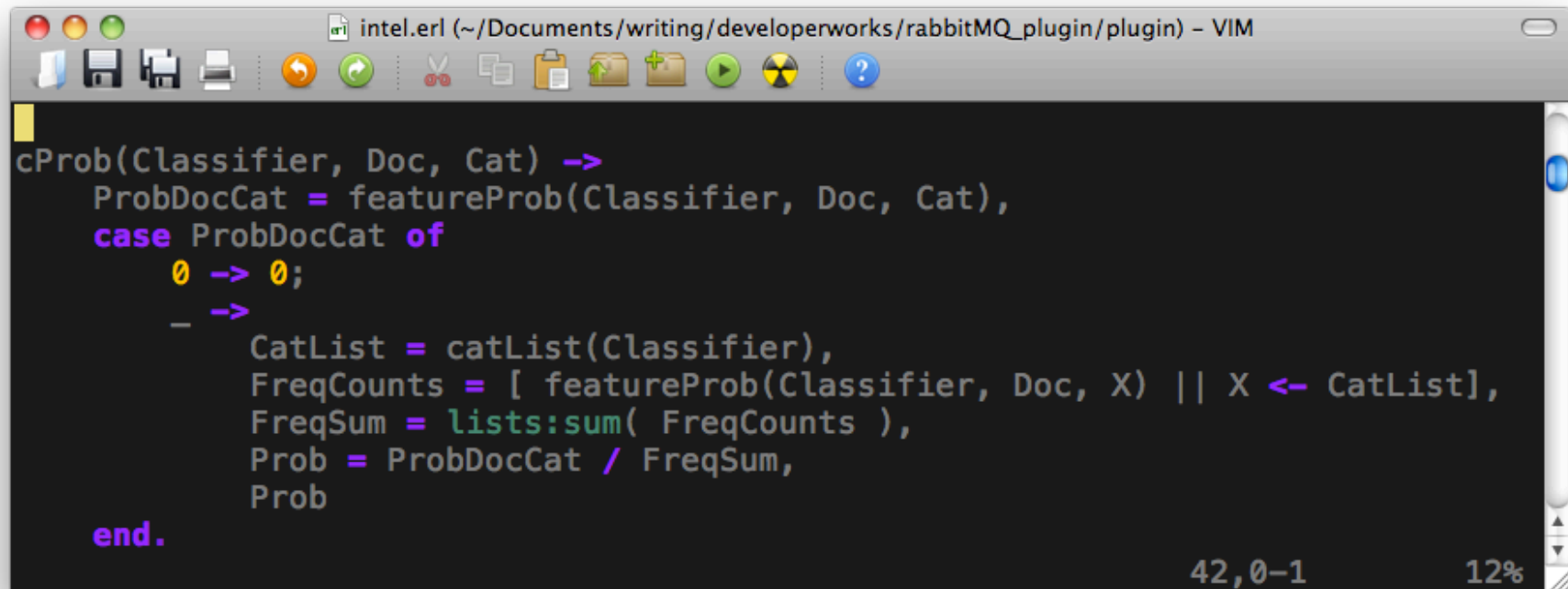
The image shows a VIM editor window with a dark background and syntax-highlighted Erlang code. The window title is 'intel.erl (~/.Documents/writing/developerworks/rabbitMQ\_plugin/plugin) - VIM'. The code defines a function 'cProb' that calculates a probability for a document and category. It uses 'featureProb' to get a probability, then a 'case' statement to handle different categories. For category 0, it returns 0. For other categories, it calculates the frequency of each category in the document and normalizes the probability. The code ends with 'end.'.

```
intel.erl (~/.Documents/writing/developerworks/rabbitMQ_plugin/plugin) - VIM

cProb(Classifier, Doc, Cat) ->
  ProbDocCat = featureProb(Classifier, Doc, Cat),
  case ProbDocCat of
    0 -> 0;
    _ ->
      CatList = catList(Classifier),
      FreqCounts = [ featureProb(Classifier, Doc, X) || X <- CatList ],
      FreqSum = lists:sum( FreqCounts ),
      Prob = ProbDocCat / FreqSum,
      Prob
  end.
```

42,0-1 12%

# Fisher Classifier: Slide E



The image shows a VIM editor window with a dark background and syntax-highlighted Erlang code. The window title is 'intel.erl (~/.Documents/writing/developerworks/rabbitMQ\_plugin/plugin) - VIM'. The code defines a function 'cProb' that calculates a probability for a document 'Doc' and a category 'Cat'. It uses 'featureProb' to get a probability, then a 'case' statement to handle different categories. For category 0, it returns 0. For other categories, it calculates the frequency of each category in the document and normalizes the probability. The code ends with 'end.'.

