Evolving your Projects with Wrangler

Simon Thompson and Huiqing Li
University of Kent, UK

Wrangler is a tool

Tool for refactoring Erlang programs.

Inside Emacs and ErlIDE.

Wrangler is a toolkit

Usable from the Erlang shell command line.

An open API for new refactorings …

… and a DSL for scripting complex changes.

API migration tool (e.g. `regexp` to `re`).

Wrangler is a toolset

Clone detection and module “bad smells”.

Plays with testing tools.

Laboratory: symbolic evaluation, slicing, concurrency, parallelisation.

erlang

rerlang

rangler

Wrangler

wrangler | ˈraŋglə |

noun

1 N. Amer. a person in charge of horses or other livestock on a ranch.

• a person who trains and takes care of animals on a film set. *they had three cow wranglers to help with the scene.*

2 a person engaging in a lengthy and complicated dispute. *he was known as the wrangler for the aplomb with which he skewered the professors.*

3 (at Cambridge University) a person placed in the first class of the mathematical tripos.

Wrangler is a tool

Refactoring

"Change how the program works, but not what it does."

Part of the programmer's standard toolkit.

Renaming, function extraction, generalisation.

Refactoring tools

Avoid the *"tedious and error-prone"*.

Keep it simple.

Cover the bureaucracy.

Whole language …

… whole projects,

… plus tests, …

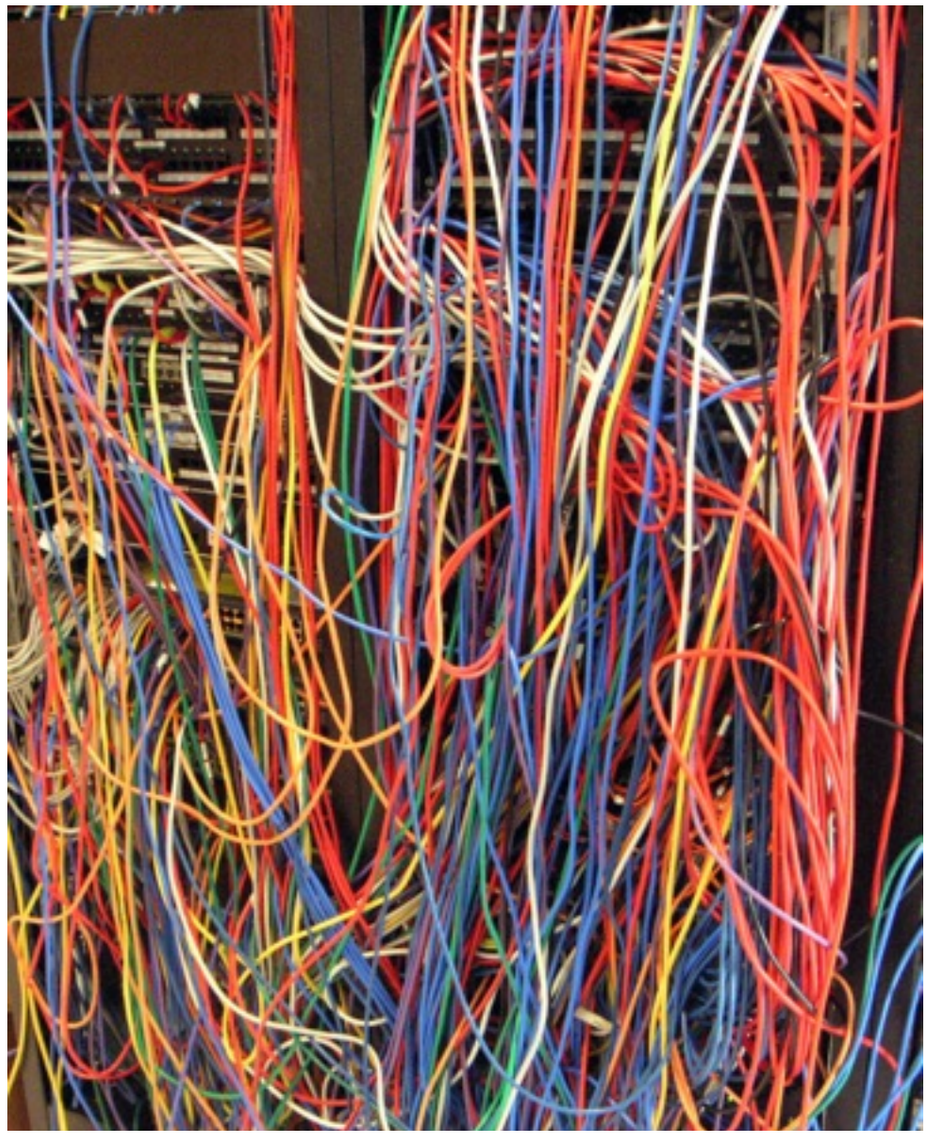
Demo

Rename,
function extraction,
generalisation,
rename again.

