# A DSL for Scripting Refactorings in Erlang

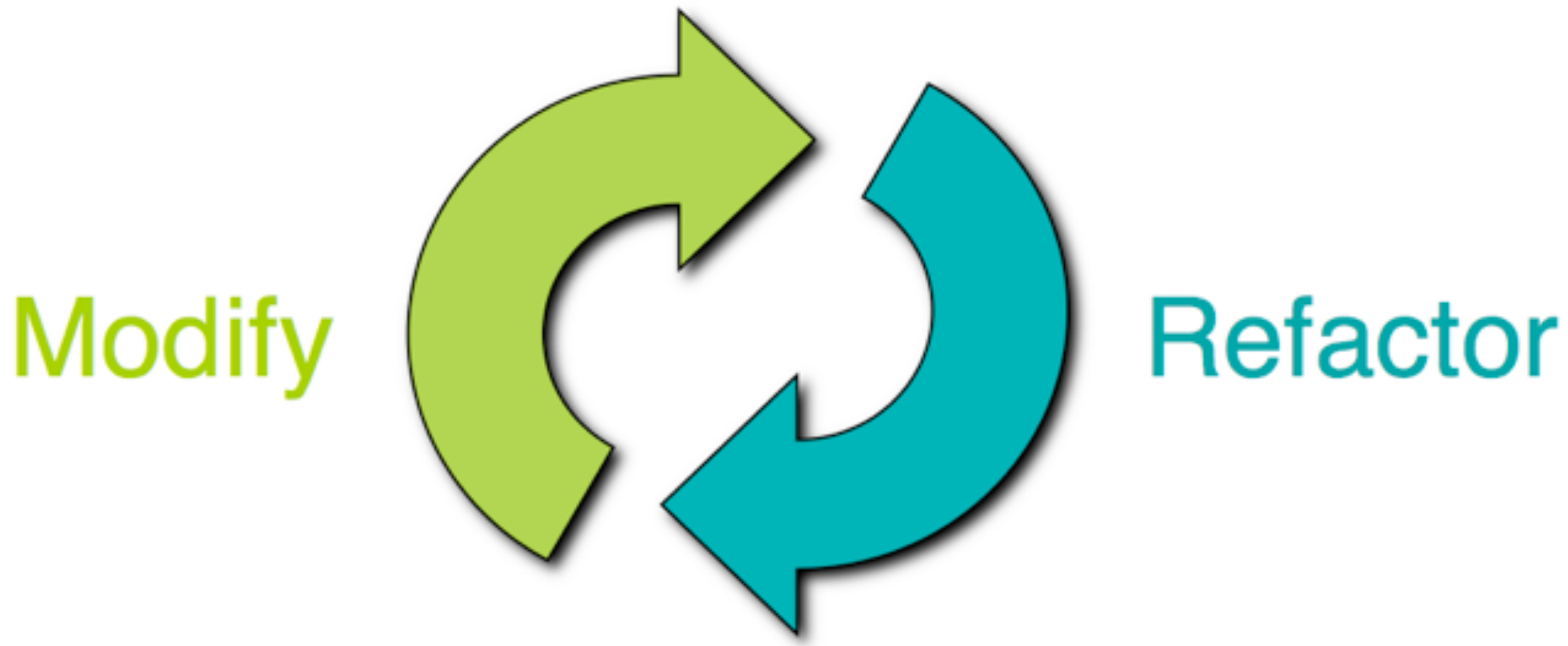
Huiqing Li and Simon Thompson

School of Computing
University of Kent

ProTest
property based testing

## Refactoring

Change how a program works without changing what it does



# Why refactor?

## Extension and reuse

```
loop_a() ->
    receive
      stop -> ok;
      {msg, _Msg, 0} -> loop_a();
      {msg, Msg, N} ->
        io:format("ping!~n"),
        timer:sleep(500),
        b ! {msg, Msg, N - 1},
        loop_a()
     end.
```

Let’s turn this into a function

# Why refactor?

## Extension and reuse

```
loop_a() ->
    receive
      stop -> ok;
      {msg, _Msg, 0} -> loop_a();
      {msg, Msg, N} ->
        io:format("ping!~n"),
        timer:sleep(500),
        b ! {msg, Msg, N - 1},
        loop_a()
     end.
```

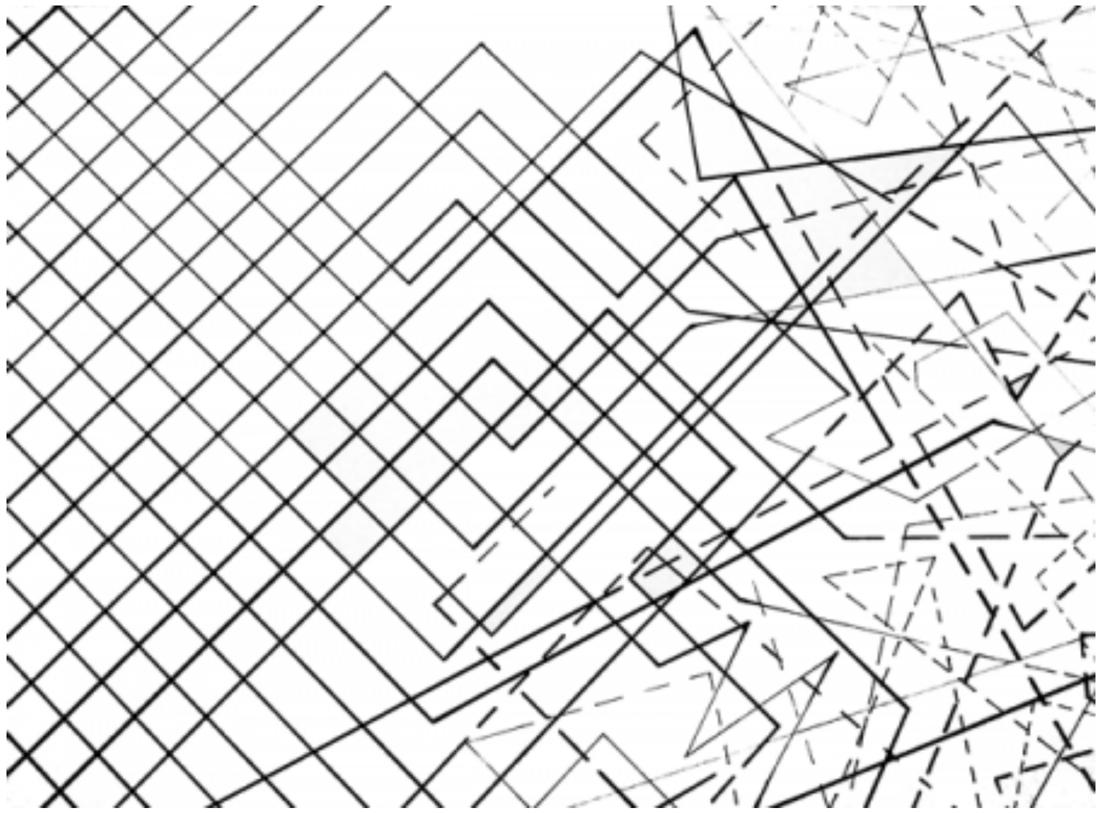
```
loop_a() ->
    receive
      stop -> ok;
      {msg, _Msg, 0} -> loop_a();
      {msg, Msg, N} ->
        body(Msg,N),
        loop_a()
     end.

body(Msg,N) ->
       io:format("ping!~n"),
       timer:sleep(500),
       b ! {msg, Msg, N - 1}.
```

# Why refactor?

Counteract decay ... comprehension

"Clones considered harmful": detect and eliminate duplicate code.

Improve the module structure: remove loops, for example.



# How to refactor?

By hand … using an editor.

- Flexible … but error-prone.
- Infeasible in the large.

Tool supported.

- Handle atoms, names, side-effects, …
- Scalable to large-code bases.
- Integrated with tests, macros, ...

