A journey to “software evolution” land

S. Ducasse
http://stephane.ducasse.free.fr

In a Nutshell

Head of RMOD team (7 permanents, 20 people)

4 years scientific deputee of Inria Lille (300 people)

Wrote several open-source books / ~ 300 articles

~ 15 K citations / H-index~59

One of the leaders of the Pharo community

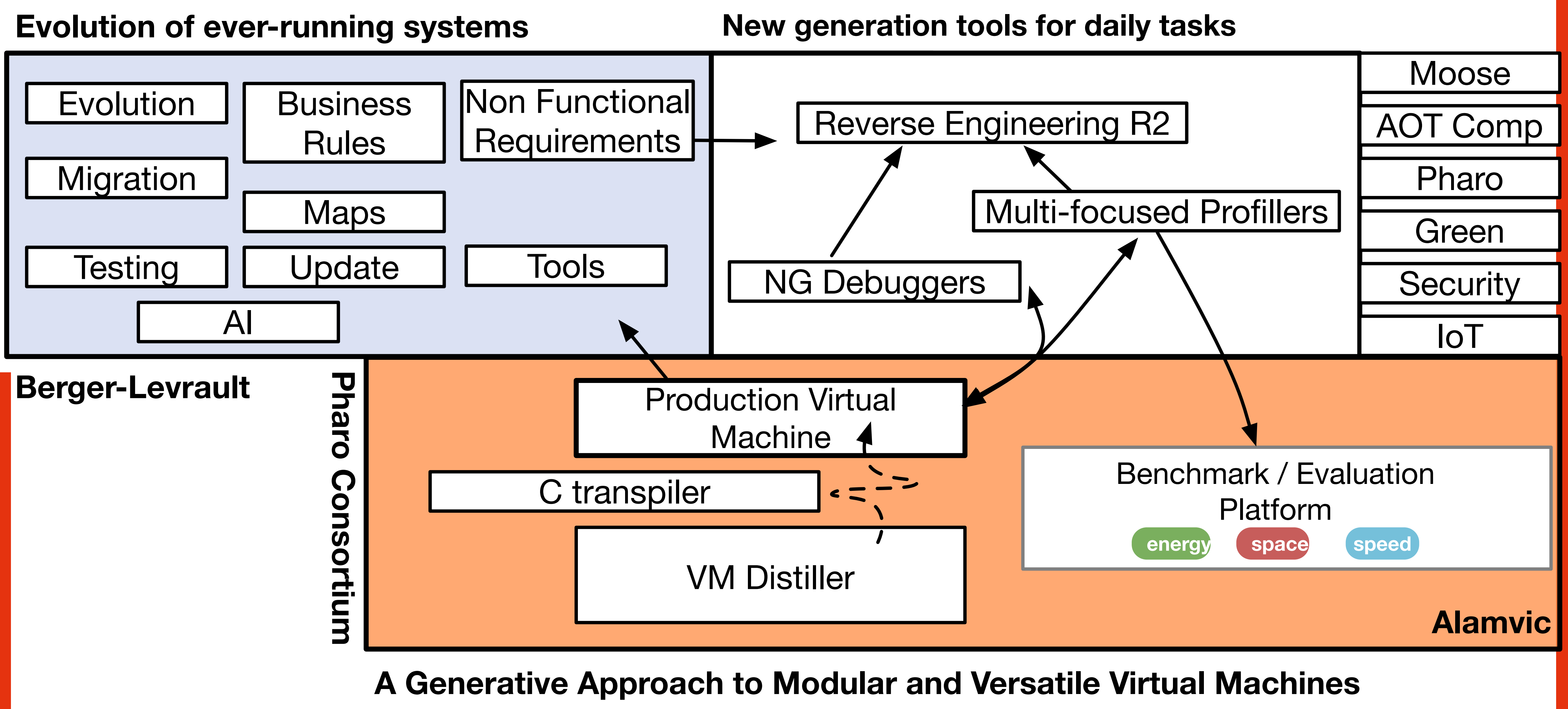
- http://www.pharo.org

Past core devs of Moose data and code analysis platform

- http://moosetechnology.org

Co-founder of http://www.synectique.eu

RMOD: 3 axes in synergy



Evolution of ever-running systems

Evolution

Business Rules

Non Functional Requirements

Migration

Maps

Testing

Update

Tools

AI

Berger-Levrault

New generation tools for daily tasks

Reverse Engineering R2

Multi-focused Profillers

NG Debuggers

Moose

AOT Comp

Pharo

Green

Security

IoT

Pharo Consortium

Production Virtual Machine

C transpiler

VM Distiller

Benchmark / Evaluation Platform

energy

space

speed

Alamvic

A Generative Approach to Modular and Versatile Virtual Machines

Roadmap

Legacy is not just Cobol

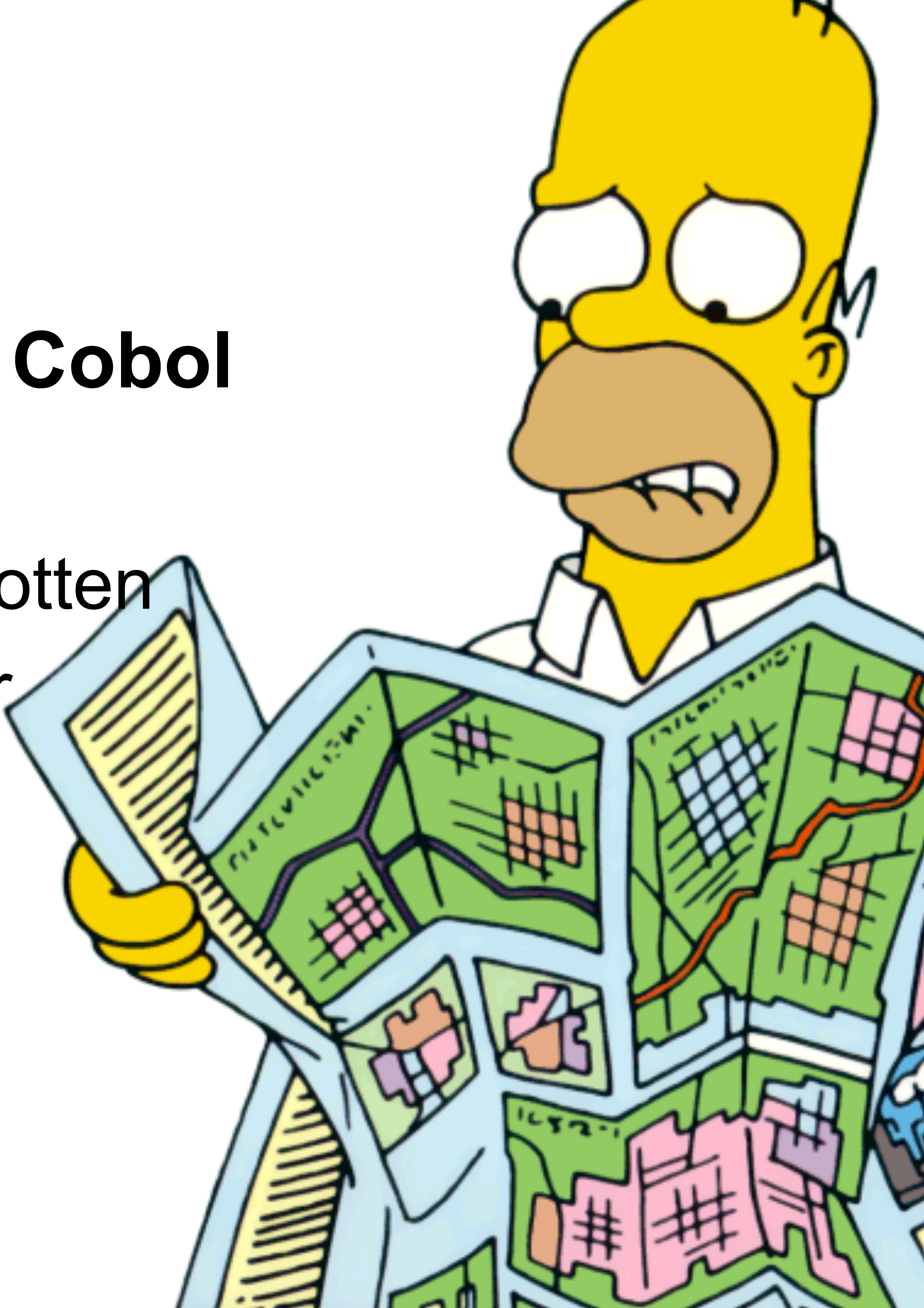
Software Maps

Green tests can be rotten

Research agenda for

Virtual Machines

Current effort



Software is Complex

Two software evolution laws

Continuing change

- A program that is used in a real-world environment must change, or become progressively less useful in that environment.

Increasing complexity

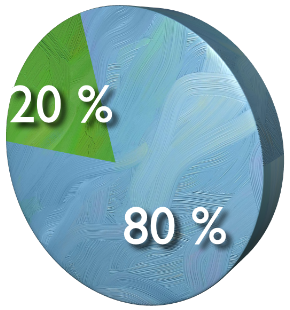
- As a program evolves, it becomes more complex, and extra resources are needed to preserve and simplify its structure.

Software is a living entity...

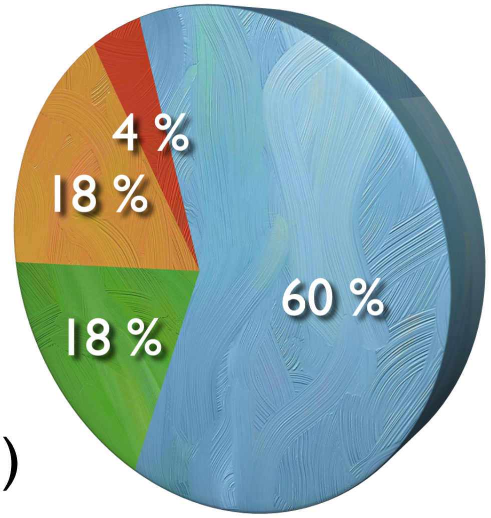
- Early decisions were certainly good at that time
- But the context changes
- Customers change
- Technology changes
- People change

We only maintain useful successful software

Maintenance is *continuous* Development



Between **70%** and **90%** of ***global***
effort is spent on "maintenance" !



4% Other

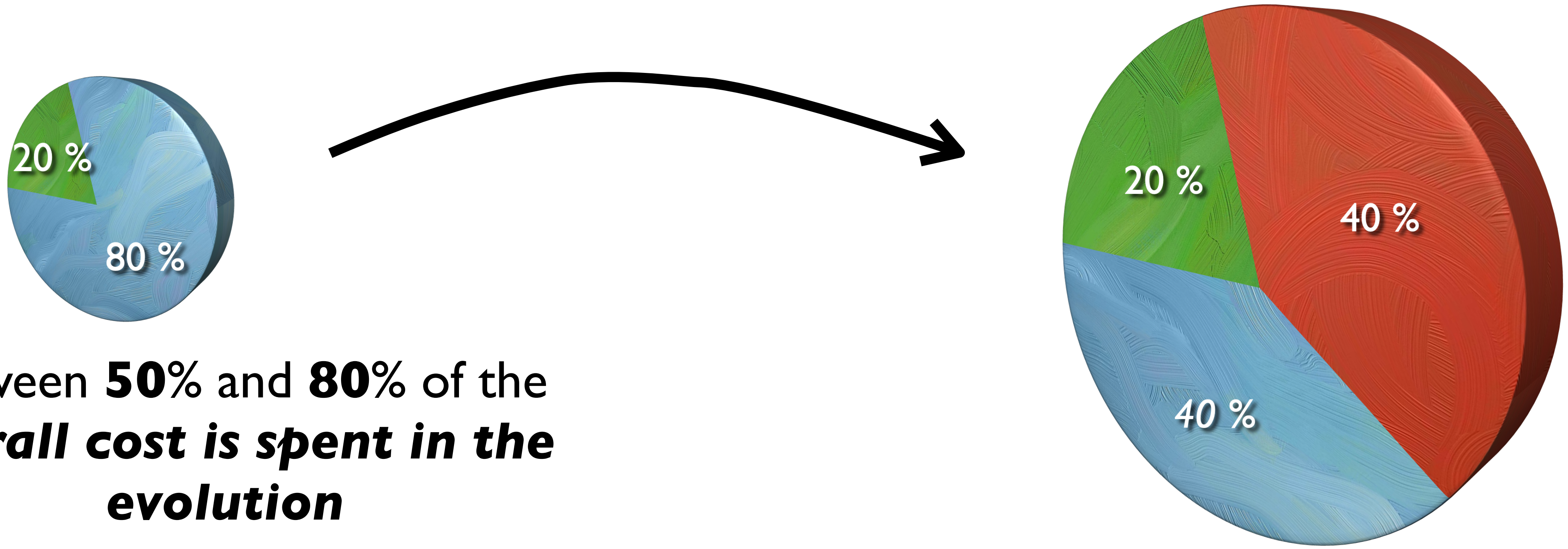
18% Adaptive
(new platforms or OS)

18% Corrective
(fixing reported errors)

60% Perfective
(new functionality)

"Maintenance"

50% of development time is lost trying to understand code !