Wrangler is a toolkit

Wrangler is a toolkit

Wrangler is written in Erlang, all the way down, so you can extend it yourself …



Wrangler is a toolkit

… but we have given you tools to make that extension much easier to deal with.

Key idea

You know Erlang …

… you don't want to have to learn a whole new language or compiler internals to get the job done.

API and DSL

An API to define a completely new refactorings from scratch

… using Erlang concrete syntax,

… also “code inspection”.

DSL for scripting refactorings

… “on steroids”

… embedded in Erlang.

What we've been asked for

`camelCase` to `camel_case`

Batch module renaming

Removing "bug preconditions" for Quviq.

API migration … for example, `regexp` to `re`.

Wrangler API

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 is used to define preconditions

Traversals describe how transformations are applied

Rules describe transformations

Templates describe expressions

Scenario: `maps` in R17

Can we use `maps` in our project?

Look for uses of `dict` … are they `map`-like?

Finding non `map`-like uses of `dict`

```
find_non_map_like_dict(input_par_prompts) ->
    [];
find_non_map_like_dict(_Args=#args{search_paths=SearchPaths}) ->
     ?FULL_TD_TU(
[?COLLECT(
    ?T("F@(Args@@)"),
    {_File@, api_refac:start_end_loc(_This@)},
    lists:member(
      api_refac:fun_define_info(F@),
      non_map_like_dict())
   )],
SearchPaths).
```

Results for `rabbitmq-server`

10 code fragments satisfying the conditions have been found.

…/rabbit_binding.erl:506.5-507.33:

…/rabbit_binding.erl:510.5-511.38:

…/rabbit_channel.erl:1213.14-1216.63:

…/rabbit_mirror_queue_slave.erl:612.10-612.78:

…/rabbit_misc.erl:759.5-759.69:

…/rabbit_msg_store.erl:1247.15-1248.56:

…/supervisor2.erl:1157.39-1157.69:

…/supervisor2.erl:1168.39-1168.69:

…/supervisor2.erl:1195.41-1195.73:

…/supervisor2.erl:1210.41-1210.73:

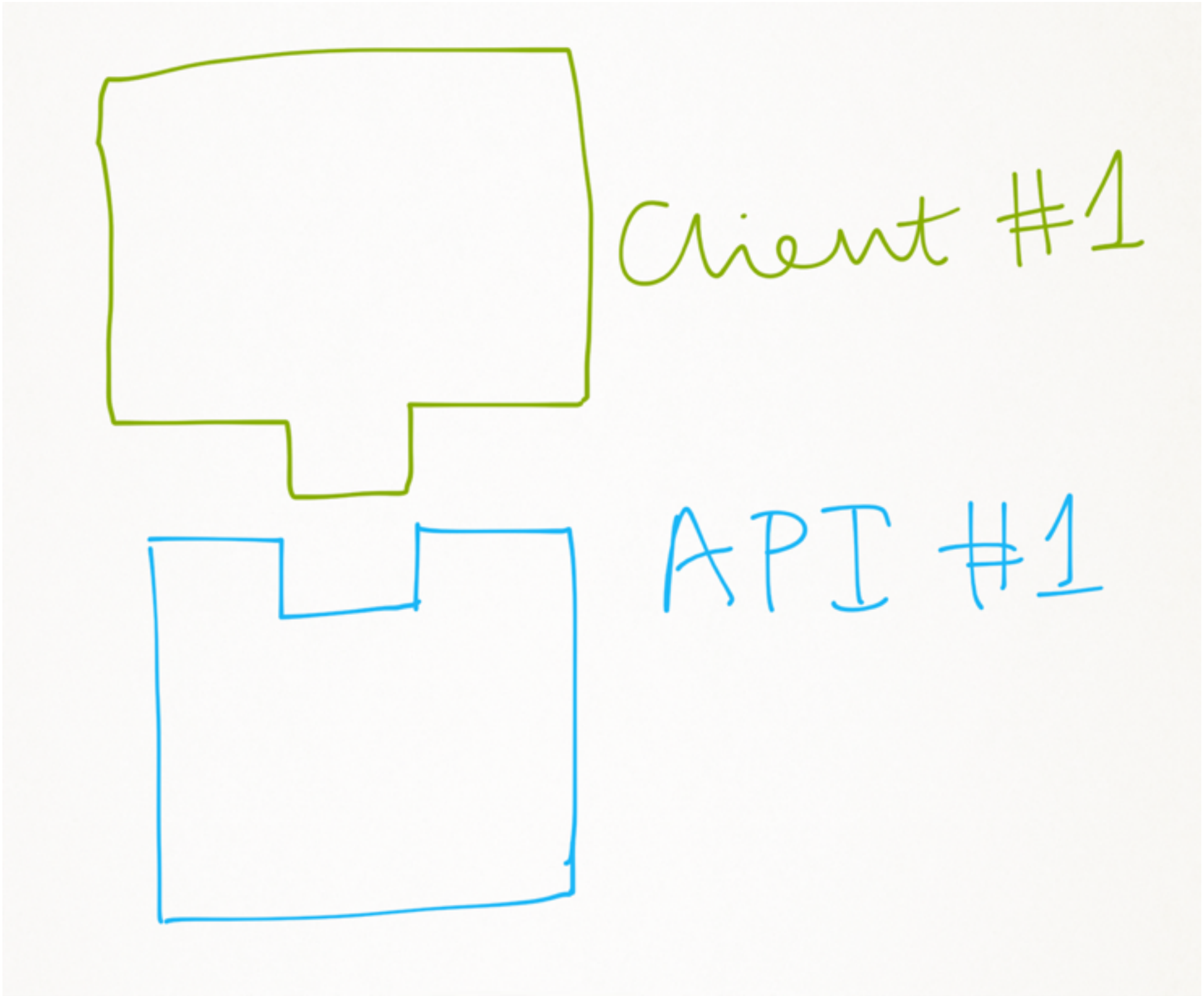
And map-like?