# Wrangler

Clone detection and removal

Module structure improvement

Basic refactorings: structural, macro, process and test-framework related

# Wrangler in a nutshell

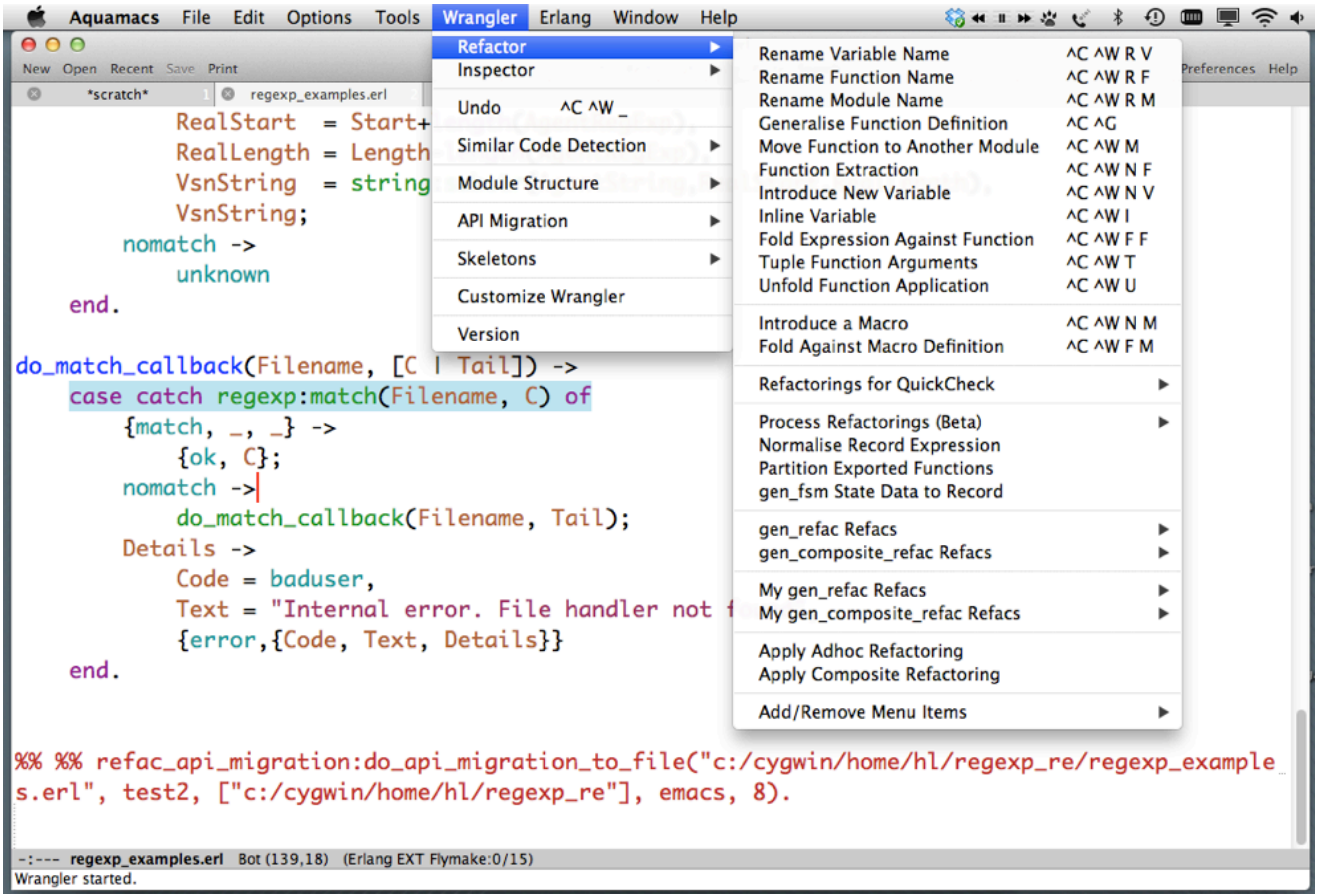
Automate the simple things, and …
… provide decision support tools otherwise.

Embed in common IDEs: emacs, eclipse, …

Handle full language, multiple modules, tests, ...

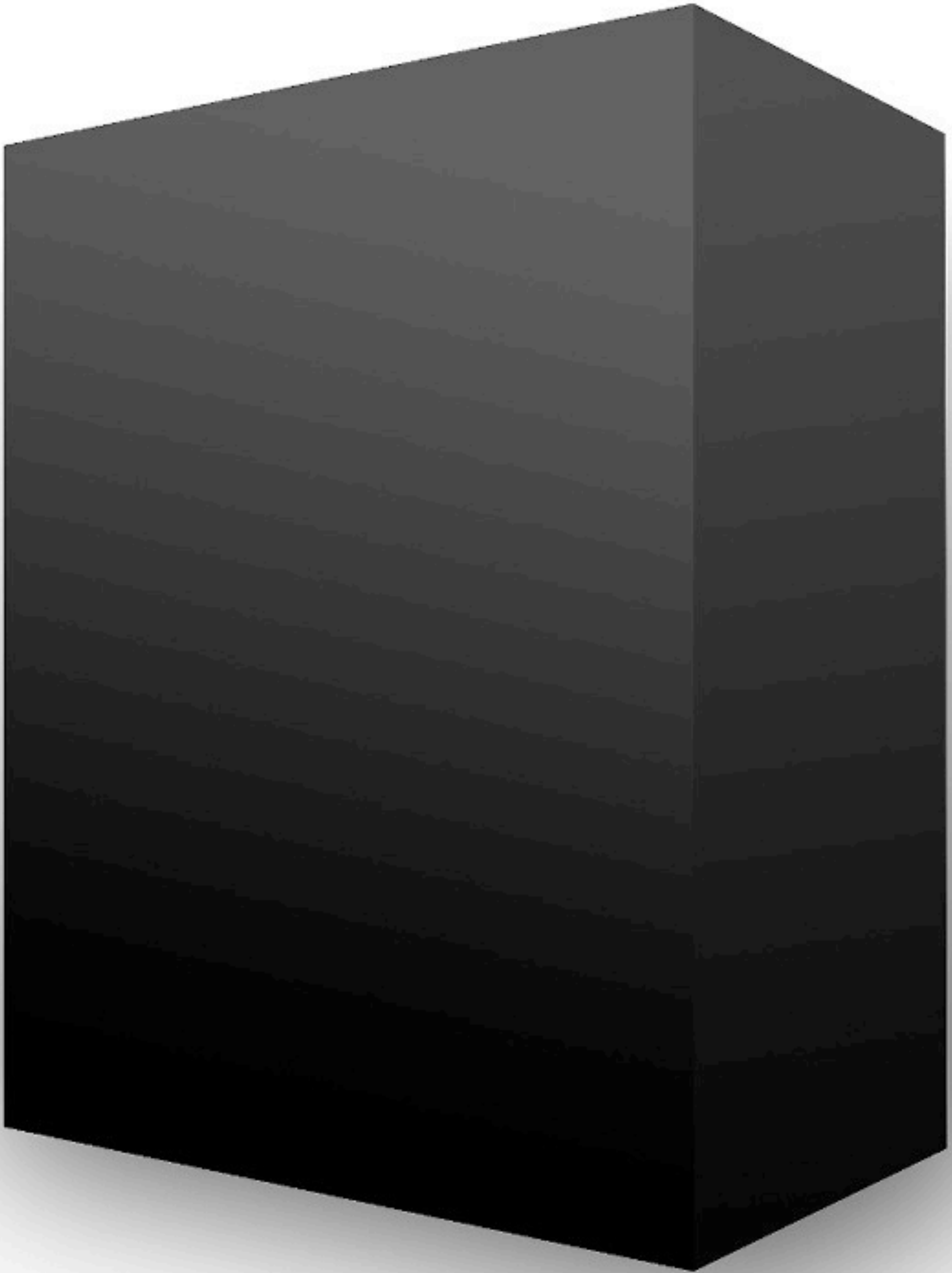
Faithful to layout and comments.