Between **50%** and **80%** of the ***overall cost is spent in the evolution***

We lose a lot of time with inappropriate and ineffective practices

Legacy systems exist in ***any*** language

Berger-Levrault by example

1 MLOCS

21 433 classes

95 164 méthodes

500 pages web

36 ans/homme
de migration

GWT →

Depuis GWT vers
Angular

Bottom up team: interested in problems

code analysis, metamodeling, software metrics, program understanding, ***program visualization***, ***reverse engineering***, evolution analysis, refactorings, quality,

changes analysis, commit,

dependencies, merging support

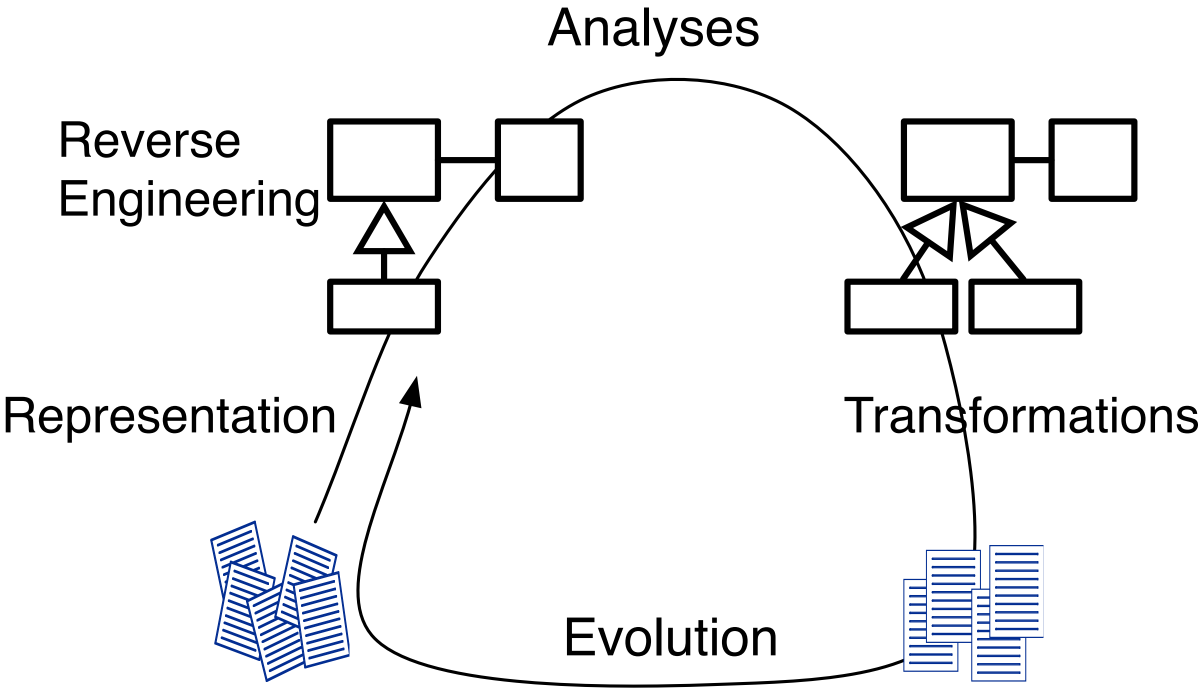
rule and bug assessment

semi-automatic migration

example-based transformations

test selection, rearchitecturing

blockchains, ***ui-migration***



Collaborations

IMT Douai, Soft (VUB), ENSTA (Bretagne)

Berger-Levrault, Siemens, Thales, CIM, Arolla, Lifeware, WordLine/ATOS

Roadmap

Legacy is not just Cobol

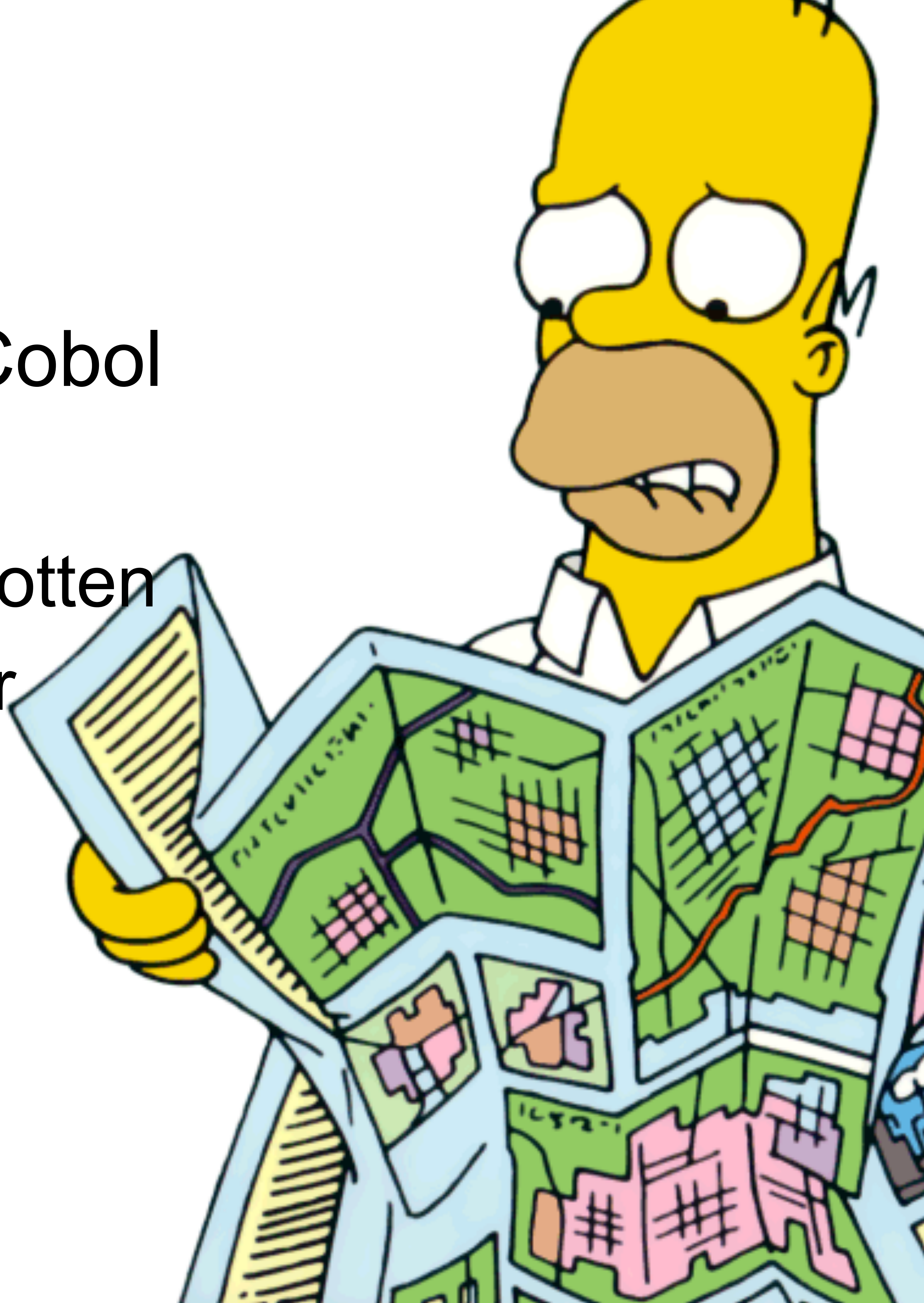
Software Maps

Green tests can be rotten

Research agenda for

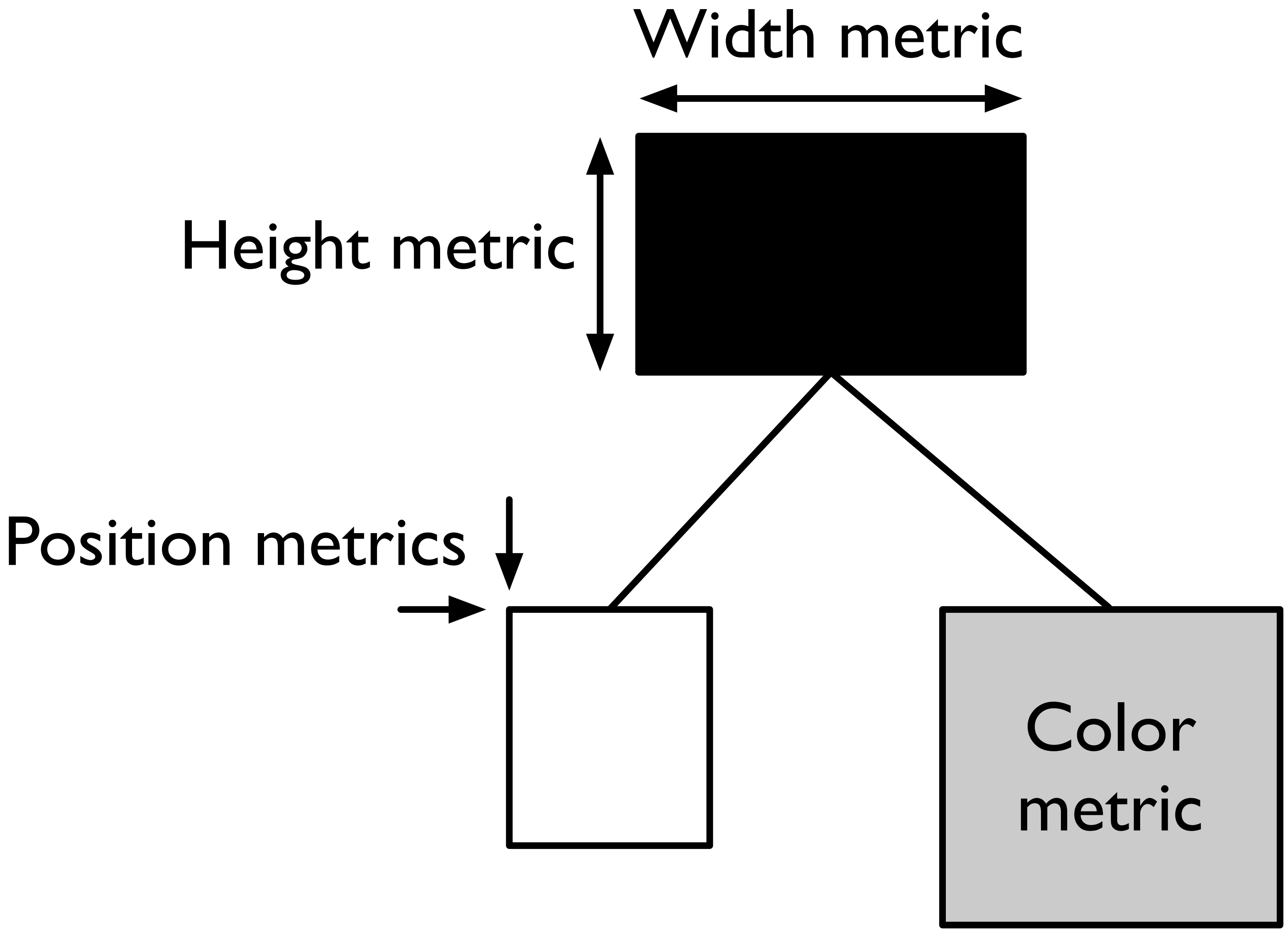
Virtual Machines

Current effort

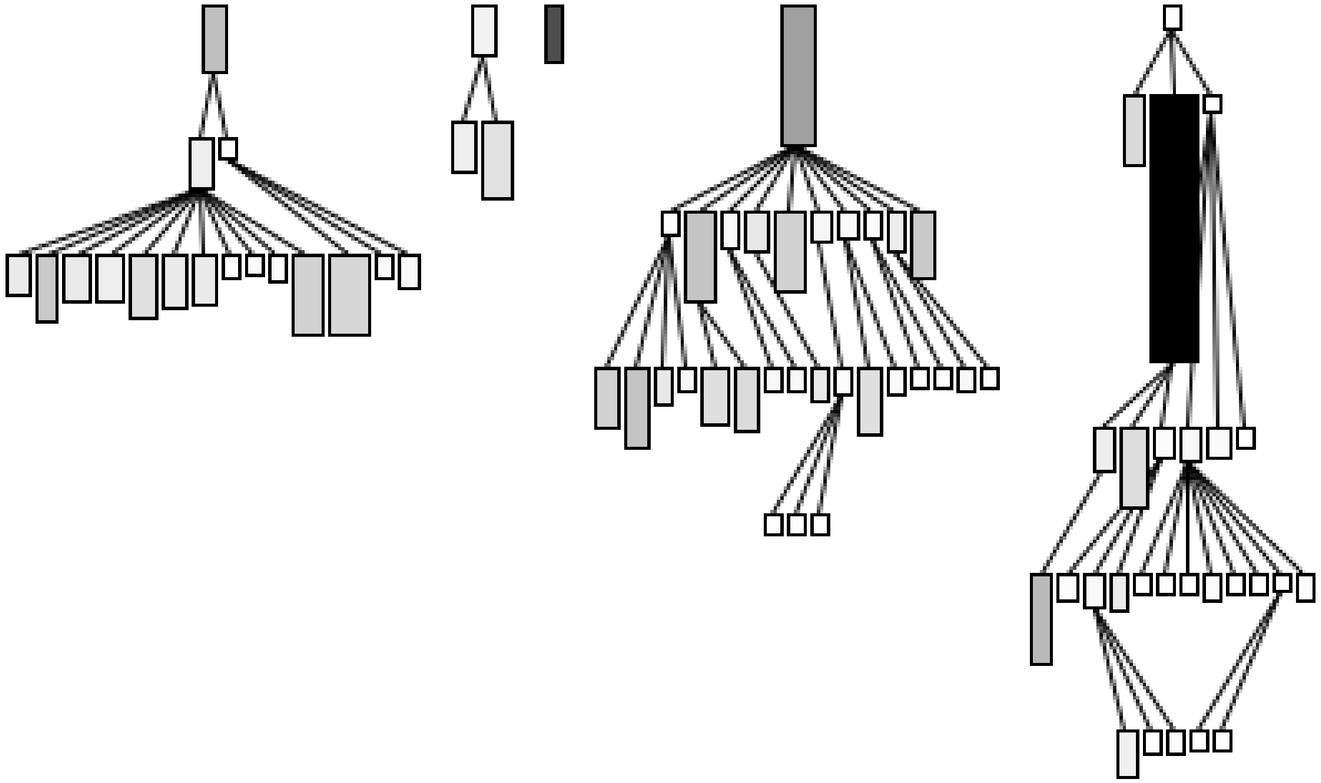


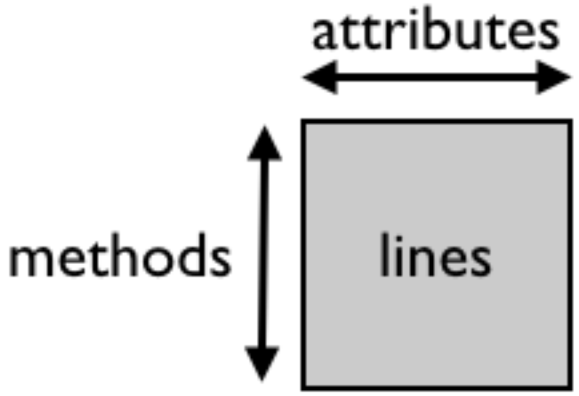
Some selected software maps
— to build **yourselves** at home