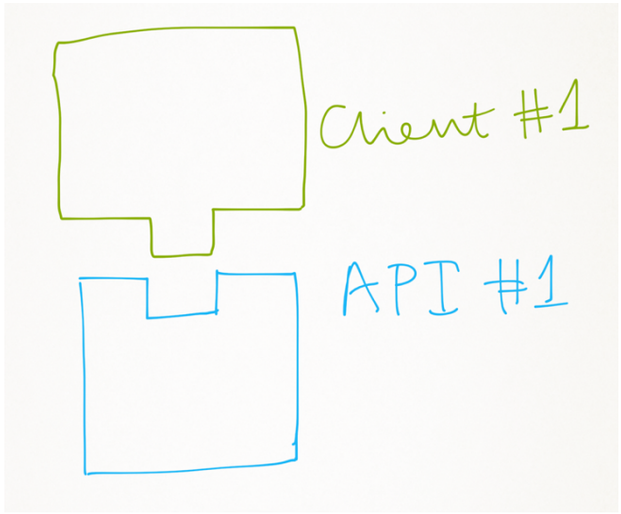
```
163 code fragments satisfying the conditions have been found.


…/delegate.erl:185.43-185.52:

…/delegate.erl:194.22-194.45:

…/delegate.erl:197.25-197.63:

…/delegate.erl:201.25-201.62:

…/delegate.erl:207.22-207.45:

…/delegate.erl:212.38-212.62:

…/delegate.erl:213.38-213.76:

…/delegate.erl:227.11-227.34:

…/delegate.erl:232.37-232.61:

……
```

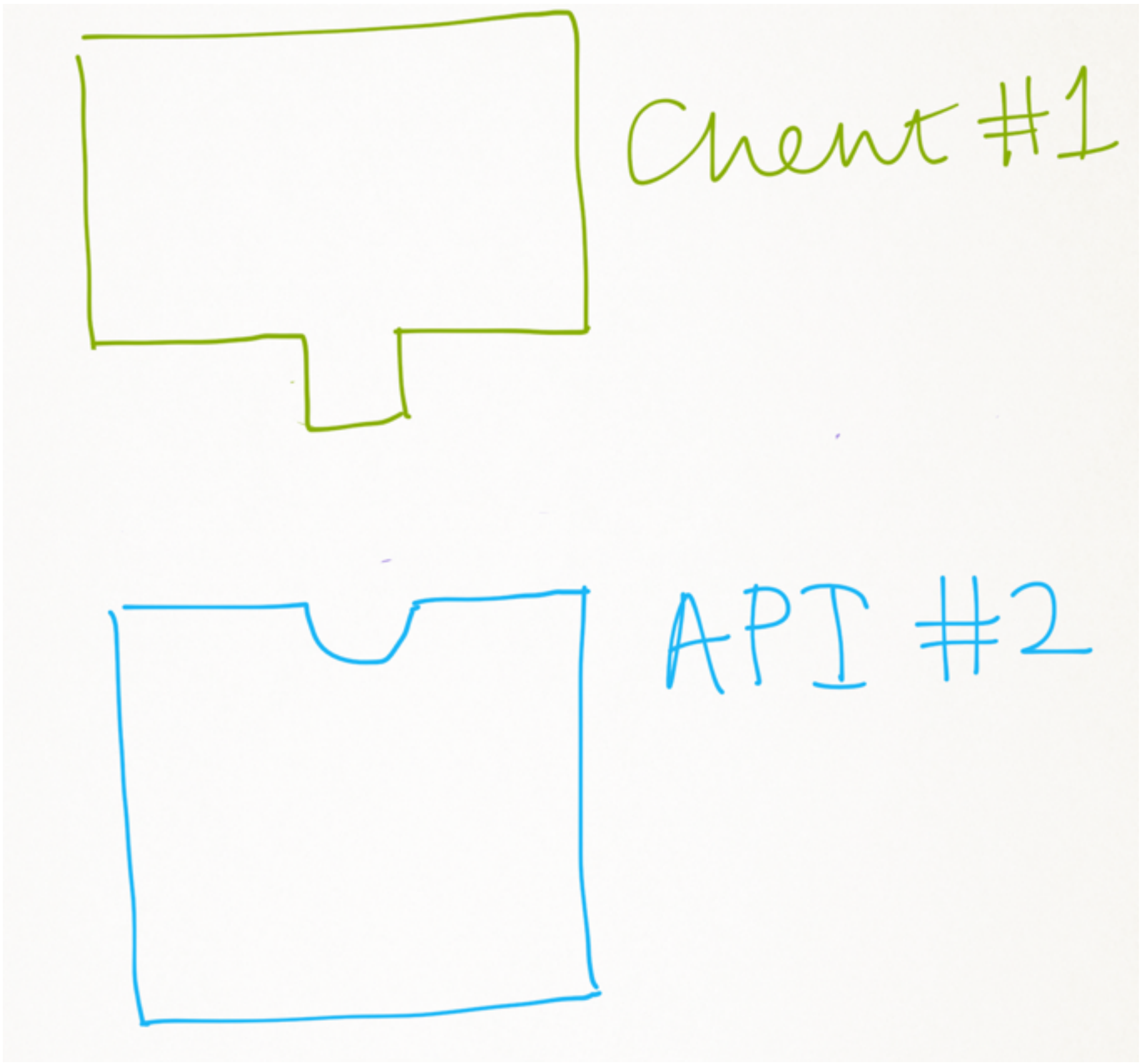
API migration in Wrangler: `maps` in R17