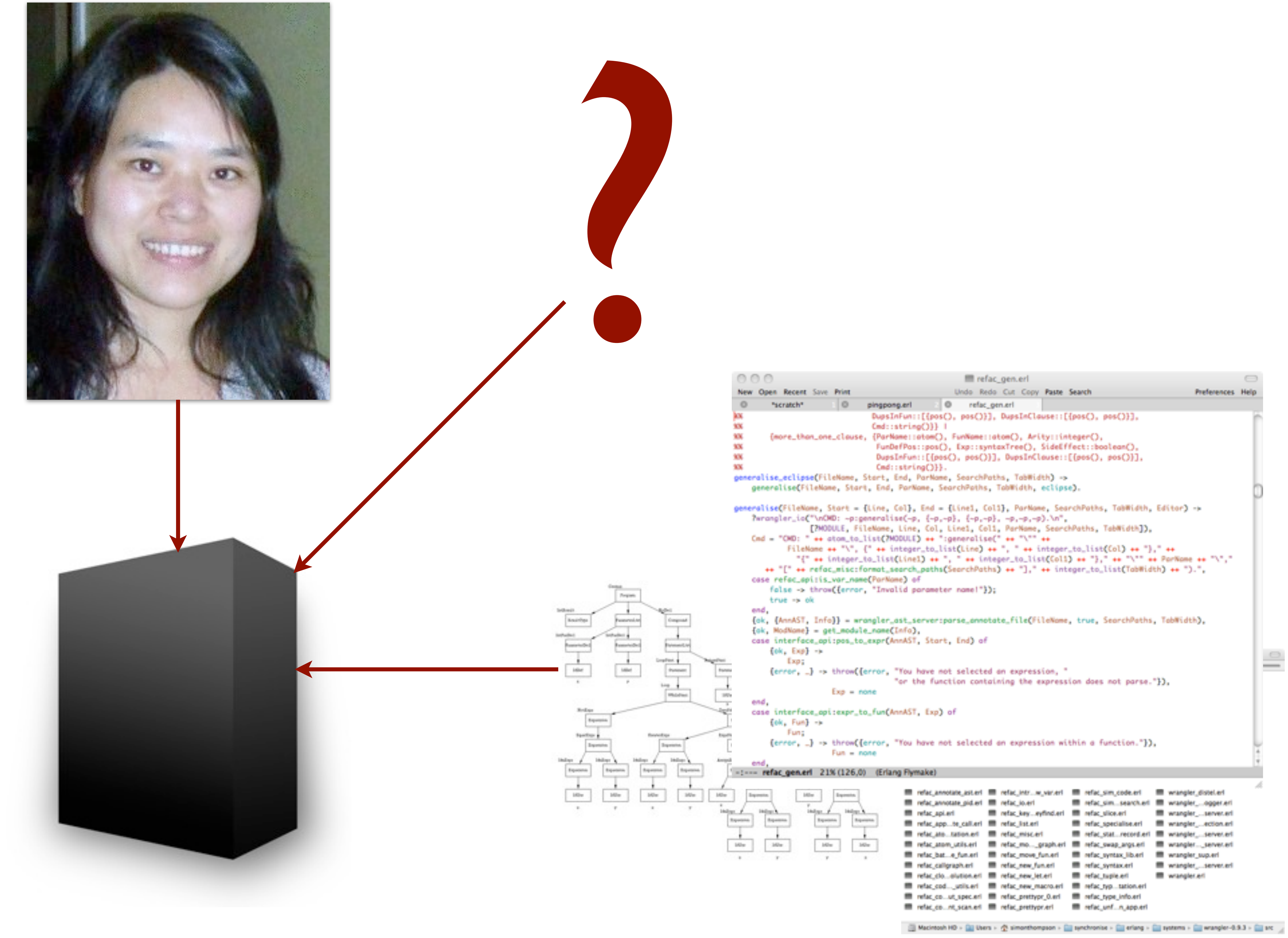
Build in Erlang and apply the tool to itself.


Aquamacs
File
Edit
Options
Tools
Wrangler
Erlang
Window
Help
New
Open
Recent
Save
Print
Preferences
Help
*scratch*
regexp_examples.erl
Refactor
Inspector
Undo ^C ^W _
Similar Code Detection
Module Structure
API Migration
Skeletons
Customize Wrangler
Version
Rename Variable Name ^C ^W R V
Rename Function Name ^C ^W R F
Rename Module Name ^C ^W R M
Generalise Function Definition ^C ^G
Move Function to Another Module ^C ^W M
Function Extraction ^C ^W N F
Introduce New Variable ^C ^W N V
Inline Variable ^C ^W I
Fold Expression Against Function ^C ^W F F
Tuple Function Arguments ^C ^W T
Unfold Function Application ^C ^W U
Introduce a Macro ^C ^W N M
Fold Against Macro Definition ^C ^W F M
Refactorings for QuickCheck
Process Refactorings (Beta)
Normalise Record Expression
Partition Exported Functions
gen_fsm State Data to Record
gen_refac Refacs
gen_composite_refac Refacs
My gen_refac Refacs
My gen_composite_refac Refacs
Apply Adhoc Refactoring
Apply Composite Refactoring
Add/Remove Menu Items

```
        RealStart   = Start+
        RealLength  = Length
        VsnString   = string
        VsnString;
    nomatch ->
        unknown
end.

do_match_callback(Filename, [C | Tail]) ->
    case catch regexp:match(Filename, C) of
        {match, _, _} ->
            {ok, C};
        nomatch ->
            do_match_callback(Filename, Tail);
        Details ->
            Code = baduser,
            Text = "Internal error. File handler not f
            {error,{Code, Text, Details}}
    end.

%% %% refac_api_migration:do_api_migration_to_file("c:/cygwin/home/hl/regexp_re/regexp_example
s.erl", test2, ["c:/cygwin/home/hl/regexp_re"], emacs, 8).
```

```
-:---  regexp_examples.erl   Bot (139,18)   (Erlang EXT Flymake:0/15)
Wrangler started.
```




?
refac_gen.erl
New Open Recent Save Print Undo Redo Cut Copy Paste Search Preferences Help
*scratch*
pingpong.erl
refac_gen.erl

```
%%                         DupsInFun::[{pos(), pos()}], DupsInClause::[{pos(), pos()}],
%%                         Cmd::string()}} |
%%      {more_than_one_clause, {ParName::atom(), FunName::atom(), Arity::integer(),
%%                          FunDefPos::pos(), Exp::syntaxTree(), SideEffect::boolean(),
%%                          DupsInFun::[{pos(), pos()}], DupsInClause::[{pos(), pos()}],
%%                          Cmd::string()}}.
generalise_eclipse(FileName, Start, End, ParName, SearchPaths, TabWidth) ->
    generalise(FileName, Start, End, ParName, SearchPaths, TabWidth, eclipse).

generalise(FileName, Start = {Line, Col}, End = {Line1, Col1}, ParName, SearchPaths, TabWidth, Editor) ->
    ?wrangler_io("\nCMD: ~p:generalise(~p, {~p,~p}, {~p,~p}, ~p,~p,~p).\n",
                 [?MODULE, FileName, Line, Col, Line1, Col1, ParName, SearchPaths, TabWidth]),
    Cmd = "CMD: " ++ atom_to_list(?MODULE) ++ ":generalise(" ++ "\"" ++
          FileName ++ "\", {" ++ integer_to_list(Line) ++ ", " ++ integer_to_list(Col) ++ "}," ++
           "{" ++ integer_to_list(Line1) ++ ", " ++ integer_to_list(Col1) ++ "}," ++ "\"" ++ ParName ++ "\","
      ++ "[" ++ refac_misc:format_search_paths(SearchPaths) ++ "]," ++ integer_to_list(TabWidth) ++ ").",
    case refac_api:is_var_name(ParName) of
        false -> throw({error, "Invalid parameter name!"});
        true -> ok
    end,
    {ok, {AnnAST, Info}} = wrangler_ast_server:parse_annotate_file(FileName, true, SearchPaths, TabWidth),
    {ok, ModName} = get_module_name(Info),
    case interface_api:pos_to_expr(AnnAST, Start, End) of
        {ok, Exp} ->
            Exp;
        {error, _} -> throw({error, "You have not selected an expression, "
                                    "or the function containing the expression does not parse."}),
                      Exp = none
    end,
    case interface_api:expr_to_fun(AnnAST, Exp) of
        {ok, Fun} ->
            Fun;
        {error, _} -> throw({error, "You have not selected an expression within a function."}),
                      Fun = none
    end,
-:--- refac_gen.erl   21% (126,0)   (Erlang Flymake)
```

# Two extensions

## API

Describe entirely new ‘atomic’ refactorings from scratch.

e.g. swap args, delete argument.

## DSL

A language to script composite refactorings on top of simpler ones.

e.g. remove clone, migrate API.

## API

## API design criteria

We assume you can program Erlang …

… but don’t want to learn the internal syntax or details of our representation and libraries.

We aim for simplicity and clarity …

… rather than complete coverage.

# Integration

Describe refactorings by a `behaviour`.

Integration with emacs for execution …

… which gives preview, undo, interactive behaviour etc. "for free".