First glance at large systems: Polymetric views [PhD Lanza]

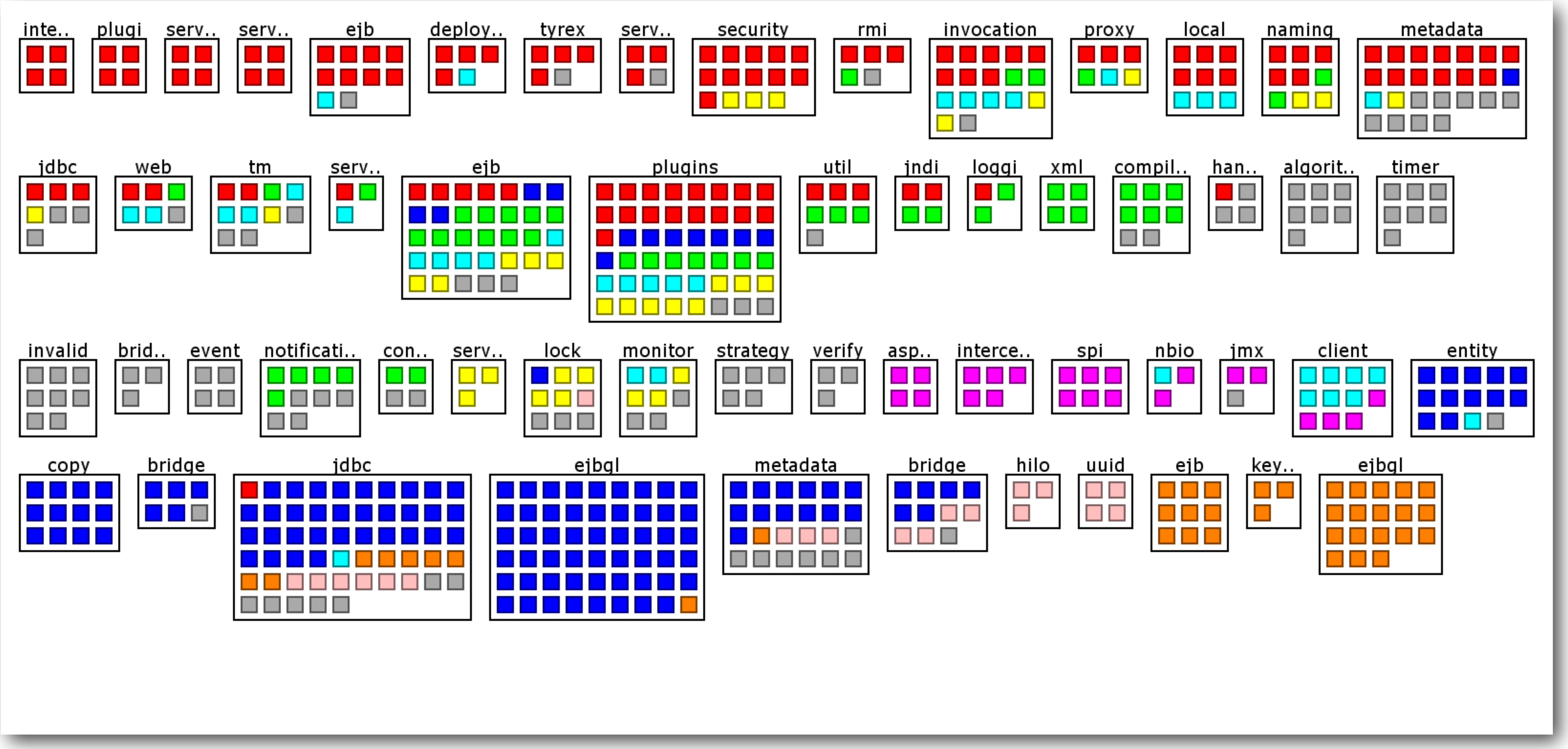


Understanding systems [PhD M. Lanza]

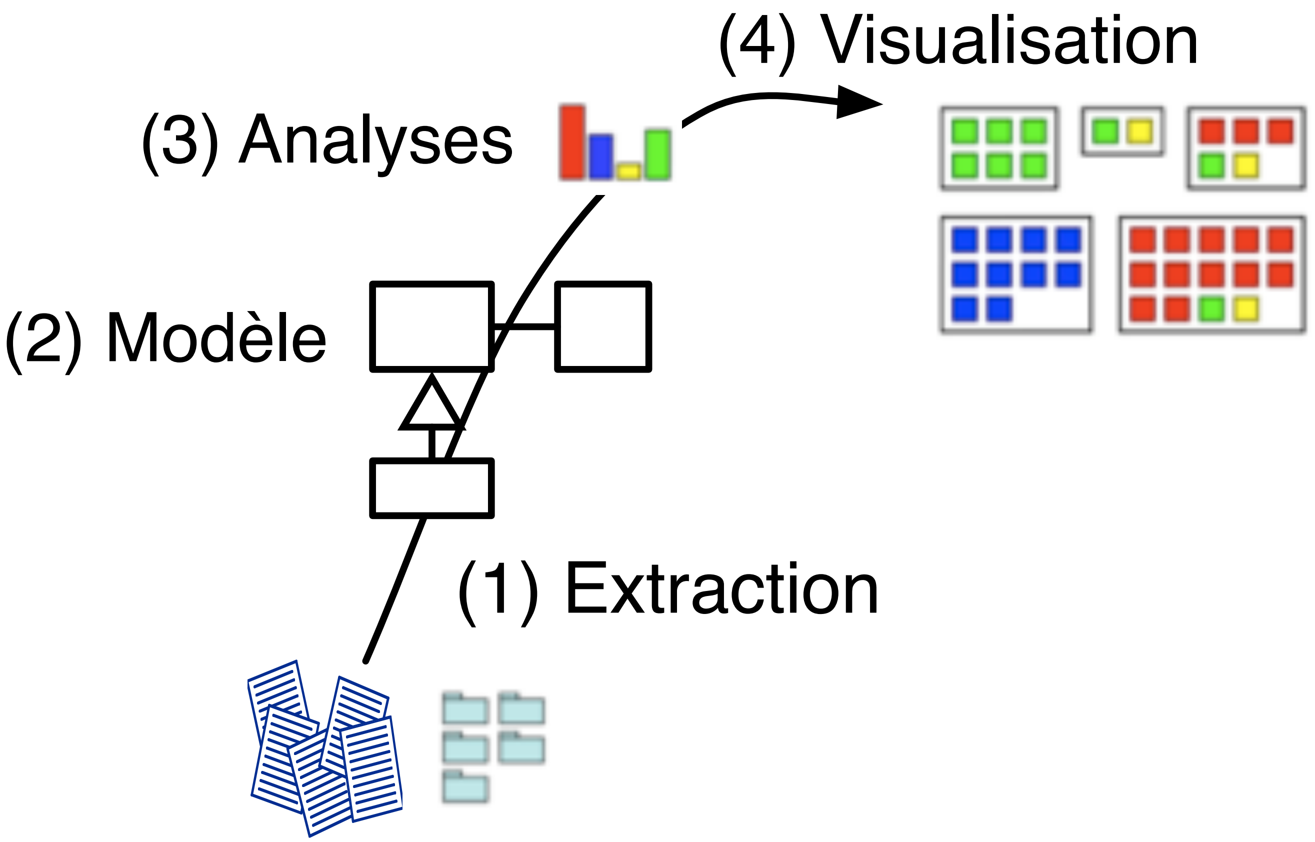




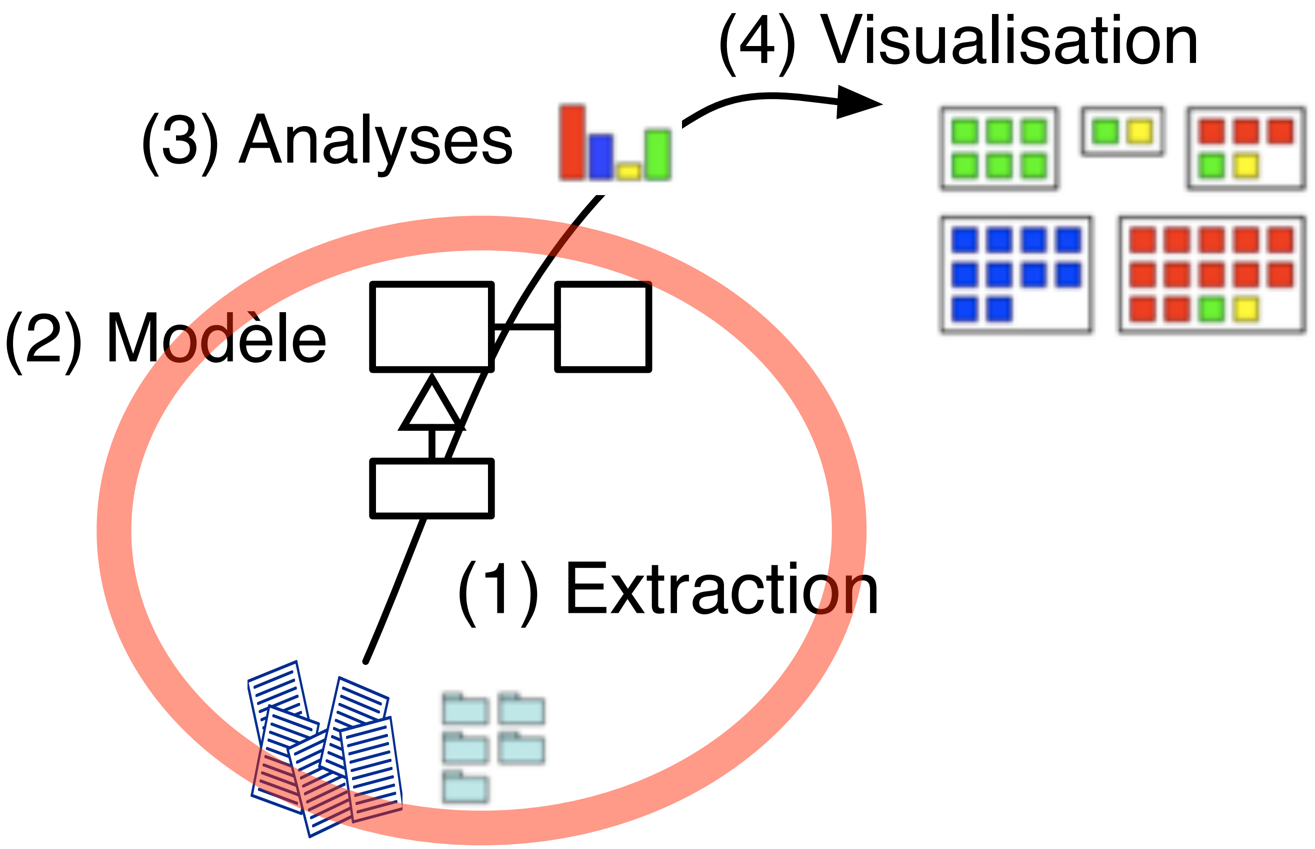
How a property spread on a system?



Example : Who is behind package X ?



