Refactoring Reflected

Simon Thompson, University of Kent


Huiqing Li


Colin Runciman


Thomas Arts

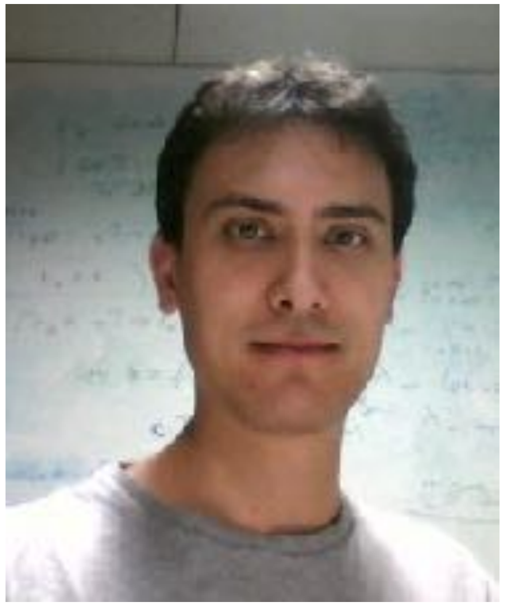

Dániel Horpácsi


Judit Kőszegi


Nik Sultana


Scott Owens


Reuben Rowe


Hugo Férée


Chris Brown


György Orosz


Melinda Tóth


Stephen Adams

Andreas Reuleaux

Claus Reinke

Pablo Lamela

Jane Street

Why should I use your refactoring tool?

What’s the ideal language supporting refactoring?

What do you mean when you say “refactoring”?

What’s so wrong with duplicated code?

Why haven’t you implemented this refactoring?

It’s just renaming … what’s all the fuss?

I don’t need a refactoring tool … … I have types!

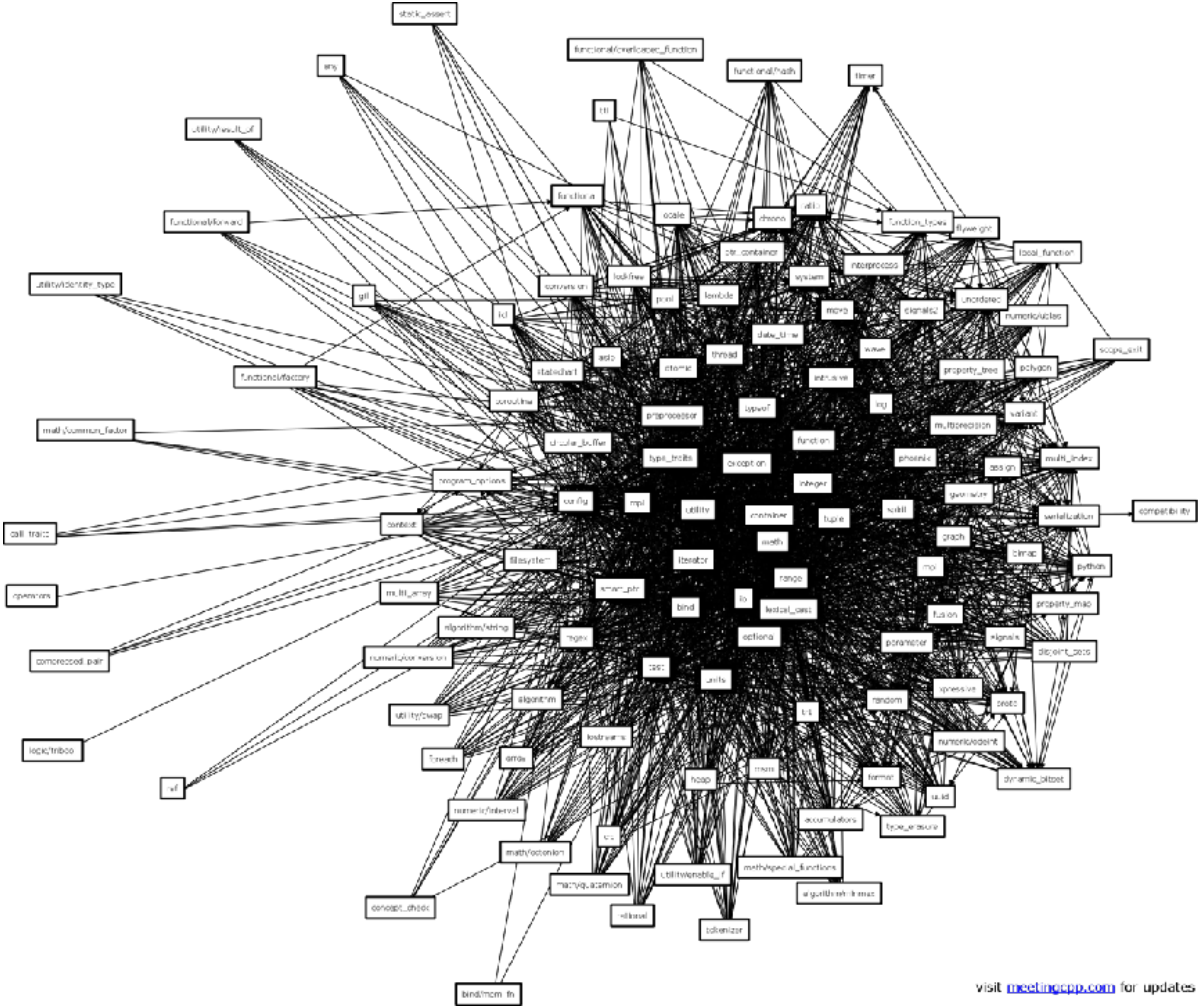
Will you integrate with this editor or IDE?

Why have you messed up the layout of my program?

Why should I trust a refactoring tool on my code?

What do you mean when you say "refactoring"?

```diff
@@ -187,11 +187,12 @@ splitOrConvert (m, r, c) sol =
      Nothing -> Nothing
 
 solveLEIntAux :: Eq a => Eq b => [([[Rational]], [a], [b])] -> Maybe [(b, Integer)]
+solveLEIntAux [] = Nothing
 solveLEIntAux (h:t) =
   case splitOrConvert h rSol of
     Just (Left nh) -> solveLEIntAux (nub (t ++ nh))
     Just (Right s) -> Just s
-    Nothing -> Nothing
+    Nothing -> solveLEIntAux t
   where
    rSol = solveLE h
 
```



visit meetingcpp.com for updates

© Sheila Terry/Science Photo Library

What does “refactoring” mean?

Minor edits or wholesale changes

Something local or of global scope

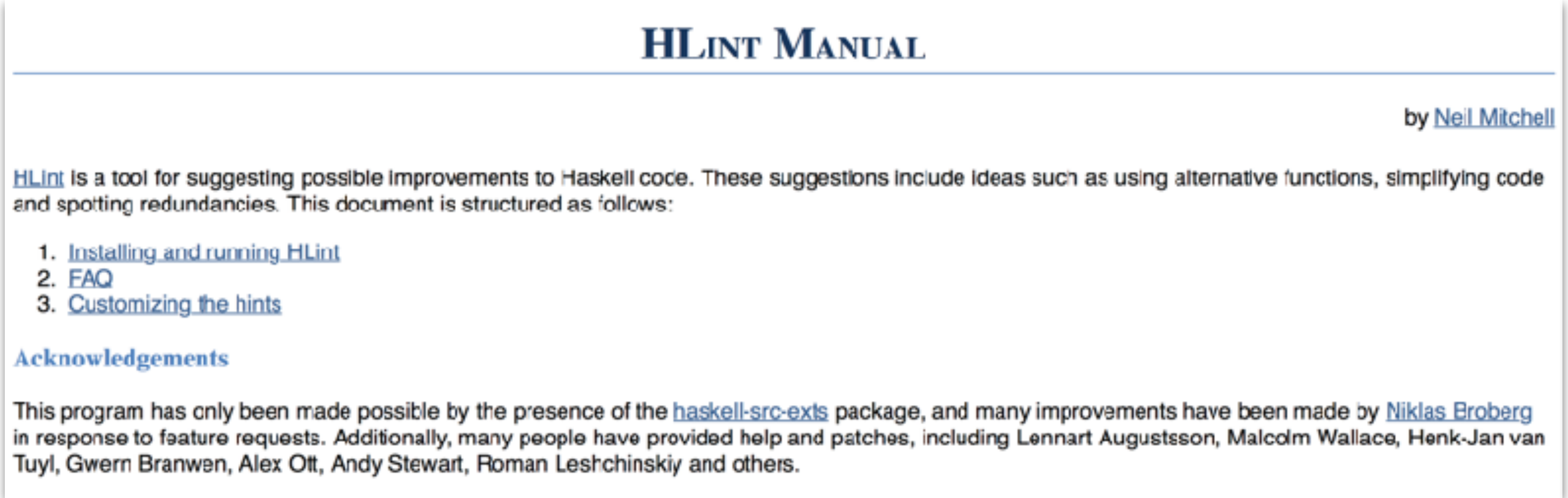
Just a general change in the software …

… or something that changes its
structure, but not its functionality?

Something chosen by a programmer …

… or chosen by an algorithm?

Expression-level refactorings



HLINT MANUAL

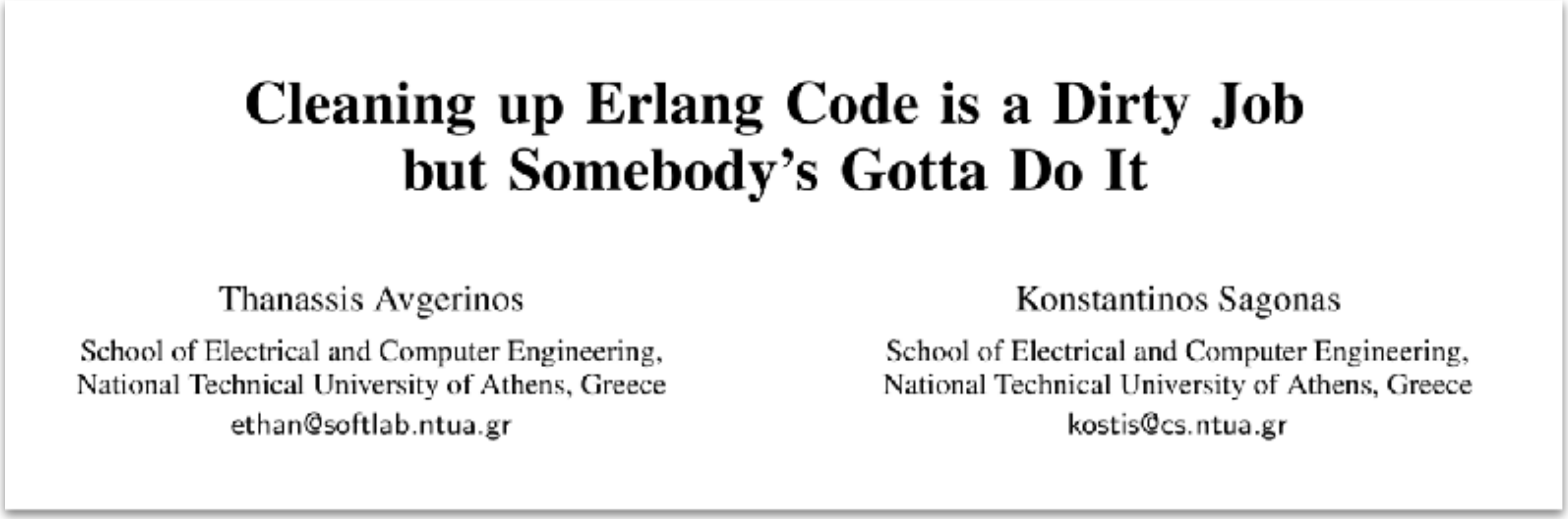
by Neil Mitchell

HLint is a tool for suggesting possible improvements to Haskell code. These suggestions include ideas such as using alternative functions, simplifying code and spotting redundancies. This document is structured as follows:

1. Installing and running HLint
2. FAQ
3. Customizing the hints

Acknowledgements

This program has only been made possible by the presence of the haskell-src-exts package, and many improvements have been made by Niklas Broberg in response to feature requests. Additionally, many people have provided help and patches, including Lennart Augustsson, Malcolm Wallace, Henk-Jan van Tuyl, Gwern Branwen, Alex Ott, Andy Stewart, Roman Leshchinskiy and others.

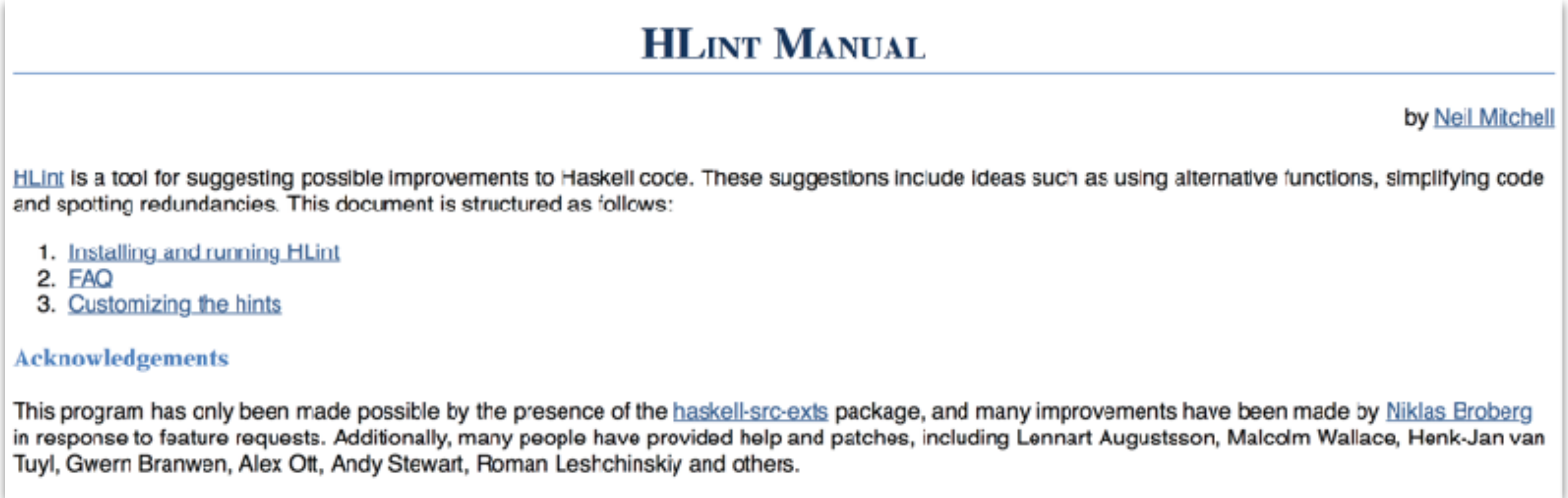


Cleaning up Erlang Code is a Dirty Job but Somebody's Gotta Do It

Thanassis Avgerinos
School of Electrical and Computer Engineering,
National Technical University of Athens, Greece
ethan@softlab.ntua.gr

Konstantinos Sagonas
School of Electrical and Computer Engineering,
National Technical University of Athens, Greece
kostis@cs.ntua.gr

Expression-level refactorings



HLint Manual

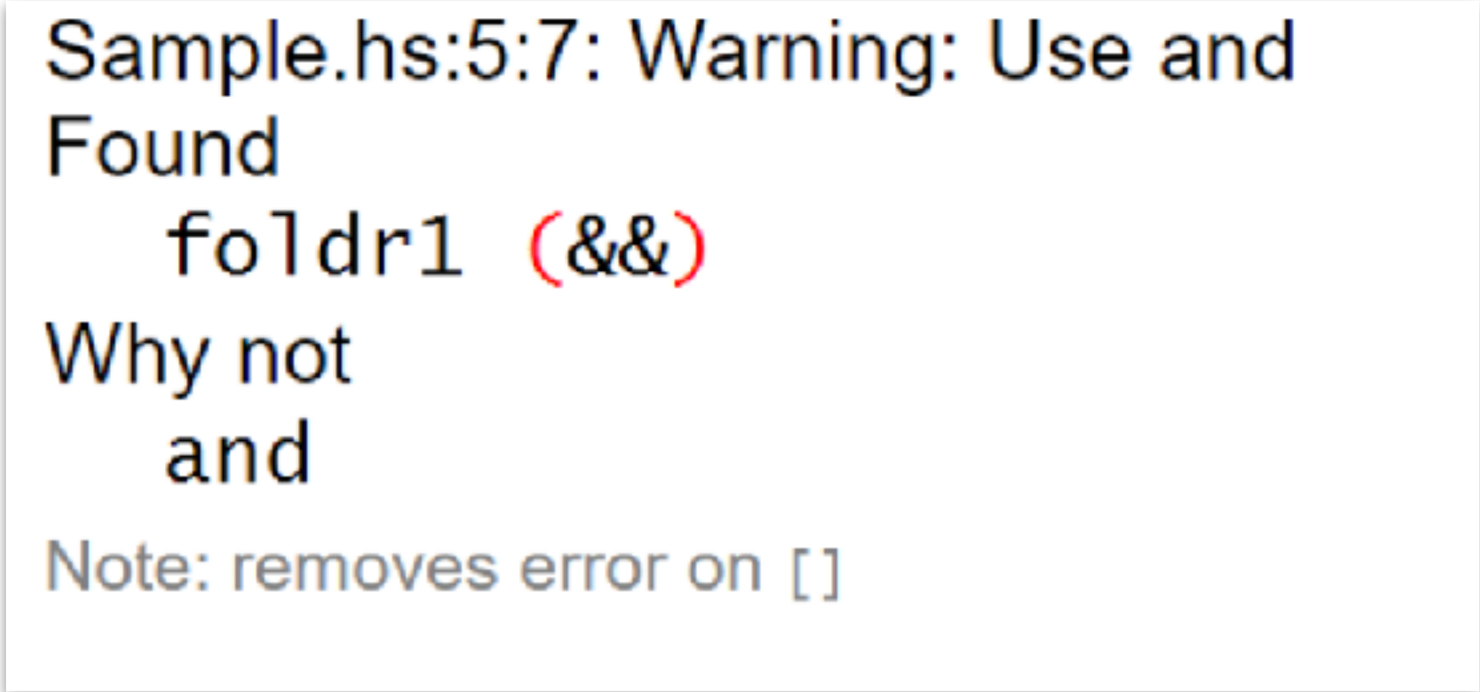
by Neil Mitchell

HLint is a tool for suggesting possible improvements to Haskell code. These suggestions include ideas such as using alternative functions, simplifying code and spotting redundancies. This document is structured as follows:

1. Installing and running HLint
2. FAQ
3. Customizing the hints

Acknowledgements

This program has only been made possible by the presence of the haskell-src-exts package, and many improvements have been made by Niklas Broberg in response to feature requests. Additionally, many people have provided help and patches, including Lennart Augustsson, Malcolm Wallace, Henk-Jan van Tuyl, Gwern Branwen, Alex Ott, Andy Stewart, Roman Leshchinskiy and others.



```
Sample.hs:5:7: Warning: Use and
Found
   foldr1 (&&)
Why not
   and
Note: removes error on []
```

What sort of refactoring interests us?

Changes beyond the purely local, which can be effected easily.

What sort of refactoring interests us?

Changes beyond the purely local, which can be effected easily.

Renaming a function / module / type / structure.

Changing a naming scheme: `camel_case` to `camelCase`, …

Generalising a function … extracting a definition.

Function extraction in Erlang

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.
```

Function extraction in Erlang

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

Function extraction in Erlang

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},
```

Function extraction in Erlang

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}.
```

What sort of refactoring interests us?

Changes beyond the purely local, which can be effected easily.

Renaming a function / module / type / structure.

Changing a naming scheme: `camel_case` to `camelCase`, …

Generalising a function … extracting a definition.

Changing a type representation.

Changing a library API.

Module restructuring: e.g. removing inclusion loops.

Why should I use your refactoring tool?

Refactoring
=
Transformation

Refactoring

=

Transformation

Refactoring
=
Transformation + Pre-condition

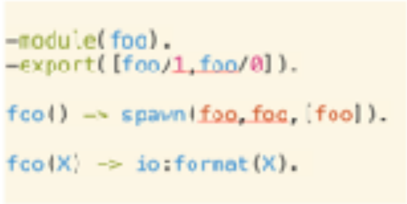
How to refactor?

By hand … using an editor

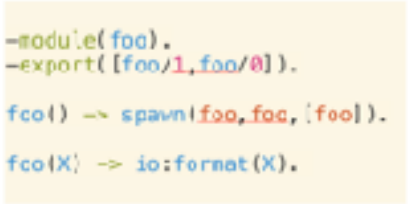
- Flexible … but error-prone.
- Infeasible in the large.

Tool-supported

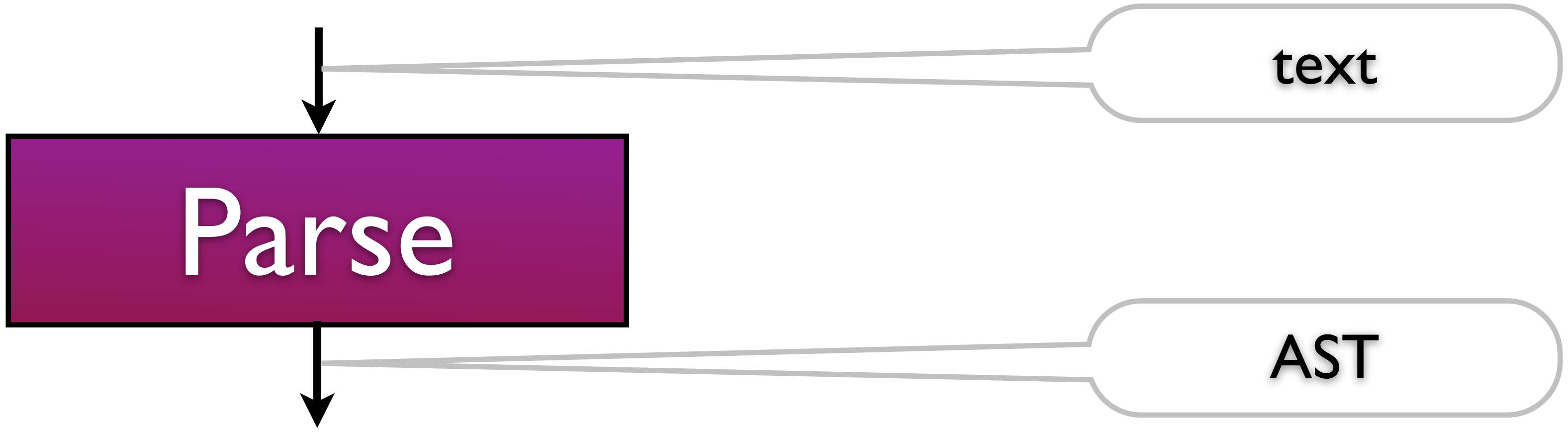
- Handles transformation *and* analysis.
- Scalable to large-code bases: module-aware.
- Integrated with tests, macros, ...