# Generalisation

Describe expressions in Erlang ...

```
loop_a() ->
    receive
      stop -> ok;
      {msg, _Msg, 0} -> loop_a();
      {msg, Msg, N} ->
       body(Msg,N),
       loop_a()
     end.

body(Msg,N) ->
        io:format("ping!~n"),
        timer:sleep(500),
        b ! {msg, Msg, N - 1}.
```

```
loop_a() ->
    receive
      stop -> ok;
      {msg, _Msg, 0} -> loop_a();
      {msg, Msg, N} ->
       body(Msg,N,"ping!~n"),
       loop_a()
     end.

body(Msg,N,Str) ->
        io:format(Str),
        timer:sleep(500),
        b ! {msg, Msg, N - 1}.
```

# Generalisation

... how expressions are transformed ...

```
loop_a() ->
    receive
      stop -> ok;
      {msg, _Msg, 0} -> loop_a();
      {msg, Msg, N} ->
       body(Msg,N),
       loop_a()
     end.

body(Msg,N) ->
      io:format("ping!~n"),
      timer:sleep(500),
      b ! {msg, Msg, N - 1}.
```

```
loop_a() ->
    receive
      stop -> ok;
      {msg, _Msg, 0} -> loop_a();
      {msg, Msg, N} ->
       body(Msg,N,"ping!~n"),
       loop_a()
     end.

body(Msg,N,Str) ->
      io:format(Str),
      timer:sleep(500),
      b ! {msg, Msg, N - 1}.
```

# Generalisation

... and its context and scope.

```
loop_a() ->
    receive
      stop -> ok;
      {msg, _Msg, 0} -> loop_a();
      {msg, Msg, N} ->
       body(Msg,N),
       loop_a()
     end.

body(Msg,N) ->
      io:format("ping!~n"),
      timer:sleep(500),
      b ! {msg, Msg, N - 1}.
```

```
loop_a() ->
    receive
      stop -> ok;
      {msg, _Msg, 0} -> loop_a();
      {msg, Msg, N} ->
       body(Msg,N,"ping!~n"),
       loop_a()
     end.

body(Msg,N,Str) ->
      io:format(Str),
      timer:sleep(500),
      b ! {msg, Msg, N - 1}.
```

# Generalisation

## Pre-conditions for refactorings

```
loop_a() ->
    receive
      stop -> ok;
      {msg, _Msg, 0} -> loop_a();
      {msg, Msg, N} ->
        body(Msg,N),
        loop_a()
     end.

body(Msg,N) ->
       io:format("ping!~n"),
       timer:sleep(500),
       b ! {msg, Msg, N - 1}.
```

Can't generalise over an expression that contains free variables ...

... or use the same name as an existing variable for the new variable.

## Wrangler API

**Context** available for pre-conditions

**Traversals** describe how rules are applied

**Rules** describe transformations

**Templates** describe expressions

# Templates

Templates are enclosed in the `?T` macro call.

Meta-variables in templates are Erlang variables ending in `@`, e.g. `F@`, `Arg@@`, `Guards@@@`.

```
?T("M:F@(1,2)")

?T("spawn(Args@@)")

?T("spawn(Arg1@,
     Arg2@,Args@@)")
```

`F@` matches a single element.

`Args@@` matches a sequence of elements of some kind.

# Apply function `f: f(1,a,3)`

```
fun test_swap_args:f/3(1,a,3).

fun f/3(1,a,3).

test_swap_args:f(1,a,3).

apply(test_swap_args, f, [1,a,3]).

spawn(test_swap_args, f, [1,a,3]).

As = [1,a,3], apply(test_swap_args, f, As).

apply(fun test_swap_args:f/3, [1,2,3]).
```

Different concrete syntax for application.

Replace with single

`?FUN_APPLY(M,F,A)`

# Rules

```
?RULE(Template, NewCode, Cond)
```

The old code, the new code and the pre-condition.

```
rule({M,F,A}, N) ->
 ?RULE(?T("F@(Args@@)"),
       begin
         NewArgs@@=delete(N, Args@@),
         ?TO_AST("F@(NewArgs@@)")
       end,
       refac_api:fun_define_info(F@) == {M,F,A}).

delete(N, List) ->  … delete Nth elem of List …
```

# Information in the AAST

Wrangler uses the `syntax_tools` AST, augmented with information about the program semantics.

API functions provide access to this.

Variables bound, free and visible at a node.

Location information.

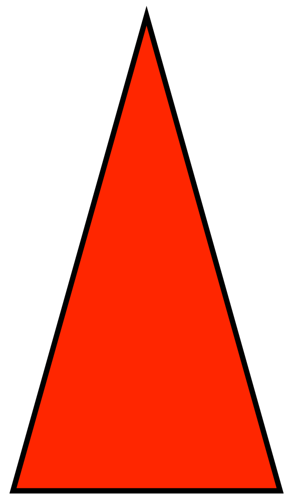
All bindings (if a vbl).

Where defined (if a fn).

Atom usage info: name, function, module etc.

Process info …

## Traversals



`?FULL_TD_TP(Rules, Scope)`

- Traverse top-down
- At each node, apply first of `Rules` to succeed …
- `TP` = “Type preserving”.

```
-module(refac_swap_args).

-behaviour(gen_refac).

-export([…]).

-include("../include/gen_refac.hrl").

-import(refac_api, [fun_define_info/1]).

input_pars() -> ["Parameter Index 1: ", "Parameter Index 2: "].

select_focus(_Args=#args{current_file_name=File,
                         cursor_pos=Pos}) ->
    interface_api:pos_to_fun_def(File, Pos).


pre_cond_check(_Args=#args{focus_sel=FunDef,
                           user_inputs=[I, J]}) ->
    …
         true ->
                ok;
         false ->
                {error, "Index 1 and Index 2 are the same."} …
    .

transform(Args=#args{current_file_name=File,focus_sel=FunDef,
                      user_inputs=[I, J]}) ->
    …
    {ok, Res}=transform_in_cur_file(Args, {M,F,A}, I1, J1),
    case refac_api:is_exported({F,A}, File) of
        true ->
            {ok, Res1}=transform_in_client_files(Args, {M,F,A}, I1, J
        …
    end.
```

Behaviour gen_refac what **callbacks** a refactoring should provide.

input_pars: prompts for interactive inputs

select_focus: what to do with the editor focus information.

pre_cond_check: check preconditions

transform: if the pre-condition is ok, do the transform.

