Getting the right module structure: using Wrangler to fix your projects

Simon Thompson, Huiqing Li
School of Computing, University of Kent, UK
Overview

Refactoring Erlang in Wrangler
Clone detection and elimination
Case study: SIP message manipulation
Improving module structure
Introduction
Refactoring means changing the design or structure of a program … without changing its behaviour.
Soft-ware

There’s no single correct design …

… different options for different situations.

Maintain flexibility as the system evolves.
From order to chaos ...

The best designs decay ...

- Clones
- Module structure "bad smells".
- ...

ProTest

property based testing

University of Kent
Computing
Generalisation and renaming

-module (test).
-export([f/1]).

add_one ([H|T]) ->
[H+1 | add_one(T)];

add_one ([]) -> [].

f(X) -> add_one(X).

-module (test).
-export([f/1]).

add_int (N, [H|T]) ->
[H+N | add_int(N,T)];

add_int (N,[]) -> [].

f(X) -> add_int(1, X).
Refactoring tool support

Bureaucratic and diffuse.

Tedious and error prone.

Semantics: scopes, types, modules, ...

Undo/redo

Enhanced creativity
Wrangler

Refactoring tool for Erlang

Integrated into Emacs and Eclipse / ErlIDE

Multiple modules

Structural, process, macro refactorings
Wrangler

Duplicate code detection …
… and elimination

Explore and improve module structure

Testing / refactoring

Property discovery

Clone detection + removal

Improve module structure

Basic refactorings
Architecture of Wrangler
local_data = {\{mux_id_1, h223_id_1\},
            \{TdmSid, LocalData, .., \} = precond_one_blade_tdm_mux_create(SidMux),
            ?CHECK(\{ok, \}, hcfTraceServerSupport, start, [[{brchDspRt, exported}]]),

% Clean up this test case
% clean_up([SidLc, SidMux, TdmSid]),
%          \{\}.
% create_5(doc) -> "Create basic VIG MUX + video + audio + TDM unframed device";
% create_5(setuping) -> "";
% create_5(name) -> /\{00,00,00,00\};
% create_5(config) -> \{\};
% create_5(main) -> \{\}.

% Test case create_5 started

SidMux = {mux_id_1, h223_id_1}.
Demo
Clone detection
Duplicate code considered harmful

It’s a *bad smell* …

• increases chance of bug propagation,
• increases size of the code,
• increases compile time, and,
• increases the cost of maintenance.

But … it’s not always a problem.
Clone detection

• Hybrid clone detector
  – relatively efficient (suffix tree)
  – no false positives (AST analysis)
• User-guided interactive removal of clones.
• Integrated into development environments.
What is ‘identical’ code?

Identical if values of literals and variables ignored, but respecting binding structure.
What is ‘similar’ code?

The anti-unification gives the (most specific) common generalisation.
<table>
<thead>
<tr>
<th>Detection</th>
<th>Expression search</th>
</tr>
</thead>
<tbody>
<tr>
<td>All clones in a project meeting the threshold parameters …</td>
<td>All instances of expressions similar to this expression …</td>
</tr>
<tr>
<td>… and their common generalisations.</td>
<td>… and their common generalisation.</td>
</tr>
<tr>
<td>Default threshold: ≥ 5 expressions and similarity of ≥ 0.8.</td>
<td>Default threshold: ≥ 20 tokens.</td>
</tr>
</tbody>
</table>
Similarity

Threshold: anti-unifier should be big enough relative to the class members:

\[
\text{similarity} = \min\left( \frac{\| X+Y \|}{\| (X+3)+4 \|}, \frac{\| X+Y \|}{\| 4+(5-(3\times X)) \|} \right)
\]

Can also threshold \( \| X+Y \| \) length of expression sequence, or number of tokens, or … .
Example: clone candidate

S1 = "This",
S2 = " is a ",
S3 = "string",
[S1,S2,S3]

S1 = "This",
S2 = "is another ",
S3 = "String",
[S3,S2,S1]

D1 = [1],
D2 = [2],
D3 = [3],
[D1,D2,D3]

D1 = [X+1],
D2 = [5],
D3 = [6],
[D3,D2,D1]

? = ?,
? = ?,
? = ?,
[?,?,?,?]