```
-module(foo).
-export([foo/1,foo/0]).

foo() -> spawn(foo,foo,[foo]).

foo(X) -> io:format(X).
```



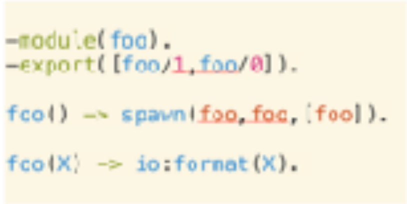
```
-module(foo).
-export([foo/1,foo/0]).

foo() -> spawn(foo,foo,[foo]).

foo(X) -> io:format(X).
```



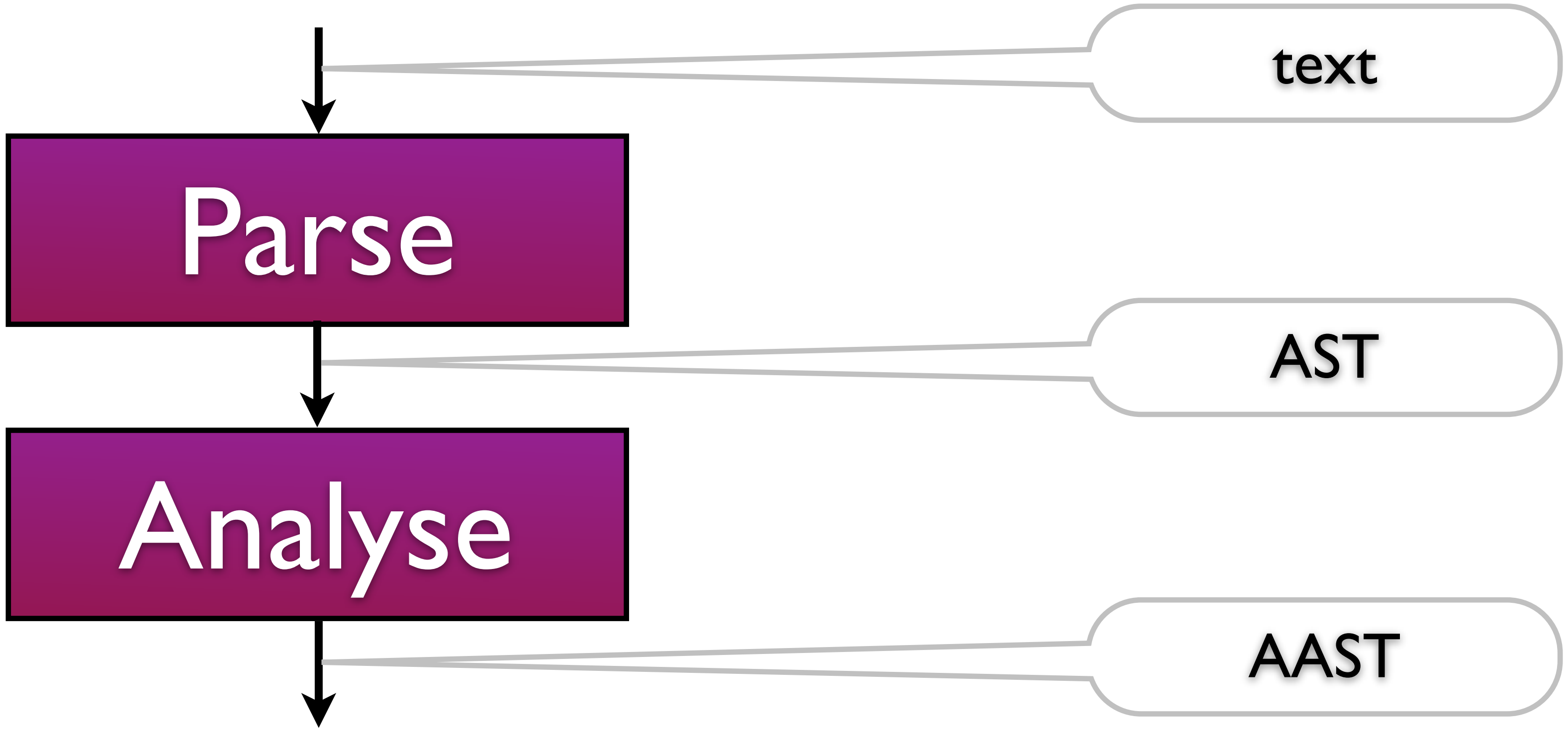
text

Parse

AST



```
-module(foo).
-export([foo/1,foo/0]).

foo() -> spawn(foo,foo,[foo]).

foo(X) -> io:format(X).
```



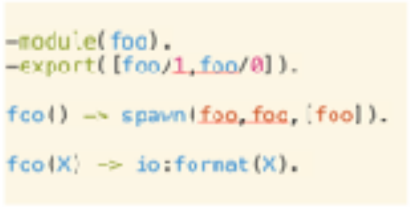
text

Parse

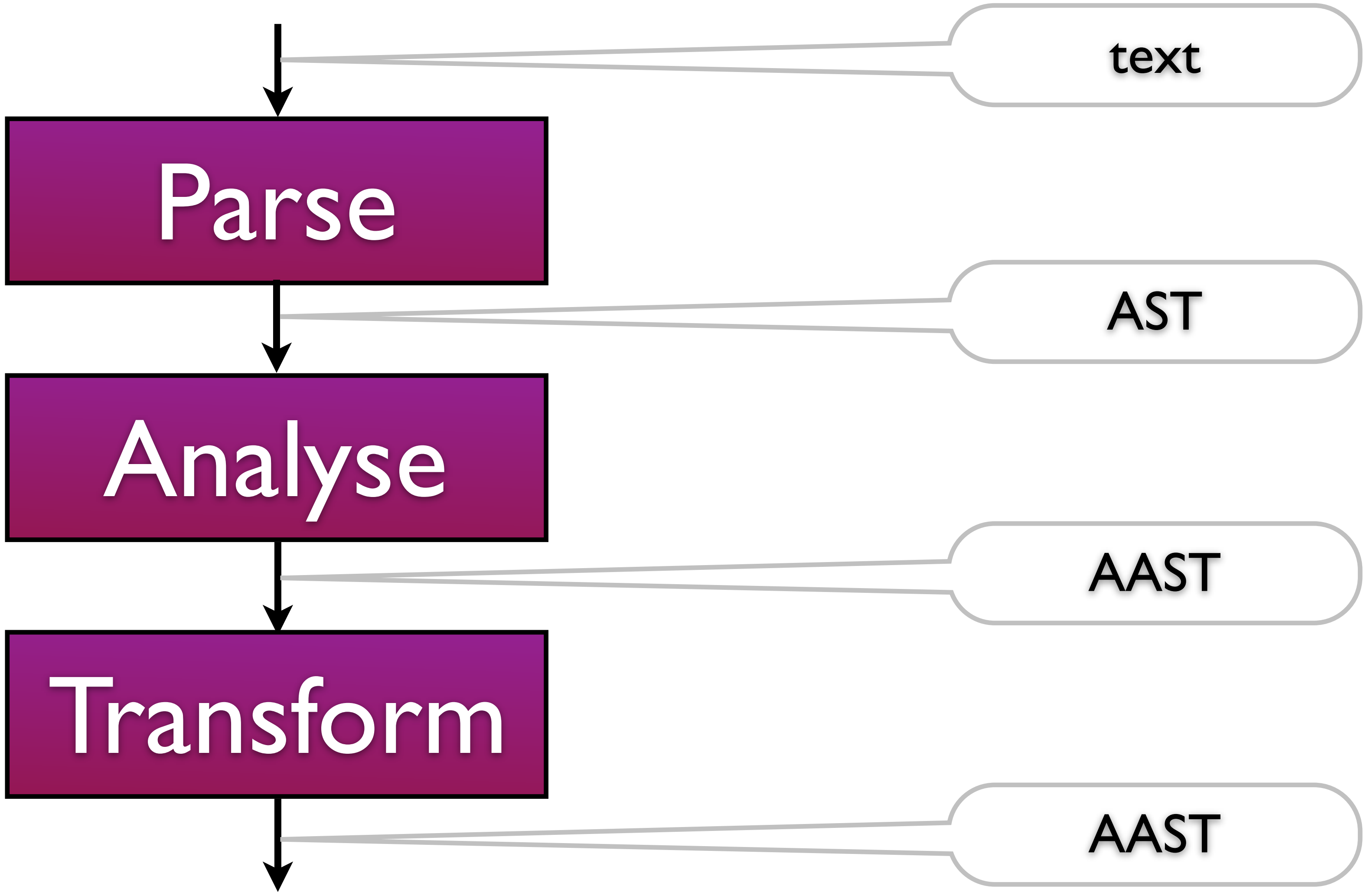
AST

Analyse

AAST



```erlang
-module(foo).
-export([foo/1,foo/0]).

foo() -> spawn(foo,foo,[foo]).

foo(X) -> io:format(X).
```



Parse

Analyse

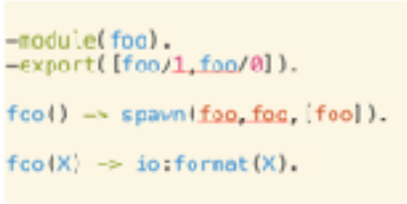
Transform

text

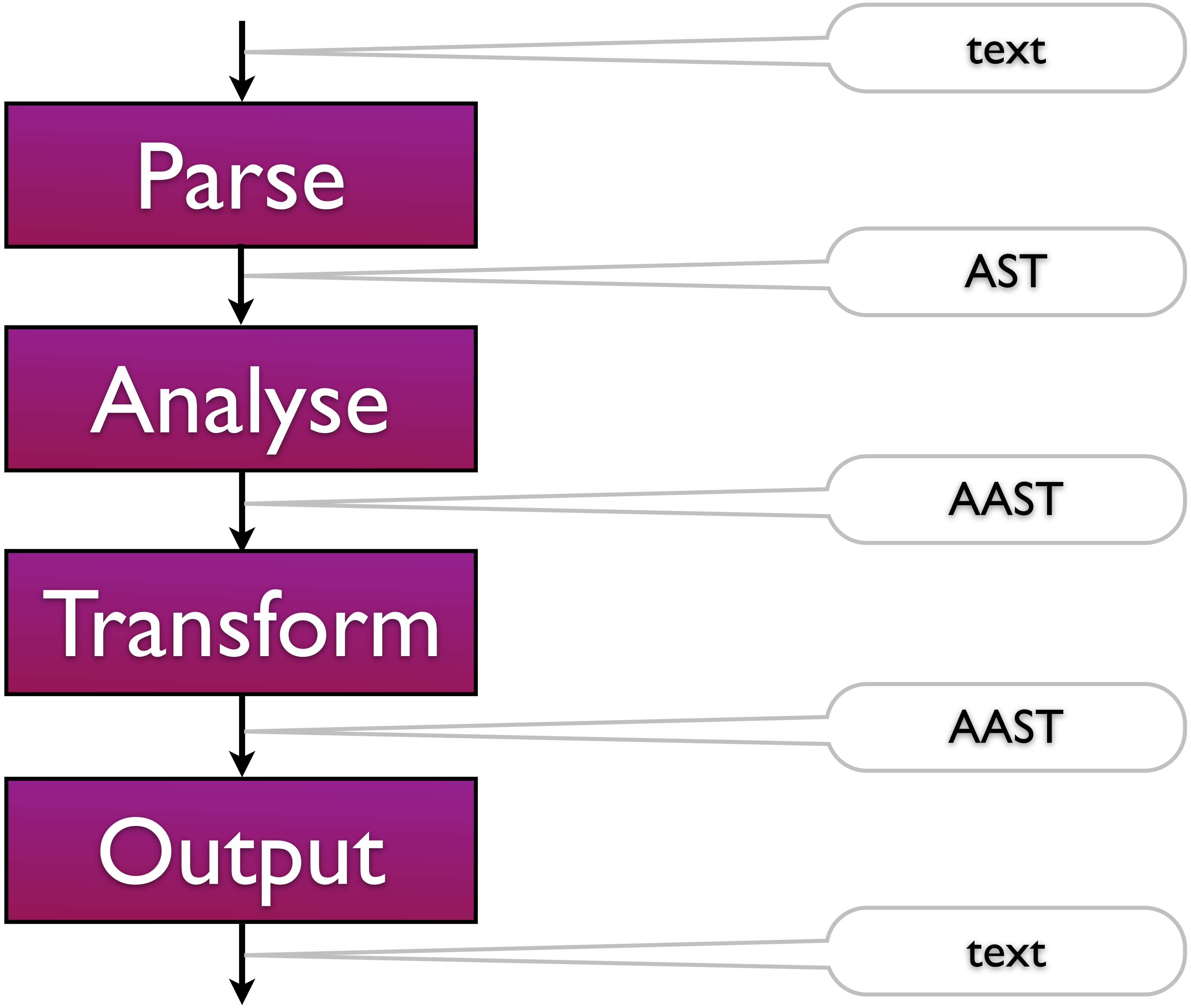
AST

AAST

AAST



```
-module(foo).
-export([foo/1,foo/0]).

foo() -> spawn(foo,foo,[foo]).

foo(X) -> io:format(X).
```



text

Parse

AST

Analyse

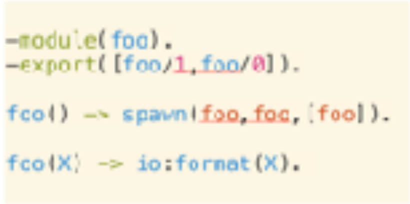
AAST

Transform

AAST

Output

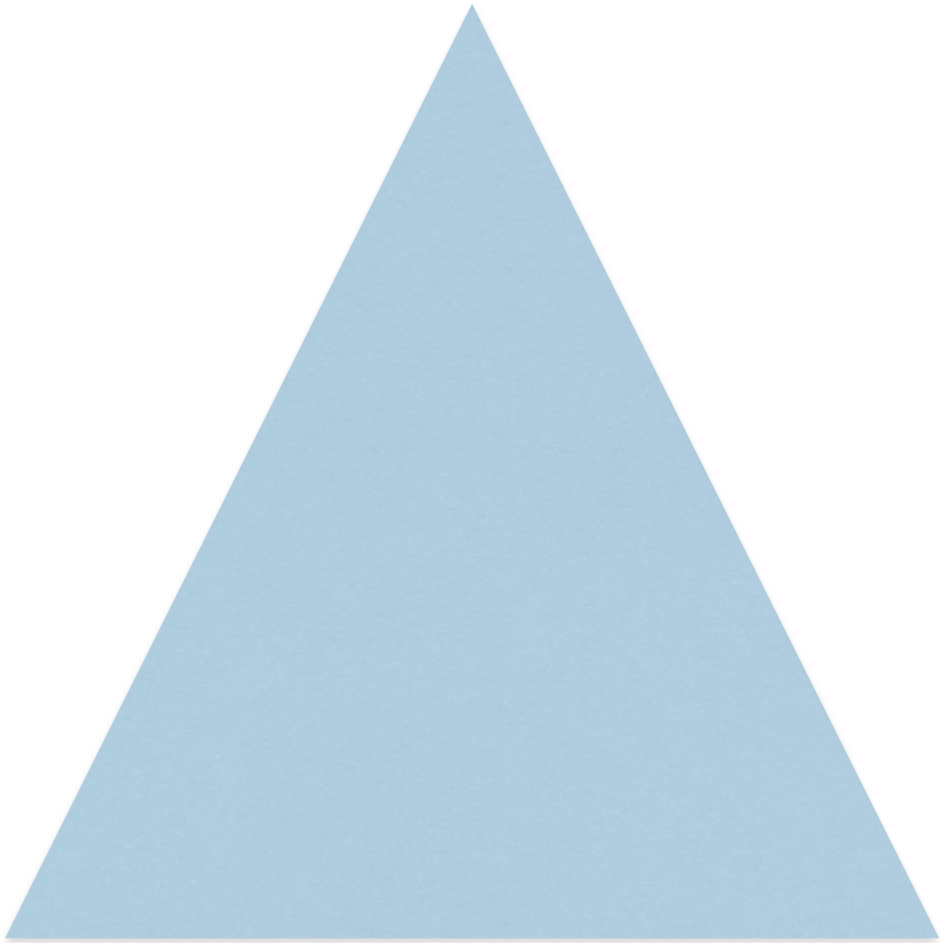
text



```
-module(foo).
-export([foo/1,foo/0]).

foo() -> spawn(foo,foo,[foo]).