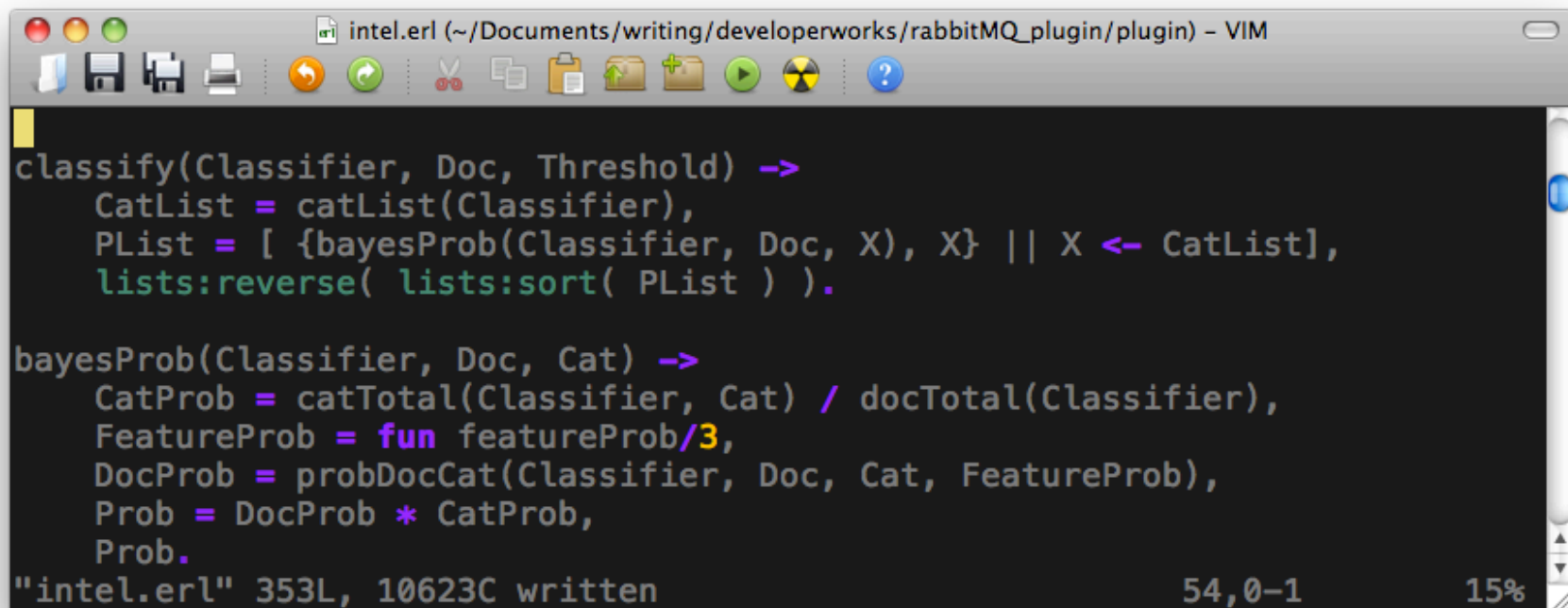
```
intel.erl (~/.Documents/writing/developerworks/rabbitMQ_plugin/plugin) - VIM

cProb(Classifier, Doc, Cat) ->
  ProbDocCat = featureProb(Classifier, Doc, Cat),
  case ProbDocCat of
    0 -> 0;
    _ ->
      CatList = catList(Classifier),
      FreqCounts = [ featureProb(Classifier, Doc, X) || X <- CatList ],
      FreqSum = lists:sum( FreqCounts ),
      Prob = ProbDocCat / FreqSum,
      Prob
  end.
```