ProTest
property based testing

University of Kent
Computing
Example: clone from sub-sequence

S1 = "This",
S2 = " is a ",
S3 = "string",
[S1,S2,S3]

S1 = "This",
S2 = "is another ",
S3 = "String",
[S3,S2,S1]

D1 = [1],
D2 = [2],
D3 = [3],
[D1,D2,D3]

D1 = [X+1],
D2 = [5],
D3 = [6],
[D3,D2,D1]

new_fun(NewVar_1,
NewVar_2,
NewVar_3) ->

S1 = NewVar_1,
S2 = NewVar_2,
S3 = NewVar_3,
{S1,S2,S3}.
Example: sub-clones

S1 = "This", S1 = "This", D1 = [1], D1 = [X+1],
S2 = " is a ", S2 = "is another ", D2 = [2], D2 = [5],
S3 = "string", S3 = "String", D3 = [3], D3 = [6],
[S1,S2,S3] [S3,S2,S1] [D1,D2,D3] [D3,D2,D1]

new_fun(NewVar_1, NewVar_2, NewVar_3) ->
S1 = NewVar_1, S1 = NewVar_1,
S2 = NewVar_2, S2 = NewVar_2,
S3 = NewVar_3, S3 = NewVar_3,
[S1,S2,S3]. [S3,S2,S1].
Demo
SIP Case Study
Why test code particularly?

Many people touch the code.

Write some tests … write more by copy, paste and modify.

Similarly with long-standing projects, with a large element of legacy code.
“Who you gonna call?”

Can reduce by 20% just by aggressively removing all the clones identified …

… what results is of no value at all.

Need to call in the domain experts.
SIP message manipulation allows rewriting rules to transform messages.

Test by `smm_SUITE.erl`, 2658 LOC.

2658 to 2042 in twelve steps.
Step 1

The largest clone class has 15 members.

The suggested function has no parameters, so the code is literally repeated.
The largest clone has 88 lines, and 2 parameters.

But what does it represent?

What to call it?

Best to work bottom up.
The general pattern

Identify a clone.

Introduce the corresponding generalisation.

Eliminate all the clone instances.

So what’s the complication?
What is the complication?

Which clone to choose?
Include all the code?
How to name functions and variables?
When and how to generalise?
'Widows' and 'orphans'
Module structure inspection
Modularity "Bad Smells"

• Module structure deteriorates over time during development.
• This can be avoided by incremental modularity maintenance.
• Not a "push button" operation …
• … need to know both the problem domain and the program.
Modularity Smells

- Cyclic module dependency.
- Export of functions that are meant to be used internally.
- Module with multiple purposes.
- Very large modules.
Modularity Smell Elimination

• Key refactoring …

  Move function(s) from one module to another.

• … but, which functions to move, and to where?

• Wrangler aims to detect modularity smells and give refactoring suggestions.
Wrangler module graph
Wrangler cycles
Ibrowse cycles

dobrace(0),
dotrace/2, get_value/3,
parse_url/1.
ibrowse_lib(0)

ibrowse(1),
spawn_connection/5, start_link/1,
get_config_value/2
ibrowse_lib(0)

ibrowse(2),
send_req/7, start/1,
start_link/1, stop/1,
get_config_value/2
ibrowse_http_client(0)
ibrowse

spawn_connection/5,
  start_link/1.

send_req/7,start/1,
  start_link/1,stop/1.

ibrowse_lib

start_link/1.

get_trace_status/2,
  get_config_value/2.
  do_trace/2,
  get_value/3,
  parse_url/1,
  get_config_value/2.