foo(X) -> io:format(X).
```

Traversals, strategies and visitors

- Multi-purpose
  - Collect and analyse info.
  - Effect a transformation.
- Separation of concerns
  - Point-wise operation …
  - … and tree traversal



Traversals, strategies and visitors

Multi-purpose

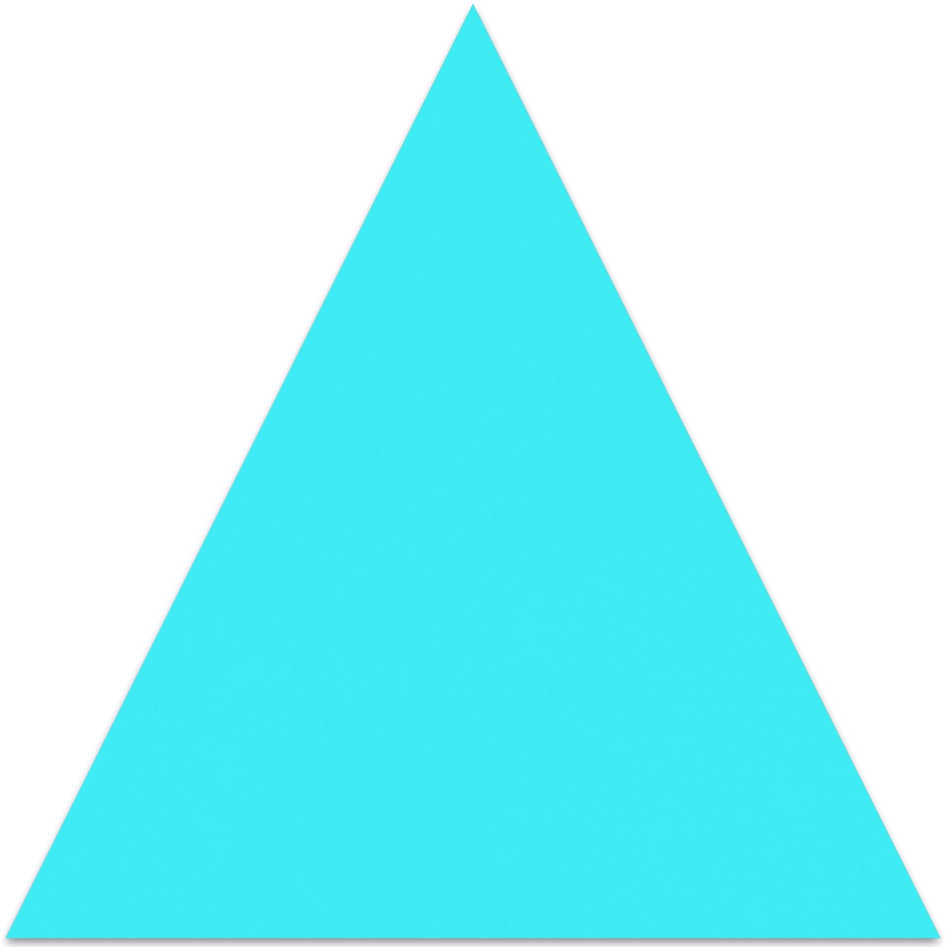
  Collect and analyse info.

  Effect a transformation.

Separation of concerns

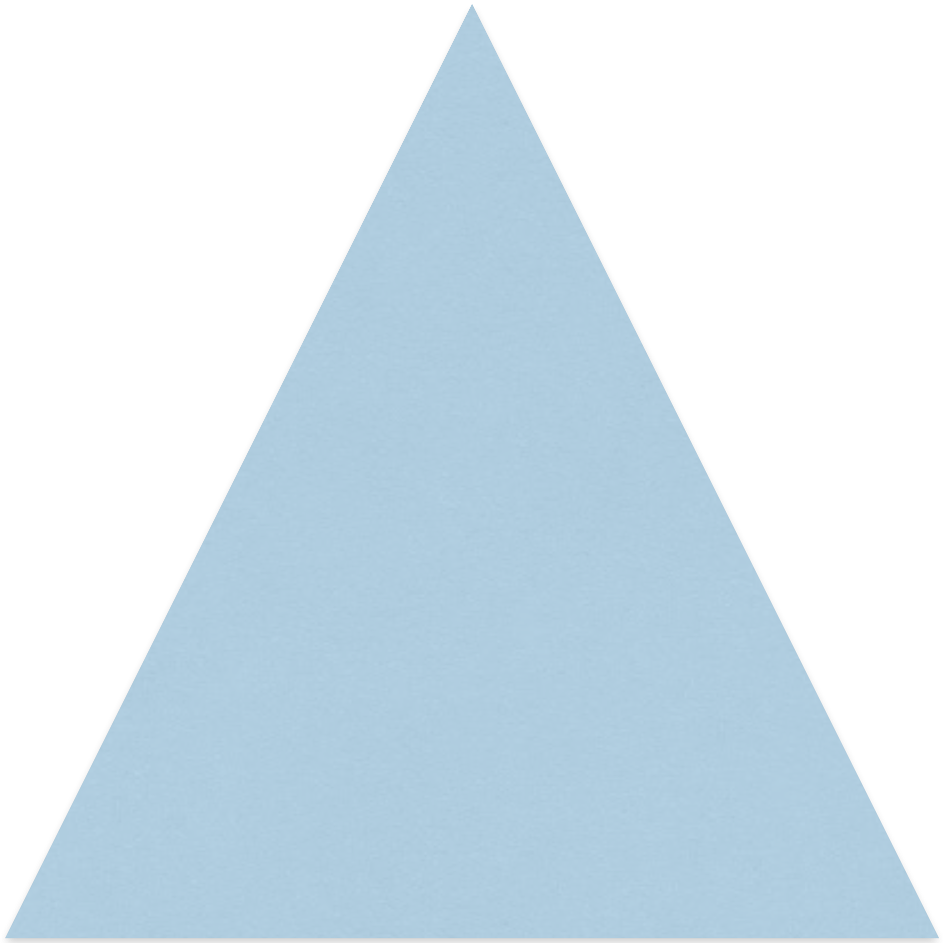
  Point-wise operation …

  … and tree traversal

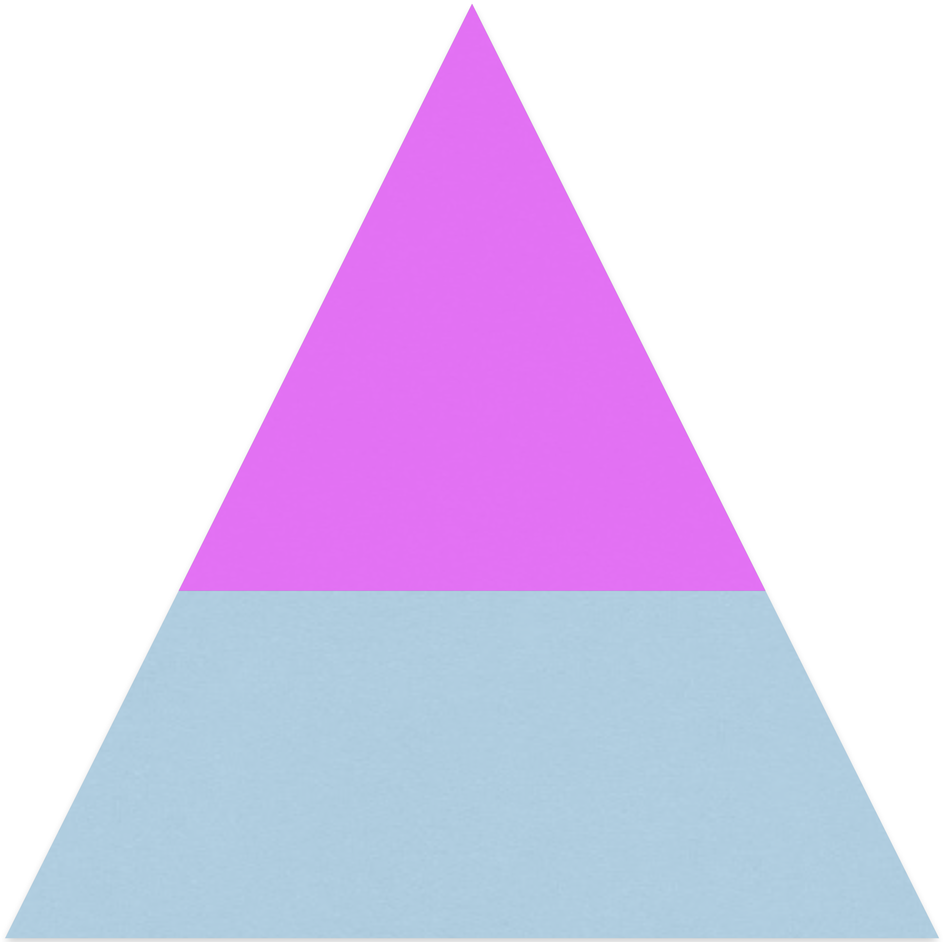


Traversals, strategies and visitors

- Multi-purpose
  - Collect and analyse info.
  - Effect a transformation.
- Separation of concerns
  - Point-wise operation …
  - … and tree traversal



Traversals, strategies and visitors



Multi-purpose

  Collect and analyse info.

  Effect a transformation.

Separation of concerns

  Point-wise operation …

  … and tree traversal

Haskell

Strongly typed
Lazy
Pure + Monads
Complex type system
Layout sensitive

Haskell

Strongly typed
Lazy
Pure + Monads
Complex type system
Layout sensitive

Erlang

Weakly typed
Strict
Some side-effects
Concurrency
Macros and idioms

Haskell

Strongly typed
Lazy
Pure + Monads
Complex type system
Layout sensitive

Erlang

Weakly typed
Strict
Some side-effects
Concurrency
Macros and idioms

OCaml

Strongly typed
Strict
Refs etc and i/o.
Modules + interfaces
Scoping/modules

Haskell

Strongly typed
Lazy
Pure + Monads
Complex type system
Layout sensitive

Erlang

Weakly typed
Strict
Some side-effects
Concurrency
Macros and idioms

OCaml

Strongly typed
Strict
Refs etc and i/o.
Modules + interfaces
Scoping/modules

HaRe

Haskell 98
GHC Haskell API …
… Alan Zimmerman
Basic refactorings,
clones, type-based, …
Strategic prog

Haskell

Strongly typed
Lazy
Pure + Monads
Complex type system
Layout sensitive

Erlang

Weakly typed
Strict
Some side-effects
Concurrency
Macros and idioms

OCaml

Strongly typed
Strict
Refs etc and i/o.
Modules + interfaces
Scoping/modules

HaRe

Haskell 98
GHC Haskell API …
… Alan Zimmerman
Basic refactorings, clones, type-based, …
Strategic prog

Wrangler

Full Erlang
Erlang, syntax_tools
HaRe + module, API + DSL,
Naive strategic prog

Haskell

Strongly typed
Lazy
Pure + Monads
Complex type system
Layout sensitive

Erlang

Weakly typed
Strict
Some side-effects
Concurrency
Macros and idioms

OCaml

Strongly typed
Strict
Refs etc and i/o.
Modules + interfaces
Scoping/modules

HaRe

Haskell 98
GHC Haskell API …
… Alan Zimmerman
Basic refactorings,
clones, type-based, …
Strategic prog

Wrangler

Full Erlang
Erlang, syntax_tools
HaRe + module, API
+ DSL,
Naive strategic prog

Rotor

(O)Caml
OCaml compiler
So far: renaming +
dependency theory.
Derived visitors

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, ...

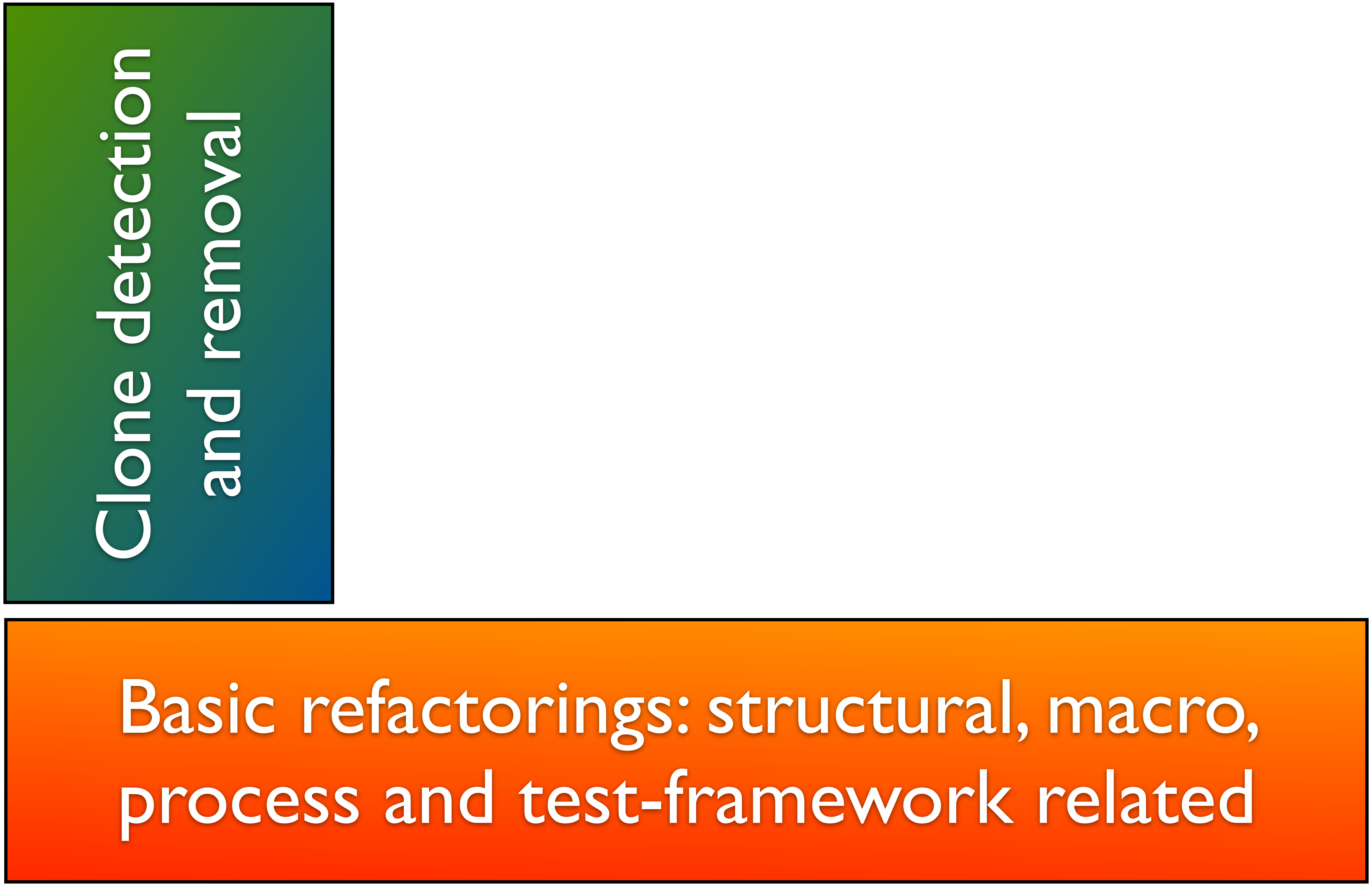
Faithful to layout and comments.

Build in Erlang and apply the tool to itself.

Wrangler

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

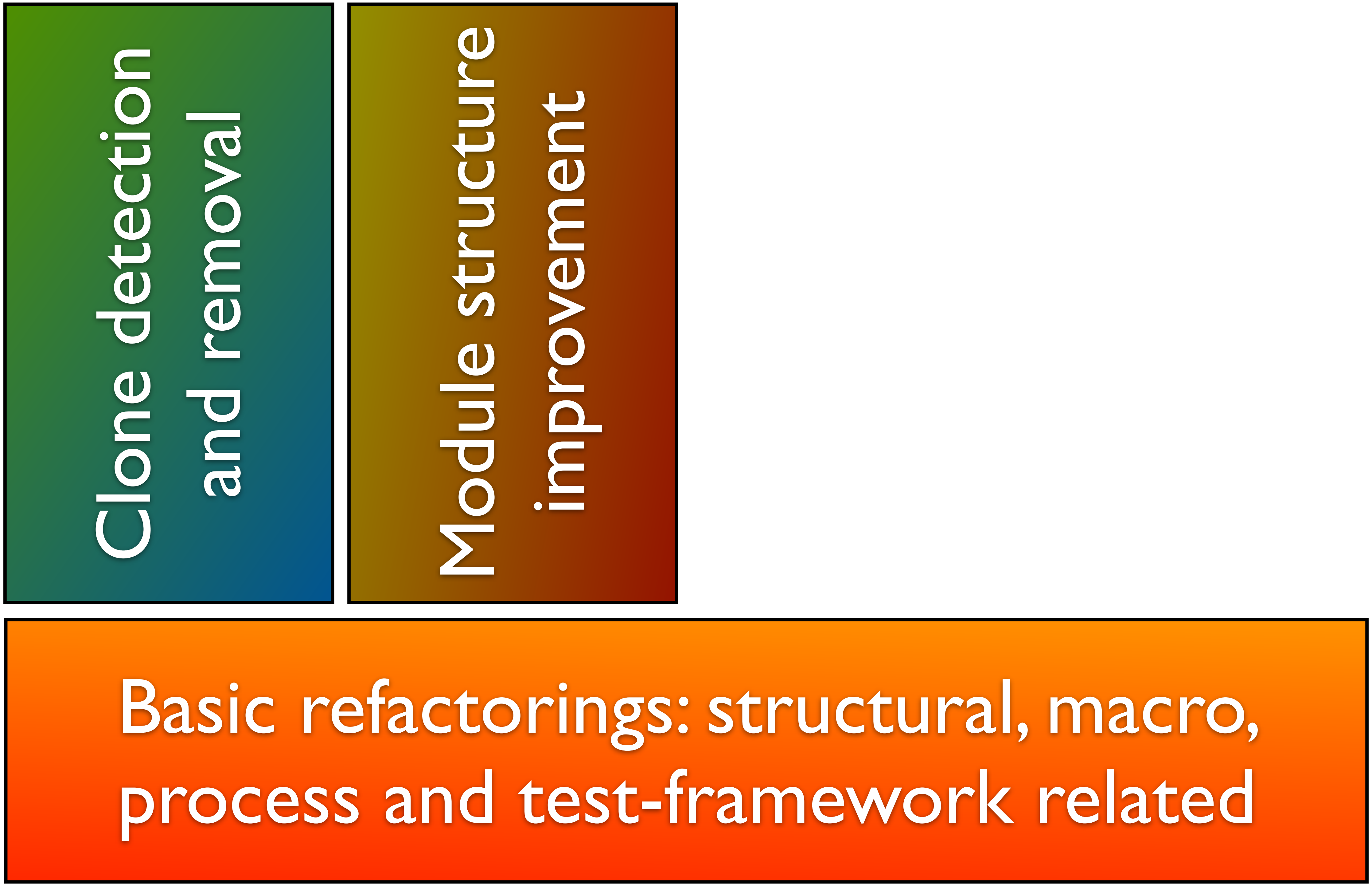
Wrangler



Clone detection and removal

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

Wrangler

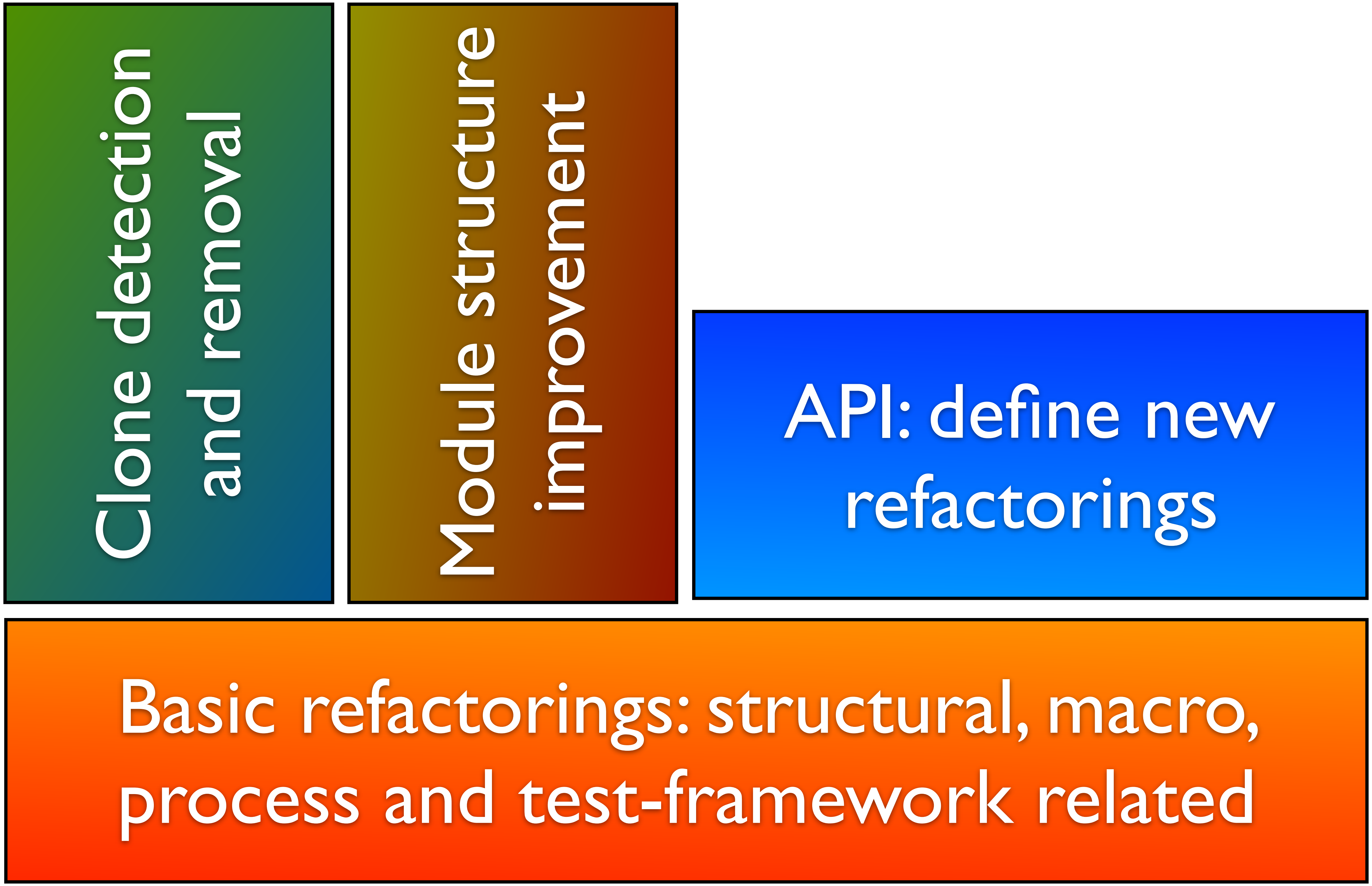


Clone detection and removal

Module structure improvement

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

Wrangler



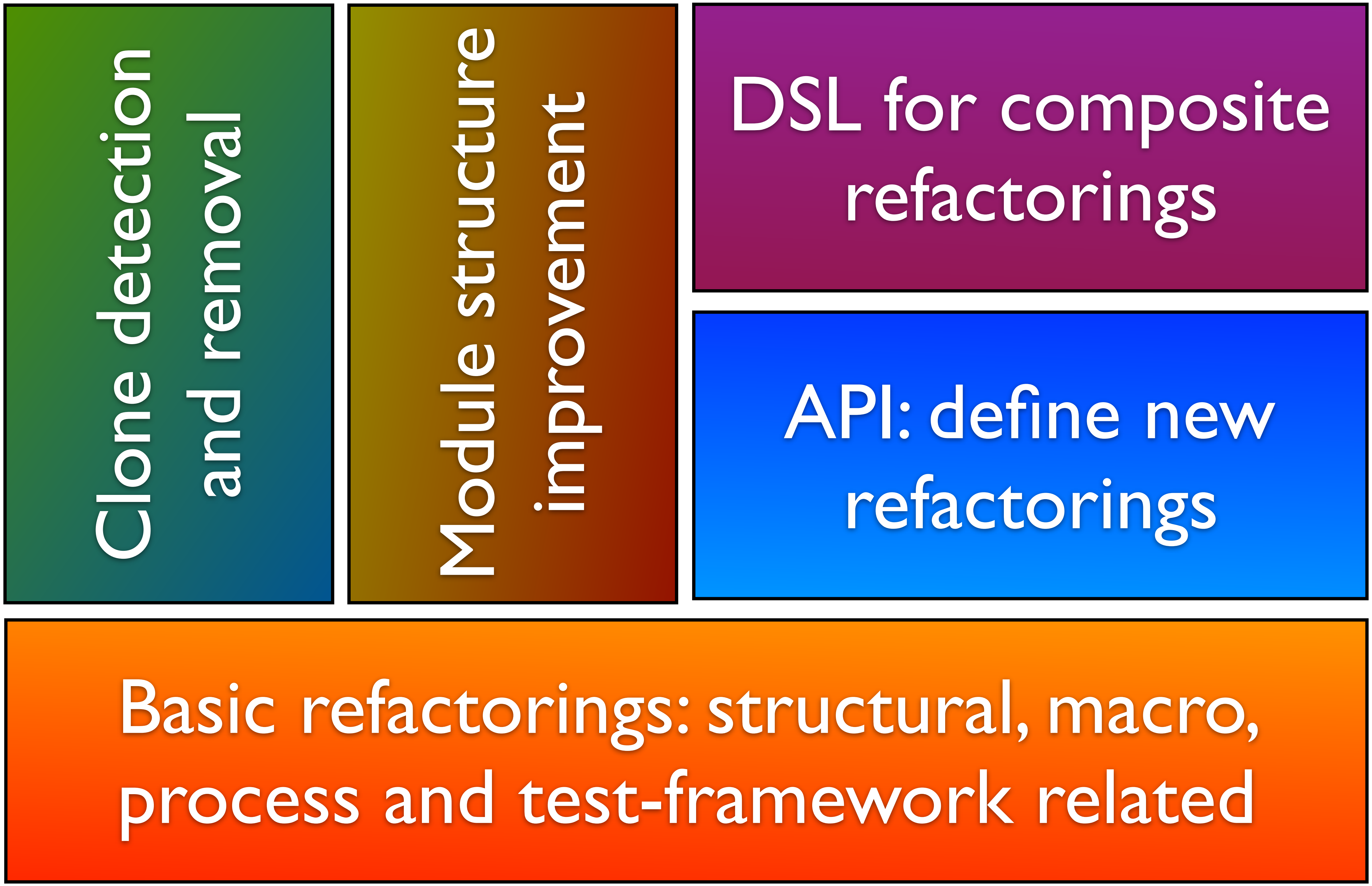
Clone detection and removal

Module structure improvement

API: define new refactorings

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

Wrangler



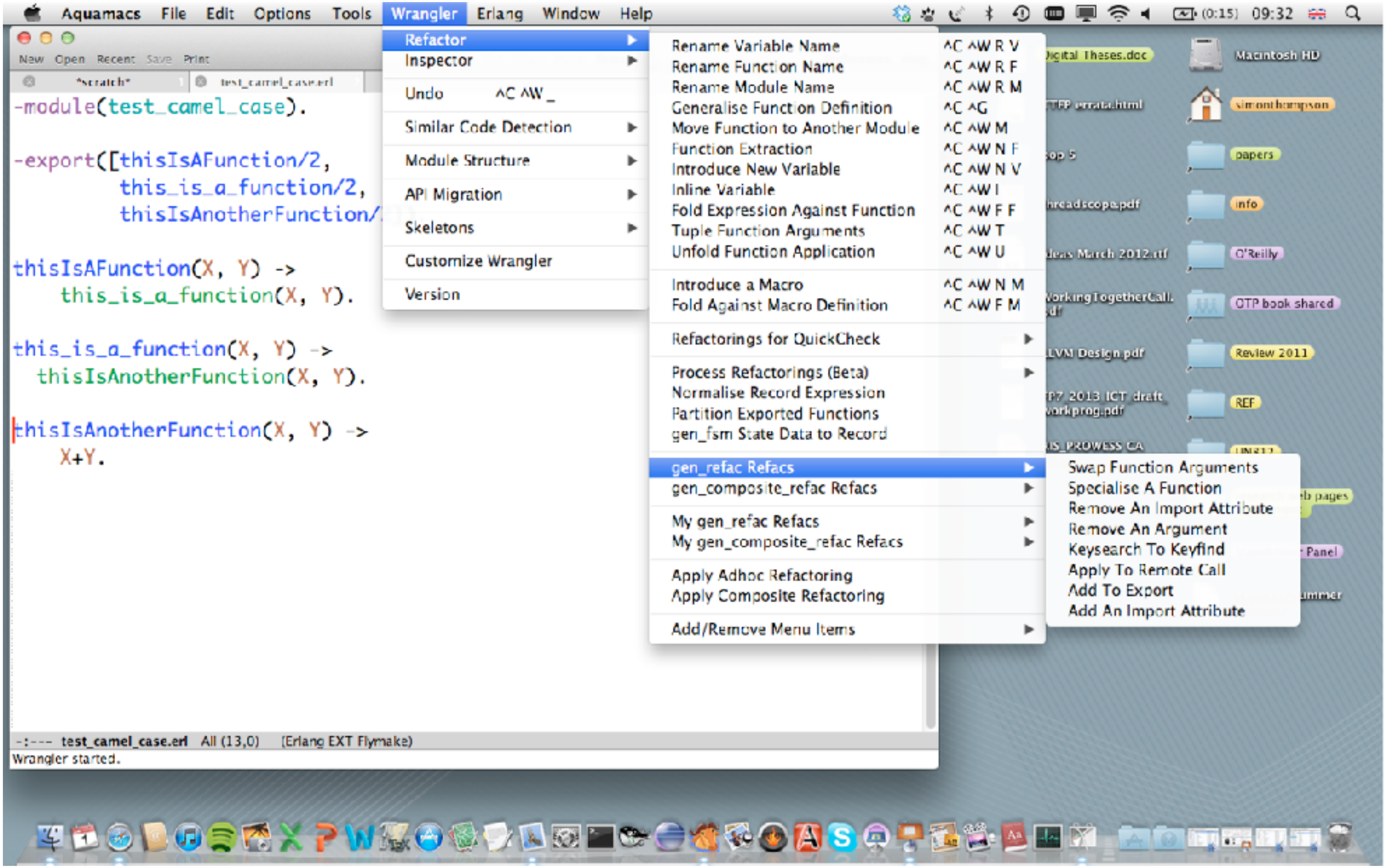
Clone detection and removal

Module structure improvement

DSL for composite refactorings

API: define new refactorings

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



```
-module(test_camel_case).

-export([thisIsAFunction/2,
         this_is_a_function/2,
         thisIsAnotherFunction/

thisIsAFunction(X, Y) ->
    this_is_a_function(X, Y).

this_is_a_function(X, Y) ->
  thisIsAnotherFunction(X, Y).

thisIsAnotherFunction(X, Y) ->
    X+Y.
```

```
-:---  test_camel_case.erl   All (13,0)    (Erlang EXT Flymake)
Wrangler started.
```

Analyses needed …

Static semantics

Types

Modules

Side-effects

Analyses needed …

Static semantics

Types

Modules

Side-effects

Atoms

Process structure

Macros

Conventions and frameworks

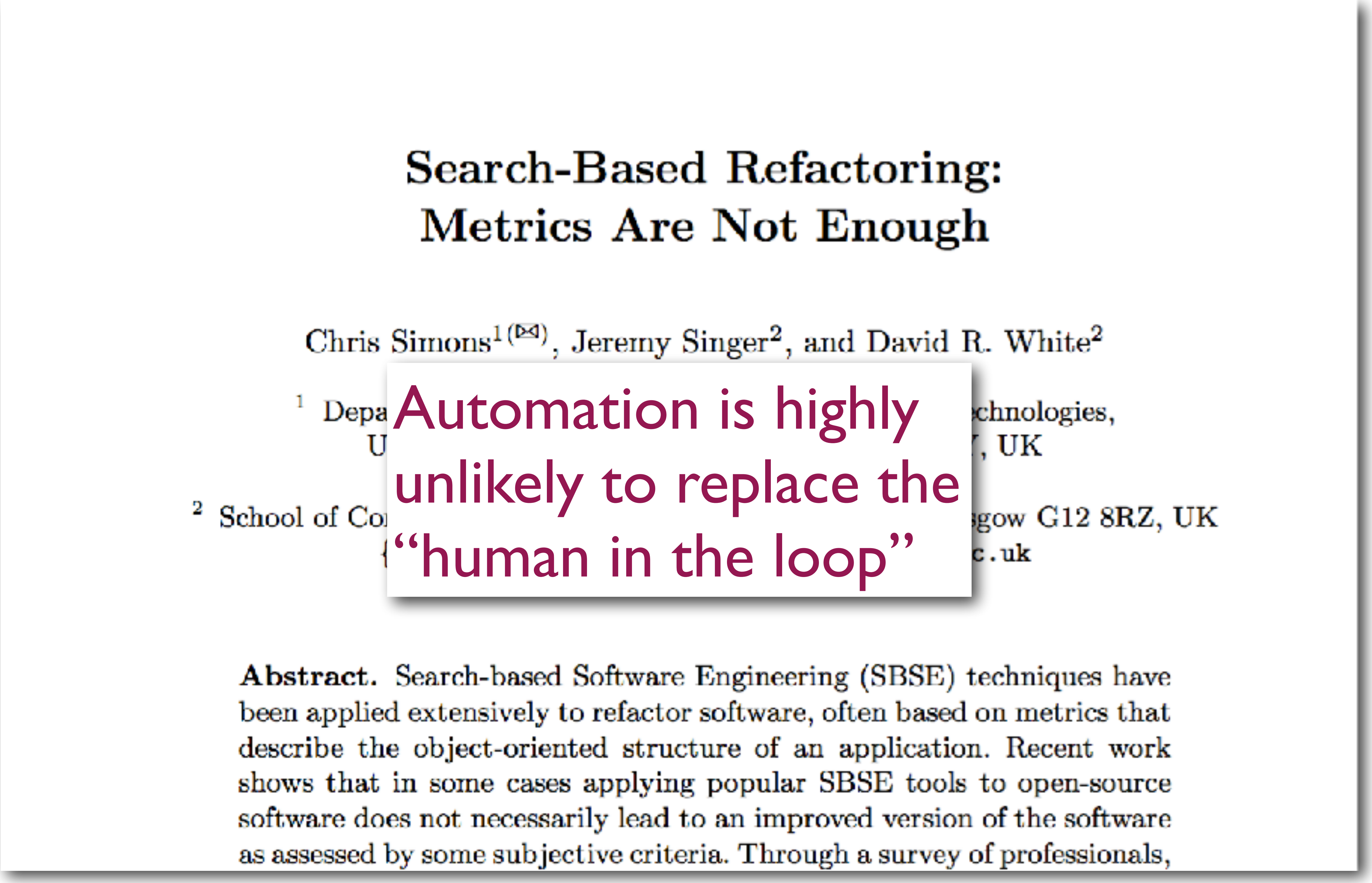
So, why use a tool?

We can do things it would take too long to do without a tool.

We can be less risk-averse: e.g. in doing generalisation.

Exploratory: try and undo if we wish.

95% ≫ 0%: hit most cases … fix the last 5% “by hand”.



Search-Based Refactoring: Metrics Are Not Enough

Chris Simons[1(✉)], Jeremy Singer[2], and David R. White[2]

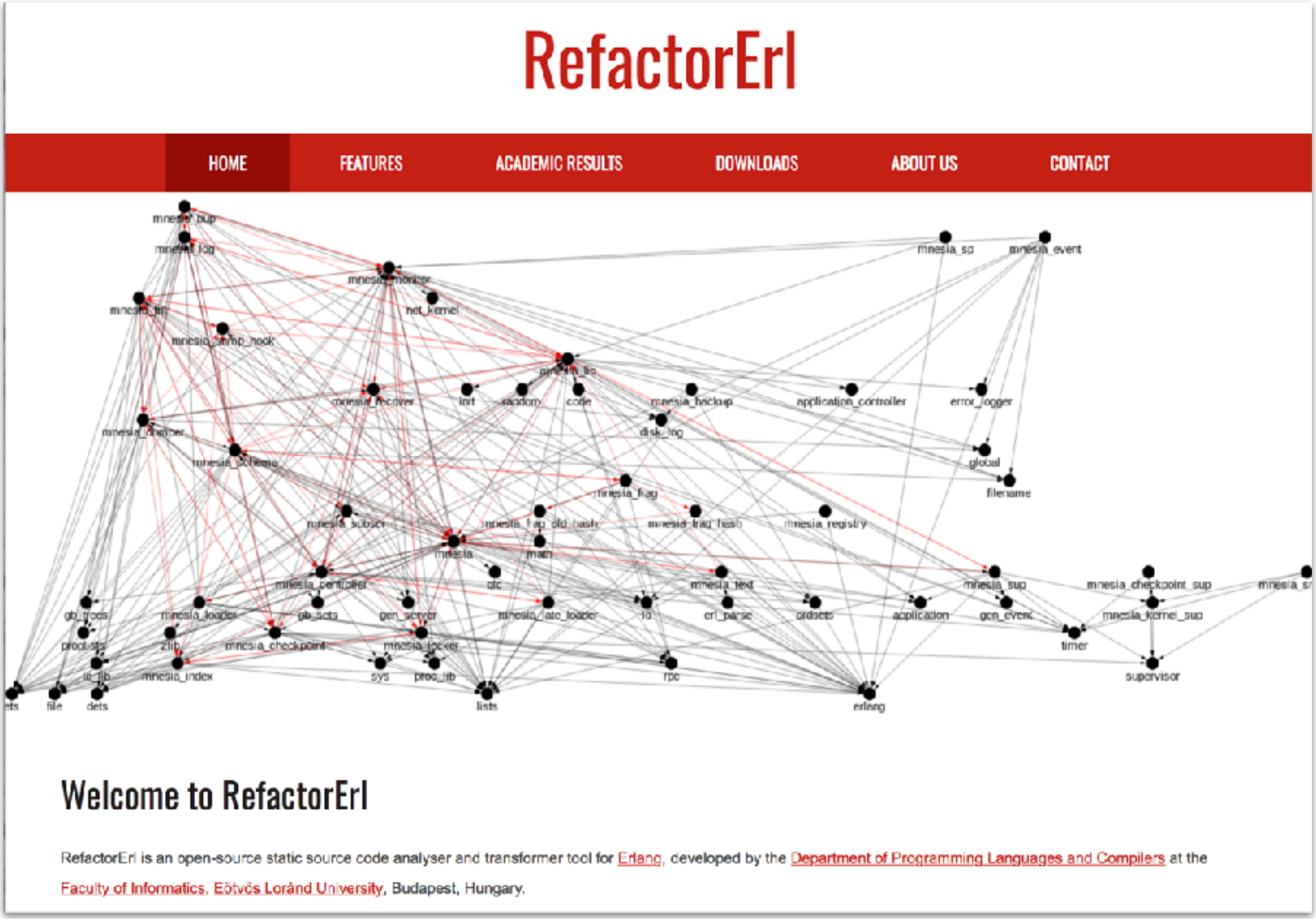
[1] Depa chnologies,
U , UK

[2] School of Co gow G12 8RZ, UK
{ c.uk

Abstract. Search-based Software Engineering (SBSE) techniques have been applied extensively to refactor software, often based on metrics that describe the object-oriented structure of an application. Recent work shows that in some cases applying popular SBSE tools to open-source software does not necessarily lead to an improved version of the software as assessed by some subjective criteria. Through a survey of professionals,

Automation is highly unlikely to replace the "human in the loop"

SSBSE 2015



RefactorErl

Welcome to RefactorErl

RefactorErl is an open-source static source code analyser and transformer tool for Erlang, developed by the Department of Programming Languages and Compilers at the Faculty of Informatics, Eötvös Loránd University, Budapest, Hungary.

We can be more adventurous with a refactoring tool!

It’s just renaming …
what’s all the fuss?

What is in a name?

Resolving names requires not just the static structure …

… but also types (polymorphism, overloading) and modules.

Beyond the wits of regexps.

Leverage other infrastructure or the compiler.

Types sneak in …