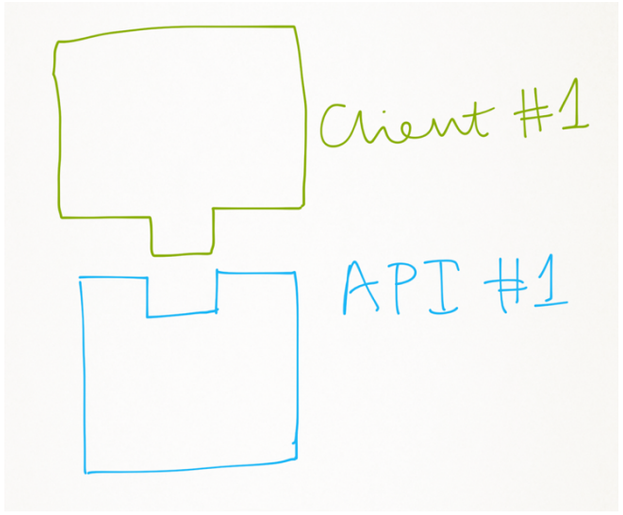
Now we look at migrating from `dict` to `map`

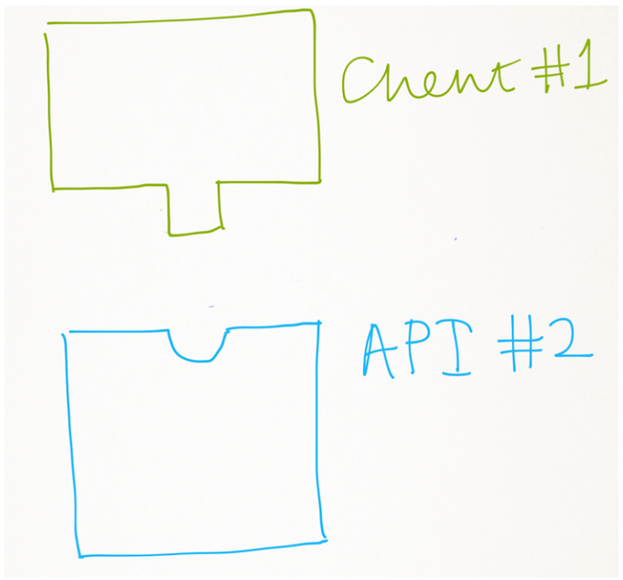
We use the Wrangler API migration tool

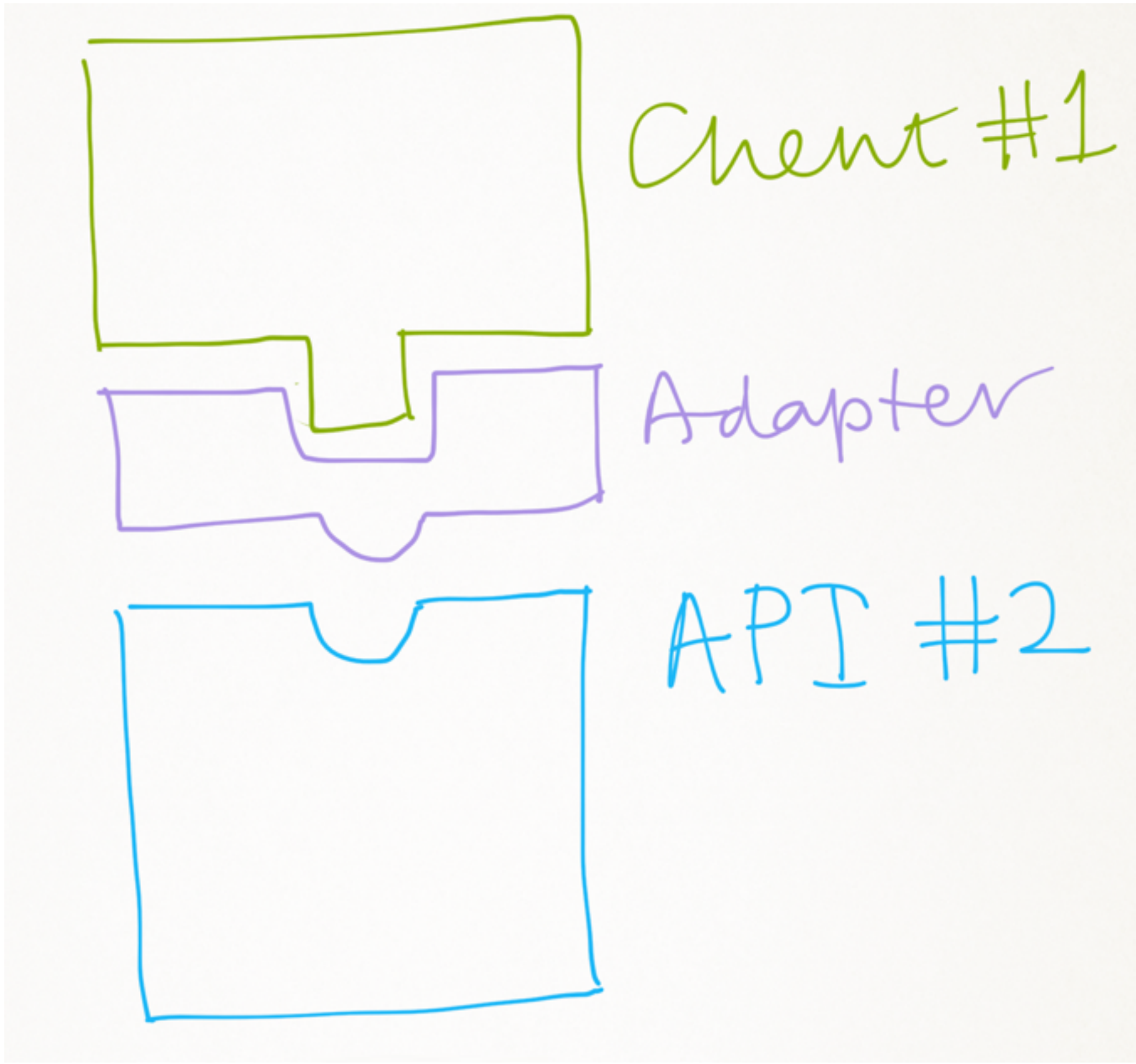

Client #1
API #1

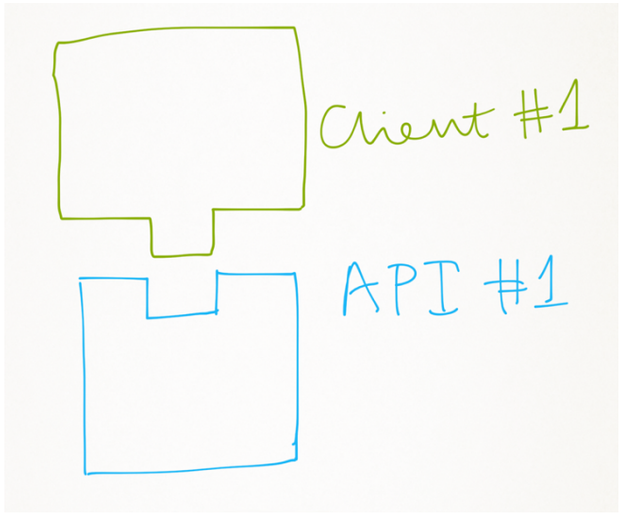

Client #1
API #1

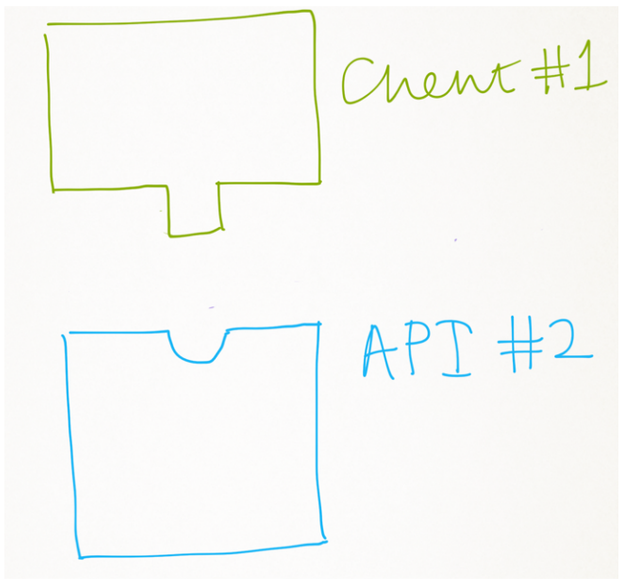

Client #1
API #2

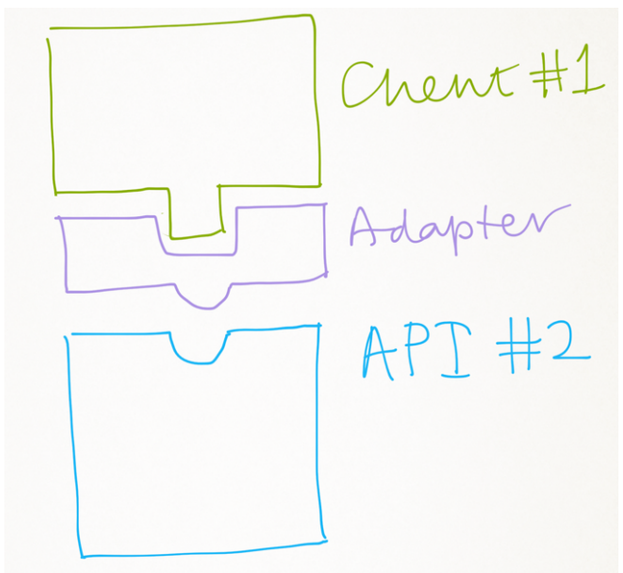

Client #1
API #1

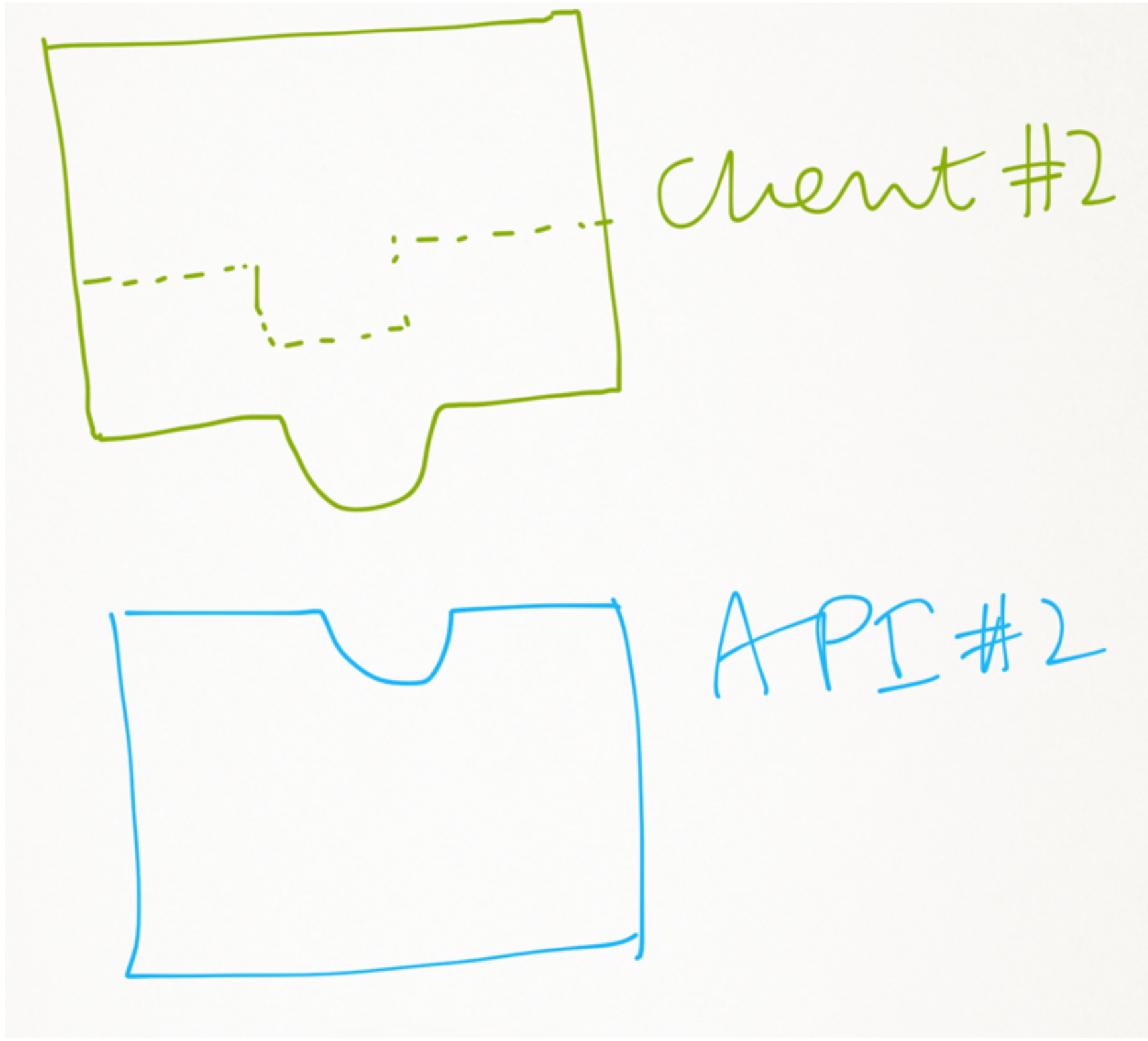

Client #1
API #2


Client #1
Adapter
API #2


Client #1
API #1


Client #1
API #2


Client #1
Adapter
API #2


Client #2
API #2

The adapter: `dict` in terms of `map`

```
is_key(Key, Dict) ->
    maps:is_key(Key, Dict).

to_list(Dict) ->
    maps:to_list(Dict).

from_list(List) ->
    maps:from_list(List).

size(Dict) ->
    maps:size(Dict).
```

The adapter: `dict` in terms of `map`

```
is_empty(Dict) ->
    maps:size(Dict)=:=0.

fetch(Key, Dict) ->
    maps:get(Key, Dict).

fetch_keys(Dict) ->
    maps:keys(Dict).

erase(Key, Dict) ->