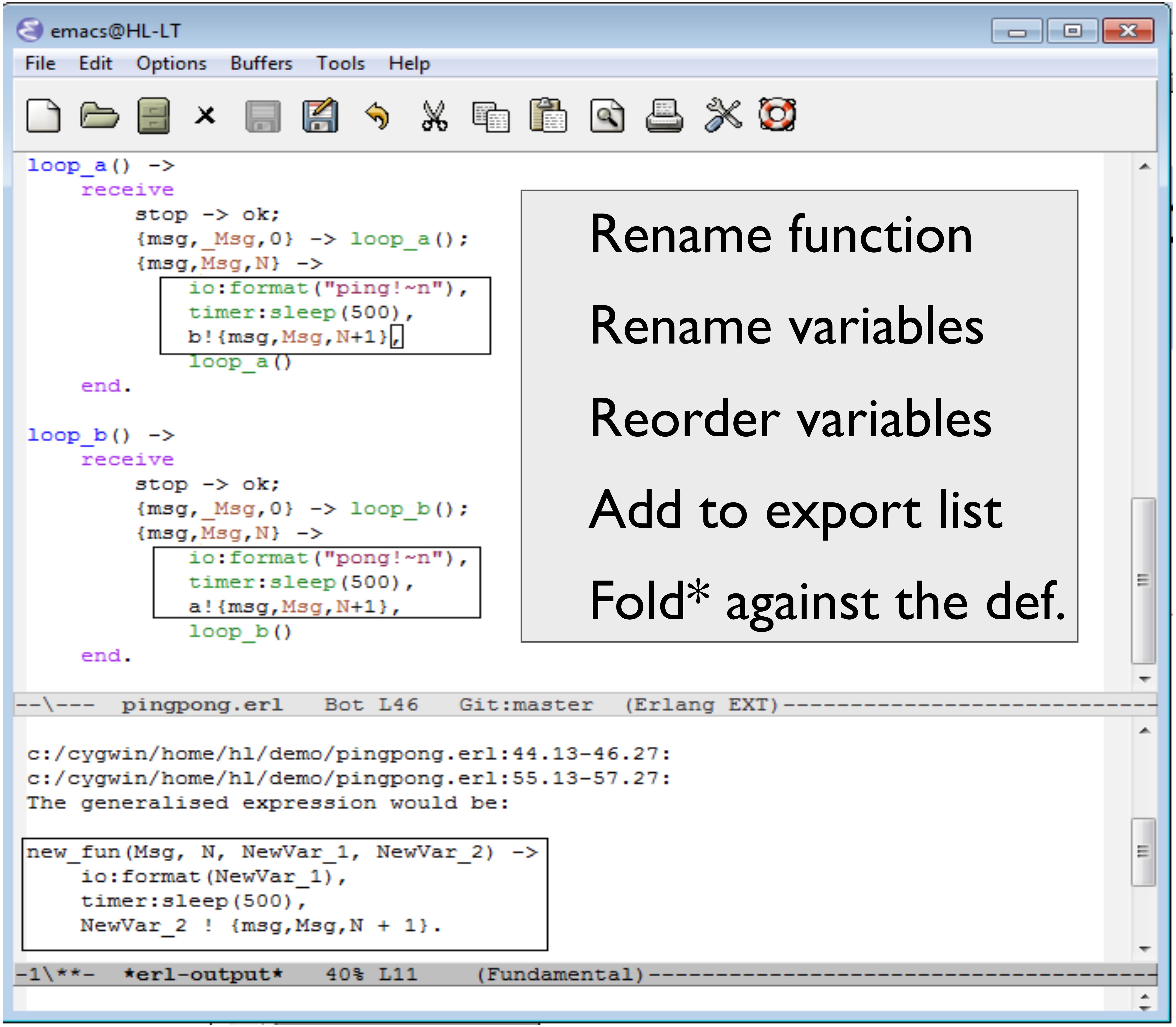
## DSL

# Composite refactorings

A sequence of simpler refactorings which together achieve a complex effect.

Example: transform all `camelCase` identifiers within a project into `camel_case`.

*How We Refactor, and How We Know It*
Murphy-Hill, Parnin & Black, *ICSE*, 2009.



```
loop_a() ->
    receive
        stop -> ok;
        {msg,_Msg,0} -> loop_a();
        {msg,Msg,N} ->
            io:format("ping!~n"),
            timer:sleep(500),
            b!{msg,Msg,N+1},
            loop_a()
    end.

loop_b() ->
    receive
        stop -> ok;
        {msg,_Msg,0} -> loop_b();
        {msg,Msg,N} ->
            io:format("pong!~n"),
            timer:sleep(500),
            a!{msg,Msg,N+1},
            loop_b()
    end.
--\---  pingpong.erl    Bot L46    Git:master   (Erlang EXT)----------------------------

c:/cygwin/home/hl/demo/pingpong.erl:44.13-46.27:
c:/cygwin/home/hl/demo/pingpong.erl:55.13-57.27:
The generalised expression would be:

new_fun(Msg, N, NewVar_1, NewVar_2) ->
    io:format(NewVar_1),
    timer:sleep(500),
    NewVar_2 ! {msg,Msg,N + 1}.
-1\**-  *erl-output*    40% L11     (Fundamental)----------------------------------------
```

Rename function

Rename variables

Reorder variables

Add to export list

Fold* against the def.

# Not just a script ...

Tracking changing names and positions.

Generating refactoring commands.

Dealing with failure.

User control of execution.

… we're dealing with the *pragmatics* of composition, rather than just the theory.

# Generators

Refactoring functions modified to take *descriptions* of arguments, rather than concrete arguments.

```
rename_fun(Module,{Fun,Arity},NewName) -> ok | error

rename_fun(fun(Module) -> boolean(),
           fun({Fun,Arity}) -> boolean(),
           fun(Module,{Fun,Arity}) -> atom(),
           boolean())
          ->
          { [ {refactoring, rename_fun, Args} ], fun}
```

# Generation: camel case

```
?refac_(CmdName, Args, Scope)
```

Args: modules, camelCase functions, new names.

```
?refac_(rename_fun,
          [{file, fun(_File)-> true end},
           fun({F, _A}) ->
                    camelCase_to_camel_case(F) /= F
           end,
           {generator, fun({_File, F,_A}) ->
                               camelCase_to_camel_case(F)
                       end}],
        SearchPaths).
```

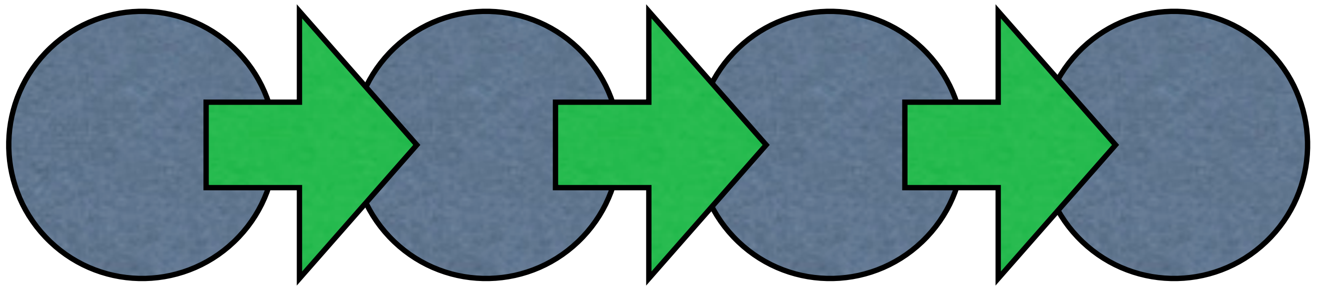
# Automation

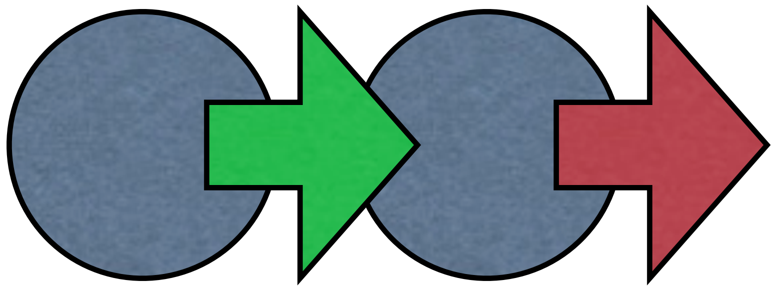
Don't have to describe each command explicitly: allow *conditions* and *generators*.

Allow lazy generation … return a refactoring command together with a *continuation*.

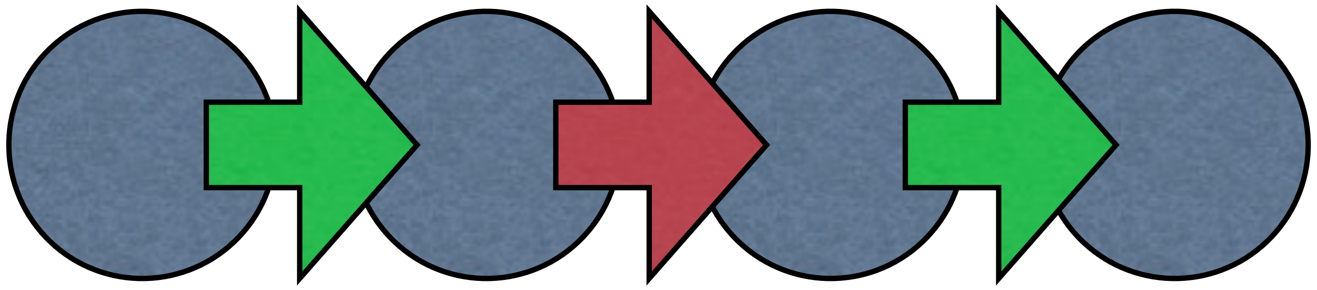
*Track names*, so that `?current(foo)` gives the 'current' name of an entity `foo` at any point in the refactoring.