```
f x = (x*x + 42) + (x + 42)

f x y = (x*x + y) + (x + y)
```

✓

Types sneak in …

```
f x = (x*x + 42) + (x + 42)

f x y = (x*x + y) + (x + y)
```

✓

```
funny = length ([[True]] ++ []) +
        length ([True] ++ [])

funny xs = length ([[True]] ++ xs) +
           length ([True] ++ xs)
```

✗

… as do different sorts of atoms

```
-module(foo).
-export([foo/1,foo/0]).

foo() -> spawn(foo,foo,[foo]).

foo(X) -> io:format("~w",[X]).
```

And some peculiarities

```
f1(P) ->
    receive
        {ok, X} -> P!thanks;
        {error,_} -> P!grr
    end,
    P!{value,X}.
```

✗

And some peculiarities

```
f1(P) ->
    receive
        {ok, X} -> P!thanks;
        {error,_} -> P!grr
    end,
    P!{value,X}.
```

✗

```
f2(P) ->
    receive
        {ok, X} -> P!thanks;
        {error,X} -> P!grr
    end,
    P!{value,X}.
```

✓

OCaml nested scopes

src/foo.ml:

```
    ⋮
  let f = ...
  let f = ...
    ⋮
  ... f ...
```

src/bar.ml:

```
  open Foo
    ⋮
  ... f ...
```

$\texttt{Foo.f} \mapsto g$

OCaml nested scopes

```
src/foo.ml:
      ⋮
  let f = ...
  let g = ...
      ⋮
  ... g ...

src/bar.ml:
  open Foo
      ⋮
  ... g ...
```

`Foo.f` ↦ g

OCaml nested scopes

$\texttt{Foo.f} \mapsto \texttt{g}$

```
src/foo.ml:

  let g = ...
  let f = ...
  let f = ...
    ⋮
  ...  ...

src/bar.ml:

  open Foo
    ⋮
  ... f ... g ...
```

OCaml nested scopes

Foo.f ↦ g

src/foo.ml:

```
let g = ...
let f = ...
let g = ...
   ⋮
... g ...
```

src/bar.ml:

```
open Foo
   ⋮
... g ... g ...
```

OCaml module signatures

src/foo.ml:

```
let f = ...
```

$\texttt{Foo.f} \mapsto g$

src/bar.ml:

```
include Foo
```

$\texttt{Bar.f} \mapsto g$

OCaml module signatures

src/foo.ml:

$\texttt{Foo.f} \mapsto g$

```
let f = ...
```

src/bar.ml:

$\texttt{Bar.f} \mapsto g$

```
include Foo
```

src/bar.mli:

```
include Sig.S
```

src/sig.ml:

```
module type S = sig val f :  ... end
```

OCaml module signatures

`src/foo.ml:` $\texttt{Foo.f} \mapsto g$

```
let f = ...
```

`src/bar.ml:` $\texttt{Bar.f} \mapsto g$

```
include Foo
```

`src/bar.mli:`

```
include Sig.S
```

`src/sig.ml:` $\texttt{Sig.S.f} \mapsto g$

```
module type S = sig val f : ... end
```

OCaml module signatures

`src/foo.ml:`

`Foo.f` ↦ g

```
let f = ...
```

`src/bar.ml:`

`Bar.f` ↦ g

```
include Foo
```

`src/bar.mli:`

```
include Sig.S
```

`src/sig.ml:`

`Sig.S.f` ↦ g

```
module type S = sig val f :  ... end
```

`src/baz.ml:`

```
module M : Sig.S = struct let f = ... end
```

OCaml module signatures

`src/foo.ml:` Foo.f ↦ g

```
let g = ...
```

`src/bar.ml:` Bar.f ↦ g

```
include Foo
```

`src/bar.mli:`

```
include Sig.S
```

`src/sig.ml:` Sig.S.f ↦ g

```
module type S = sig val g :  ... end
```

`src/baz.ml:`

```
module M : Sig.S = struct let g = ... end
```

There is more …

Punning

Module (type) aliases

Using structures to define signatures

Functors

A theory of refactoring dependencies

Towards Large-scale Refactoring for OCaml

REUBEN N. S. ROWE, University of Kent, UK
SIMON J. THOMPSON, University of Kent, UK

Refactoring is the process of changing the way a program works without changing its overall behaviour. The functional programming paradigm presents its own unique challenges to refactoring. For the OCaml language in particular, the expressiveness of its module system makes this a highly non-trivial task. The use of PPX preprocessors, other language extensions, and idiosyncratic build systems complicates matters further.

We begin to address the question of how to refactor large OCaml programs by looking at a particular refactoring—value binding renaming—and implementing a prototype tool to carry it out. Our tool, Rotor, is developed in OCaml itself and combines several features to manage the complexities of refactoring OCaml code. Firstly it defines a rich, hierarchical way of identifying bindings which distinguishes between structures and functors and their associated module types, and is able to refer directly to functor parameters. Secondly it makes use of the recently developed `visitors` library to perform generic traversals of abstract syntax trees. Lastly it implements a notion of 'dependency' between renamings, allowing refactorings to be computed in a modular fashion. We evaluate Rotor using a snapshot of Jane Street's core library and its dependencies, comprising some 900 source files across 80 libraries, and a test suite of around 3000 renamings.

We propose that the notion of dependency is a general one for refactoring, distinct from a refactoring 'precondition'. Dependencies may actually be *mutual*, in that all must be applied together for each one individually to be correct, and serve as declarative specifications of refactorings. Moreover, refactoring dependency graphs can be seen as abstract (semantic) representations.

CCS Concepts: • **Software and its engineering** → **Software notations and tools**; **Software maintenance tools**; • **Theory of computation** → **Semantics and reasoning**; **Abstraction**; *Program constructs*; *Functional constructs*;

Additional Key Words and Phrases: Refactoring, Renaming, Dependencies, Binding Structure, OCaml

ACM Reference Format:
Reuben N. S. Rowe and Simon J. Thompson. 2018. Towards Large-scale Refactoring for OCaml. *Proc. ACM Program. Lang.* 1, 1 (March 2018), 29 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Refactoring is a necessary and ongoing process in both the development and maintenance of any codebase [Fowler et al. 1999]. Individual refactoring steps are often conceptually very simple (e.g. rename this function from `foo` to `bar`, swap the order of parameters `x` and `y`). However applying them in practice can be complex, involving many repeated but subtly varying changes across the entire codebase. Moreover, refactorings are, by and large, context sensitive, meaning that even powerful low-tech utilities (e.g. `grep` and `sed`) are only effective up to a point.

Take as an example the renaming of a function, which is the refactoring that we focus on in this paper. As well as renaming the function at its definition point, every call of the function

---

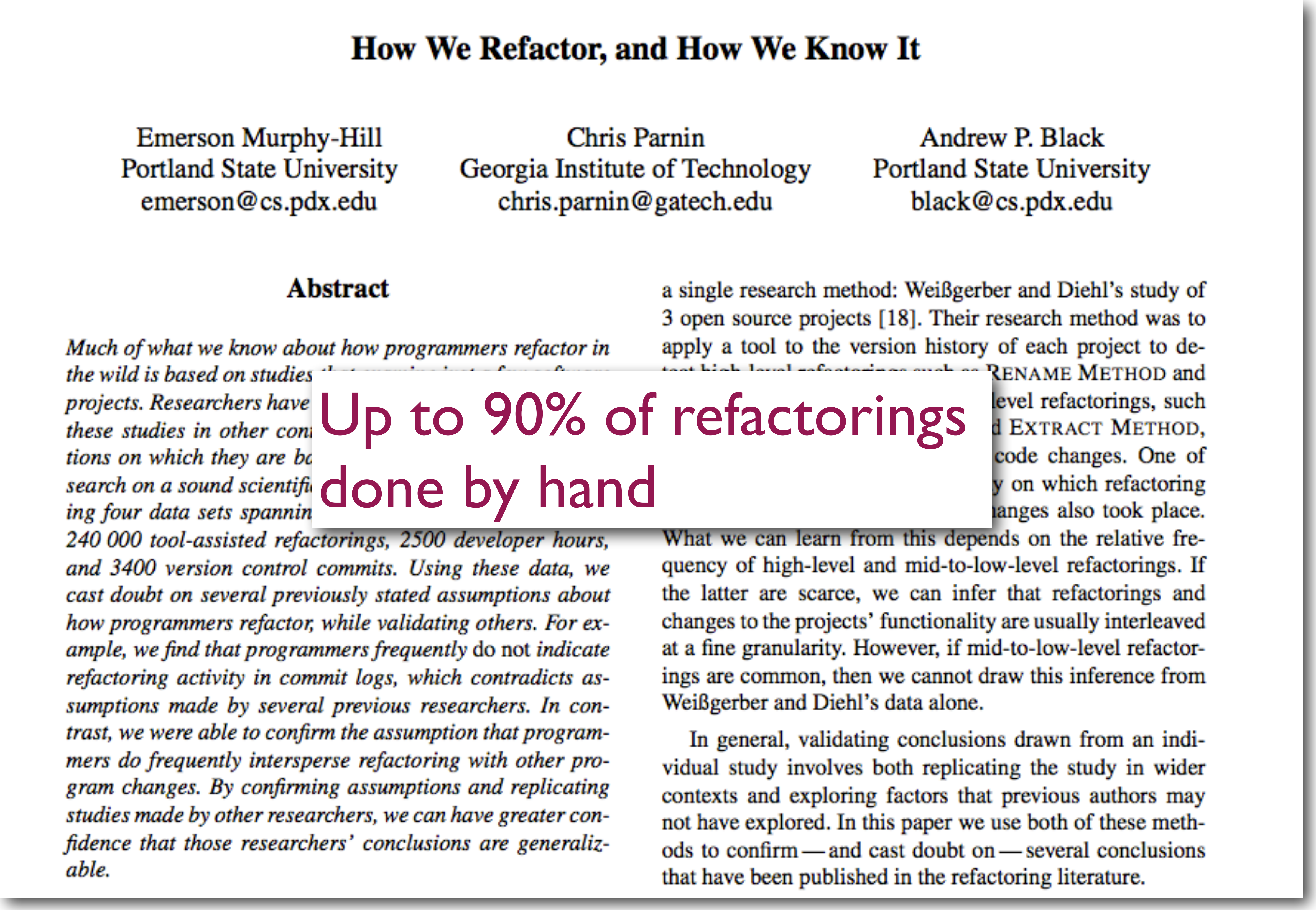
Authors' addresses: Reuben N. S. Rowe, University of Kent, Canterbury, Kent CT2 7NZ, UK, r.n.s.rowe@kent.ac.uk; Simon J. Thompson, University of Kent, Canterbury, Kent, CT2 7NZ, UK, s.j.thompson@kent.ac.uk.

2475-1421/2018/3-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Abandon any idea of building language-independent refactoring tools!

I don't need a refactoring tool …

… I have types!



How We Refactor, and How We Know It

Emerson Murphy-Hill
Portland State University
emerson@cs.pdx.edu

Chris Parnin
Georgia Institute of Technology
chris.parnin@gatech.edu

Andrew P. Black
Portland State University
black@cs.pdx.edu

Abstract

Much of what we know about how programmers refactor in the wild is based on studies
projects. Researchers have
these studies in other con
tions on which they are ba
search on a sound scientifi
ing four data sets spannin
240 000 tool-assisted refactorings, 2500 developer hours, and 3400 version control commits. Using these data, we cast doubt on several previously stated assumptions about how programmers refactor, while validating others. For example, we find that programmers frequently do not *indicate refactoring activity in commit logs, which contradicts assumptions made by several previous researchers. In contrast, we were able to confirm the assumption that programmers do frequently intersperse refactoring with other program changes. By confirming assumptions and replicating studies made by other researchers, we can have greater confidence that those researchers' conclusions are generalizable.*

a single research method: Weißgerber and Diehl's study of 3 open source projects [18]. Their research method was to apply a tool to the version history of each project to detect high-level refactorings such as RENAME METHOD and
level refactorings, such
d EXTRACT METHOD,
code changes. One of
y on which refactoring
hanges also took place.
What we can learn from this depends on the relative frequency of high-level and mid-to-low-level refactorings. If the latter are scarce, we can infer that refactorings and changes to the projects' functionality are usually interleaved at a fine granularity. However, if mid-to-low-level refactorings are common, then we cannot draw this inference from Weißgerber and Diehl's data alone.

In general, validating conclusions drawn from an individual study involves both replicating the study in wider contexts and exploring factors that previous authors may not have explored. In this paper we use both of these methods to confirm — and cast doubt on — several conclusions that have been published in the refactoring literature.

Up to 90% of refactorings done by hand

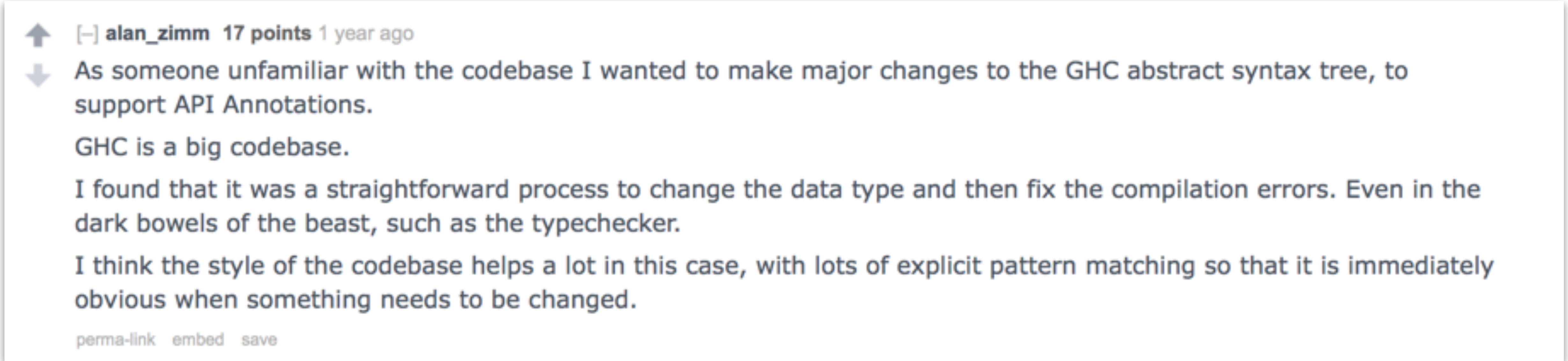
ICSE 2009

```
pipeline.
formation -> Effect IO ()
= fetchRawInputs runInfo.
> preprocessInputs
> addForecasts
> crea

from the database.
nInformation -> Producer RawInputs IO ()
eportId, runId) = do
. connectToDatabase $ rsaConfig^.db.dbHost
. runDbAction mongoPipe . handleErr $ getRunKeys runId
. runDbAction mongoPipe $ getRawInputs reportId keys
goPipe

inputs and yield them downstream.
```

SOFTWARE PROJECT MAINTENANCE IS WHERE HASKELL SHINES.

Posted by Chris Done - 31 December, 2016

https://www.fpcomplete.com/blog/2016/12/software-project-maintenance-is-where-haskell-shines



[–] **alan_zimm** **17 points** 1 year ago

As someone unfamiliar with the codebase I wanted to make major changes to the GHC abstract syntax tree, to support API Annotations.

GHC is a big codebase.

I found that it was a straightforward process to change the data type and then fix the compilation errors. Even in the dark bowels of the beast, such as the typechecker.

I think the style of the codebase helps a lot in this case, with lots of explicit pattern matching so that it is immediately obvious when something needs to be changed.

perma-link embed save

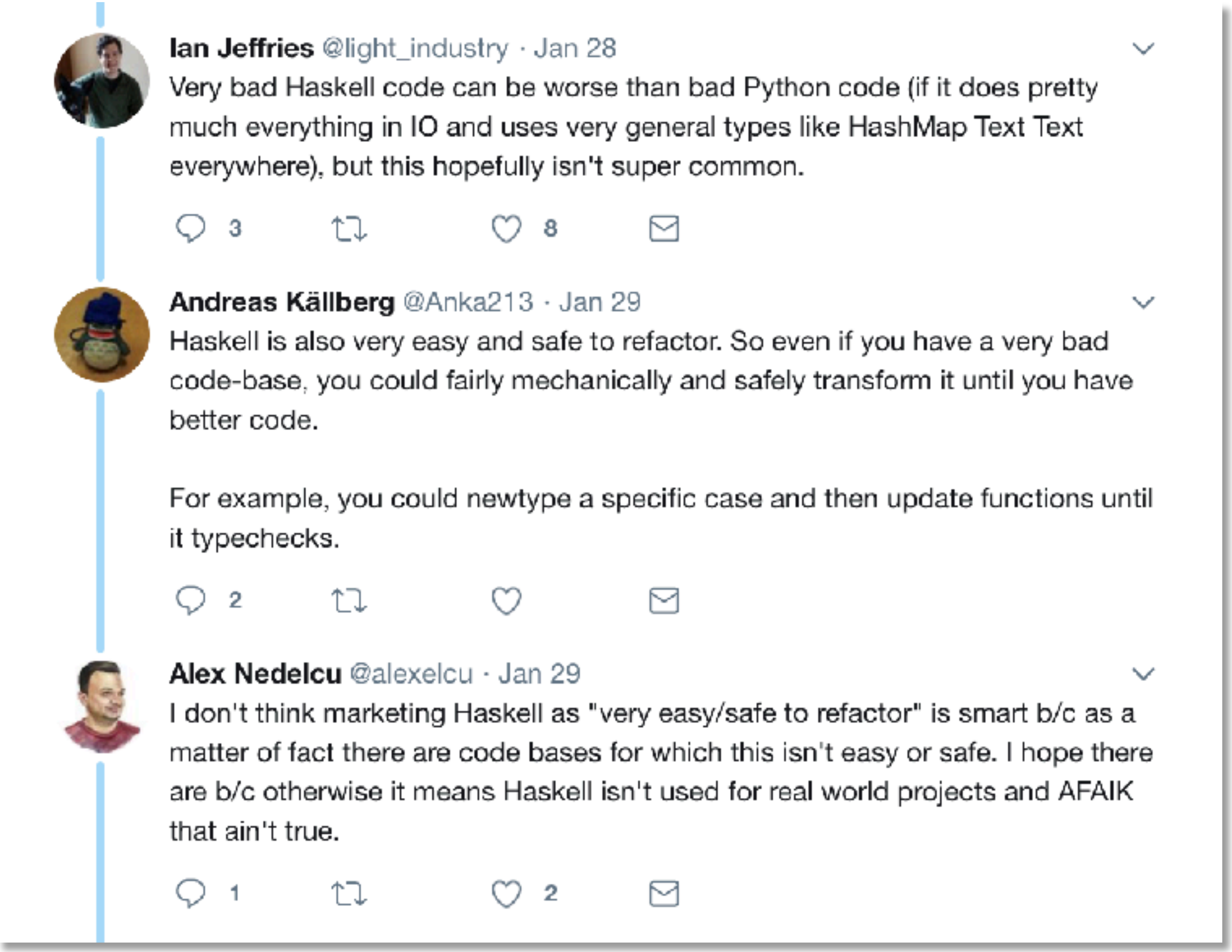
https://www.reddit.com/r/haskell/comments/65d510/experience_reports_on_refactoring_haskell_code/

But is it really as simple as that … ?

Changes in bindings – e.g. name capture – can give code that compiles and type checks, but gives different results.

Are you really prepared to fix 1,000 type error messages?

Maybe just be risk averse …



Ian Jeffries @light_industry · Jan 28

Very bad Haskell code can be worse than bad Python code (if it does pretty much everything in IO and uses very general types like HashMap Text Text everywhere), but this hopefully isn't super common.

Andreas Källberg @Anka213 · Jan 29

Haskell is also very easy and safe to refactor. So even if you have a very bad code-base, you could fairly mechanically and safely transform it until you have better code.

For example, you could newtype a specific case and then update functions until it typechecks.

Alex Nedelcu @alexelcu · Jan 29

I don't think marketing Haskell as "very easy/safe to refactor" is smart b/c as a matter of fact there are code bases for which this isn't easy or safe. I hope there are b/c otherwise it means Haskell isn't used for real world projects and AFAIK that ain't true.

Replace lists with “Hughes lists”

```
explode :: [a] -> [a]
explode lst = concat (map (\x -> replicate (length lst) x) lst)
```

Replace lists with “Hughes lists”

```haskell
explode :: [a] -> [a]
explode lst = concat (map (\x -> replicate (length lst) x) lst)
```

```haskell
explode :: DList a -> DList a
explode lst =
    DL.concat
        (DL.map
            (\x -> DL.replicate (length (DL.toList lst)) x) lst)
```

From `Monad` to `Applicative`

```
moduleDef :: LParser Module
moduleDef = do
    reserved "module"
    modName <- identifier
    reserved "where"
    imports <- layout importDef (return ()) decls <- layout decl (return ())
    cnames <- get
    return $ Module modName imports decls cnames
```

From Monad to Applicative

```haskell
moduleDef :: LParser Module
moduleDef = do
    reserved "module"
    modName <- identifier
    reserved "where"
    imports <- layout importDef (return ()) decls <- layout decl (return ())
    cnames <- get
    return $ Module modName imports decls cnames
```

```haskell
moduleDef :: LParser Module
moduleDef = Module
    <$> (reserved "module" *> identifier <* reserved "where")
    <*> layout importDef (return ())
    <*> layout decl (return ())
    <*> get
```

From `List` to `Vector`

```
map :: (a -> b) -> [a] -> [b]
app :: [a] -> [a] -> [a]
filter :: (a -> Bool) -> [a] -> [a]
take :: Int -> [a] -> [a]
```

From `List` to `Vector`

```
map :: (a -> b) -> [a] -> [b]
app :: [a] -> [a] -> [a]
filter :: (a -> Bool) -> [a] -> [a]
take :: Int -> [a] -> [a]
```