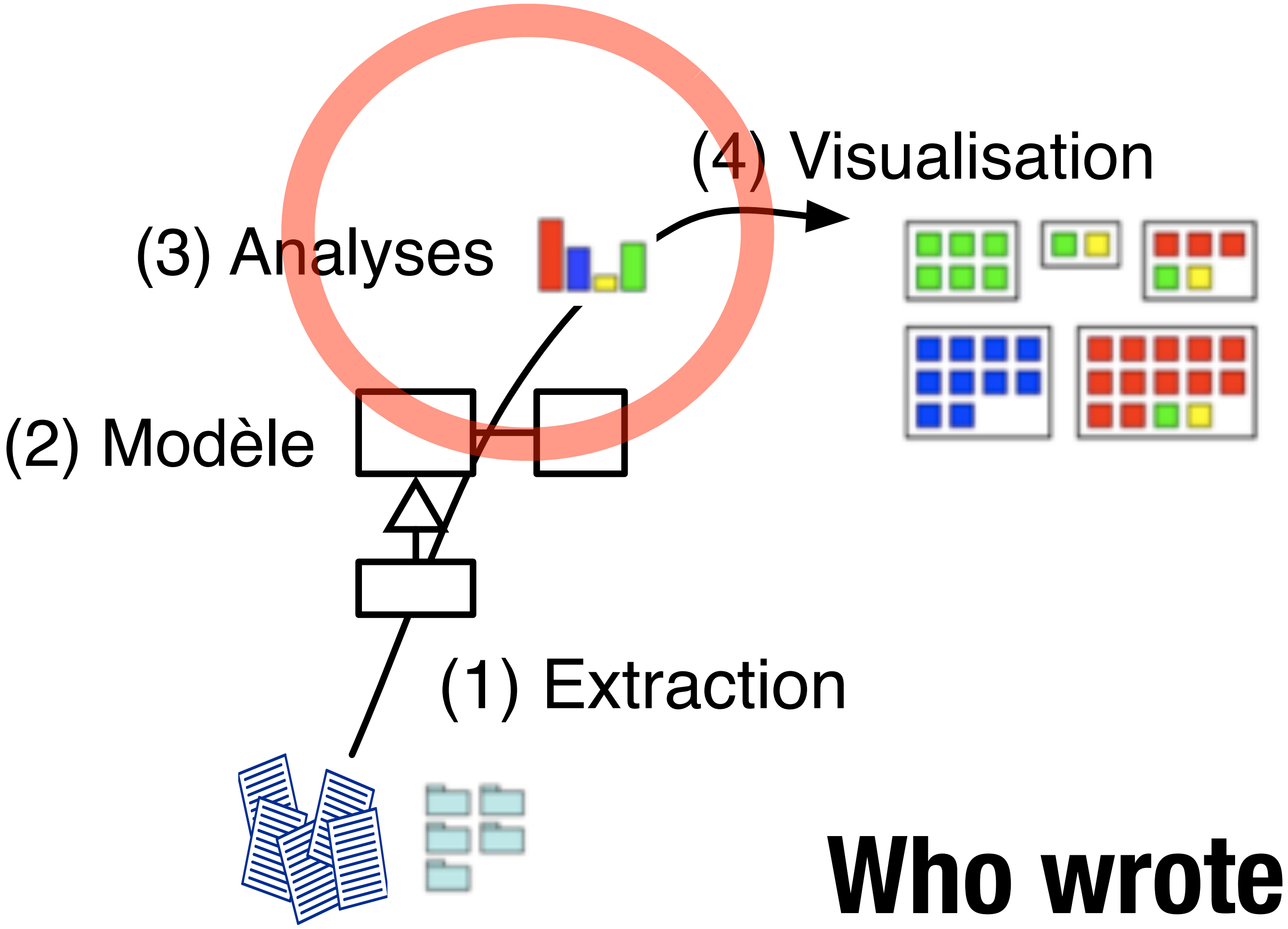
Step 1 - Model Creation/Import



Definition of a model to represent entities

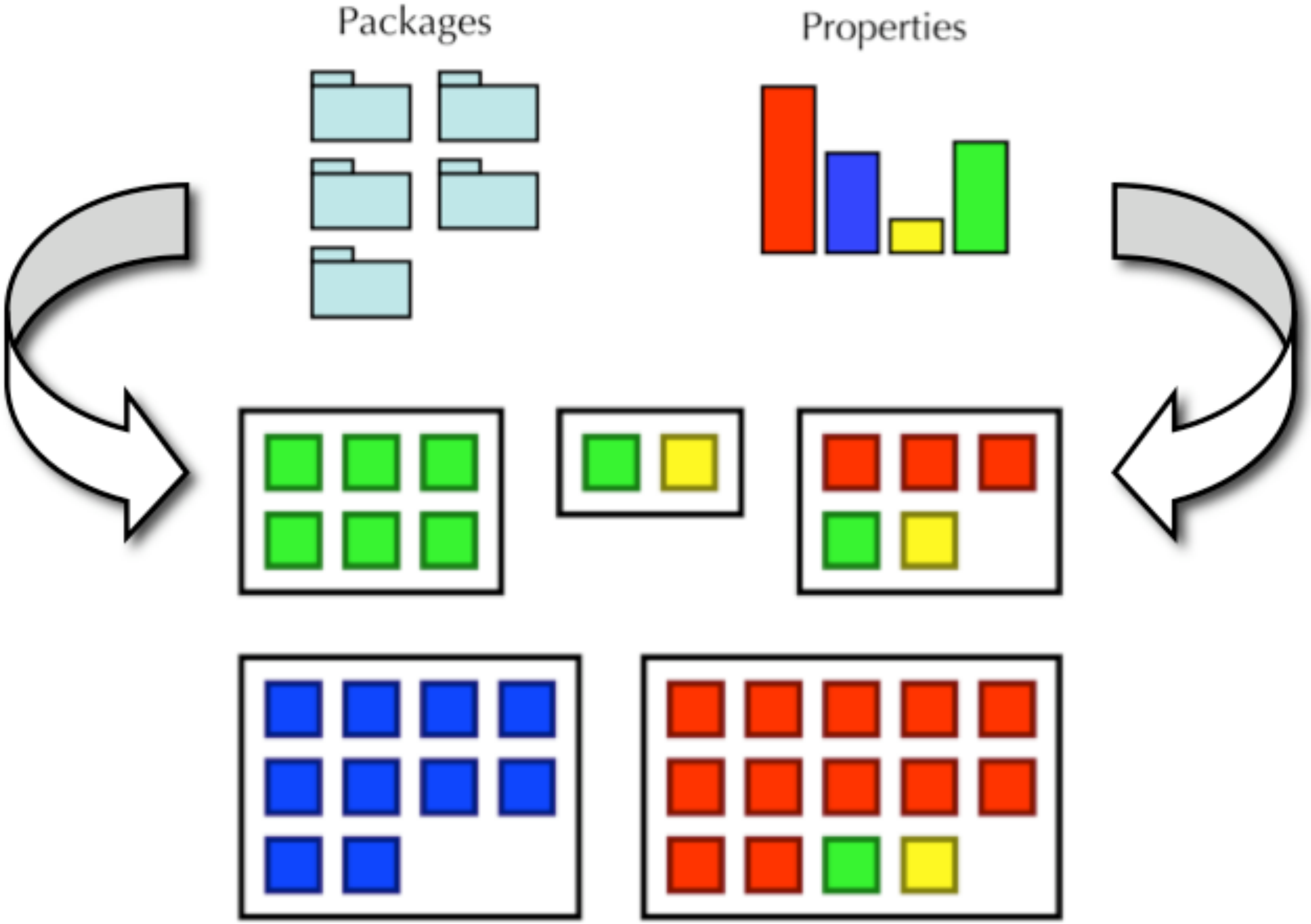
Data Extraction (CVS...)

Step 2 - Analyses



Who wrote how many lines of code?