## Handling failure



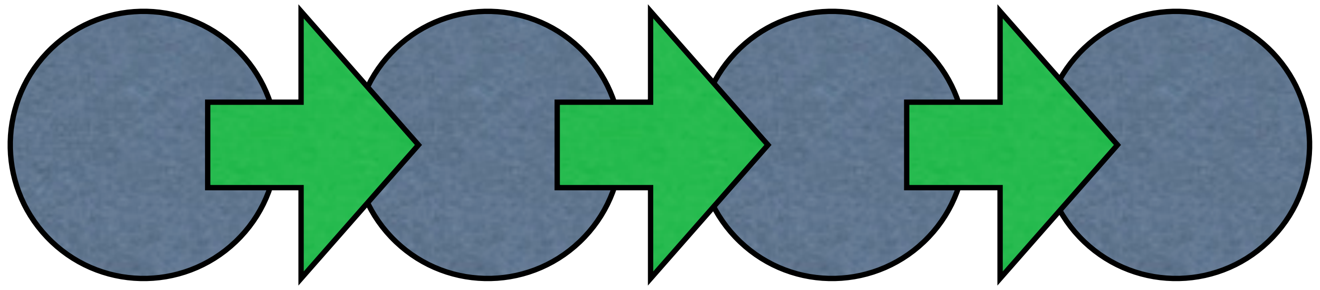


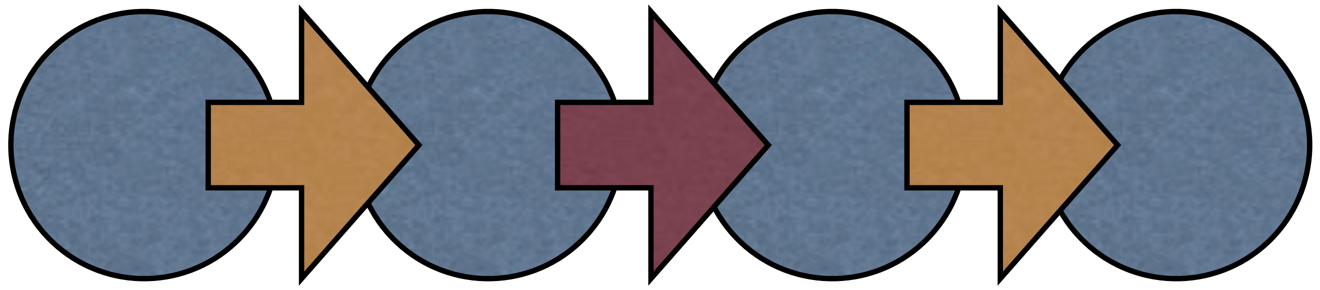
Atomic: if one part fails, *abandon* the whole.



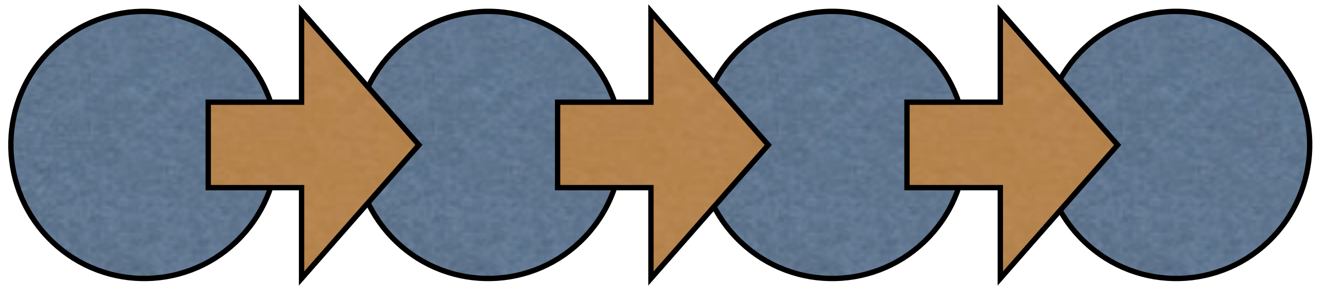
Non-atomic: *continue* even when failure.

# Handling interaction



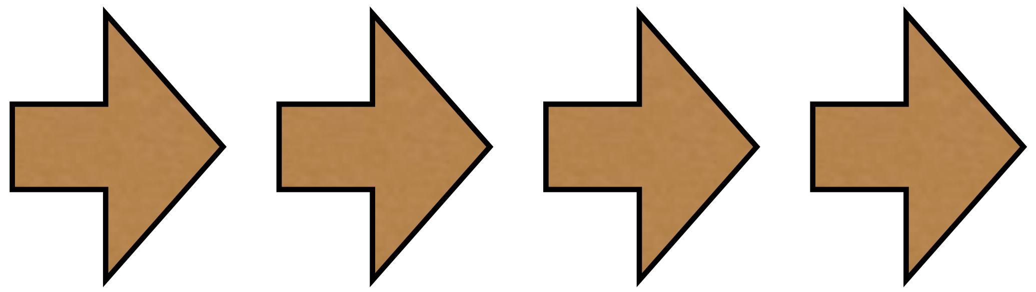


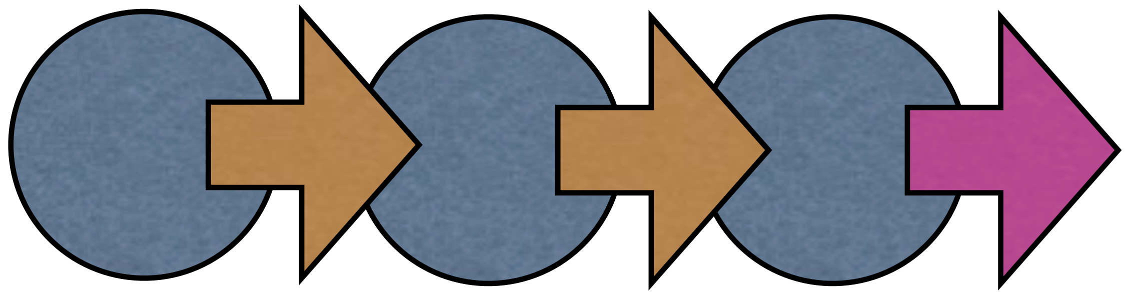
Interactive: *choose* the cases to perform.



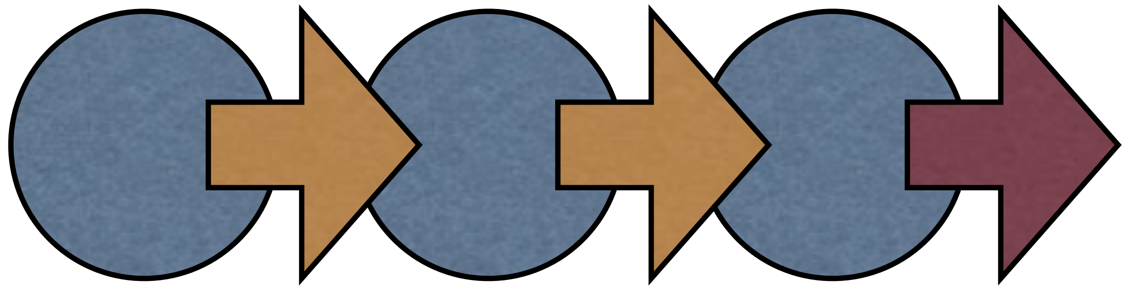
Non-interactive: *perform all cases.*

# Handling repetition





Condition: *repeat* while
a condition is true.