ibrowse_http_client

dec2hex/2,do_trace/2,
  encode_base64/1,
  get_trace_status/2,
  get_value/2,get_value/3,
  get_config_value/2.

ibrowse_lib
Cyclic Module Dependency

- Reasons for cyclic module dependency:
  - Mutual recursive function definition across multiple modules.
  - API Functions from different logical layers of the system coexist in the same module.
- Some cyclic module dependencies might be legitimate.
Some CouchDB cycles
Some terminology

- **Intra-layer dependency**: mutually recursive functions across multiple modules.
- **Inter-layer dependency**: mutually recursive modules, but not mutually recursive functions.
Resolving inter-module cycle

-module(m1).
-export([foo/0,bar/0]).

foo() -> 1.
bar() -> m2:blah().

-module(m2).
-export([blah/0]).

blah() -> m1:foo().

-module(m3).
-export([foo/0]).

foo() -> 1.
Resolving inter-module cycle

-module(m1).
-export([foo/0, bar/0]).
foo() -> 1.
bar() -> m2:blah().

-module(m2).
-export([blah/0]).
blah() -> m1:foo().

-module(m3).
-export([bar/0]).
bar() -> m2:blah().
Cyclic Module Dependency

- For each cyclic module dependency, Wrangler gives refactoring suggestions.

  e.g.

  Inter-layer cyclic module dependency: [refac_prettypr, refac_util]
  Refactoring suggestion:
  move_fun(refac_util, [{write_refactored_files, 1},
                        {write_refactored_files, 3}, {write_refactored_files, 4}],
                        user_supplied_target_mod).
Identifying "API" functions

• Identify by examining call graph.

• API functions are those …
  • … not used internally,
  • … "close to" other API functions

• Others are seen as internal, external calls to these are deemed improper.
Improper inter-module calls

wrangler_code_inspector:improper_inter_module_calls("/Users/simonthompson/Desktop/improper_module_dependency.dot", ["/Users/simonthompson/erlang/systems/wrangler-0.8.8/src"]).

Refactoring suggestions:
refac_move_fun:move_fun({refac_register_pid,spawn_funs,0},[refac_syntax_lib,refac_misc,refac_annotate_pid,refac_slice,refac_syntax,ast_traverse_api,interface_api,refac_util]).
Large Modules

• A module should not contain more than 400 lines of source code according to the Erlang programming rules.

• A very large module is likely to serve more than one purpose or contain too many internal functions.
Large Modules

- A large module could be partitioned into two or more smaller modules.
Large Modules

- Partition the exports of a module into groups using similarity metrics, each group forms an export attribute.
- Agglomerative hierarchical algorithm using Jaccard similarity coefficient.
- Functions specified in an export attribute can be moved to another module together.
Demo
Going further
Property discovery in Wrangler

Find (test) code that is similar ...

... build a common abstraction

... accumulate the instances

... and generalise the instances.

Example:

Test code from Ericsson: different media and codecs.

Generalisation to all medium/codec combinations.
www.cs.kent.ac.uk/projects/wrangler/
→ GettingStarted
ProTest
property based testing
Next steps

Refine the notion of similarity …
… to take account of insert / delete in command seqs.

Scaling up: look for incremental version; check vs. libraries …

Refactorings of tests and properties themselves.

Extracting FSMs from sets of tests.

Support property extraction from 'free' and EUnit tests.
Systems test: FSM discovery

Use FSM to model expected behaviour.

Test random paths through the FSM to test system function.

Extract the FSM from sets of existing test cases.

Use +ve and -ve cases.
Refactoring and testing

Refactor tests e.g.
- Tests into EUnit tests.
- Group EUnit tests into a single test generator.
- Move EUnit tests into a separate test module.
- Normalise EUnit tests.
- Extract setup and tear-down into EUnit fixtures.

Respect test code in EUnit, QuickCheck and Common Test … … and refactor tests along with refactoring the code itself.
www.cs.kent.ac.uk/projects/wrangler/
→ GettingStarted