    maps:remove(Key, Dict).

store(Key, Value, Dict) ->
    maps:put(Key, Value, Dict).

update(Key, Fun, Dict1) ->
    maps:update(Key, Fun(maps:get(Key, Dict1)), Dict1).
```

Before

```
predef_macros(File) ->
    Ms0 = dict:new(),
    true = dict:is_empty(Ms0),
    Ms1 = dict:store({atom,'FILE'}, {none,[{string,1,File}]}, Ms0),
    Ms2 = dict:store({atom,'LINE'}, {none,[{integer,1,1}]}, Ms1),
    Ms3 = dict:store({atom,'MODULE'}, undefined, Ms2),
    Ms4 = dict:store({atom,'MODULE_STRING'}, undefined, Ms3),
    Ms5 = dict:store({atom,'BASE_MODULE'}, undefined, Ms4),
    Ms6 = dict:store({atom,'BASE_MODULE_STRING'}, undefined, Ms5),
    Machine = list_to_atom(erlang:system_info(machine)),
    Ms7 = dict:store({atom,'MACHINE'}, {none,[{atom,1,Machine}]}, Ms6),
    Ms8 = dict:store({atom,Machine}, {none,[{atom,1,true}]}, Ms7),
    Size = dict:size(Ms8),
    io:format("Number of defined macros:~p\n", [Size]),
    Ms8.
```

After

```
predef_macros(File) ->
    Ms0 = maps:new(),