```
vmap :: (a -> b) -> (Vec n a) -> (Vec n b)
vapp :: (Vec n a) -> (Vec m a) -> (Vec n+m a)
vfilter :: (a -> Bool) -> (Vec n a) -> (Vecs n a)
vtake :: (n :: Int) -> (Vec m a) -> (Vec (min n m) a)
vtake :: (n :: Int) -> (Vec m a) -> (Vecs n a)
```

Types vs refactorings?

The more precise the typings, the more fragile the structure.

Difficulty of getting it right first time: `Vec` vs `Vecs` vs …

```
vmap :: (a -> b) -> (Vec n a) -> (Vec n b)
vapp :: (Vec n a) -> (Vec m a) -> (Vec n+m a)
vfilter :: (a -> Bool) -> (Vec n a) -> (Vecs n a)
vtake :: (n :: Int) -> (Vec m a) -> (Vec (min n m) a)
vtake :: (n :: Int) -> (Vec m a) -> (Vecs n a)
```

Types might both help and hinder effective refactoring

What’s so wrong with duplicated code?

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.

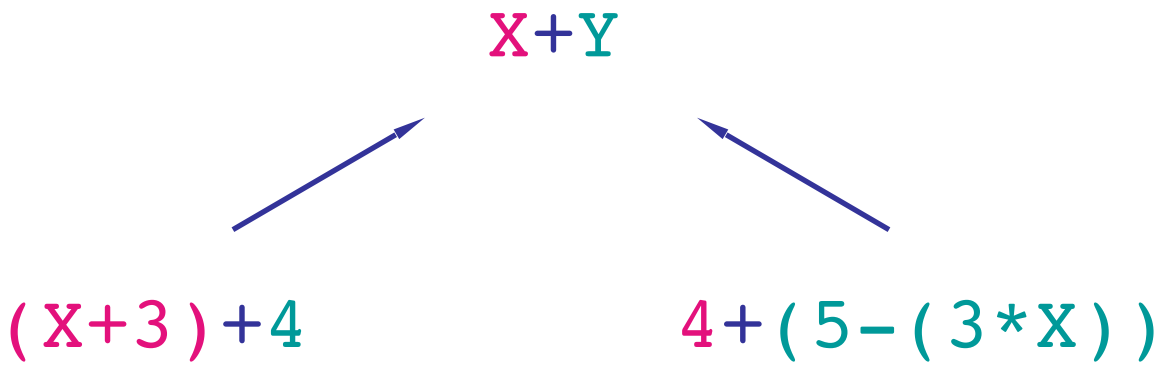
But … it’s not always a problem.

What is similar code?

`(X+3)+4`

`4+(5-(3*X))`

What is similar code?

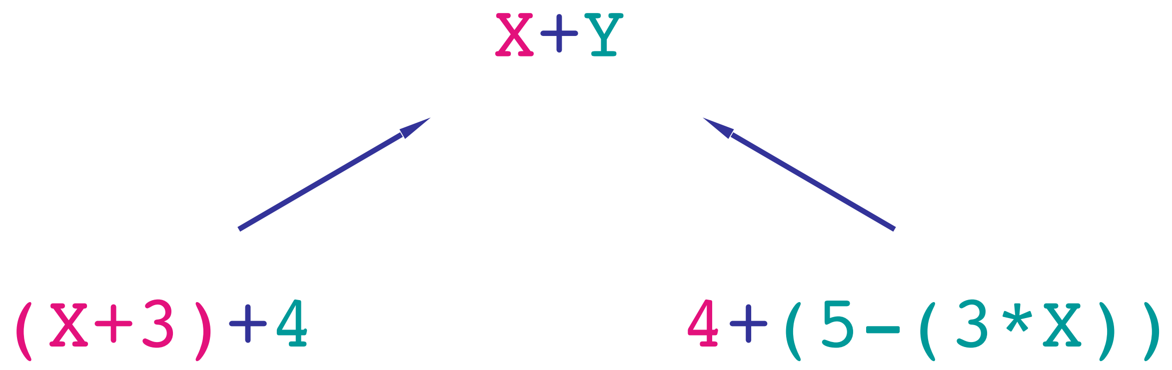


`X+Y`

`(X+3)+4`

`4+(5-(3*X))`

What is similar code?



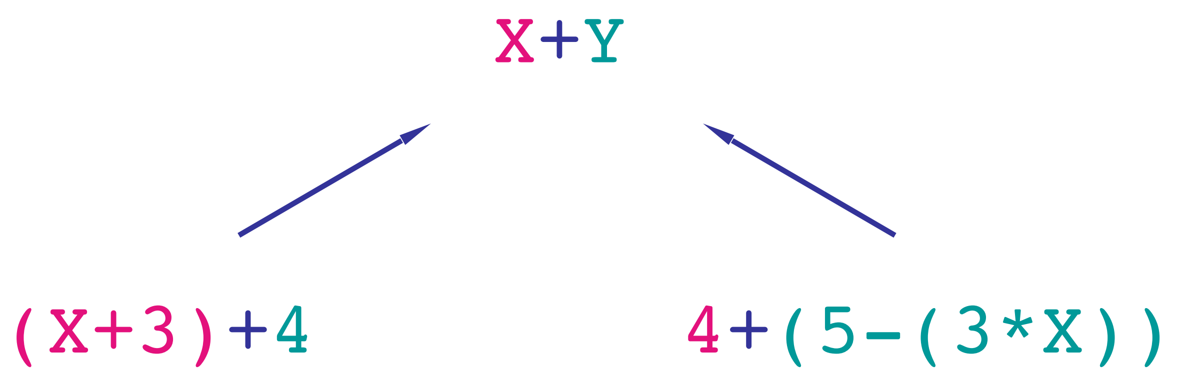
`X+Y`

`(X+3)+4`

`4+(5-(3*X))`

The anti-unification gives the (most specific) common generalisation.

What is similar code?



`X+Y`

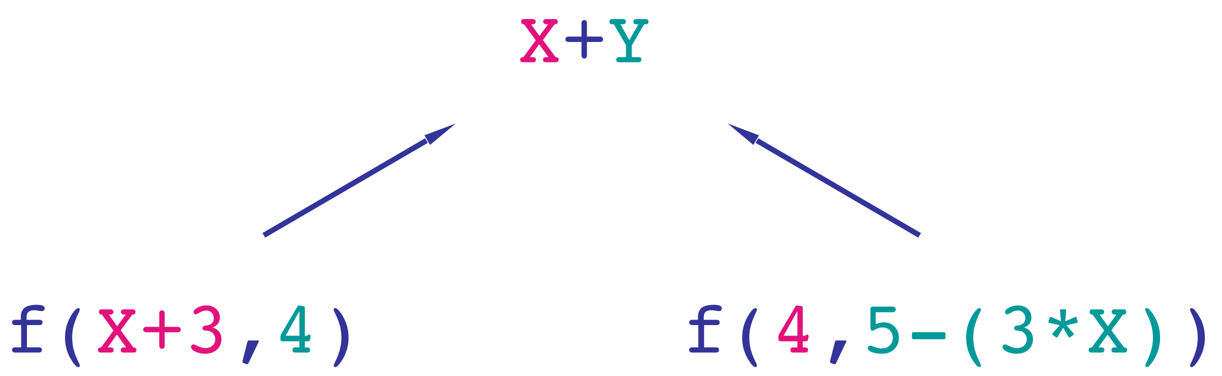
`(X+3)+4`

`4+(5-(3*X))`

```
f(Z,W) -> X+Y.
```

The anti-unification gives the (most specific) common generalisation.

What is similar code?



`X+Y`

`f(X+3,4)`

`f(4,5-(3*X))`

```
f(Z,W) -> X+Y.
```

The anti-unification gives the (most specific) common generalisation.

What makes a clone (in Erlang)?

Thresholds

- Number of expressions
- Number of tokens
- Number of variables introduced
- Similarity = $\min_{i=1..n}(\text{size}(Gen)/\text{size}(E_i))$

What makes a clone (in Erlang)?

Thresholds … and their defaults

- Number of expressions ≥ 5
- Number of tokens ≥ 20
- Number of variables introduced ≤ 4
- Similarity = $\min_{i=1..n}(\mathrm{size}(Gen)/\mathrm{size}(E_i)) \geq 0.8$

Clone detection and removal

Find a clone, name it and its parameters, and eliminate.

What could go wrong?

What could go wrong?

Naming can’t be automated, nor the order of eliminating.

Bottom-up or top-down?

Widows and orphans, sub-clones, premature generalisation, …

What could go wrong?

```
new_fun(FilterName, NewVar_1) ->
  FilterKey = ?SMM_CREATE_FILTER_CHECK(FilterName),
  %%Add rulests to filter
  RuleSetNameA = "a",
  RuleSetNameB = "b",
  RuleSetNameC = "c",
  RuleSetNameD = "d",
  ... 16 lines which handle the rules sets are elided ...
  %%Remove rulesets
  NewVar_1,
{RuleSetNameA, RuleSetNameB, RuleSetNameC, RuleSetNameD, FilterKey}.
```

Widows and orphans, sub-clones, premature generalisation, …

```
new_fun(FilterName, FilterKey) ->
  %%Add rulests to filter
  RuleSetNameA = "a",
  RuleSetNameB = "b",
  RuleSetNameC = "c",
  RuleSetNameD = "d",
  ... 16 lines which handle the rules sets are elided ...
  %%Remove rulesets

{RuleSetNameA, RuleSetNameB, RuleSetNameC, RuleSetNameD}.
```

What could go wrong?

Naming can’t be automated, nor the order of eliminating.

Bottom-up or top-down?

Widows and orphans, sub-clones, premature generalisation, …

Bring in the experts

With a domain expert …

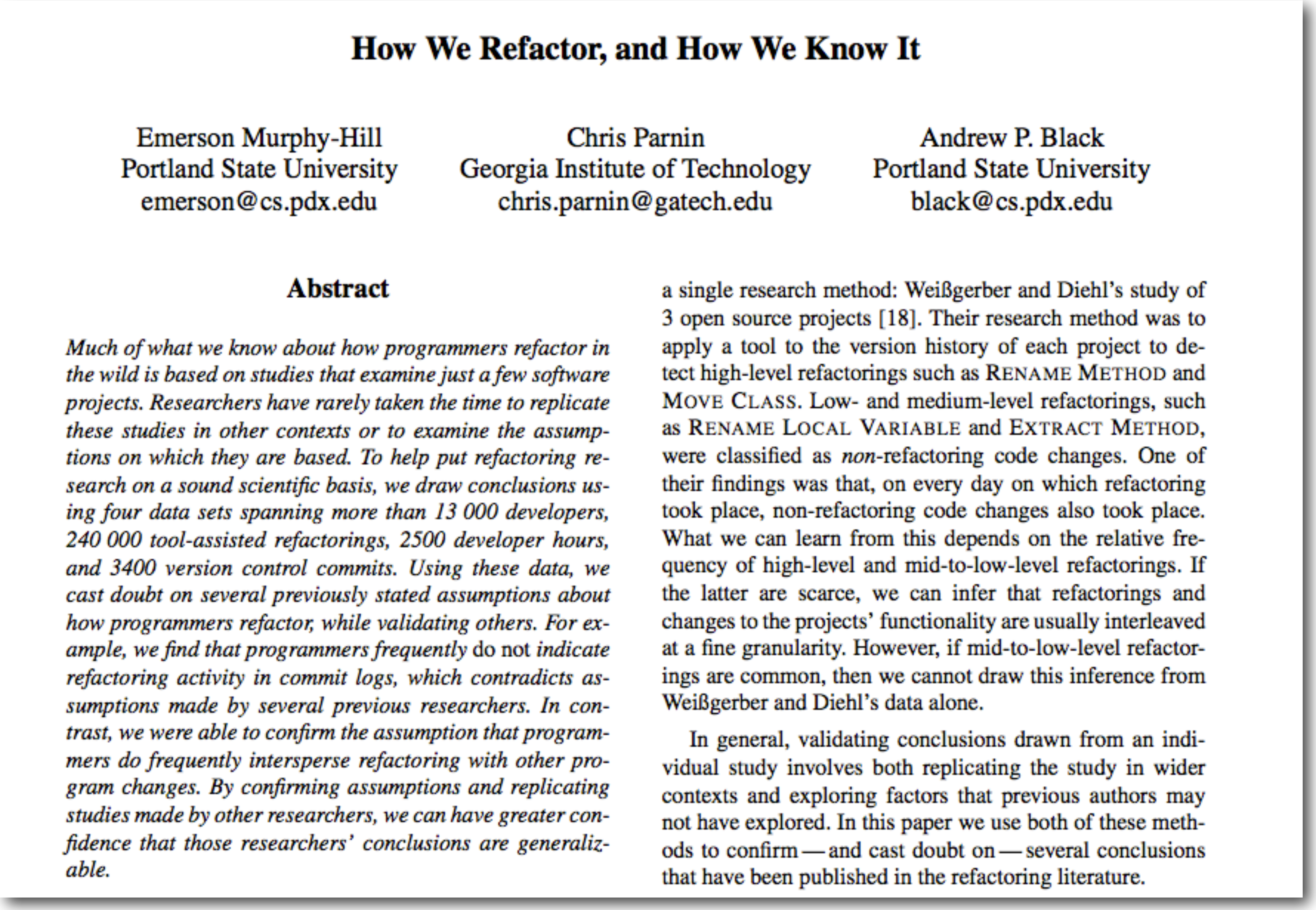
can choose in the right order,

name the clones and their parameters, …

And the domain expert can learn in the process …

e.g. test code example from Ericsson.

Why haven't you implemented this refactoring?



How We Refactor, and How We Know It

Emerson Murphy-Hill
Portland State University
emerson@cs.pdx.edu

Chris Parnin
Georgia Institute of Technology
chris.parnin@gatech.edu

Andrew P. Black
Portland State University
black@cs.pdx.edu

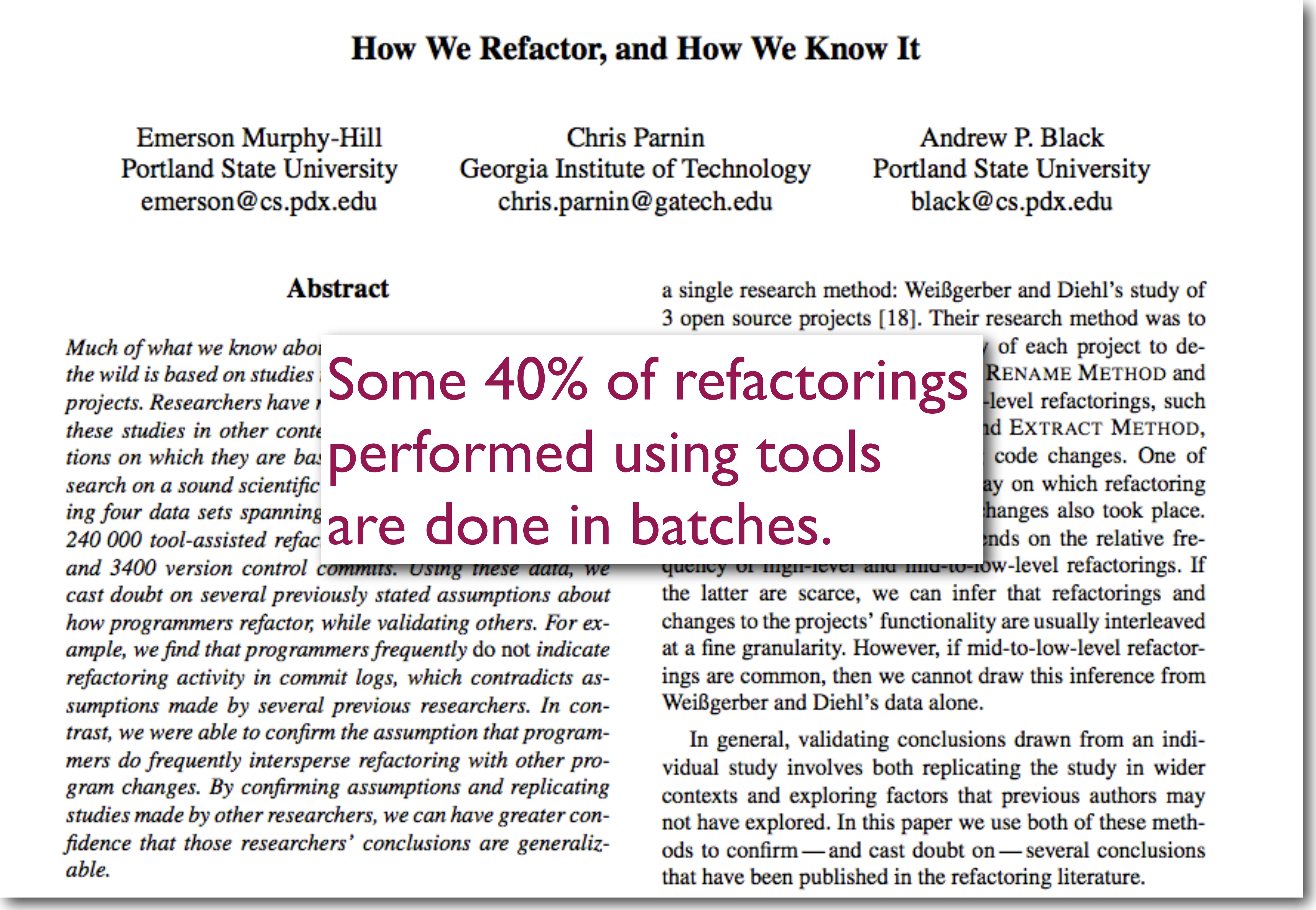
Abstract

Much of what we know about how programmers refactor in the wild is based on studies that examine just a few software projects. Researchers have rarely taken the time to replicate these studies in other contexts or to examine the assumptions on which they are based. To help put refactoring research on a sound scientific basis, we draw conclusions using four data sets spanning more than 13 000 developers, 240 000 tool-assisted refactorings, 2500 developer hours, and 3400 version control commits. Using these data, we cast doubt on several previously stated assumptions about how programmers refactor, while validating others. For example, we find that programmers frequently do not *indicate refactoring activity in commit logs, which contradicts assumptions made by several previous researchers. In contrast, we were able to confirm the assumption that programmers do frequently intersperse refactoring with other program changes. By confirming assumptions and replicating studies made by other researchers, we can have greater confidence that those researchers' conclusions are generalizable.*

a single research method: Weißgerber and Diehl's study of 3 open source projects [18]. Their research method was to apply a tool to the version history of each project to detect high-level refactorings such as RENAME METHOD and MOVE CLASS. Low- and medium-level refactorings, such as RENAME LOCAL VARIABLE and EXTRACT METHOD, were classified as *non*-refactoring code changes. One of their findings was that, on every day on which refactoring took place, non-refactoring code changes also took place. What we can learn from this depends on the relative frequency of high-level and mid-to-low-level refactorings. If the latter are scarce, we can infer that refactorings and changes to the projects' functionality are usually interleaved at a fine granularity. However, if mid-to-low-level refactorings are common, then we cannot draw this inference from Weißgerber and Diehl's data alone.

In general, validating conclusions drawn from an individual study involves both replicating the study in wider contexts and exploring factors that previous authors may not have explored. In this paper we use both of these methods to confirm — and cast doubt on — several conclusions that have been published in the refactoring literature.

ICSE 2009



How We Refactor, and How We Know It

Emerson Murphy-Hill
Portland State University
emerson@cs.pdx.edu

Chris Parnin
Georgia Institute of Technology
chris.parnin@gatech.edu

Andrew P. Black
Portland State University
black@cs.pdx.edu

Abstract

Much of what we know abo
the wild is based on studies
projects. Researchers have
these studies in other conte
tions on which they are bas
search on a sound scientific
ing four data sets spanning
240 000 tool-assisted refac
and 3400 version control commits. Using these data, we cast doubt on several previously stated assumptions about how programmers refactor, while validating others. For example, we find that programmers frequently do not *indicate refactoring activity in commit logs, which contradicts assumptions made by several previous researchers. In contrast, we were able to confirm the assumption that programmers do frequently intersperse refactoring with other program changes. By confirming assumptions and replicating studies made by other researchers, we can have greater confidence that those researchers' conclusions are generalizable.*

a single research method: Weißgerber and Diehl's study of 3 open source projects [18]. Their research method was to
of each project to de-
RENAME METHOD and
-level refactorings, such
d EXTRACT METHOD,
code changes. One of
ay on which refactoring
hanges also took place.
nds on the relative fre-
quency of high-level and mid-to-low-level refactorings. If the latter are scarce, we can infer that refactorings and changes to the projects' functionality are usually interleaved at a fine granularity. However, if mid-to-low-level refactorings are common, then we cannot draw this inference from Weißgerber and Diehl's data alone.

In general, validating conclusions drawn from an individual study involves both replicating the study in wider contexts and exploring factors that previous authors may not have explored. In this paper we use both of these methods to confirm — and cast doubt on — several conclusions that have been published in the refactoring literature.

Some 40% of refactorings performed using tools are done in batches.

ICSE 2009

API: templates and rules … in Erlang

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

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

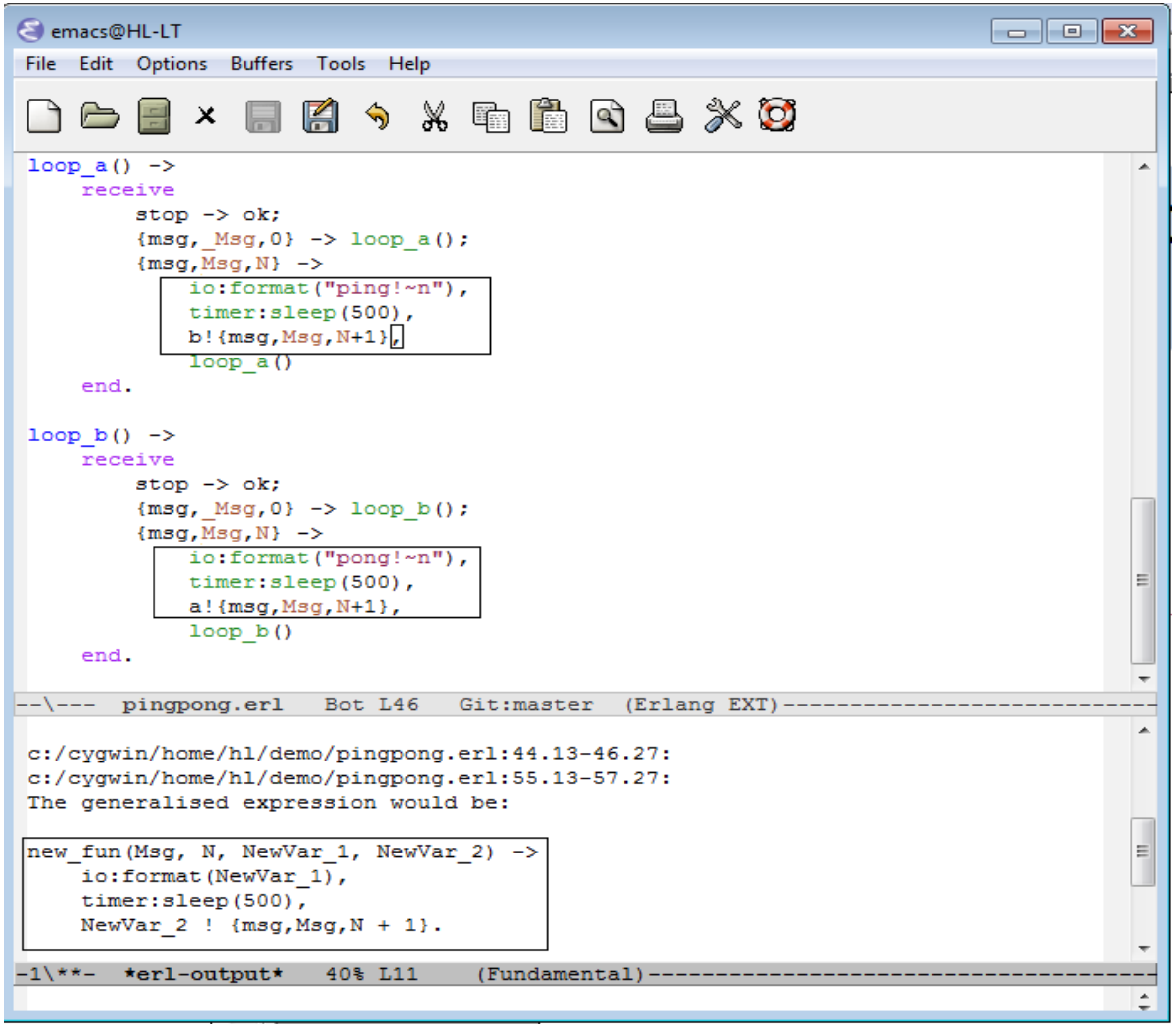
API: templates and rules … in Erlang