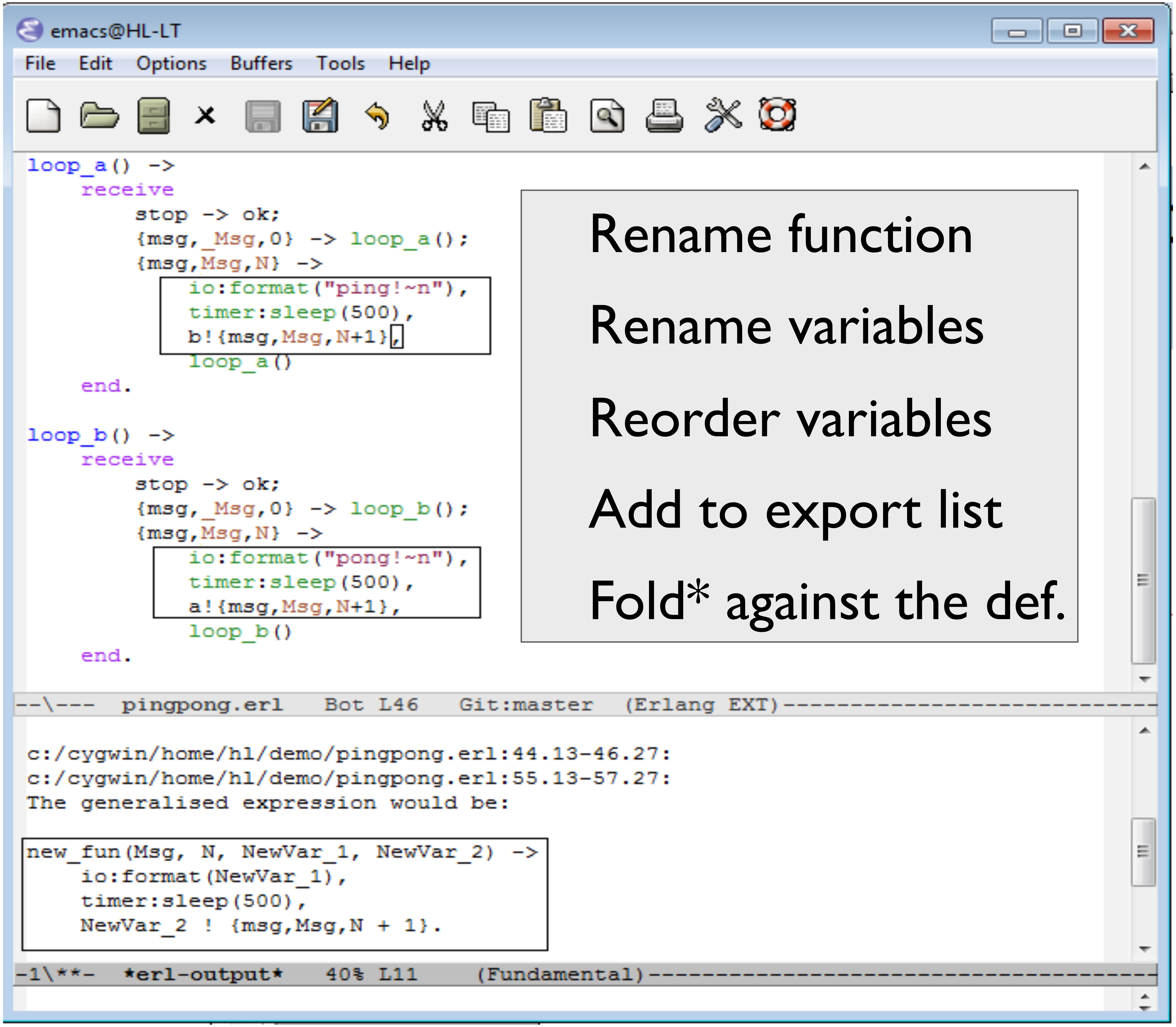
Interactively:
*ask* whether to repeat.

## Building a DSL

Domain specific language to support options of atomicity, interactivity etc.

*Embed* in Erlang to leverage the language e.g. to define conditions and generators.

Use Erlang to represent the language, and *macros* to support.



```
loop_a() ->
    receive
        stop -> ok;
        {msg,_Msg,0} -> loop_a();
        {msg,Msg,N} ->
            io:format("ping!~n"),
            timer:sleep(500),
            b!{msg,Msg,N+1},
            loop_a()
    end.

loop_b() ->
    receive
        stop -> ok;
        {msg,_Msg,0} -> loop_b();
        {msg,Msg,N} ->
            io:format("pong!~n"),
            timer:sleep(500),
            a!{msg,Msg,N+1},
            loop_b()
    end.
--\---  pingpong.erl   Bot L46   Git:master  (Erlang EXT)---------------------------

c:/cygwin/home/hl/demo/pingpong.erl:44.13-46.27:
c:/cygwin/home/hl/demo/pingpong.erl:55.13-57.27:
The generalised expression would be:

new_fun(Msg, N, NewVar_1, NewVar_2) ->
    io:format(NewVar_1),
    timer:sleep(500),
    NewVar_2 ! {msg,Msg,N + 1}.
-1\**-  *erl-output*   40% L11    (Fundamental)-----------------------------------
```

Rename function

Rename variables

Reorder variables

Add to export list

Fold* against the def.

# Clone removal: top level

Atomic as a whole … non-atomic components OK.

Repetition, interactivity, conditionality.

```
?atomic([?interactive( RENAME FUNCTION )
         ?refac_( RENAME ALL VARIABLES OF THE
                  FORM NewVar* )
         ?repeat_interactive( SWAP ARGUMENTS )
         ?if_then( EXPORT IF NOT ALREADY )
         ?non_atomic( FOLD INSTANCES OF THE CLONE )
        ]).
```

# Erlang and DSL

```
?refac_(rename_var,
        [M,
         begin
           {_, F1, A1} = ?current(M,F,A),
           {F1, A1}
         end,
         fun(X) ->
             re:run(atom_to_list(X), "NewVar*")/=nomatch
         end,
         {user_input, fun({_, _, V}) ->
                          lists:flatten(io_lib:format
                                  "Rename variable ~p to: ", [V]))
                      end},
         SearchPaths])
```

## Demo

## Remove bug preconditions

*Scenario*: building Erlang models for C code.

For buggy code, want to avoid hitting the same bugs all the time

Add bug precondition macros …

… but want to remove in delivered code.

*DSL*: fuses 3 steps. *API*: first two. *Built in*: third.

# Step 1: simple rules

```
replace_bug_cond_macro_rule() ->
    ?RULE(?T("Expr@"),
          ?TO_AST("false"),
          is_bug_cond_macro(Expr@)).

logic_rule_1() ->
    ?RULE(?T("not false"),?TO_AST("true"),true).
```

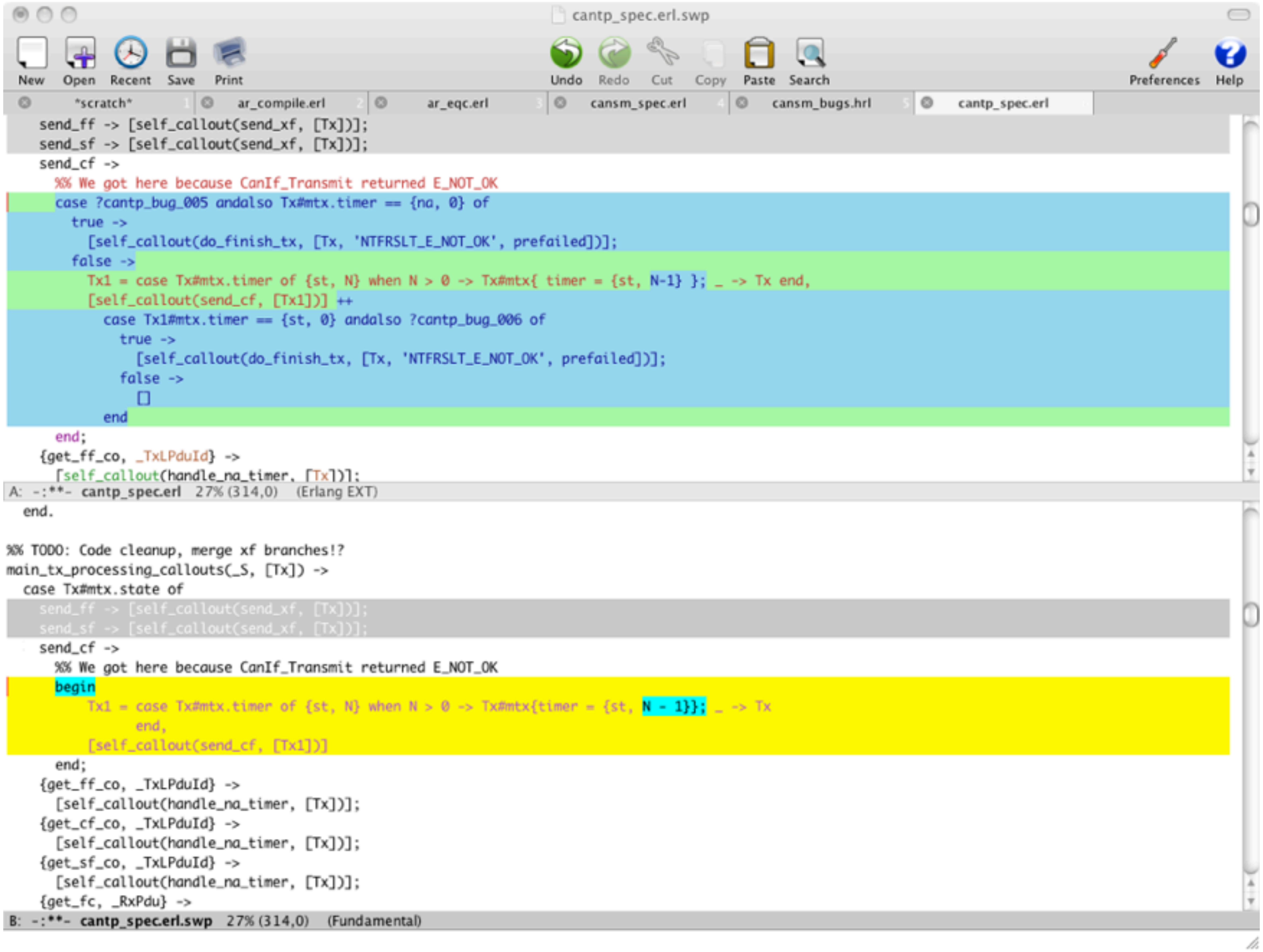
## Step 2: tidy up

```
case false of
  true  -> com_cfg:initial_value(Sig);
  false -> get_shadow_value(Id, S)
end.
```

Simplifies to `get_shadow_value(Id, S)`.