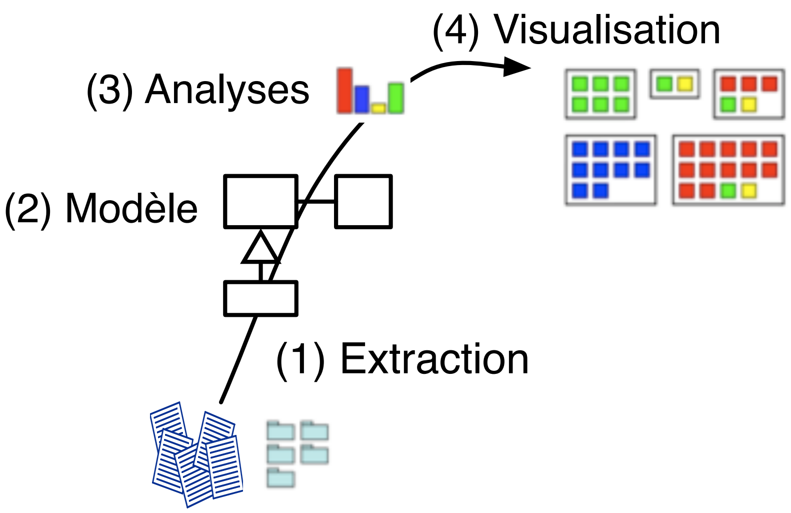
Step : 3 - Creating the Map

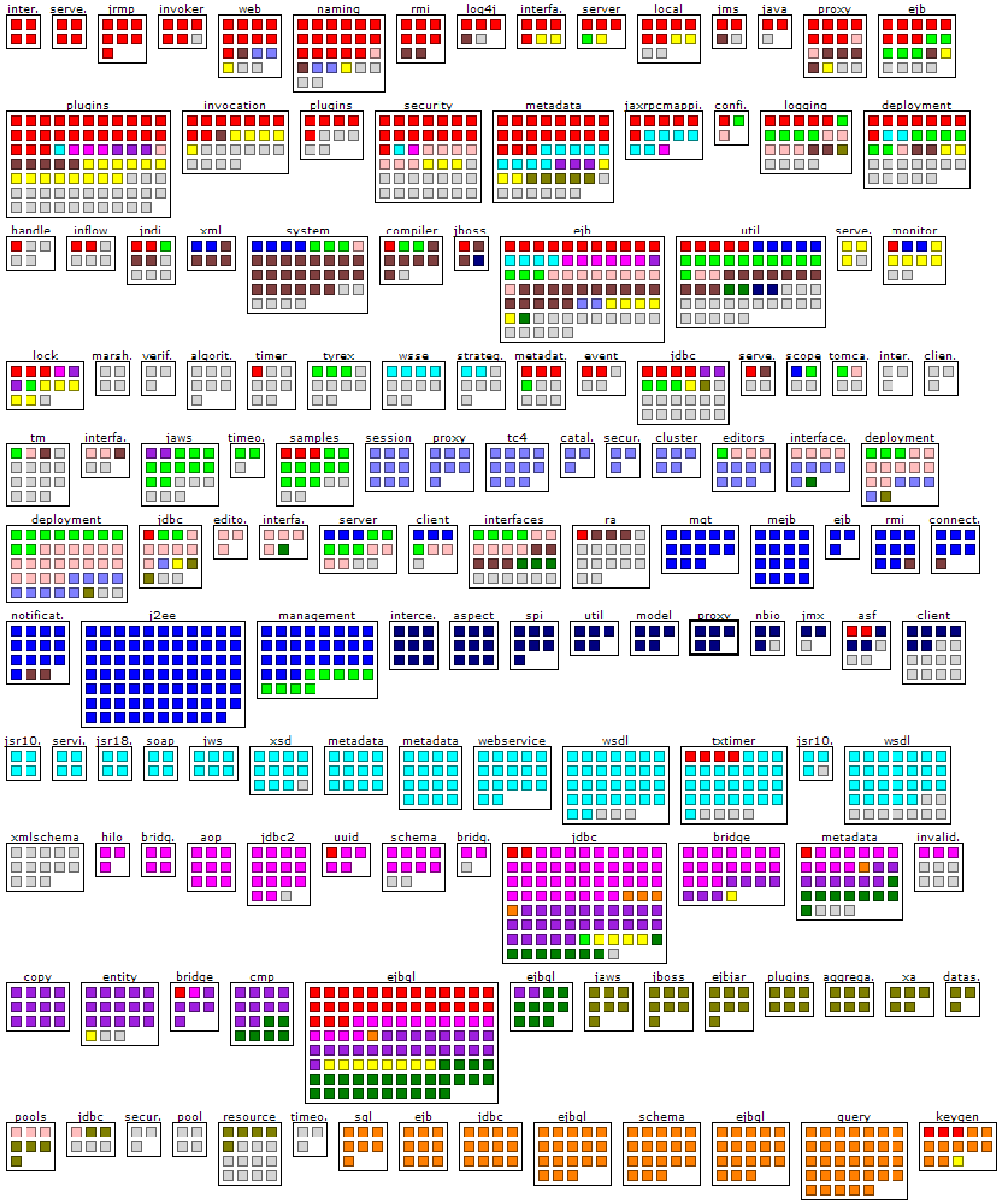


JBoss at a glance

Interactive tool

Data in perspective

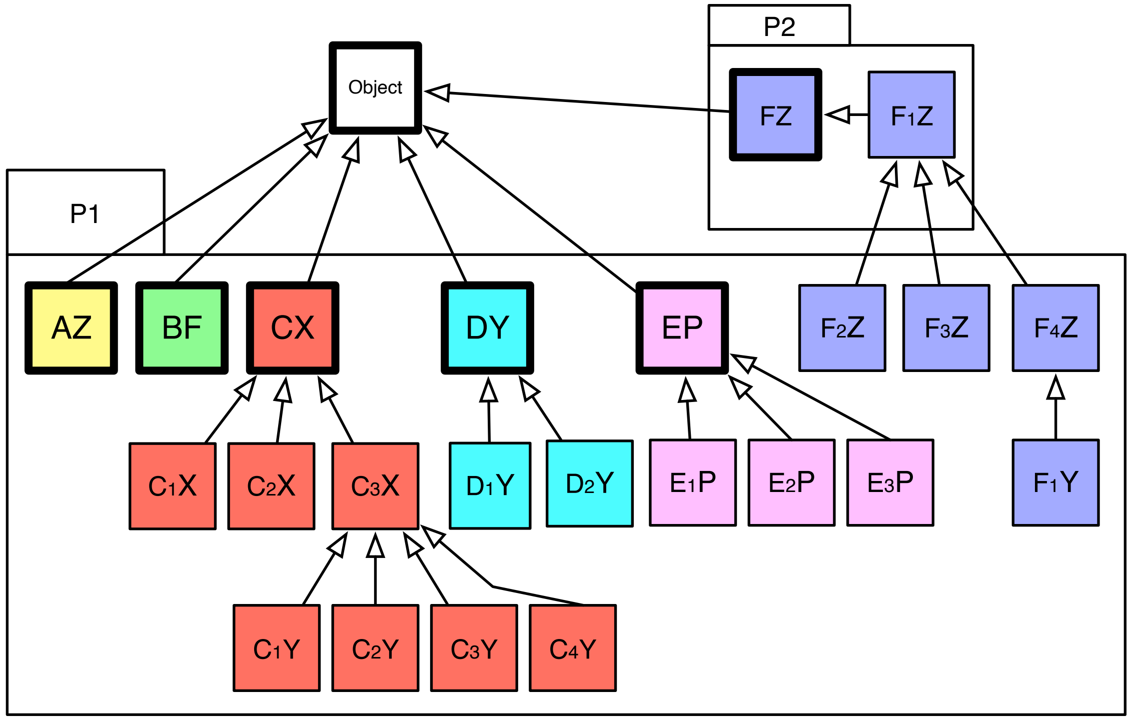


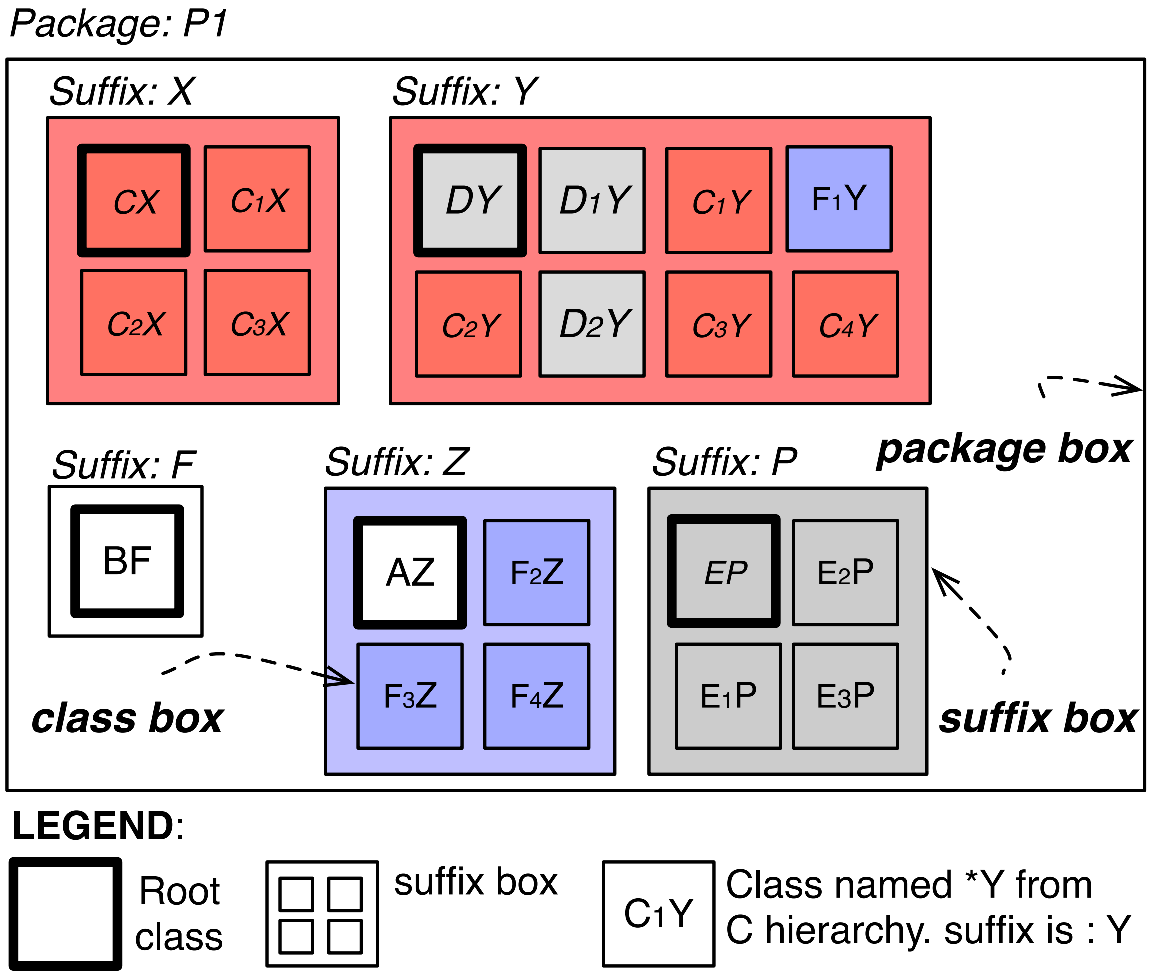


Currently: How to support understand classnames? [PhD N.-J. Agouf]

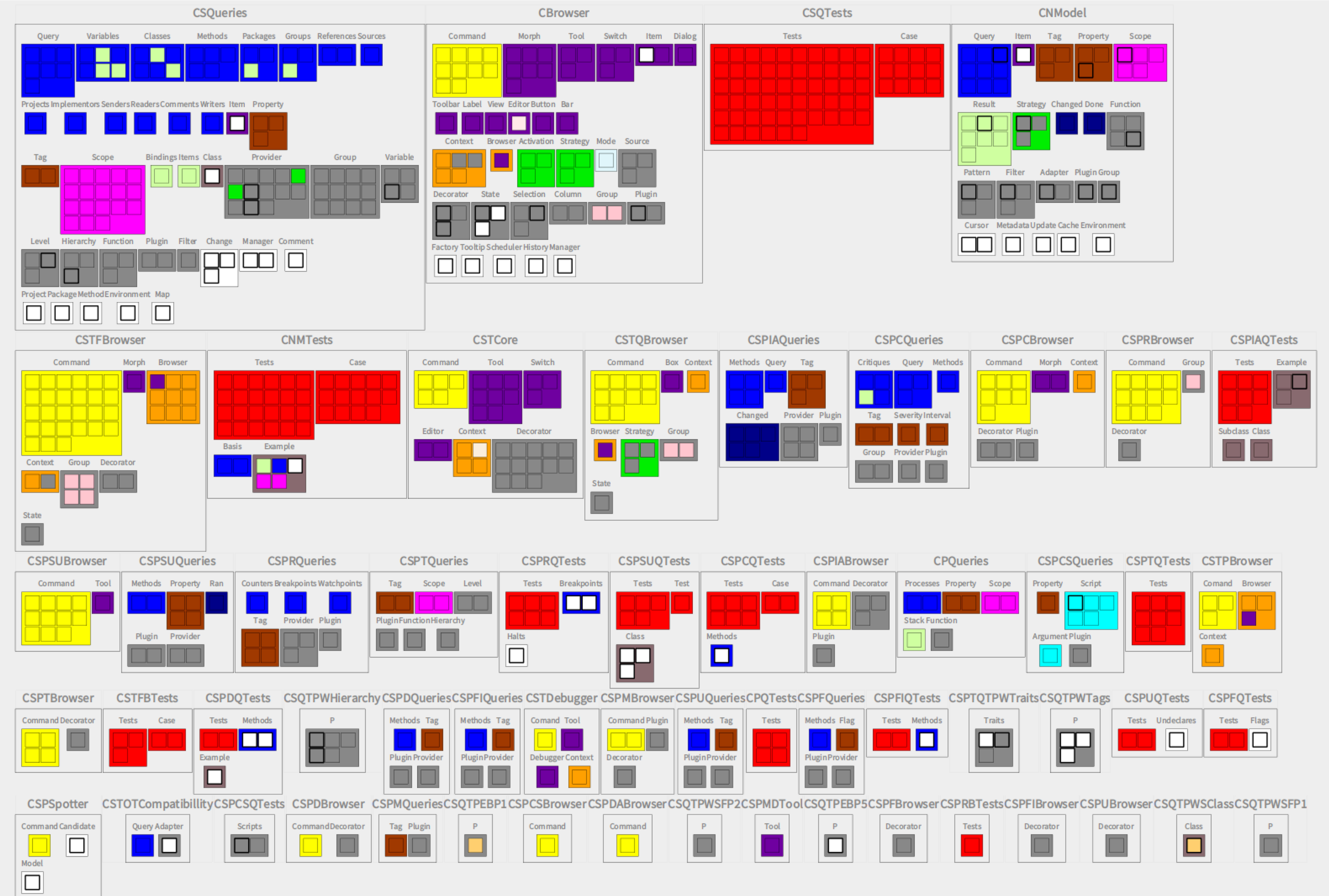
- How class are named?
  - is inheritance conveyed through names
- Is naming consistent?

One color = one hierarchy
One middle box = one suffix





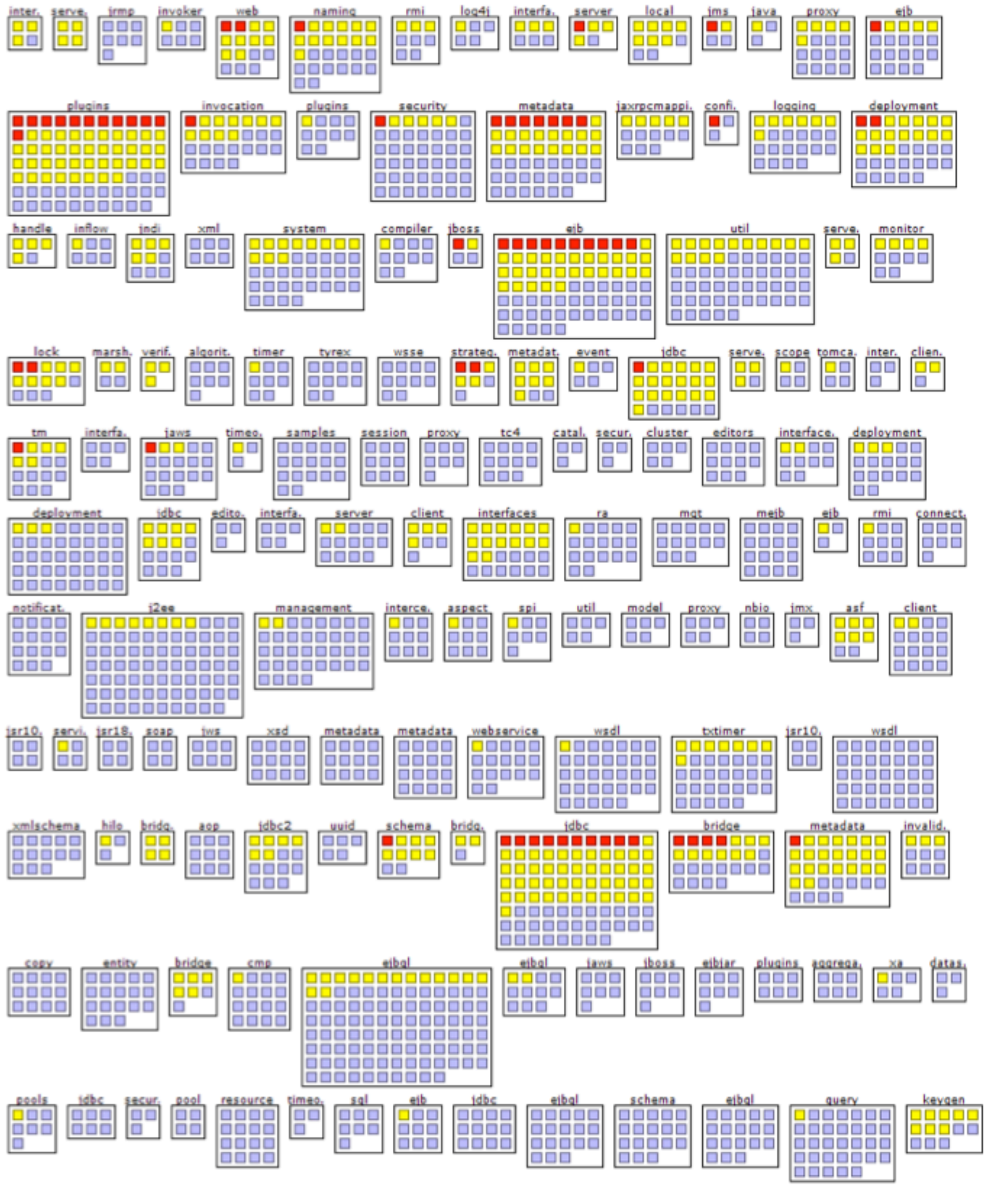
One color = one hierarchy
One middle box = one suffix



What about security (dreams so far)?

What are the maps we want to see?