    true = maps:size(Ms0) =:= 0,
    Ms1 = maps:put({atom,'FILE'}, {none,[{string,1,File}]}, Ms0),
    Ms2 = maps:put({atom,'LINE'}, {none,[{integer,1,1}]}, Ms1),
    Ms3 = maps:put({atom,'MODULE'}, undefined, Ms2),
    Ms4 = maps:put({atom,'MODULE_STRING'}, undefined, Ms3),
    Ms5 = maps:put({atom,'BASE_MODULE'}, undefined, Ms4),
    Ms6 = maps:put({atom,'BASE_MODULE_STRING'}, undefined, Ms5),
    Machine = list_to_atom(erlang:system_info(machine)),
    Ms7 = maps:put({atom,'MACHINE'}, {none,[{atom,1,Machine}]}, Ms6),
    Ms8 = maps:put({atom,Machine}, {none,[{atom,1,true}]}, Ms7),
    Size = maps:size(Ms8),
    io:format("Number of defined macros:~p\n", [Size]),
    Ms8.
```

Adapting to `maps` - how does it work?

Write adapter module. e.g. `dict_to_map.erl`

Generate API migration rules, e.g. `refac_dict_to_map.erl`

Compile the rules (to `refac_dict_to_map.beam`)

Select API migration

→ Apply API migration to current file

Input name of adaptor module `refac_dict_to_map`

Using Wrangler outside emacs/ErlIDE

Wrangler from the Erlang shell … use module `api_wrangler.erl`

For example:

```
1> api_wrangler:start().

2> api_wrangler:rename_mod("main.erl",
                           test,
                           ["c:/cygwin/home/hl/test"]).

3> api_wrangler:stop().
```

More info: `http://refactoringtools.github.io/wrangler`

Wrangler is a toolset

Wrangler as a toolset

Clone detection

- Parametrisable
- Incremental
  - Automated support using the DSL

Module “bad smell” detection

- Size, cycles, exports

Other inspection and refactoring functions

Demo

Clone detection
simple example
rabbitmq
automation

emacs@HL-LT

File Edit Options Buffers Tools Help

```
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.
```

Rename function

Rename variables

Reorder variables

Add to export list

Fold* against the def.

```
--\---  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)-----------------------------------
```

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.