```
?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 …
```

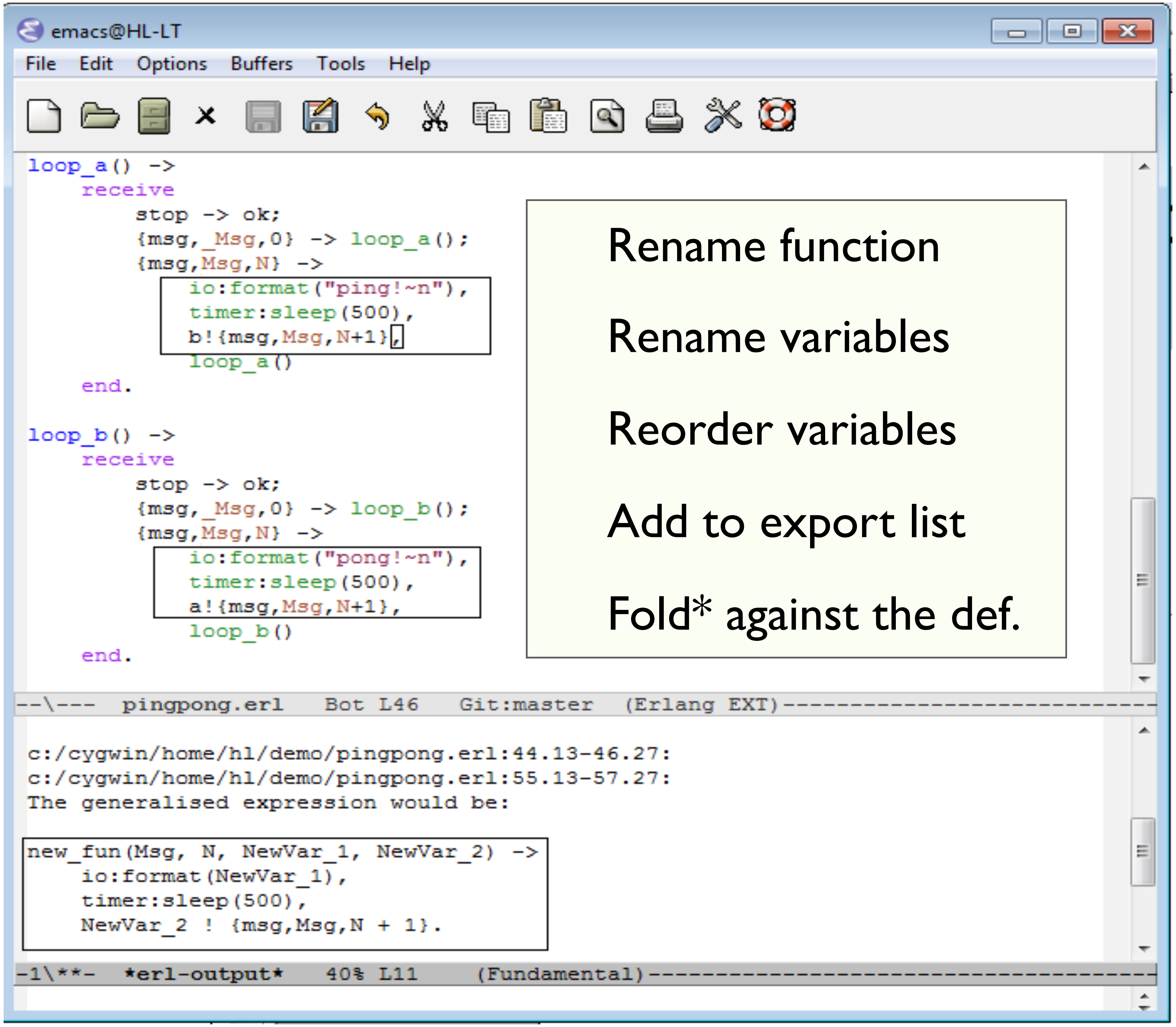
Clone removal



```
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)------------------------------------
```

Clone removal



```
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)----------------------------
```

- Rename function
- Rename variables
- Reorder variables
- Add to export list
- Fold* against the def.

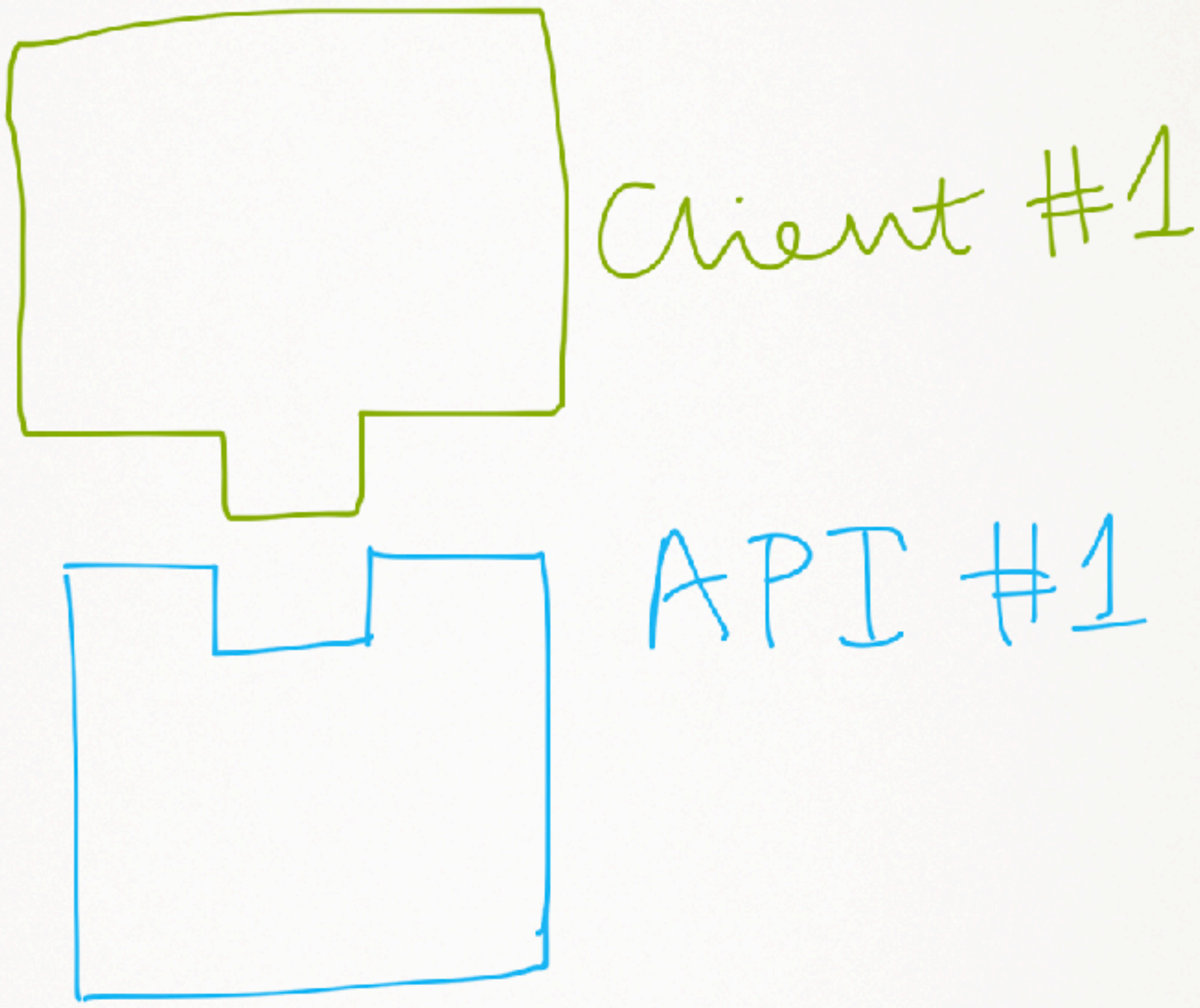
```
c:/cygwin/home/hl/demo/pingpong.erl:44.13-46.27:
c:/cygwin/home/hl/demo/pingpong.erl:55.13-57.27:
The generalised expression would be:

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)-------------------------------------
```

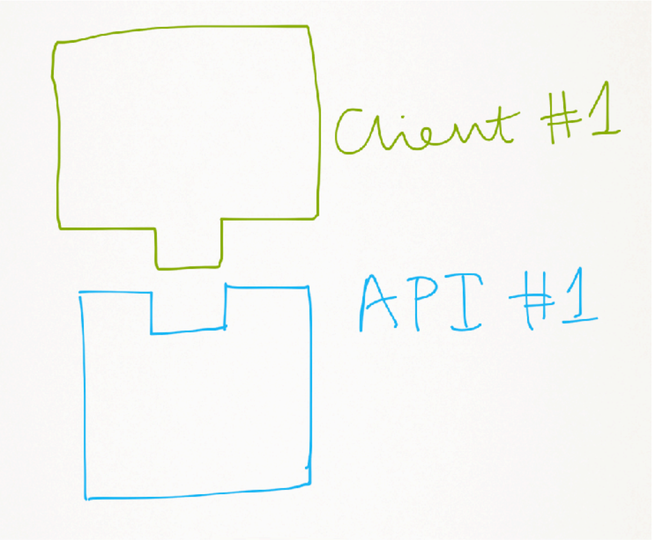
Clone removal in the DSL

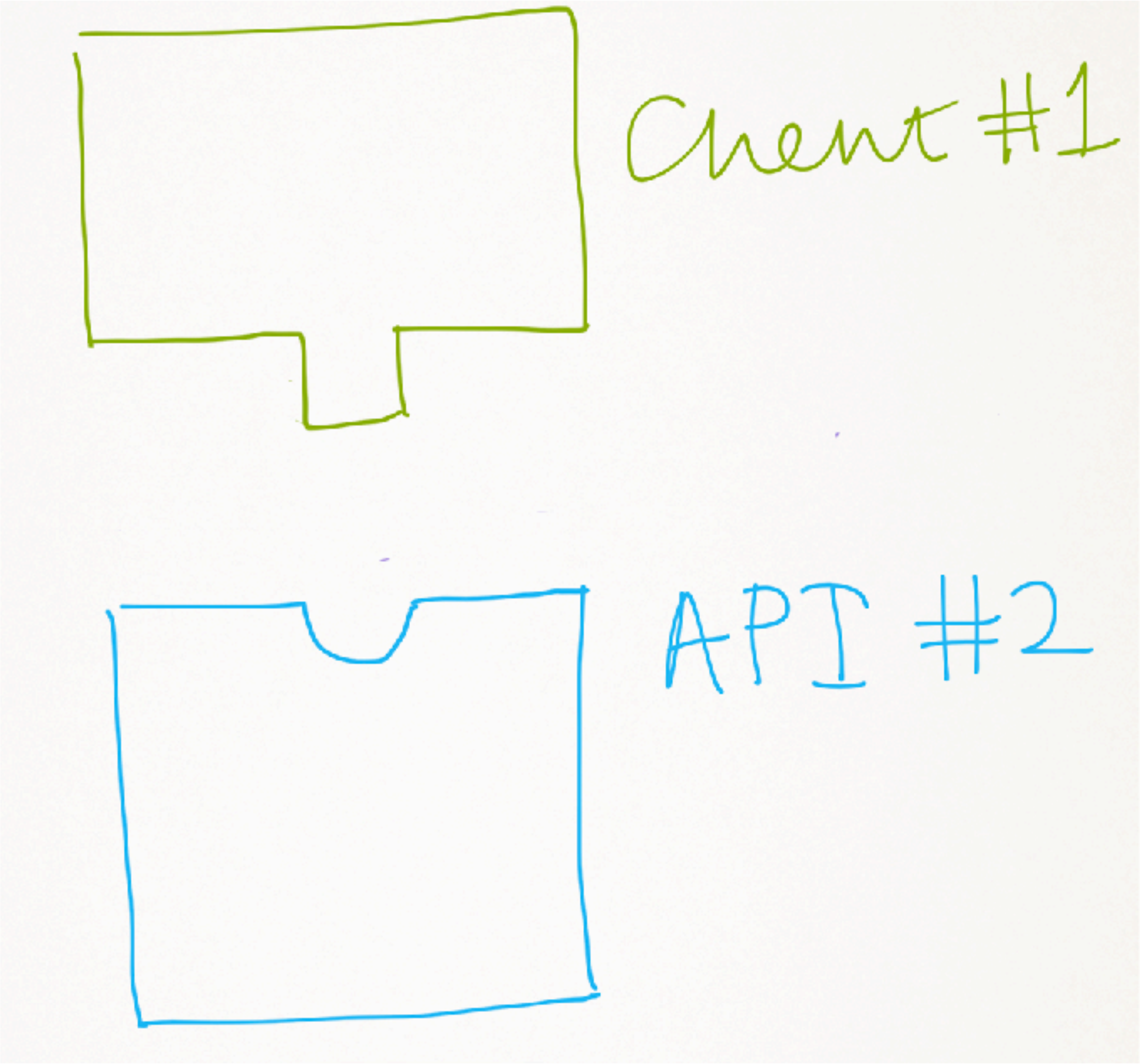
Transaction as a whole … non-transactional components OK.

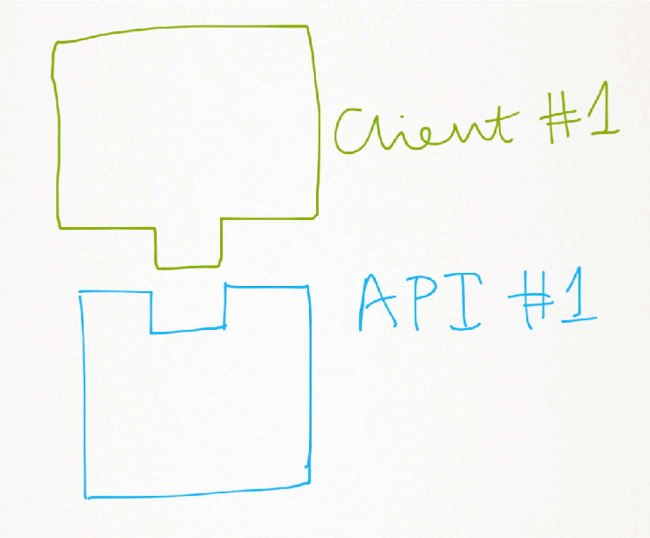
Not just an API: `?transaction` etc. modify interpretation of what they enclose …

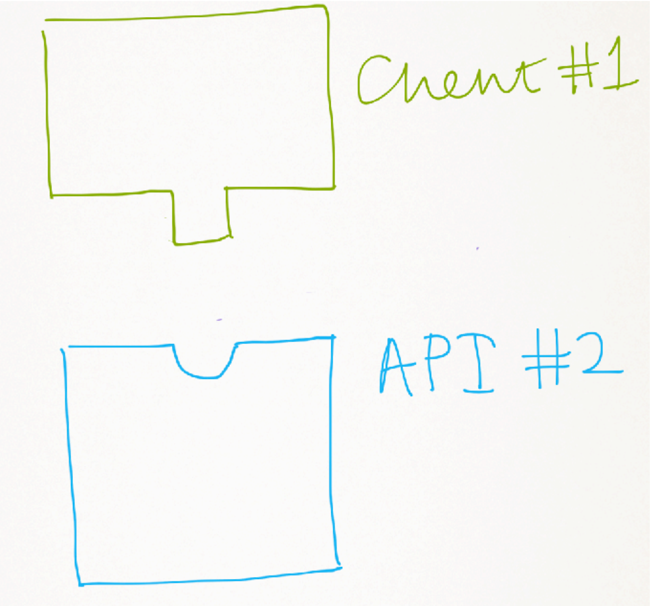
```
?transaction(
     [?interactive( RENAME FUNCTION )
      ?refac_( RENAME ALL VARIABLES OF THE FORM NewVar*)
      ?repeat_interactive( SWAP ARGUMENTS )
      ?if_then( EXPORT IF NOT ALREADY )
      ?non_transaction( FOLD INSTANCES OF THE CLONE )
     ]).
```

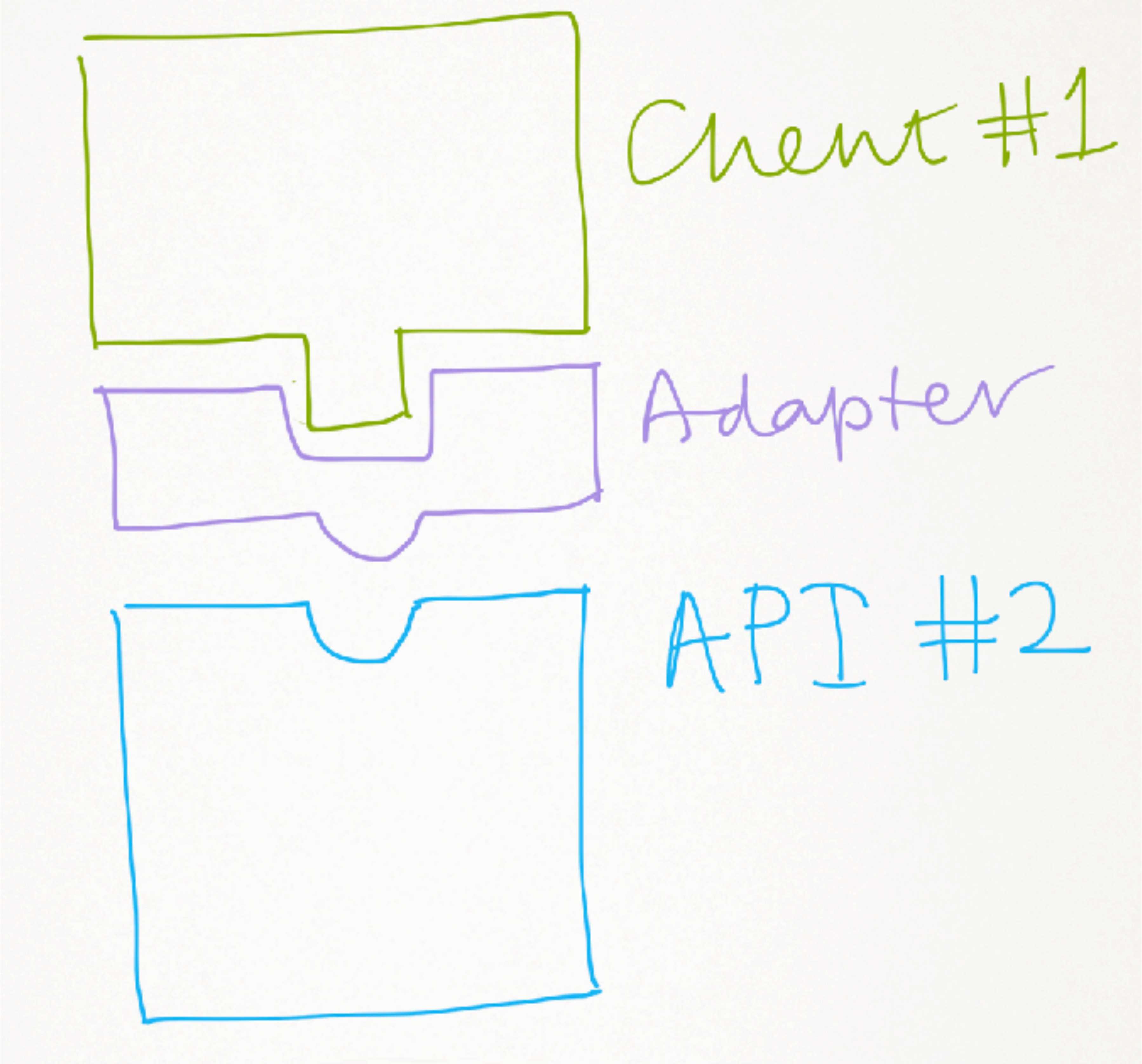

Client #1
API #1

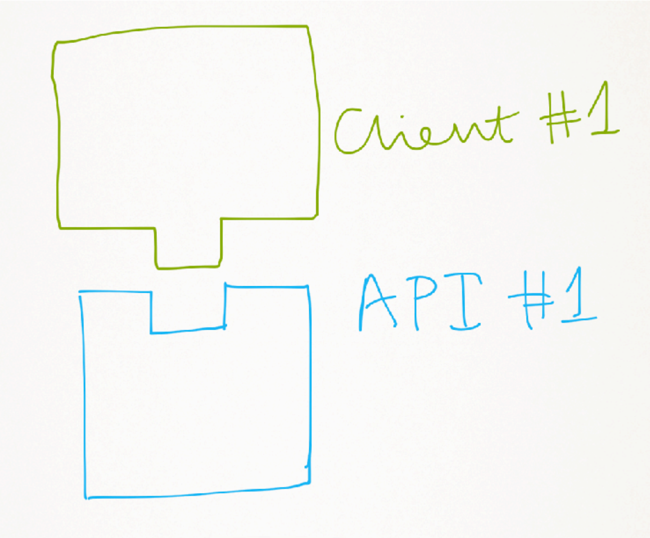

Client #1
API #1

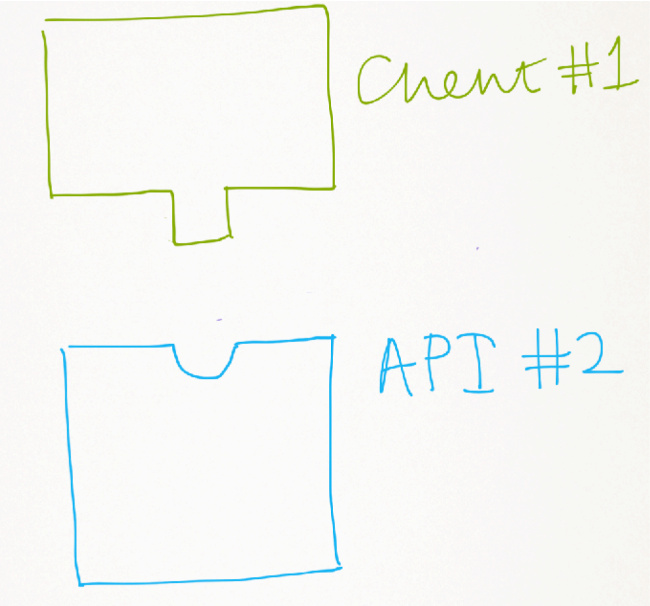

Client #1
API #2

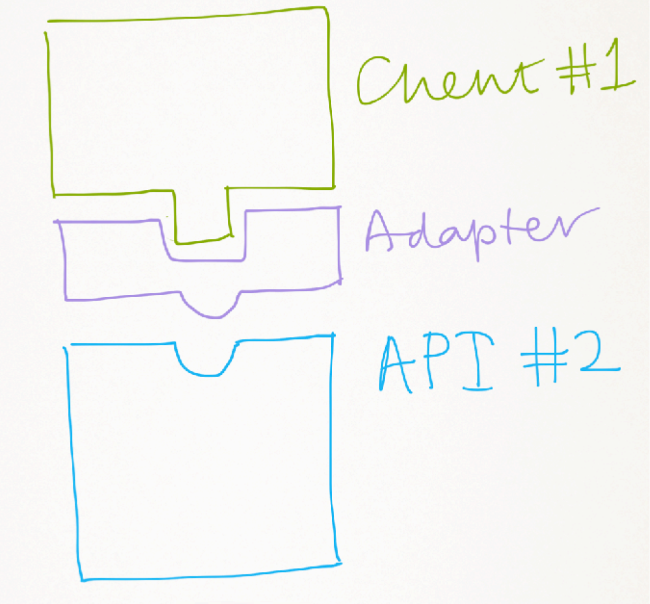

Client #1
API #1

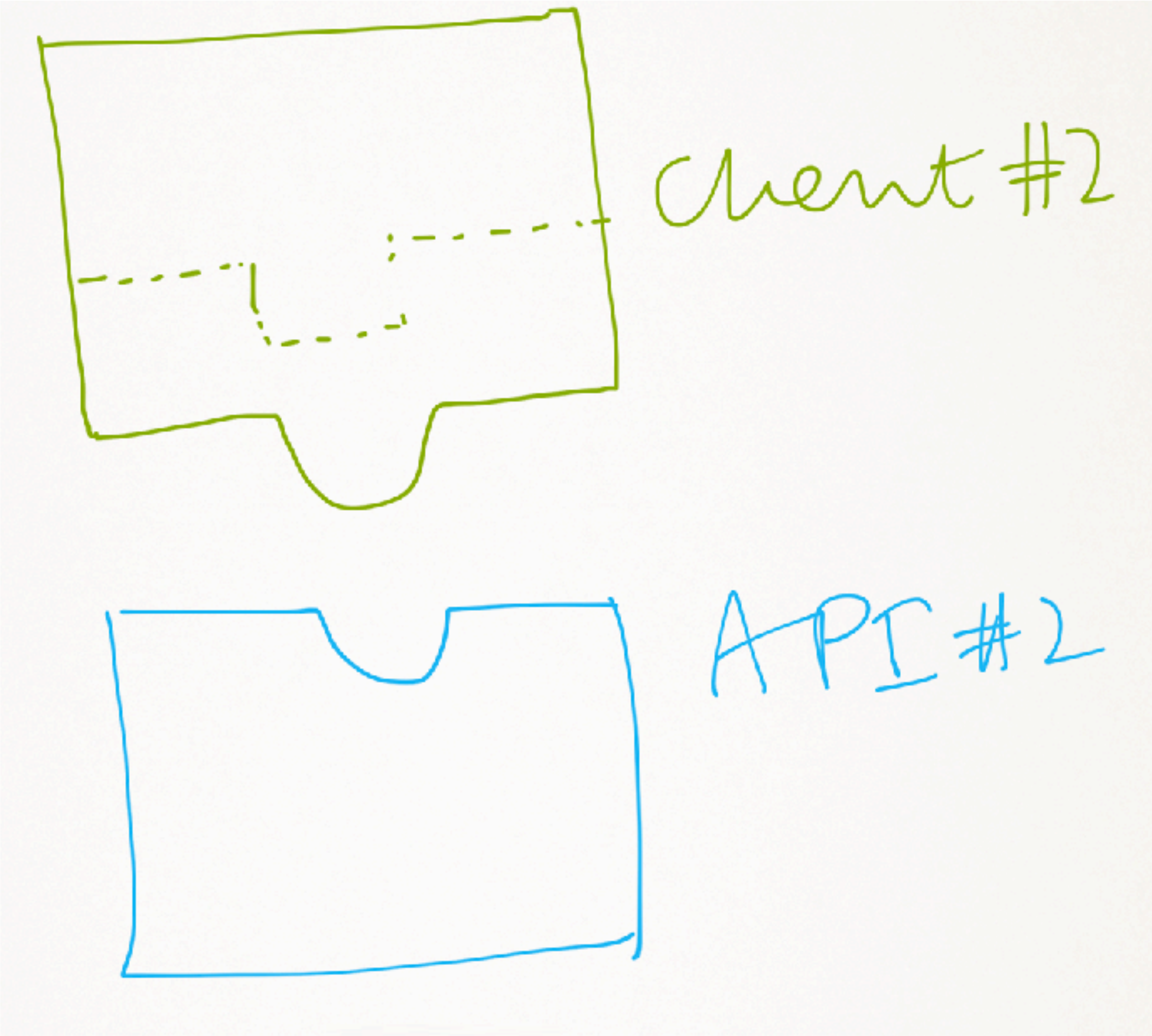

Client #1
API #2

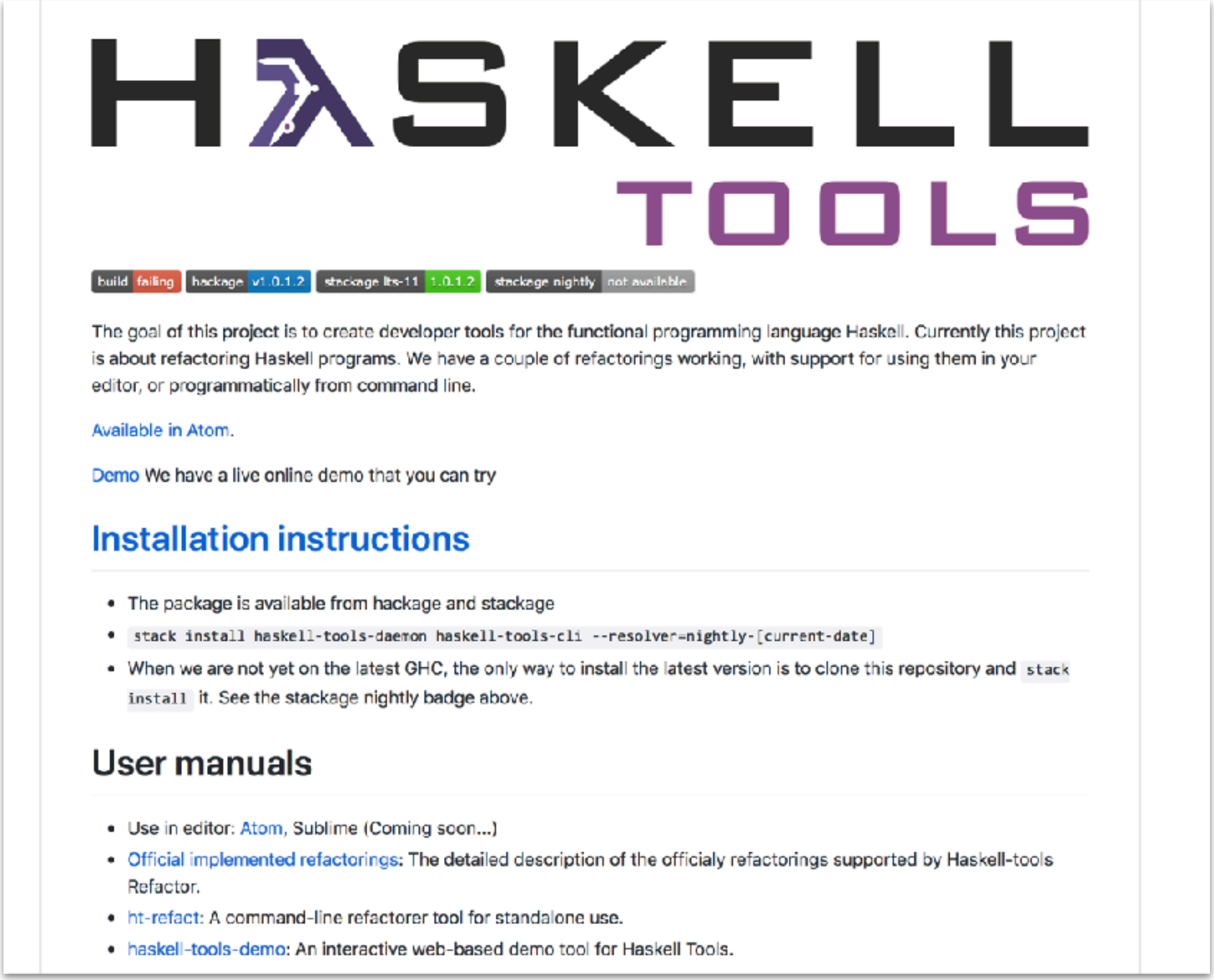

Client #1
Adapter
API #2


Client #1
API #1


Client #1
API #2


Client #1
Adapter
API #2


Client #2
API #2



The goal of this project is to create developer tools for the functional programming language Haskell. Currently this project is about refactoring Haskell programs. We have a couple of refactorings working, with support for using them in your editor, or programmatically from command line.

Available in Atom.

Demo We have a live online demo that you can try

Installation instructions

- The package is available from hackage and stackage
- `stack install haskell-tools-daemon haskell-tools-cli --resolver=nightly-[current-date]`
- When we are not yet on the latest GHC, the only way to install the latest version is to clone this repository and `stack install` it. See the stackage nightly badge above.

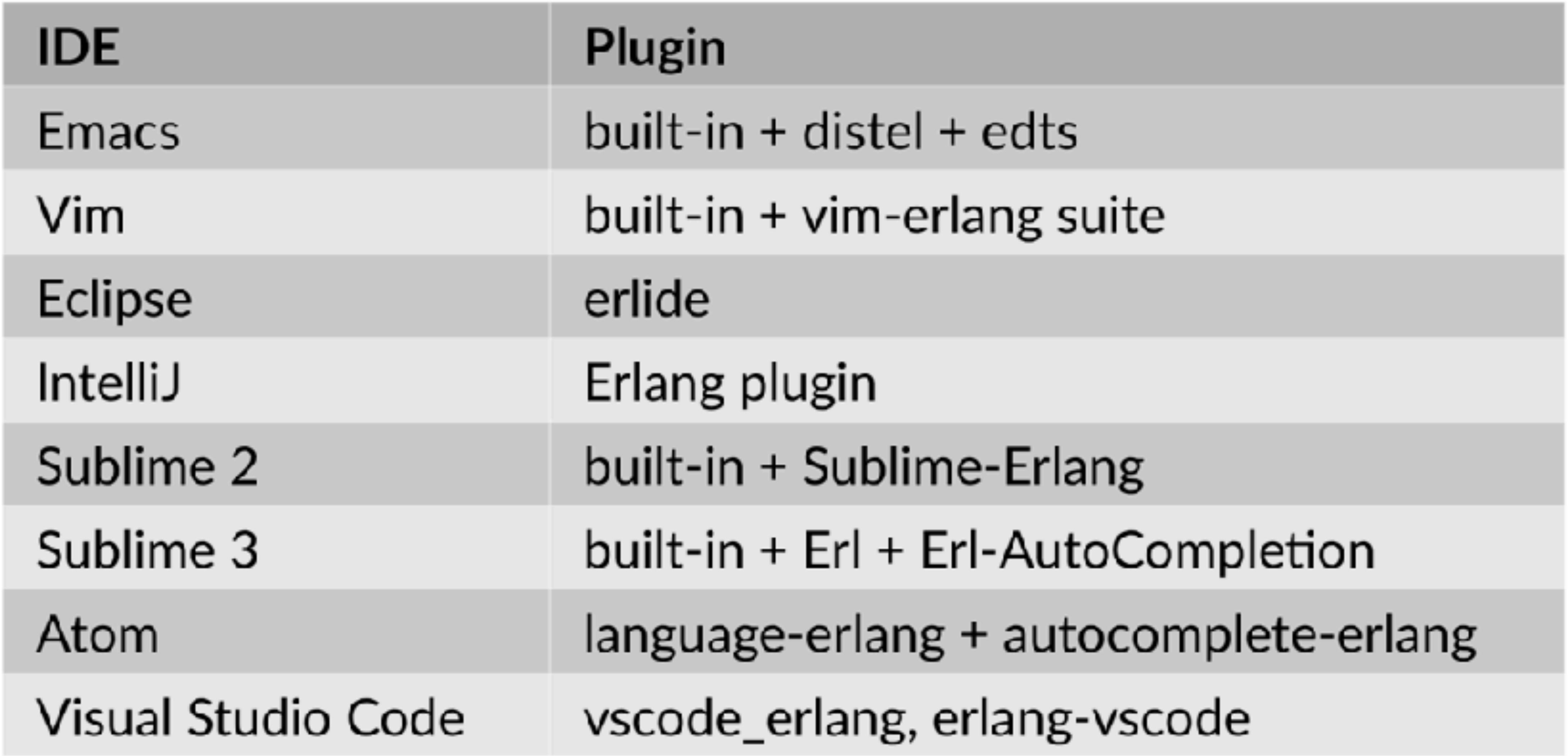
User manuals

- Use in editor: Atom, Sublime (Coming soon...)
- Official implemented refactorings: The detailed description of the officialy refactorings supported by Haskell-tools Refactor.
- ht-refact: A command-line refactorer tool for standalone use.
- haskell-tools-demo: An interactive web-based demo tool for Haskell Tools.

It's better to implement libraries, APIs and DSLs than individual refactorings

Will you integrate with this editor or IDE?

Can we integrate with every editor/IDE?



IDE	Plugin
Emacs	built-in + distel + edts
Vim	built-in + vim-erlang suite
Eclipse	erlide
IntelliJ	Erlang plugin
Sublime 2	built-in + Sublime-Erlang
Sublime 3	built-in + Erl + Erl-AutoCompletion
Atom	language-erlang + autocomplete-erlang
Visual Studio Code	vscode_erlang, erlang-vscode

Thanks to Csaba Hoch: this table from his CODE BEAM 2018 talk.

Can we integrate with every editor/IDE?

Hard work!

Keep it simple? Command-line tooling.

Some support from Language Server Protocol?

Support from Open Source collaborators

Shout out for

- Richard Carlsson of Klarna for Wrangler contributions
- Alan Zimmerman for porting HaRe to GHC API


Please help!

Why have you messed up the layout of my program?

Appearance must be right