- constructs maps
- “dangerous” expressions?
- inputs
- sequence of expressions



Roadmap

Legacy is not just Cobol

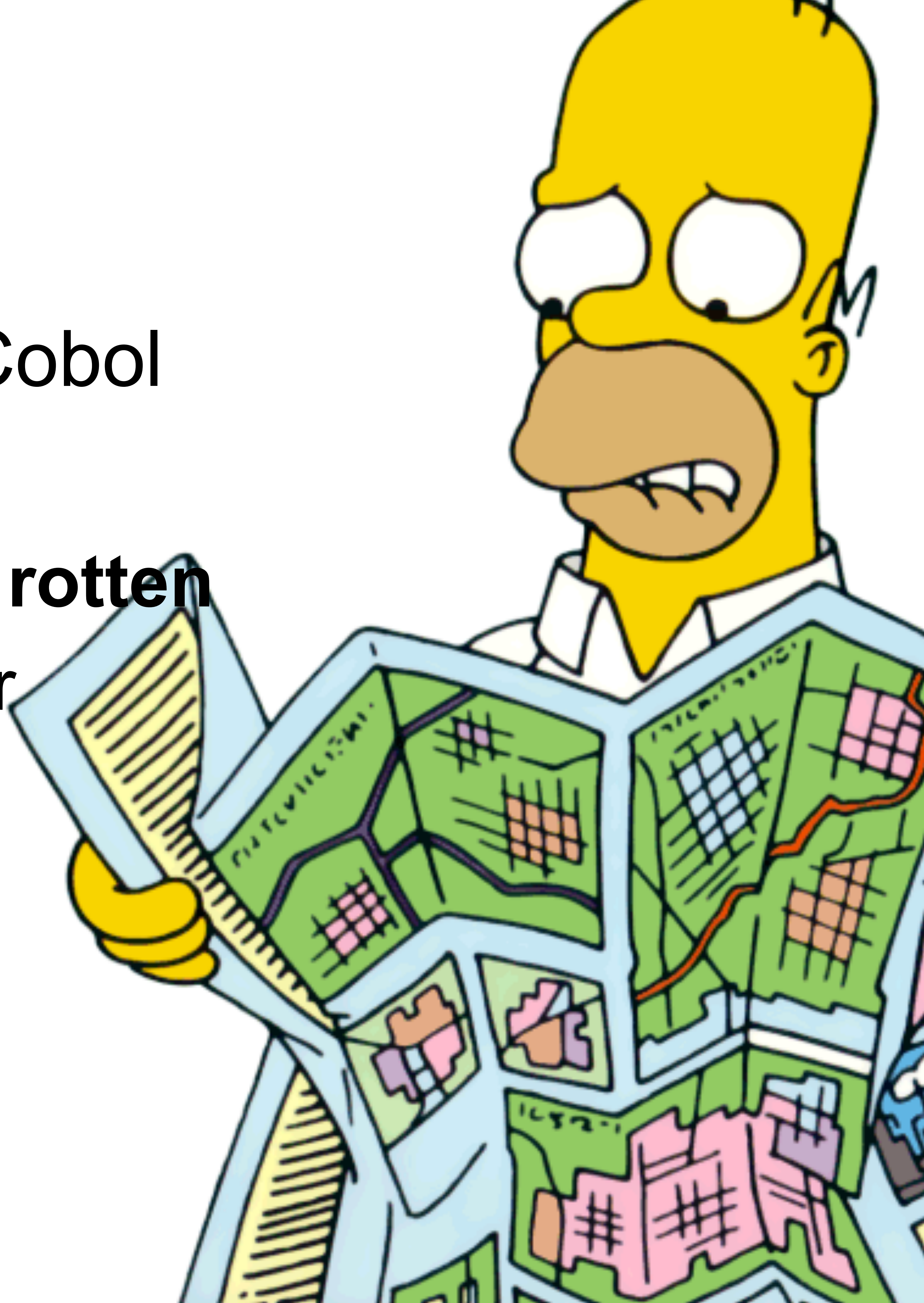
Software Maps

Green tests can be rotten

Research agenda for

Virtual Machines

Current effort



WHAT IS A ROTTEN GREEN TEST? (ICSE'19)

J. Delplanque, S. Ducasse, G. Polito, A. P. Black and A. Etien

*Univ. Lille, CNRS, Centrale Lille, **Inria**, UMR 9189 - CRIStAL*
Dept of Computer Science, Portland State University, Oregon, USA

ANATOMY OF A TEST

```
class SetTest {
    method testSetAdd {
        def s = Set.new()
        s.add(1)
        s.add(1)
        self.assertEquals(s.size(),1)
        self.assert(s.includes(1))
    }
}
```

NOT TALKING ABOUT A SMOKE TEST!

```
SetTest » testSetAddSmokeTest
    | s |
    s := Set new.
    s add: 1.
    s add: 1
```

- ➤ No assertion
- ➤ Not a rotten green test

A ROTTEN GREEN TEST IS

- A test ***passing (green)***
- A test that contains at least one ***assertion***
- One or more assertions is ***not*** executed when test runs

A LITTLE SKETCH OF A ROTTEN GREEN TEST

```
class RottenTest {
    method testABC {
     if (false) then {self.assert(x)}
    }

}
```

A REAL ONE

```
TPrintOnSequencedTest » testPrintOnDelimiter
    | aStream result allElementsAsString |
    result := ''.
    aStream := ReadWriteStream on: result.
    self nonEmpty printOn: aStream delimiter: ', '.
    allElementsAsString := result findBetweenSubstrings: ', ‘.
    allElementsAsString withIndexDo: [:el :i |
        self assert: el equals: ((self nonEmpty at:i) asString) ]
```

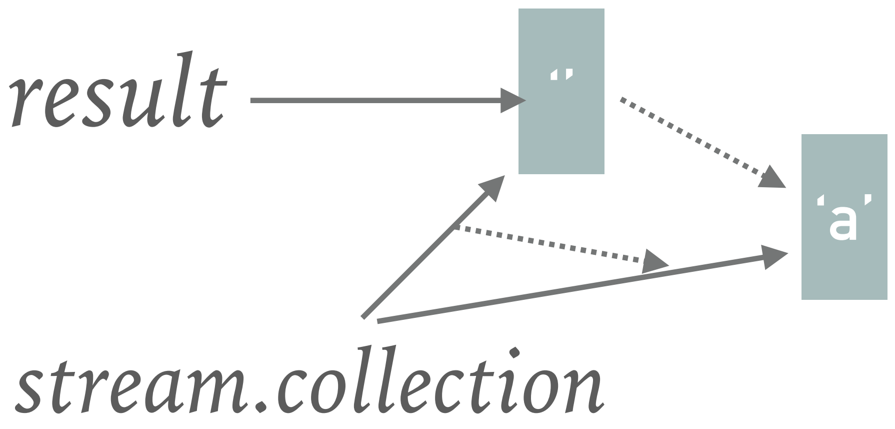
A REAL ONE

```
TPrintOnSequencedTest » testPrintOnDelimiter
    | aStream result allElementsAsString |
    result := ''.
    aStream := ReadWriteStream on: result.
    self nonEmpty printOn: aStream delimiter: ', '.
    allElementsAsString := result findBetweenSubstrings: ', ‘.
    allElementsAsString withIndexDo: [:el :i |
            self assert: el equals: ((self nonEmpty at:i) asString) ]
```

Not executed!

The programmer believed that the object on which the stream is working is “magically” mutated on stream growth

```
TPrintOnSequencedTest » testPrintOnDelimiter
    | aStream result allElementsAsString |
    result := ''.
    aStream := ReadWriteStream on: result.
    self nonEmpty printOn: aStream delimiter: ', '.
    allElementsAsString := result findBetweenSubstrings: ', '.
    allElementsAsString withIndexDo: [:el :i |
        self assert: el equals: ((self nonEmpty at:i) asString) ]
```



result stays empty

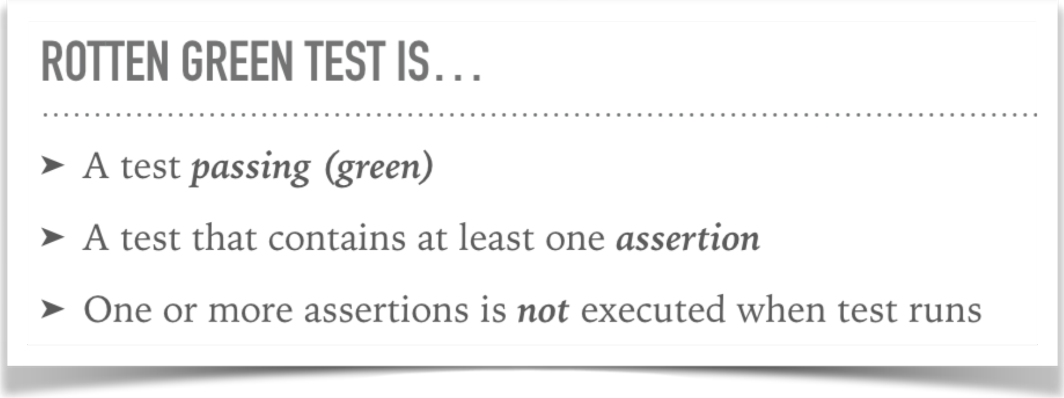
Iterator does not run

ROTTEN GREEN TEST WRITERS

- Rotten green tests are NOT intentional
- We say: this is ***not*** the programmer’s fault
- Instead: it is the fault of testing tools that **do not report** them

WHY ARE ROTTEN GREEN TESTS BAD?

- Give a false sense of security
- Can easily pass unnoticed
- Not reported by testing frameworks prior to ***DrTest***

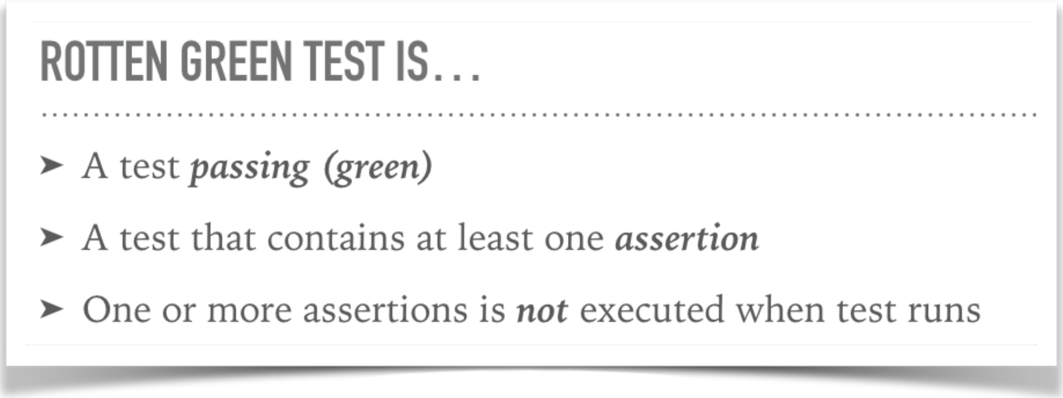


ROTTEN GREEN TEST IS…

- A test *passing (green)*
- A test that contains at least one ***assertion***
- One or more assertions is ***not*** executed when test runs

MAINLY CAUSED BY

- Conditional code not executing a branch
- Iterating over an empty collection



ROTTEN GREEN TEST IS…

- A test ***passing (green)***
- A test that contains at least one ***assertion***
- One or more assertions is ***not*** executed when test runs

HOW TO IDENTIFY THEM?

HANDLING HELPERS

```
class RottenTest {
    method testABC {
        if (false) then {self.helper()}
    }

    method helper {
        self.secondHelper()
    }

    method secondHelper {
        self.assert(x)
    }

}
```

HANDLING HELPERS

```
class RottenTest {
    method testABC {
        if (false) then {self.helper()}
    }                   Not executed!

    method helper {
        self.secondHelper()
    }

    method secondHelper {
        self.assert(x)
    }                   Not executed!

}
```

ABOUT THE NEED FOR CALL SITE ANALYSIS

```
class RottenTest {
    method testDEF {
        self.badHelper()
        self.assert(true)
    }
    method badHelper {
        if (false) then {
            self.secondHelper()
        }
    }
    method secondHelper {
        self.assert(x)
    }
}
```

ABOUT THE NEED FOR CALL SITE ANALYSIS

```
class RottenTest {
    method testDEF {
        self.badHelper()
        self.assert(true)
    }

    method badHelper {
        if (false) then {
            self.secondHelper()
         }
    }

    method secondHelper {
        self.assert(x)
    }

}
```

Executed!

Not executed!

Not executed!