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.

Clone removal: top level

Transaction as a whole … non-“atomic” components OK.

Not just an API: `?atomic` etc. modify interpretation of what they enclose …

```
?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])
```

Refactoring tools for functional languages

JFP **23** (3): 293–350, 2013.
doi:10.1017/S0956796813000117

Refactoring tools for functional languages

SIMON THOMPSON and HUIQING LI
School of Computing, University of Kent, Kent, UK
(*e-mail:* {s.j.thompson,h.li}@kent.ac.uk)

Abstract

Refactoring is the process of changing the design of a program without changing what it does. Typical refactorings, such as function extraction and generalisation, are intended to make a program more amenable to extension, more comprehensible and so on. Refactorings differ from other sorts of program transformation in being applied to source code, rather than to a 'core' language within a compiler, and also in having an effect across a code base, rather than to a single function definition, say. Because of this, there is a need to give automated support to the process. This paper reflects on our experience of building tools to refactor functional programs written in Haskell (HaRe) and Erlang (Wrangler). We begin by discussing what refactoring means for functional programming languages, first in theory, and then in the context of a larger example. Next, we address system design and details of system implementation as well as contrasting the style of refactoring and tooling for Haskell and Erlang. Building both tools led to reflections about what particular refactorings mean, as well as requiring analyses of various kinds, and we discuss both of these. We also discuss various extensions to the core tools, including integrating the tools with test frameworks; facilities for detecting and eliminating code clones; and facilities to make the systems extensible by users. We then reflect on our work by drawing some general conclusions, some of which apply particularly to functional languages, while many others are of general value.

http://journals.cambridge.org/repo_A90unxgK


Wrangelska palatset
Svea livgarde

Wrangelska palatset
Wrangelska Palace

Inför Karl XII:s kröning 1697 förvandlades palatsets gård till en praktsal, där kungen satt på en silvertron. Wrangelska palatset hade blivit kungafamiljens tillfälliga bostad. Sedan 1755 använder Svea hovrätt palatset.

Fältherren Carl Gustav Wrangel satte i gång palatsbygget 1652. Fyra år tidigare hade han fått tomten i belöning för sina insatser i 30-åriga kriget av drottning Kristina. Här stod då ett stenhus i två våningar som Lars Sparre låtit bygga 20 år tidigare. I ena hörnet ner mot stranden ingick ett kanontorn från Gustav Vasas dagar.

Wrangel anlitade arkitekterna Jean De la Vallée och Nicodemus Tessin den äldre och stenhuset byggdes ut till ett storslaget palats. Fasaderna fick rika dekorationer i sandsten och den ståtligaste sidan mot vattnet balanserades upp med ännu ett rundtorn. In mot torget tillkom de två flyglarna med avslutande fyrkantsstorn. Palatsets inre var inte helt klart då Wrangel och hustrun Anna Margareta von Haugwitz avled på 1670-talet. Men det var stadens största och mest ståndsmässiga bostad.

År 1697 brann slottet Tre Kronor och fram till 1754 var Wrangelska palatset kungafamiljens bostad, kallad "Kungshuset". År 1755 flyttade flera ämbetsverk in, däribland Svea hovrätt, instiftat 1614. Vid en brand 1802 förstördes stora delar. Palatset byggdes om efter ritningar av Carl Christoffer Gjörwell 1804 och fick då dagens utseende. Med höga krav på säkerhet och tillgänglighet för alla gjordes en ombyggnad 2004.

For the coronation of Karl XII in 1697 the palace was turned into a hall of splendour, where the king sat on a silver throne. The Wrangel Palace had become the temporary residence of the Royal Family. Since 1755 it is used by Svea Hovrätt Court of Appeal.

Count Carl Gustav Wrangel started constructing the palace in 1652. Four years earlier Queen Kristina had granted him the land for his exploits in the Thirty Years' War. A 20-year-old, two-storey stone house stood here, featuring at one corner facing the shore a cannon turret from 1530.

Wrangel enlisted architects Jean De la Vallée and Nicodemus Tessin the Elder to turn the stone house into a magnificent palace. The façades were decorated in sandstone and the most imposing side overlooking the water was given balance through a second round tower. Facing the square, two wings were added, ending in square towers. Wrangel died in 1676, the palace still incomplete, but it was the biggest and grandest dwelling in the town.

In 1697 the castle Tre Kronor burned down, and the palace became home to the Royal Family until 1754. In 1755 several public bodies moved in, including Svea Hovrätt, founded in 1614. Much was destroyed by fire in 1802. In 1804, the palace was redesigned to drawings by Carl Christoffer Gjörwell, giving it today's appearance. Alterations were carried out in 2004 to stringent specifications on security and public accessibility.

Getting involved

https://github.com/RefactoringTools/Wrangler

RELEASE

ProTest
property based testing

Questions?