```
my_list() ->
    [ foo,
      bar,
      baz,
      wombat
    ]

my_funny_list() ->
    [ foo
      ,bar
      ,baz
      ,wombat
    ]
```

Appearance must be right

```
my_list() ->
    [ foo,
      bar,
      baz,
      wombat
    ]

my_funny_list() ->
    [ foo
      ,bar
      ,baz
      ,wombat
    ]
```

```
{v1, v2, v3}

{v1,v2,v3}
```

Appearance must be right

```
my_list() ->
    [ foo,
      bar,
      baz,
      wombat
    ]

my_funny_list() ->
    [ foo
      ,bar
      ,baz
      ,wombat
    ]
```

```
{v1, v2, v3}

{v1,v2,v3}
```

```
f (g x y)

f $ g x y
```

Appearance must be right

```
my_list() ->
    [ foo,
      bar,
      baz,
      wombat
    ]

my_funny_list() ->
    [ foo
      ,bar
      ,baz
      ,wombat
    ]
```

```
{v1, v2, v3}

{v1,v2,v3}
```

```
f (g x y)

f $ g x y
```

```
data MyType = Foo |
              Bar |
              Baz

data HerType = Foo
             | Bar
             | Baz
```

Preserving appearance

Preserve precisely parts not touched.

Pretty print … or use lexical details.

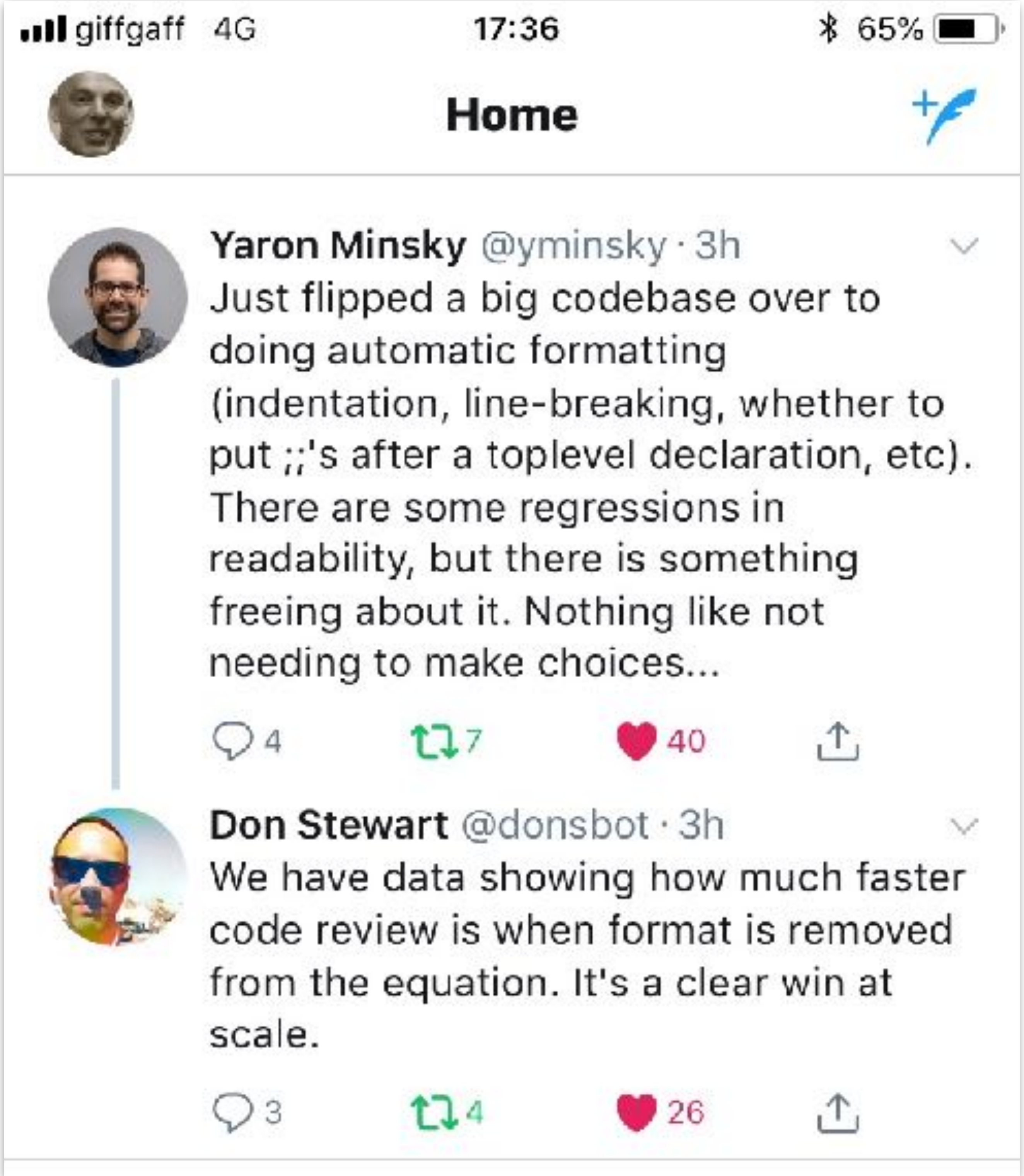
Preserving appearance isn’t built in

Compilers throw away some / all layout info, comments, …

Need to build infrastructure to hide layout manipulations.

Learn layout for synthesised code from existing codebase?

Scrap Your Reprinter by Orchard *et al*



Yaron Minsky @yminsky · 3h

Just flipped a big codebase over to doing automatic formatting (indentation, line-breaking, whether to put ;;'s after a toplevel declaration, etc). There are some regressions in readability, but there is something freeing about it. Nothing like not needing to make choices...

Don Stewart @donsbot · 3h

We have data showing how much faster code review is when format is removed from the equation. It's a clear win at scale.

“but there is something freeing about it. Nothing like not needing to make choices …”

Why should I trust
a refactoring tool
on my code?

POINT

Refactoring Tools Are Trustworthy Enough

John Brant

Refactoring tools don't have to guarantee correctness to be useful. Sometimes imperfect tools can be particularly helpful.

A COMMON DEFINITION of refactoring is "a behavior-preserving transformation that improves the overall code quality." Code quality is subjective, and a particular refactoring in a sequence of refactorings often might temporarily make the code worse. So, the code-quality-improvement part of the definition is often omitted, which leaves that refactorings are simply behavior-preserving transformations.

From that definition, the most important part of tool-supported refactorings appears to be correctness in behavior preservation. However, from a developer's viewpoint, the most important part is the refactoring's usefulness: can it help developers get their job done better and faster? Although absolute correctness is a great feature to have, it's neither a necessary nor sufficient condition for developers to use an automated refactoring tool.

Consider an imperfect refactoring tool. If a developer needs to perform a refactoring that the tool provides, he or she has two options. The developer can either use the tool and fix the bugs it introduced or perform manual refactoring and fix the bugs the manual changes introduced. If the time spent using the tool and fixing the bugs is less than the time doing it manually, the tool is useful. Furthermore, if the tool supports preview and undo, it can be more useful. With previewing, the developer can double-check that the changes look correct before they're saved; with undo, the developer can quickly revert the changes if they introduced any bugs.

Often, even a buggy refactoring tool is more useful than an automated refactoring tool that never introduces bugs. For example, automated tools often can't check all the preconditions for a refactoring. The preconditions might be undecidable, or no efficient algorithm exists for checking them. In this case, the buggy tool might check as much as it can and proceed with the refactoring, whereas the correct version sees that it can't check everything it needs and aborts the refactoring, leaving the developer to perform it manually. Depending on the buggy tool's defect rate and the developer's abilities, the buggy tool might introduce fewer errors than the correct tool paired with manual refactoring.

Even when a refactoring can be implemented without bugs, it can be beneficial to relax some preconditions to allow non-behavior-preserving transformations. For example, after implementing Extract Method in the Smalltalk Refactoring Browser, my colleagues and I received an email requesting that we allow the extracted method to override

continued on page 82

0740-7459/15/$31.00 © 2015 IEEE

COUNTERPOINT

Trust Must Be Earned

Friedrich Steimann

Creating bug-free refactoring tools is a real challenge. However, tool developers will have to meet this challenge for their tools to be truly accepted.

WHEN I ASK people about the progress of their programming projects, I often get answers like "I got it to work—now I need to do some refactoring!" What they mean is that they managed to tweak their code so that it appears to do what it's supposed to do, but knowing the process, they realize all too well that its result won't pass even the lightest code review. In the following refactoring phase, whether it's manual or tool supported, minor or even larger behavior changes go unnoticed, are tolerated, or are even welcomed (because refactoring the code has revealed logical errors). I assume that this conception of refactoring is by far the most common, and I have no objections to it (other than, perhaps, that I would question such a software process per se).

Now imagine a scenario in which code has undergone extensive (and expensive) certification. If this code is touched in multiple locations, chances are that the entire certification must be repeated. Pervasive changes typically become necessary if the functional requirements change and the code's current design can't accommodate the new requirements in a form that would allow isolated certification of the changed code. If, however, we had refactoring tools that have been certified to preserve behavior, we might be able to refactor the code so that the necessary functional changes remain local and don't require global recertification of the software. Unfortunately, we don't have such tools.

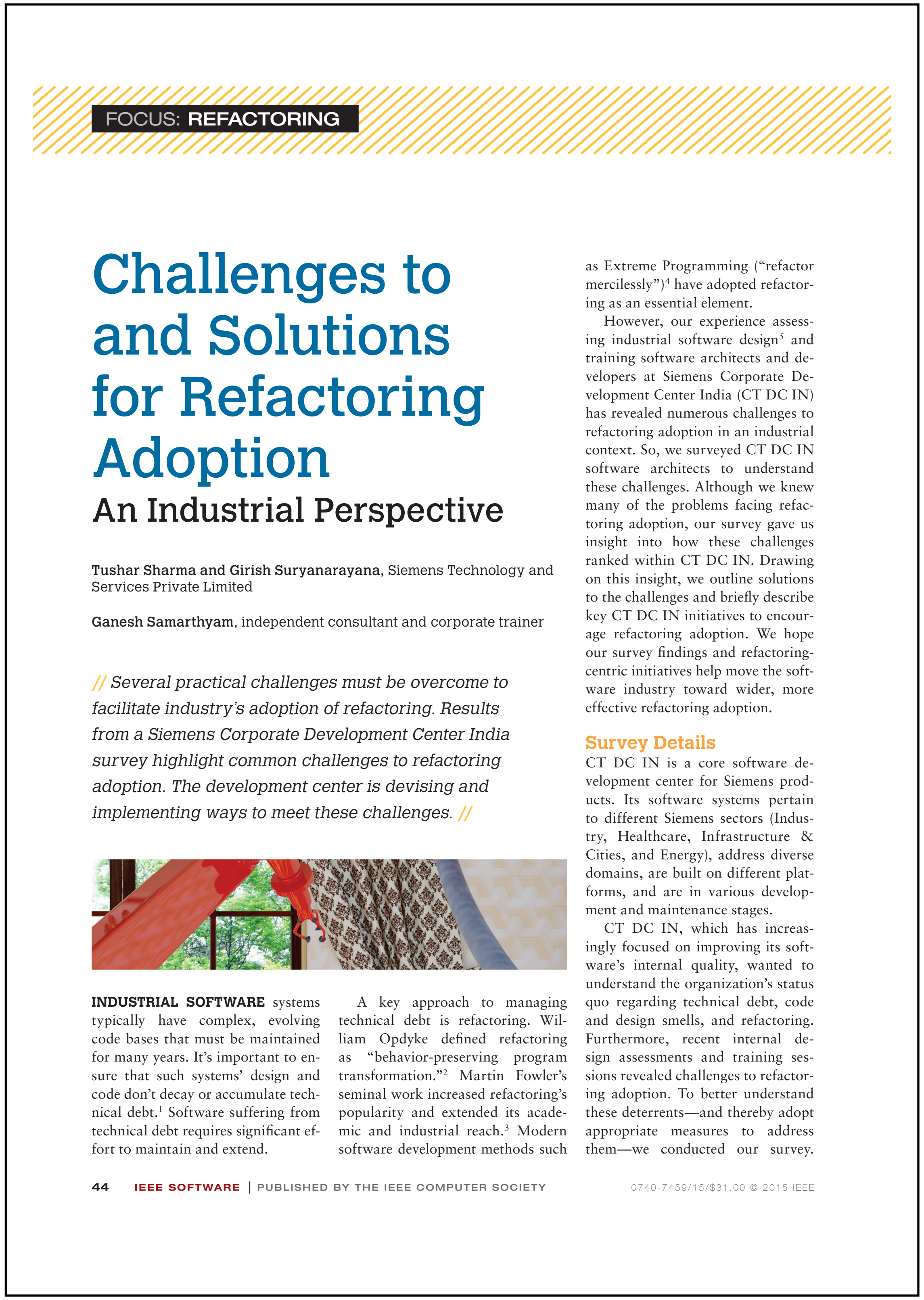
There's also a third perspective—the one I care about most. As an engineer, and even more so as a researcher, I want to do things that are state-of-the-art. Where the state-of-the-art leaves something to be desired, I want to push it further. If that's impossible, I want to know why, and I want people to understand why so that they can adjust their expectations. Refactoring-tool users will more easily accept limitations if these limitations are inherent in the nature of the matter and aren't engineering shortcomings.

What we have today is the common sentiment that "if only the tool people had enough resources, they would fix the refactoring bugs," suggesting that no fundamental obstacles to fixing them exist. This of course has the corollary that the bugs aren't troubling enough to be fixed (because otherwise, the necessary resources would be made available). For this corollary, two explanations are common: "Hardly anyone uses refactoring tools anyway, so who cares about the bugs?" and "The bugs aren't a real problem; my compiler and test suite will catch them as I go." I reject both expla-

continued on page 82

0740-7459/15/$31.00 © 2015 IEEE

IEEE Software, Nov/Dec 2015



FOCUS: REFACTORING

Challenges to and Solutions for Refactoring Adoption

An Industrial Perspective

Tushar Sharma and Girish Suryanarayana, Siemens Technology and Services Private Limited

Ganesh Samarthyam, independent consultant and corporate trainer

// Several practical challenges must be overcome to facilitate industry's adoption of refactoring. Results from a Siemens Corporate Development Center India survey highlight common challenges to refactoring adoption. The development center is devising and implementing ways to meet these challenges. //

INDUSTRIAL SOFTWARE systems typically have complex, evolving code bases that must be maintained for many years. It's important to ensure that such systems' design and code don't decay or accumulate technical debt.[1] Software suffering from technical debt requires significant effort to maintain and extend.

A key approach to managing technical debt is refactoring. William Opdyke defined refactoring as "behavior-preserving program transformation."[2] Martin Fowler's seminal work increased refactoring's popularity and extended its academic and industrial reach.[3] Modern software development methods such as Extreme Programming ("refactor mercilessly")[4] have adopted refactoring as an essential element.

However, our experience assessing industrial software design[5] and training software architects and developers at Siemens Corporate Development Center India (CT DC IN) has revealed numerous challenges to refactoring adoption in an industrial context. So, we surveyed CT DC IN software architects to understand these challenges. Although we knew many of the problems facing refactoring adoption, our survey gave us insight into how these challenges ranked within CT DC IN. Drawing on this insight, we outline solutions to the challenges and briefly describe key CT DC IN initiatives to encourage refactoring adoption. We hope our survey findings and refactoring-centric initiatives help move the software industry toward wider, more effective refactoring adoption.

Survey Details

CT DC IN is a core software development center for Siemens products. Its software systems pertain to different Siemens sectors (Industry, Healthcare, Infrastructure & Cities, and Energy), address diverse domains, are built on different platforms, and are in various development and maintenance stages.

CT DC IN, which has increasingly focused on improving its software's internal quality, wanted to understand the organization's status quo regarding technical debt, code and design smells, and refactoring. Furthermore, recent internal design assessments and training sessions revealed challenges to refactoring adoption. To better understand these deterrents—and thereby adopt appropriate measures to address them—we conducted our survey.

0740-7459/15/$31.00 © 2015 IEEE

IEEE Software, Nov/Dec 2015

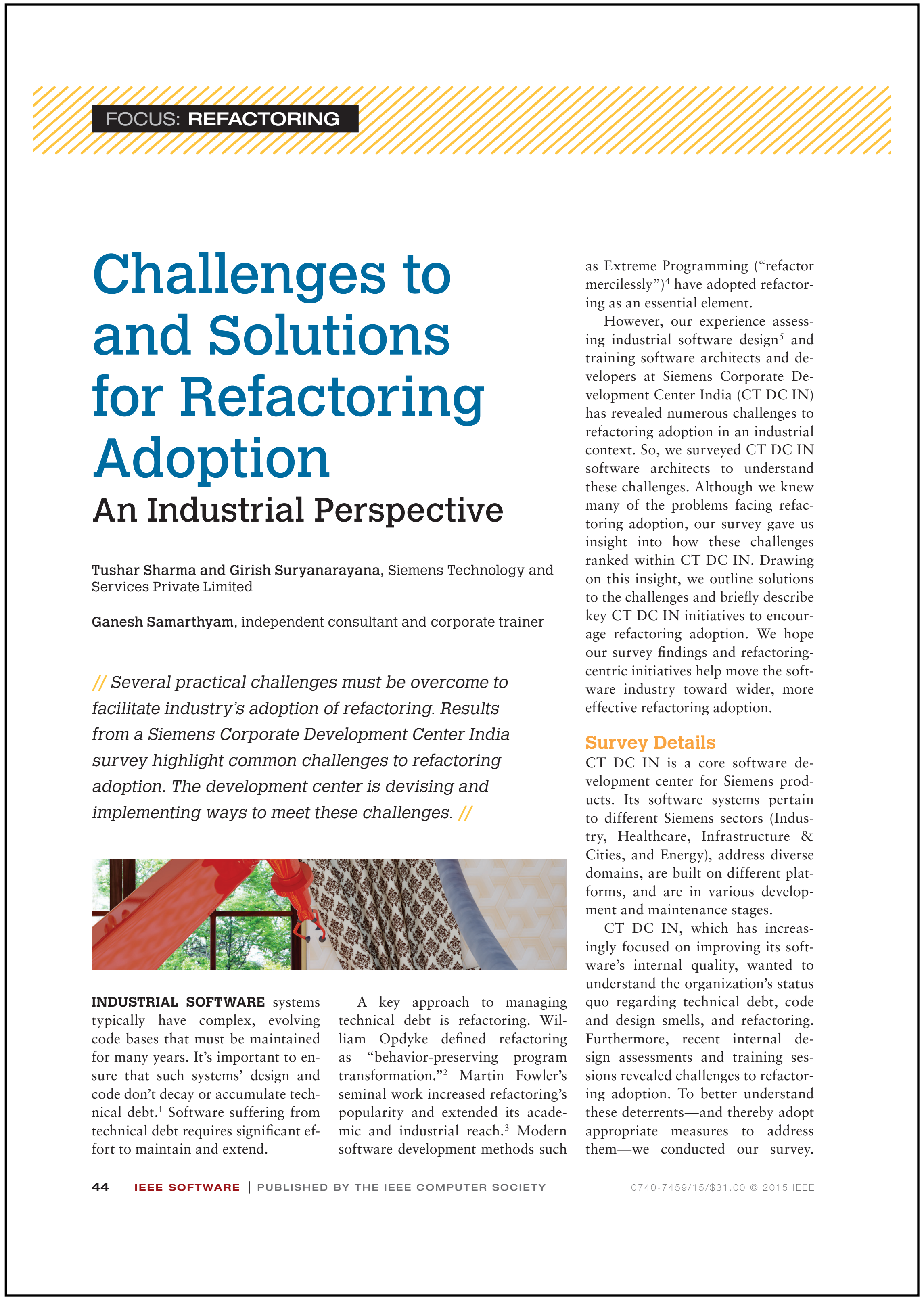
Breaking code

Cannot justify the time spent

Unpredictable impact

Difficult to review

Inadequate tools



FOCUS: REFACTORING

Challenges to and Solutions for Refactoring Adoption

An Industrial Perspective

Tushar Sharma and Girish Suryanarayana, Siemens Technology and Services Private Limited

Ganesh Samarthyam, independent consultant and corporate trainer