IDENTIFYING ROTTEN GREEN TESTS

- We use both
  - *Static analysis,* to identify helpers and inherited methods
  - *Dynamic analysis,* to identify ***call*** sites that are not executed

BEFORE TEST EXECUTION: FIRST IDENTIFYING THE HELPERS

```
class RottenTest {
    method testDEF {
        self.badHelper()
        self.assert(true)
    }
    method badHelper {
        if (false) then {
            self.secondHelper()
        }
    }
    method secondHelper {
        self.assert(x)
    }
}
```

BEFORE TEST EXECUTION: FIRST IDENTIFYING THE HELPERS

```
class RottenTest {
    method testDEF {
        self.badHelper()
        self.assert(true)
    }
    method badHelper {   is an helper
        if (false) then {
            self.secondHelper()
        }
    }
    method secondHelper {   is an helper
        self.assert(x)
    }
}
```

BEFORE TEST EXECUTION: INSTALLING CALL SITE SPIES

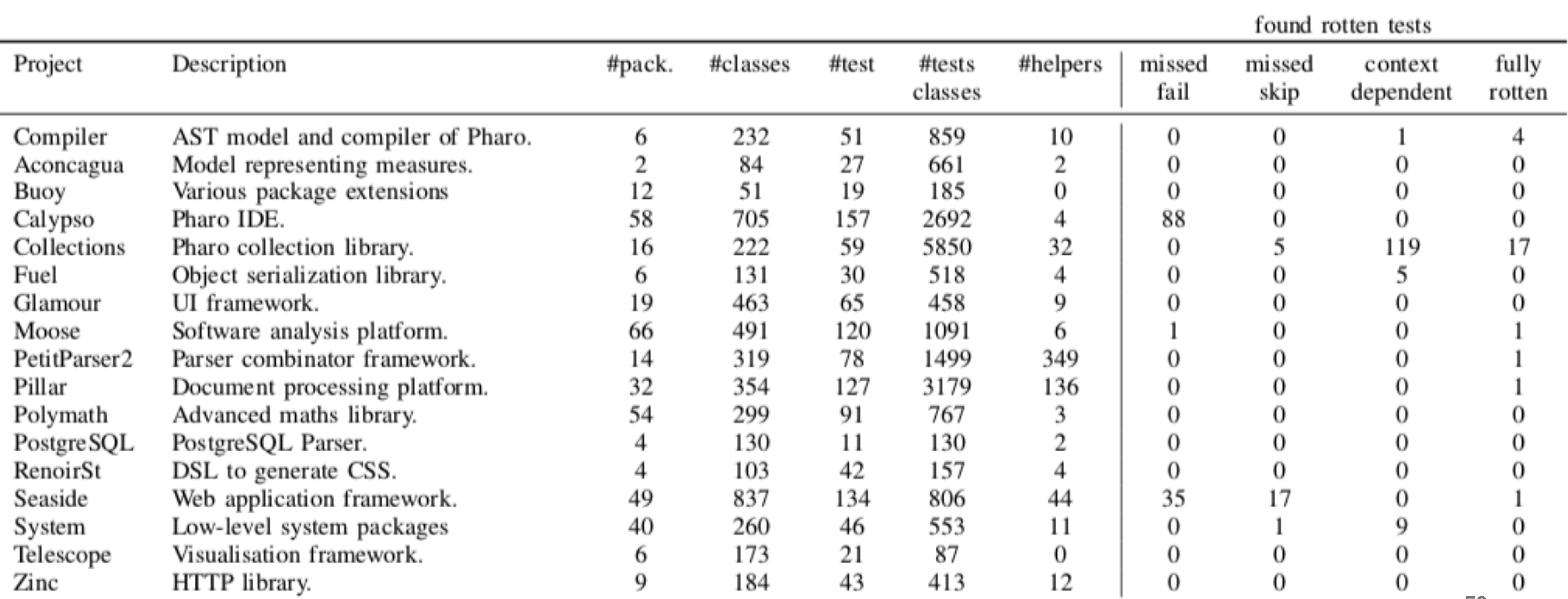
```
class RottenTest {
    method testDEF {
        self.badHelper()
        self.assert(true)    spy
    }
    method badHelper {
        if (false) then {
            self.secondHelper()
        }
    }
    method secondHelper {
        self.assert(x)    spy
    }
}
```

DURING EXECUTION

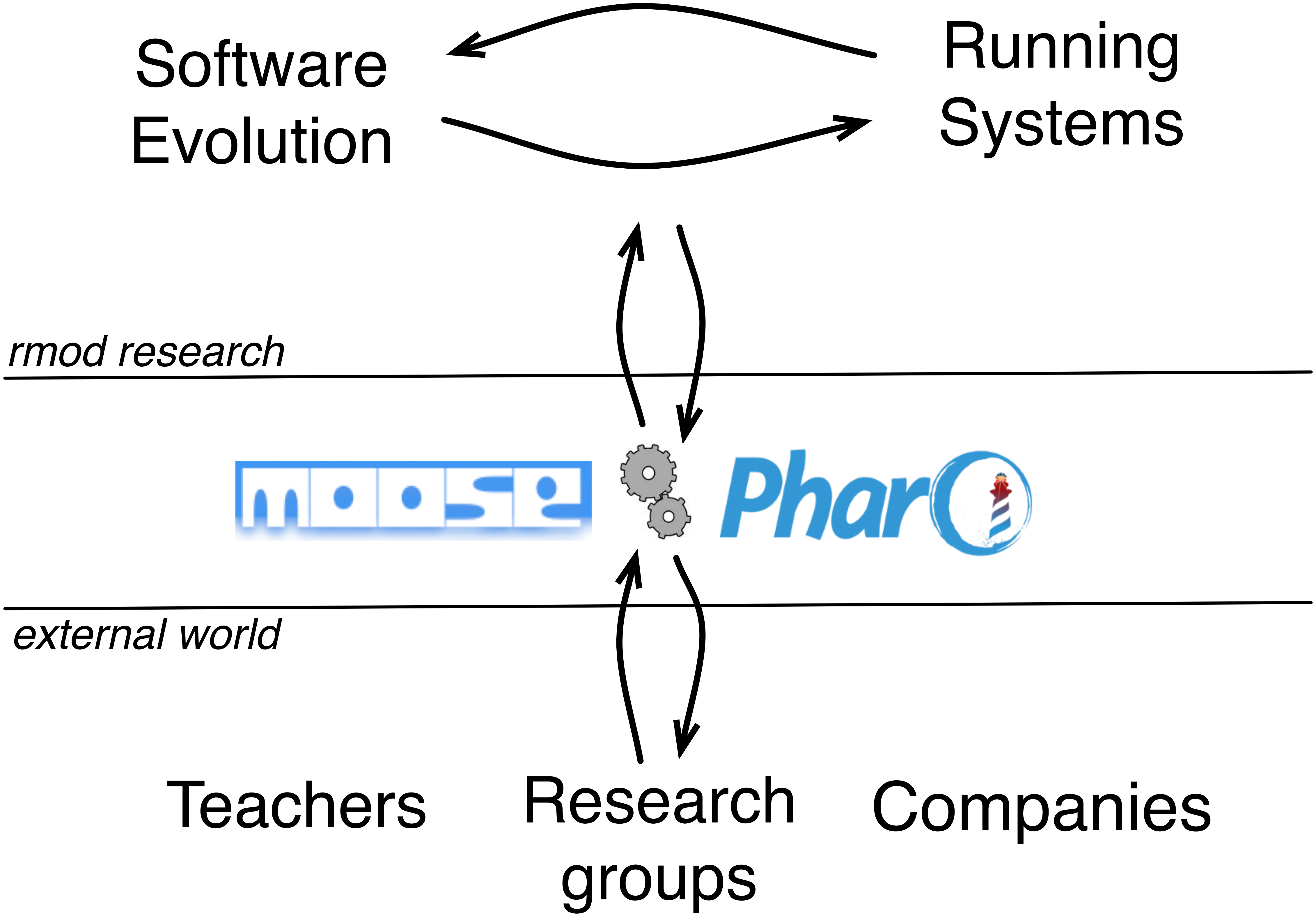
```
class RottenTest {
    method testDEF {
        self.badHelper()
        self.assert(true)      spy
    }
    method badHelper {
        if (false) then {
            self.secondHelper()
        }
    }
    method secondHelper {
        self.assert(x)      spy
    }
}
```

CASE STUDIES (CHECK THE PAPER AND THE FOLLOWING ONE)

- 19,905 tests analysed on mature projects
- 294 rotten (25 fully rotten)
- Found rotten green tests in Java and Python projects



							found rotten tests			
Project	Description	#pack.	#classes	#test	#tests classes	#helpers	missed fail	missed skip	context dependent	fully rotten
Compiler	AST model and compiler of Pharo.	6	232	51	859	10	0	0	1	4
Aconcagua	Model representing measures.	2	84	27	661	2	0	0	0	0
Buoy	Various package extensions	12	51	19	185	0	0	0	0	0
Calypso	Pharo IDE.	58	705	157	2692	4	88	0	0	0
Collections	Pharo collection library.	16	222	59	5850	32	0	5	119	17
Fuel	Object serialization library.	6	131	30	518	4	0	0	5	0
Glamour	UI framework.	19	463	65	458	9	0	0	0	0
Moose	Software analysis platform.	66	491	120	1091	6	1	0	0	1
PetitParser2	Parser combinator framework.	14	319	78	1499	349	0	0	0	1
Pillar	Document processing platform.	32	354	127	3179	136	0	0	0	1
Polymath	Advanced maths library.	54	299	91	767	3	0	0	0	0
PostgreSQL	PostgreSQL Parser.	4	130	11	130	2	0	0	0	0
RenoirSt	DSL to generate CSS.	4	103	42	157	4	0	0	0	0
Seaside	Web application framework.	49	837	134	806	44	35	17	0	1
System	Low-level system packages	40	260	46	553	11	0	1	9	0
Telescope	Visualisation framework.	6	173	21	87	0	0	0	0	0
Zinc	HTTP library.	9	184	43	413	12	0	0	0	0



Software Evolution

Running Systems

rmod research

external world

Teachers

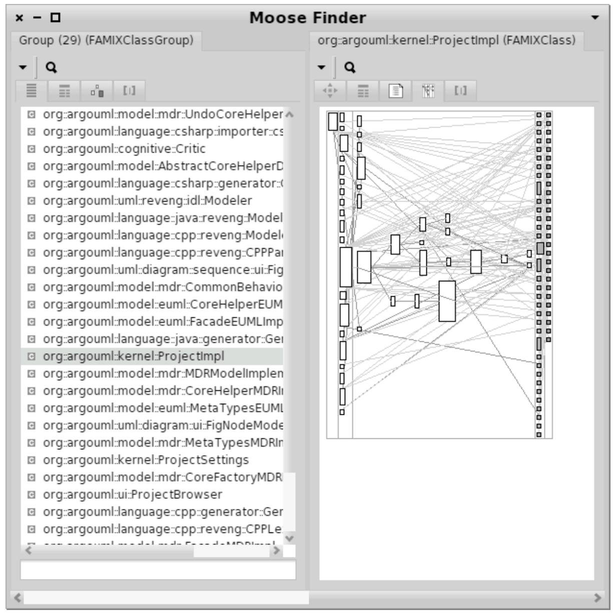
Research groups

Companies

classes select: #isGod

McCabe = 21

LOC = 753,000



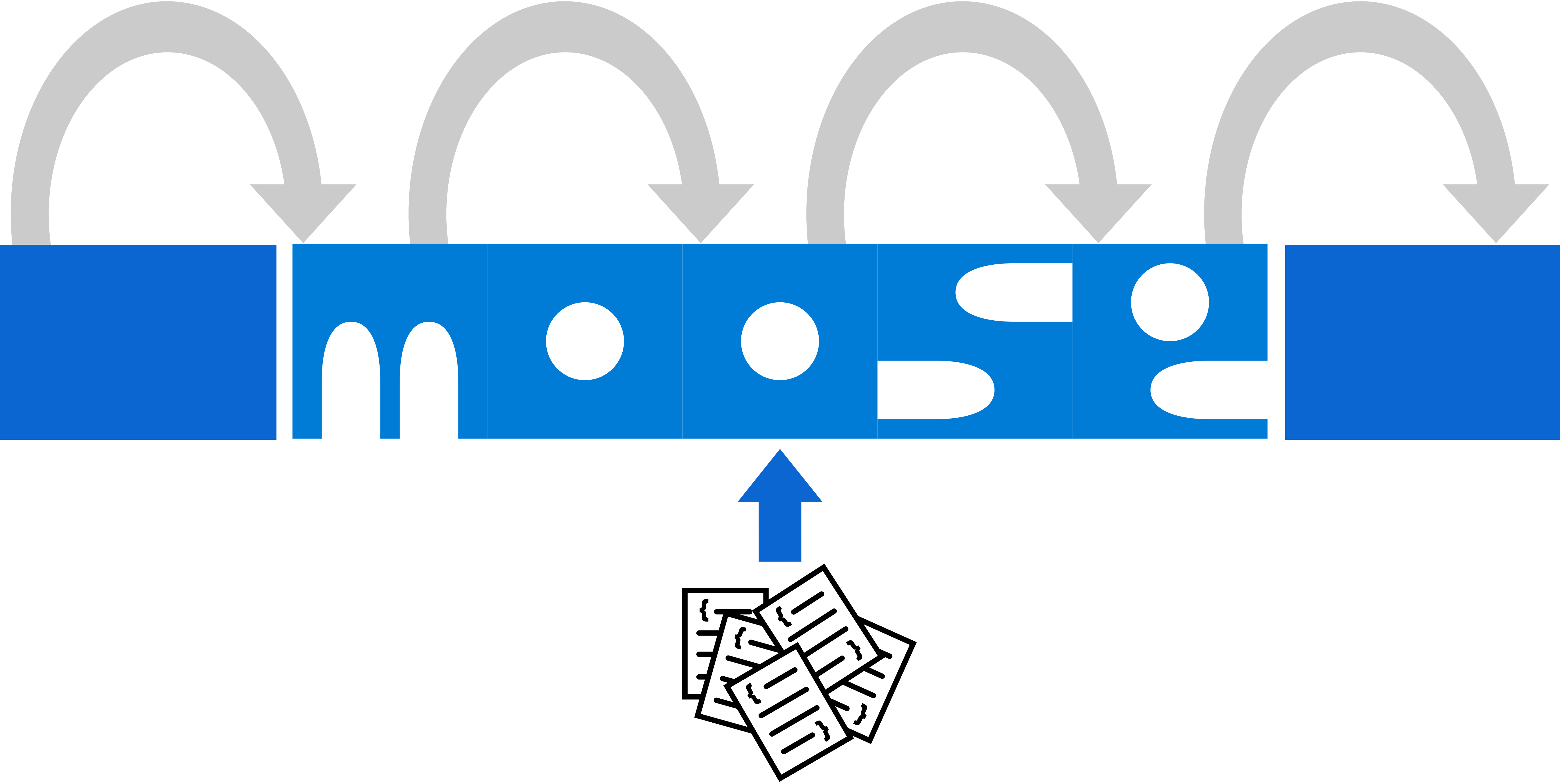
Moose Finder

Group (29) (FAMIXClassGroup)

org::argouml::model::mdr::UndoCoreHelper
org::argouml::language::csharp::importer::cs
org::argouml::cognitive::Critic
org::argouml::model::AbstractCoreHelperD
org::argouml::language::csharp::generator::
org::argouml::uml::reveng::idl::Modeler
org::argouml::language::java::reveng::Model
org::argouml::language::cpp::reveng::Model
org::argouml::language::cpp::reveng::CPPPar
org::argouml::uml::diagram::sequence::ui::Fig
org::argouml::model::mdr::CommonBehavio
org::argouml::model::euml::CoreHelperEUM
org::argouml::model::euml::FacadeEUMLImp
org::argouml::language::java::generator::Ger
org::argouml::kernel::ProjectImpl
org::argouml::model::mdr::MDRModelImplen
org::argouml::model::mdr::CoreHelperMDRI
org::argouml::model::euml::MetaTypesEUML
org::argouml::uml::diagram::ui::FigNodeMode
org::argouml::model::mdr::MetaTypesMDRIr
org::argouml::kernel::ProjectSettings
org::argouml::model::mdr::CoreFactoryMDR
org::argouml::ui::ProjectBrowser
org::argouml::language::cpp::generator::Ger
org::argouml::language::cpp::reveng::CPPLe

org::argouml::kernel::ProjectImpl (FAMIXClass)