42,0-1 12%



# Fisher Classifier: Slide F



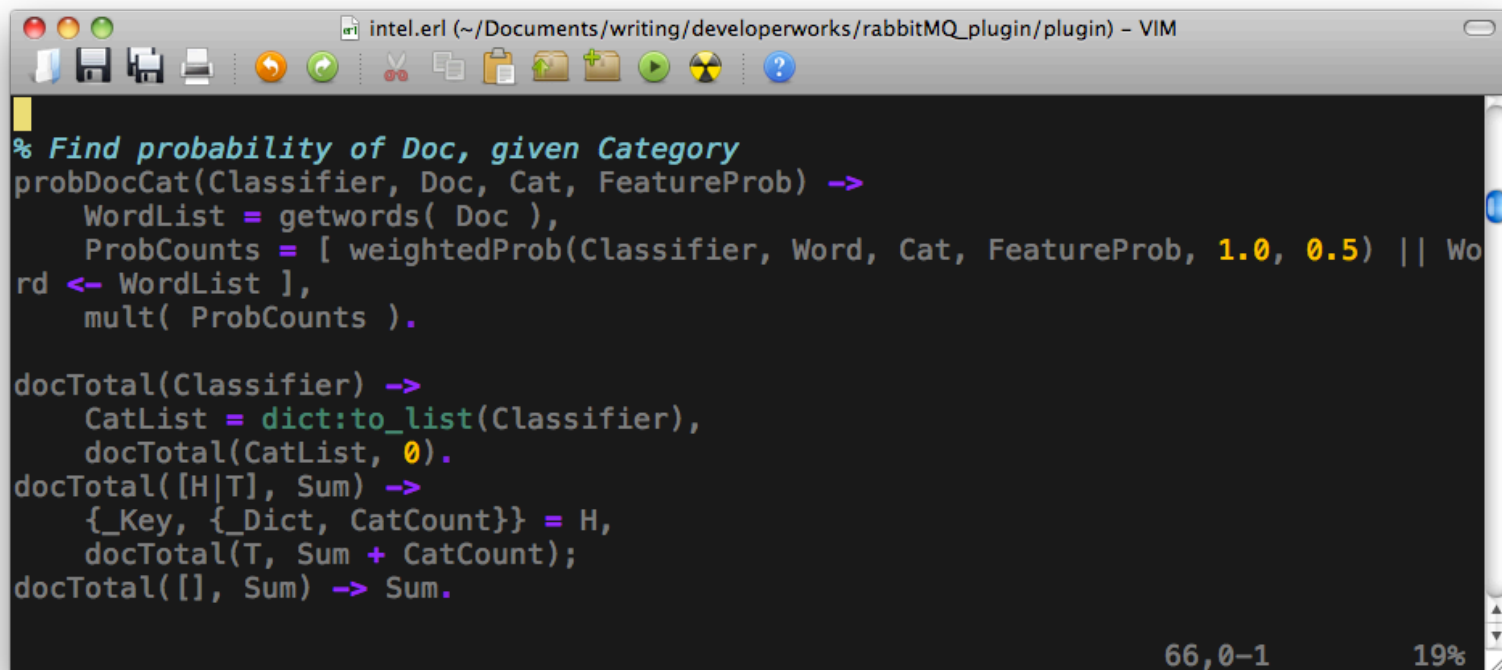
The image shows a VIM editor window titled "intel.erl (~/.Documents/writing/developerworks/rabbitMQ\_plugin/plugin) - VIM". The window contains Erlang code for a Fisher Classifier. The code defines a `classify` function and a `bayesProb` function. The `classify` function takes a `Classifier`, a `Doc`, and a `Threshold` as arguments. It calculates the probability for each category in `CatList` and returns the category with the highest probability. The `bayesProb` function calculates the probability for a given category `Cat` based on the document `Doc` and the classifier. The code is written in Erlang and uses standard VIM syntax highlighting.

```
classify(Classifier, Doc, Threshold) ->
    CatList = catList(Classifier),
    PList = [ {bayesProb(Classifier, Doc, X), X} || X <- CatList],
    lists:reverse( lists:sort( PList ) ).

bayesProb(Classifier, Doc, Cat) ->
    CatProb = catTotal(Classifier, Cat) / docTotal(Classifier),
    FeatureProb = fun featureProb/3,
    DocProb = probDocCat(Classifier, Doc, Cat, FeatureProb),
    Prob = DocProb * CatProb,
    Prob.

"intel.erl" 353L, 10623C written                    54,0-1                    15%
```

# Fisher Classifier: Slide G



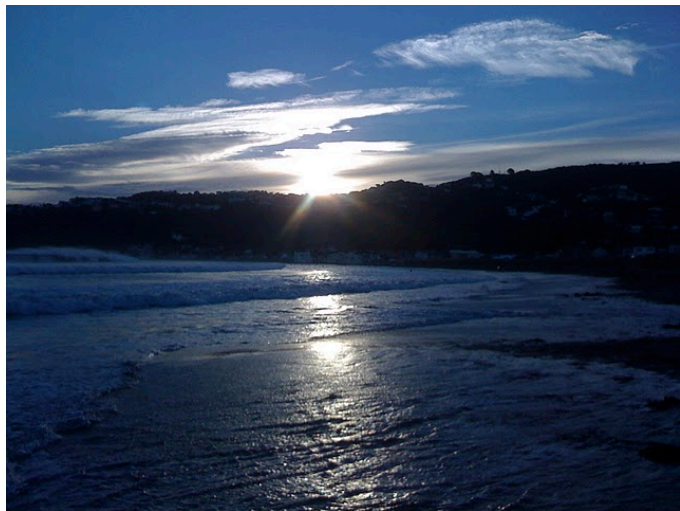
```
intel.erl (~/.Documents/writing/developerworks/rabbitMQ_plugin/plugin) - VIM

% Find probability of Doc, given Category
probDocCat(Classifier, Doc, Cat, FeatureProb) ->
    WordList = getwords( Doc ),
    ProbCounts = [ weightedProb(Classifier, Word, Cat, FeatureProb, 1.0, 0.5) || Word <- WordList ],
    mult( ProbCounts ).

docTotal(Classifier) ->
    CatList = dict:to_list(Classifier),
    docTotal(CatList, 0).
docTotal([H|T], Sum) ->
    {_Key, {_Dict, CatCount}} = H,
    docTotal(T, Sum + CatCount);
docTotal([], Sum) -> Sum.
```

66,0-1 19%

# Conclusion



- ❑ AMQP
- ❑ Erlang
- ❑ Python
- ❑ RSS

# Questions

- ▣ Code: [https://github.com/mikev/rabbitmq\\_rss\\_plugin](https://github.com/mikev/rabbitmq_rss_plugin)
- ▣ Contact: [“Noah Gift” noah.gift@gmail.com](mailto:noah.gift@gmail.com),  
[“Michael Vierling” mvierling@gmail.com](mailto:mvierling@gmail.com)