// Several practical challenges must be overcome to facilitate industry's adoption of refactoring. Results from a Siemens Corporate Development Center India survey highlight common challenges to refactoring adoption. The development center is devising and implementing ways to meet these challenges. //

INDUSTRIAL SOFTWARE systems typically have complex, evolving code bases that must be maintained for many years. It's important to ensure that such systems' design and code don't decay or accumulate technical debt.[1] Software suffering from technical debt requires significant effort to maintain and extend.

A key approach to managing technical debt is refactoring. William Opdyke defined refactoring as "behavior-preserving program transformation."[2] Martin Fowler's seminal work increased refactoring's popularity and extended its academic and industrial reach.[3] Modern software development methods such as Extreme Programming ("refactor mercilessly")[4] have adopted refactoring as an essential element.

However, our experience assessing industrial software design[5] and training software architects and developers at Siemens Corporate Development Center India (CT DC IN) has revealed numerous challenges to refactoring adoption in an industrial context. So, we surveyed CT DC IN software architects to understand these challenges. Although we knew many of the problems facing refactoring adoption, our survey gave us insight into how these challenges ranked within CT DC IN. Drawing on this insight, we outline solutions to the challenges and briefly describe key CT DC IN initiatives to encourage refactoring adoption. We hope our survey findings and refactoring-centric initiatives help move the software industry toward wider, more effective refactoring adoption.

Survey Details

CT DC IN is a core software development center for Siemens products. Its software systems pertain to different Siemens sectors (Industry, Healthcare, Infrastructure & Cities, and Energy), address diverse domains, are built on different platforms, and are in various development and maintenance stages.

CT DC IN, which has increasingly focused on improving its software's internal quality, wanted to understand the organization's status quo regarding technical debt, code and design smells, and refactoring. Furthermore, recent internal design assessments and training sessions revealed challenges to refactoring adoption. To better understand these deterrents—and thereby adopt appropriate measures to address them—we conducted our survey.

IEEE Software, Nov/Dec 2015

Breaking code

Cannot justify the time spent

Unpredictable impact

Difficult to review

Inadequate tools

Building trust more widely

Open Source … confidence in the code … other committers.

Openness of the system …

… you can check the changes that a refactoring makes,

… and for the DSL can see which refactorings performed.

GHC *vs* Haskell standards *vs* other Haskell implementations.

Editor and IDE integration

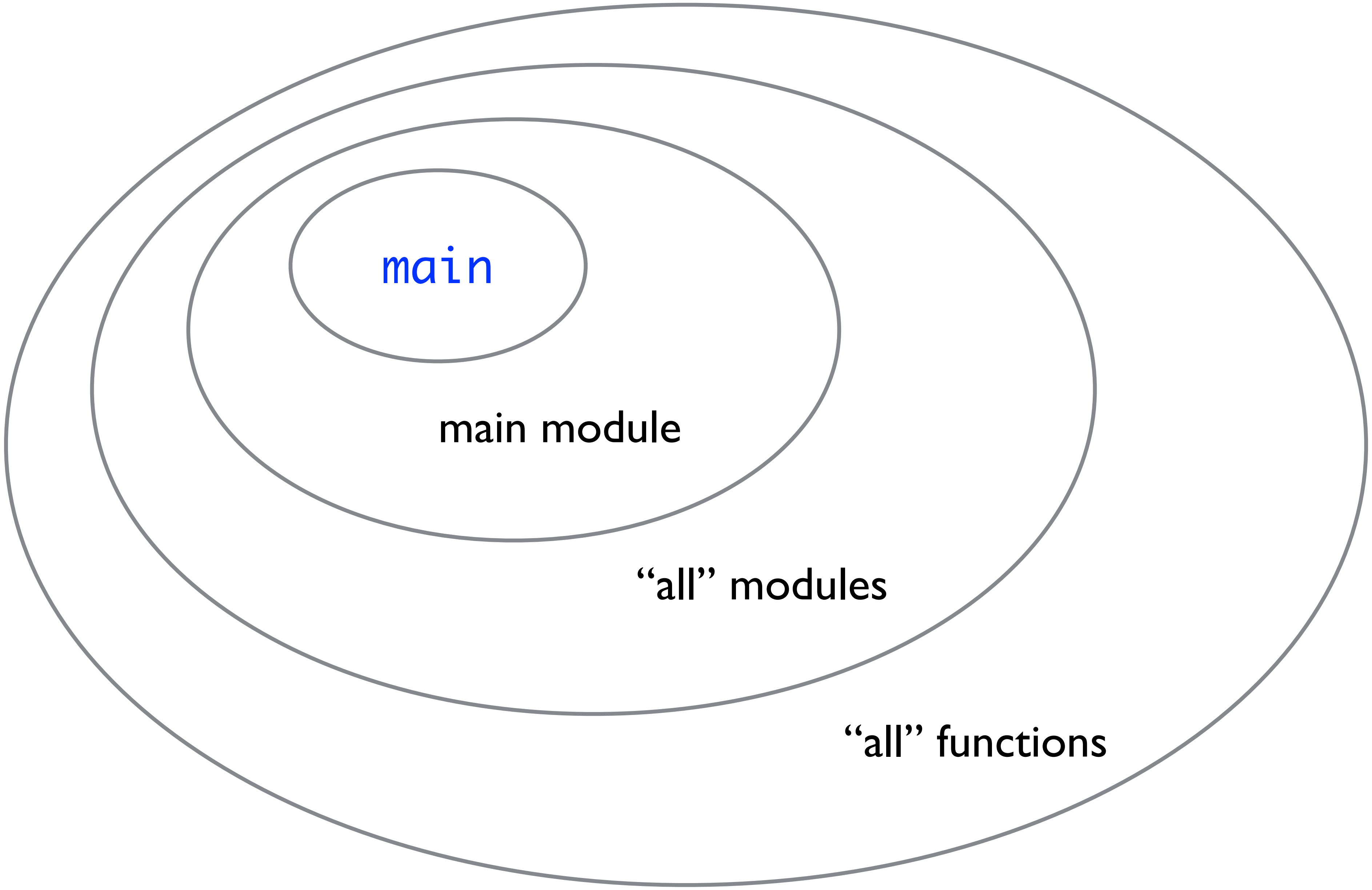
Preserving meaning

Do these two programs mean the same thing?

Difficult to examine and compare the meanings directly …

… so we look at other ways of trying to answer this.

Different scopes



`main`

main module

“all” modules

“all” functions

Different contexts

All tests for the project.

Refactorings need to be test-framework aware

- Naming conventions: `foo` and `foo_test` …
- Macro use, etc.

The `makefile` for the project.

Using these versions of these libraries … which we don't control.

Assuring meaning preservation

	test	verify
instances of the refactoring		
the refactoring itself		

Assuring meaning preservation

	test	verify
instances of the refactoring	Rename foo to bar in this project.	
the refactoring itself		

Assuring meaning preservation

	test	verify
instances of the refactoring	Rename `foo` to `bar` in this project.	
the refactoring itself	Renaming for all names, functions and projects.	

	test	verify
instances of the refactoring	✓	✓
the refactoring itself	✓	✓

Testing

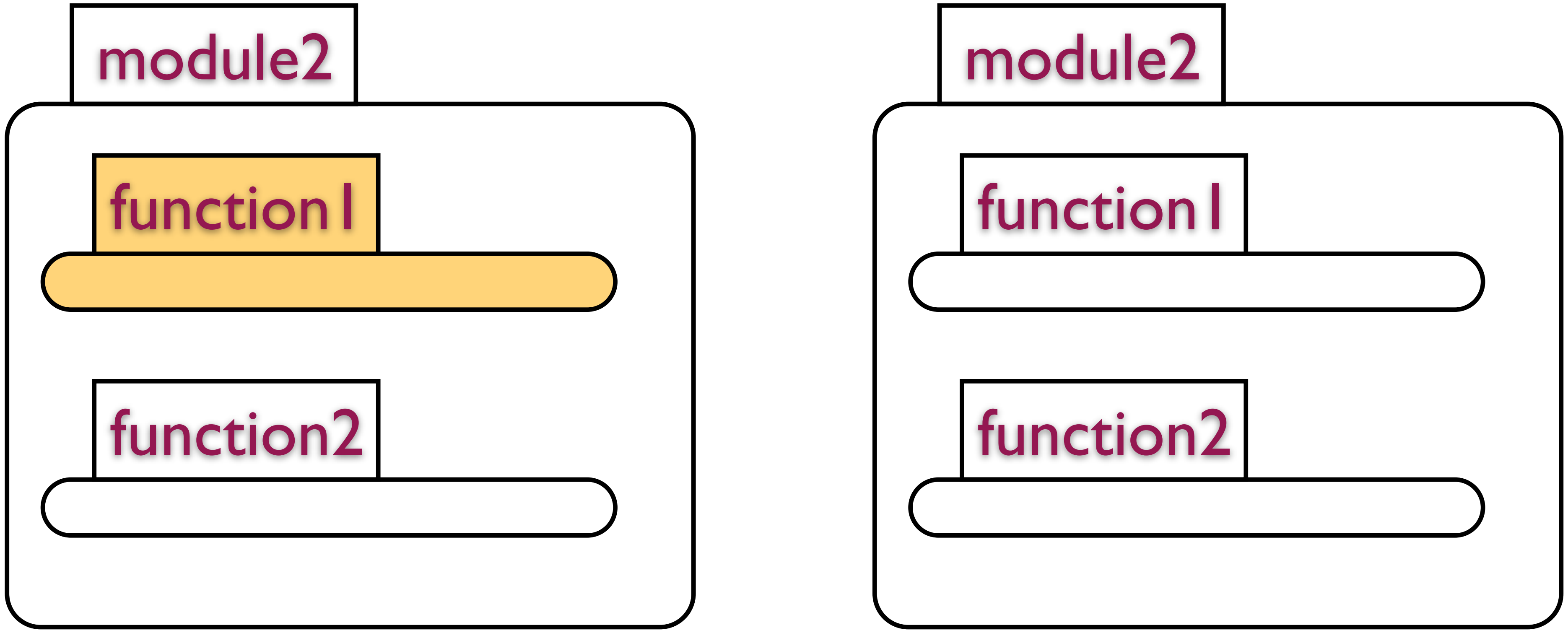
	test	verify
instances of the refactoring	✓	
the refactoring itself		

Testing new *vs* old (with Huiqing Li)

Compare the results of function1 and function1 (unmodified) …

… using existing unit tests, and randomly-generated inputs

… could compare ASTs as well as behaviour (in former case).



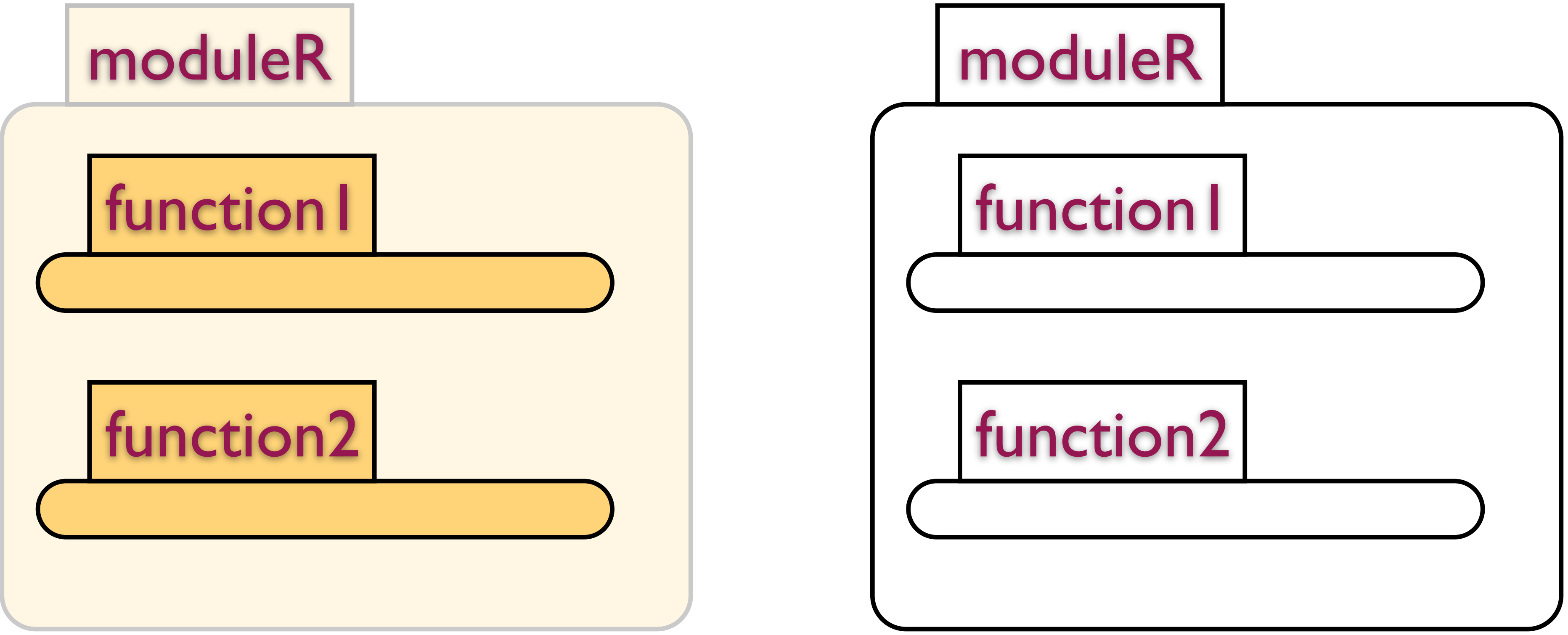
	test	verify
instances of the refactoring		
the refactoring itself	✓	

Fully random

Generate random modules,

… generate random refactoring commands,

… and check ≡ with random inputs. (w/ Drienyovszky, Horpácsi).



Verification

	test	verify
instances of the refactoring		
the refactoring itself		✓

Tool verification (with Nik Sultana)

$$\forall p.\ (Q\,p) \longrightarrow (T\,p) \simeq p$$

Deep embeddings of small languages:

… potentially name-capturing λ-calculus

… PCF with unit and sum types.

Isabelle/HOL: LCF-style secure proof checking.

Formalisation of meta-theory: variable binding, free / bound variables, capture, fresh variables, typing rules, etc …

… principally to support pre-conditions.

Shallow embedding

	test	verify
instances of the refactoring		✓
the refactoring itself		

Automatically verify instances of refactorings

Prove the equivalence of the particular pair of functions / systems using an SMT solver …

… SMT solvers linked to Haskell by `Data.SBV` (Levent Erkok).

Manifestly clear what is being checked.

The approach delegates trust to the SMT solver …

… can choose other solvers, and examine counter-examples.

DEMUR work with Colin Runciman

Example

```
module Before where

h :: Integer->Integer->Integer

h x y = g y + f (g y)

g :: Integer->Integer

g x = 3*x + f x

f :: Integer->Integer

f x = x + 1
```

Example: renaming

```
module Before where

h :: Integer->Integer->Integer

h x y = g y + f (g y)

g :: Integer->Integer

g x = 3*x + f x

f :: Integer->Integer

f x = x + 1
```

```
module After where

h :: Integer->Integer->Integer

h x y = k y + f (k y)

k :: Integer->Integer

k x = 3*x + f x

f :: Integer->Integer

f x = x + 1
```

```
{-# LANGUAGE ScopedTypeVariables #-}

module RefacProof where

import Data.SBV
```

```
{-# LANGUAGE ScopedTypeVariables #-}

module RefacProof where

import Data.SBV
```

```
h :: Integer->Integer->Integer

h x y = g y + f (g y)

g :: Integer->Integer

g x = 3*x + f x
```

```
{-# LANGUAGE ScopedTypeVariables #-}

module RefacProof where

import Data.SBV
```

```
h :: Integer->Integer->Integer

h x y = g y + f (g y)

g :: Integer->Integer

g x = 3*x + f x
```

```
h' :: Integer->Integer->Integer

h' x y = k y + f (k y)

k :: Integer->Integer

k x = 3*x + f x
```

```
{-# LANGUAGE ScopedTypeVariables #-}

module RefacProof where

import Data.SBV
```

```
h :: Integer->Integer->Integer

h x y = g y + f (g y)

g :: Integer->Integer

g x = 3*x + f x
```

```
h' :: Integer->Integer->Integer

h' x y = k y + f (k y)

k :: Integer->Integer

k x = 3*x + f x
```

```
-- f can be treated as an uninterpreted symbol

f = uninterpret "f"

-- Properties

propertyk = prove $ \(x::SInteger) -> g x .== k x
propertyh = prove $ \(x::SInteger) (y::SInteger) -> h x y .== h' x y
```

```haskell
h :: Integer->Integer->Integer

h x y = g y + f (g y)

g :: Integer->Integer

g x = 3*x + f x
```

```haskell
h' :: Integer->Integer->Integer

h' x y = k y + f (k y)

k :: Integer->Integer

k x = 3*x + f x
```

```haskell
-- f can be treated as an uninterpreted symbol

f = uninterpret "f"

-- Properties

propertyk = prove $ \(x::SInteger) -> g x .== k x
propertyh = prove $ \(x::SInteger) (y::SInteger) -> h x y .== h' x y
```

```
*Refac2> propertyk
Q.E.D.
*Refac2> propertyh
Q.E.D.
```

```
h :: Integer->Integer->Integer

h x y = g y + f (g y)
        where
        g z = z*z

g :: Integer->Integer

g x = 3*x + f x
```

```
h :: Integer->Integer->Integer

h x y = g y + f (g y)
        where
        g z = z*z

g :: Integer->Integer

g x = 3*x + f x
```

```
h' :: Integer->Integer->Integer

h' x y = k y + f (k y)
         where
         g z = z*z

k :: Integer->Integer

k x = 3*x + f x
```

```
h :: Integer->Integer->Integer

h x y = g y + f (g y)
        where
        g z = z*z

g :: Integer->Integer

g x = 3*x + f x
```

```
h' :: Integer->Integer->Integer

h' x y = k y + f (k y)
         where
         g z = z*z

k :: Integer->Integer

k x = 3*x + f x
```

```
f = uninterpret "f"

propertyk = prove $ \(x::SInteger) -> g x .== k x
propertyh = prove $ \(x::SInteger) (y::SInteger) -> h x y .== h' x y
```

```
h :: Integer->Integer->Integer

h x y = g y + f (g y)
        where
        g z = z*z

g :: Integer->Integer

g x = 3*x + f x
```

```
h' :: Integer->Integer->Integer

h' x y = k y + f (k y)
         where
         g z = z*z

k :: Integer->Integer

k x = 3*x + f x
```

```
f = uninterpret "f"

propertyk = prove $ \(x::SInteger) -> g x .== k x
propertyh = prove $ \(x::SInteger) (y::SInteger) -> h x y .== h' x y
```

```
*Refac2> propertyk
Q.E.D.
*Refac2> propertyh
Falsifiable. Counter-example:
  s0 = 0 :: SInteger
  s1 = -1 :: SInteger
```

Trustworthy refactoring project

CAKEML
A Verified Implementation of ML

Fully formally verified refactorings for a certified language and compiler: CakeML …

… plus a tool for OCaml providing high-assurance refactorings, through proof, SMT solving and testing.

	test	verify
instances of the refactoring	✓	✓
the refactoring itself	✓	✓

Trust is a complicated, multi-dimensional issue … but we're working on it.

What's the ideal language supporting refactoring?

What's the ideal language for refactoring?

Changes are first class.

No layout choice: you have to conform to layout rules.

No macros, reflection, …

Compiler stability

Integration with a semantically-aware change management tool.

Theory of patches, …

© Sheila Terry/Science Photo Library

Why should I use your refactoring tool?

What's the ideal language supporting refactoring?

What do you mean when you say "refactoring"?

What's so wrong with duplicated code?

Why haven't you implemented this refactoring?

It's just renaming … what's all the fuss?

I don't need a refactoring tool … … I have types!

Will you integrate with this editor or IDE?

Why have you messed up the layout of my program?

Why should I trust a refactoring tool on my code?

Trust is a complicated, multi-dimensional issue … but we're working on it.

We can be more adventurous with a refactoring tool!

It's better to implement libraries, APIs and DSLs than individual refactorings

Types might both help and hinder effective refactoring

"but there is something freeing about it. Nothing like not needing to make choices …"

Set aside any thoughts of building language-independent refactoring tools!

https://github.com/alanz/HaRe

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

https://gitlab.com/trustworthy-refactoring/refactorer