# Step 3: inline variables

```
route_data_next(S,_, [{{SrcKind,SrcId}, Dst, Val}], _) ->
    % clear gateway pending flag
    NewS = set_gateway_pending(S, SrcKind, SrcId, false),
    S2   = NewS,
    copy_to_destination(S2, Dst, Val).
```

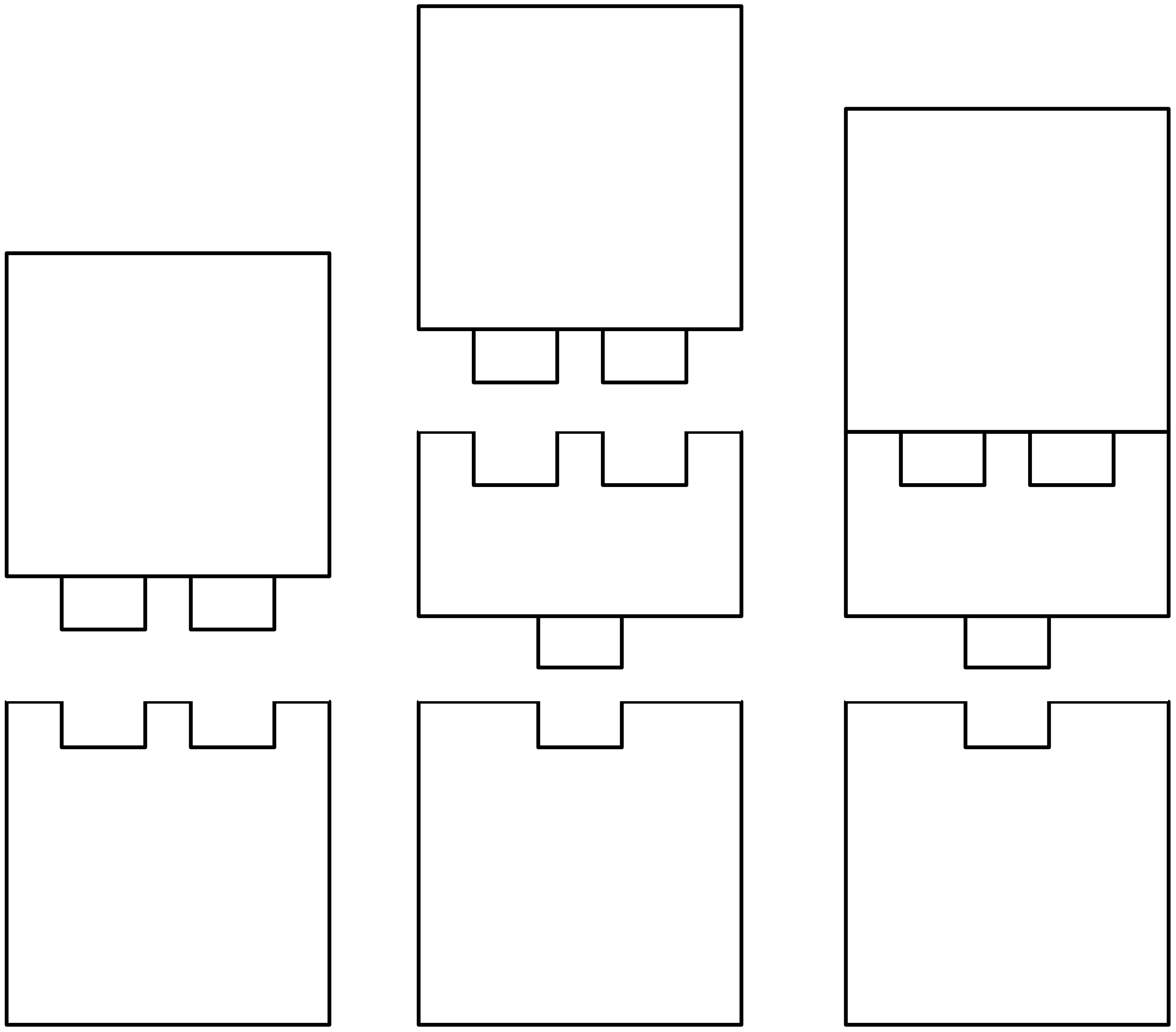


```
  send_ff -> [self_callout(send_xf, [Tx])];
  send_sf -> [self_callout(send_xf, [Tx])];
  send_cf ->
   %% We got here because CanIf_Transmit returned E_NOT_OK
   case ?cantp_bug_005 andalso Tx#mtx.timer == {na, 0} of
    true ->
     [self_callout(do_finish_tx, [Tx, 'NTFRSLT_E_NOT_OK', prefailed])];
    false ->
     Tx1 = case Tx#mtx.timer of {st, N} when N > 0 -> Tx#mtx{ timer = {st, N-1} }; _ -> Tx end,
     [self_callout(send_cf, [Tx1])] ++
      case Tx1#mtx.timer == {st, 0} andalso ?cantp_bug_006 of
       true ->
        [self_callout(do_finish_tx, [Tx, 'NTFRSLT_E_NOT_OK', prefailed])];
       false ->
        []
      end
   end;
  {get_ff_co, _TxLPduId} ->
   [self_callout(handle_na_timer, [Tx])];
A: -:**- cantp_spec.erl  27% (314,0)  (Erlang EXT)
 end.

%% TODO: Code cleanup, merge xf branches!?
main_tx_processing_callouts(_S, [Tx]) ->
 case Tx#mtx.state of
  send_ff -> [self_callout(send_xf, [Tx])];
  send_sf -> [self_callout(send_xf, [Tx])];
  send_cf ->
   %% We got here because CanIf_Transmit returned E_NOT_OK
   begin
     Tx1 = case Tx#mtx.timer of {st, N} when N > 0 -> Tx#mtx{timer = {st, N - 1}}; _ -> Tx
        end,
     [self_callout(send_cf, [Tx1])]
   end;
  {get_ff_co, _TxLPduId} ->
   [self_callout(handle_na_timer, [Tx])];
  {get_cf_co, _TxLPduId} ->
   [self_callout(handle_na_timer, [Tx])];
  {get_sf_co, _TxLPduId} ->
   [self_callout(handle_na_timer, [Tx])];
  {get_fc, _RxPdu} ->
B: -:**- cantp_spec.erl.swp  27% (314,0)  (Fundamental)
```

# API migration

Scenario: system upgrade accompanied with a change in API, e.g. `regexp` to `re`.

How to refactor client code to accommodate this?



# regexp to re

```
match(String, RegExp) ->
    try re:run(String, RegExp, [global]) of
         {match, Match} ->
            {Start0, Len} = lists:last(
                                 lists:ukeysort(
                                   2, lists:append(Match))),
            Start = Start0+1,
            {match, Start, Len};
        nomatch -> nomatch
    catch
        error:_->
            {error, Error}=re:compile(RegExp),
            {error, Error}
    end.
```

## API migration

We generate a set of refactorings for client code from the adapter functions.

Look at a representative set of examples from recent versions of OTP.

## Demo

# What next?

Refining the detailed design.

User contributions.

Application to other languages …

# Questions?

www.cs.kent.ac.uk/projects/wrangler

ProTest
property based testing