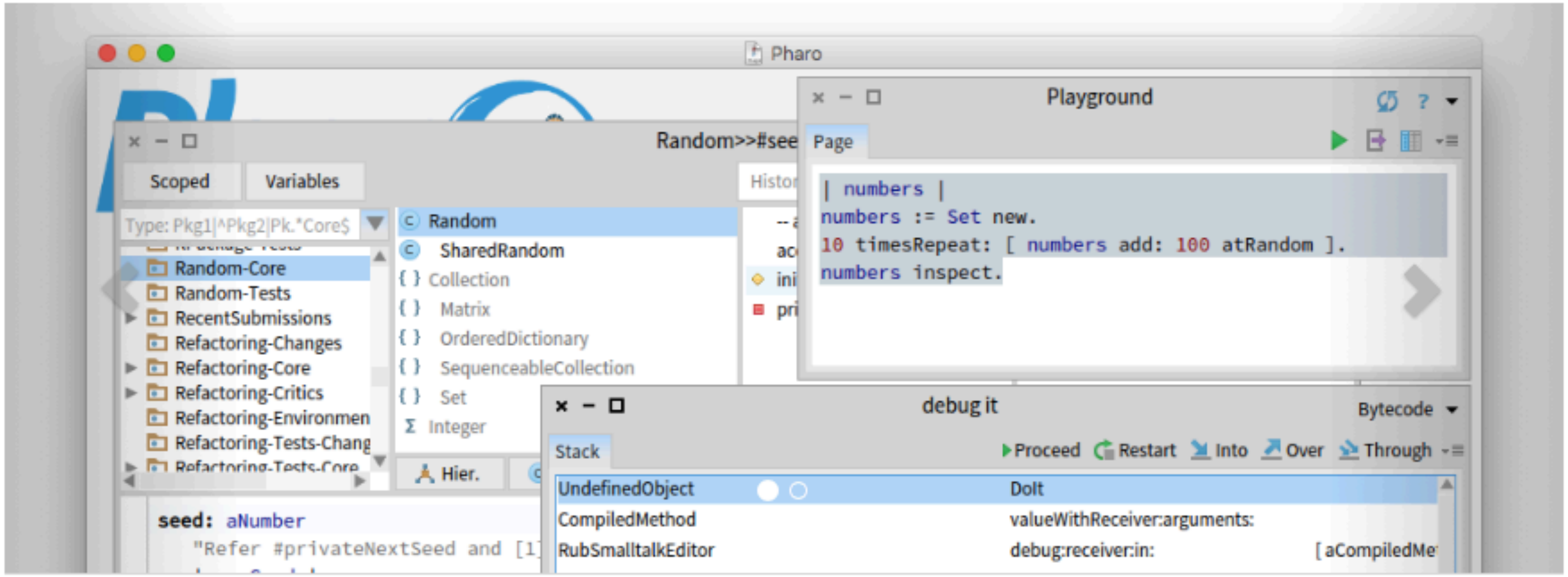
...



Pharo

The immersive programming experience

Pharo is a pure object-oriented programming language *and* a powerful environment, focused on simplicity and immediate feedback (think IDE and OS rolled into one).



```
| numbers |
numbers := Set new.
10 timesRepeat: [ numbers add: 100 atRandom ].
numbers inspect.
```

```
seed: aNumber
    "Refer #privateNextSeed and [1]
```

```
UndefinedObject        DoIt
CompiledMethod         valueWithReceiver:arguments:
RubSmalltalkEditor     debug:receiver:in:        [aCompiledMe
```

Pharo 90
~740 packages
- 9 000 classes
- 120 000 methods

250 forks sur Github
up to 100 contributors
30 regulars
- 8 sub projets
- **graphics**
- **vcs**
- **tools**

Consortium
~ 28 companies
~ 25 academic

Discover

Learn more about Pharo's key features and elegant design.

Download

Download latest version (8.0)!
Read more about here

Learn

Access the Pharo Mooc!
3000 people registered and follow the Pharo Mooc. You can find it here.
Watch the teaser!

Subscribe to the Pharo Newsletter

email address | Subscribe

Follow us on Twitter: @pharoproject

Roadmap

Legacy is not just Cobol

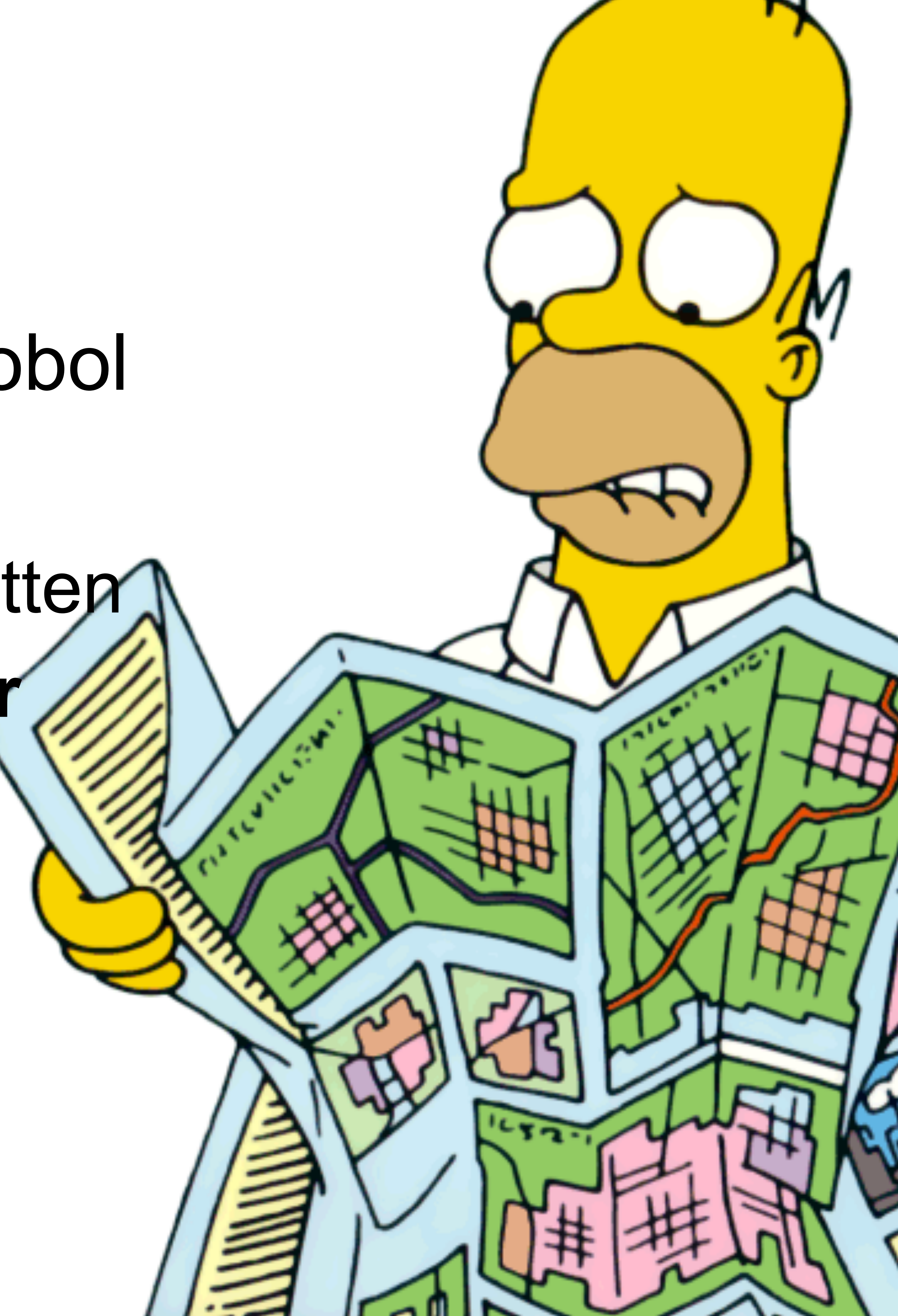
Software Maps

Green tests can be rotten

Research agenda for

Virtual Machines

Current effort



Virtual Machines

Modern Language Implementations

Managed **Execution**

Runtime Binary Translation

Managed **Memory**

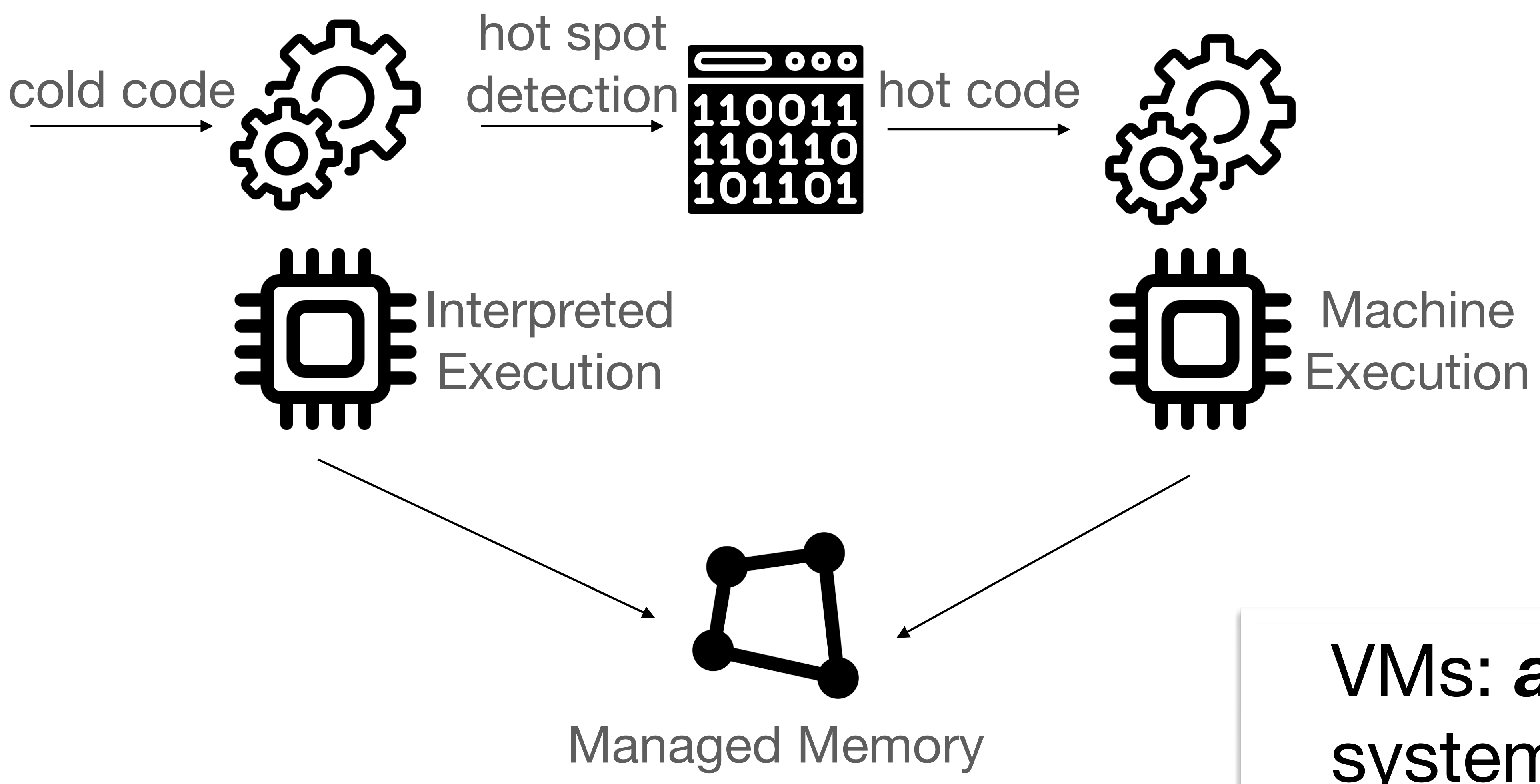
Hardware/System Interaction

ORACLE

Microsoft

Virtual Machines

Typical Architecture Overview



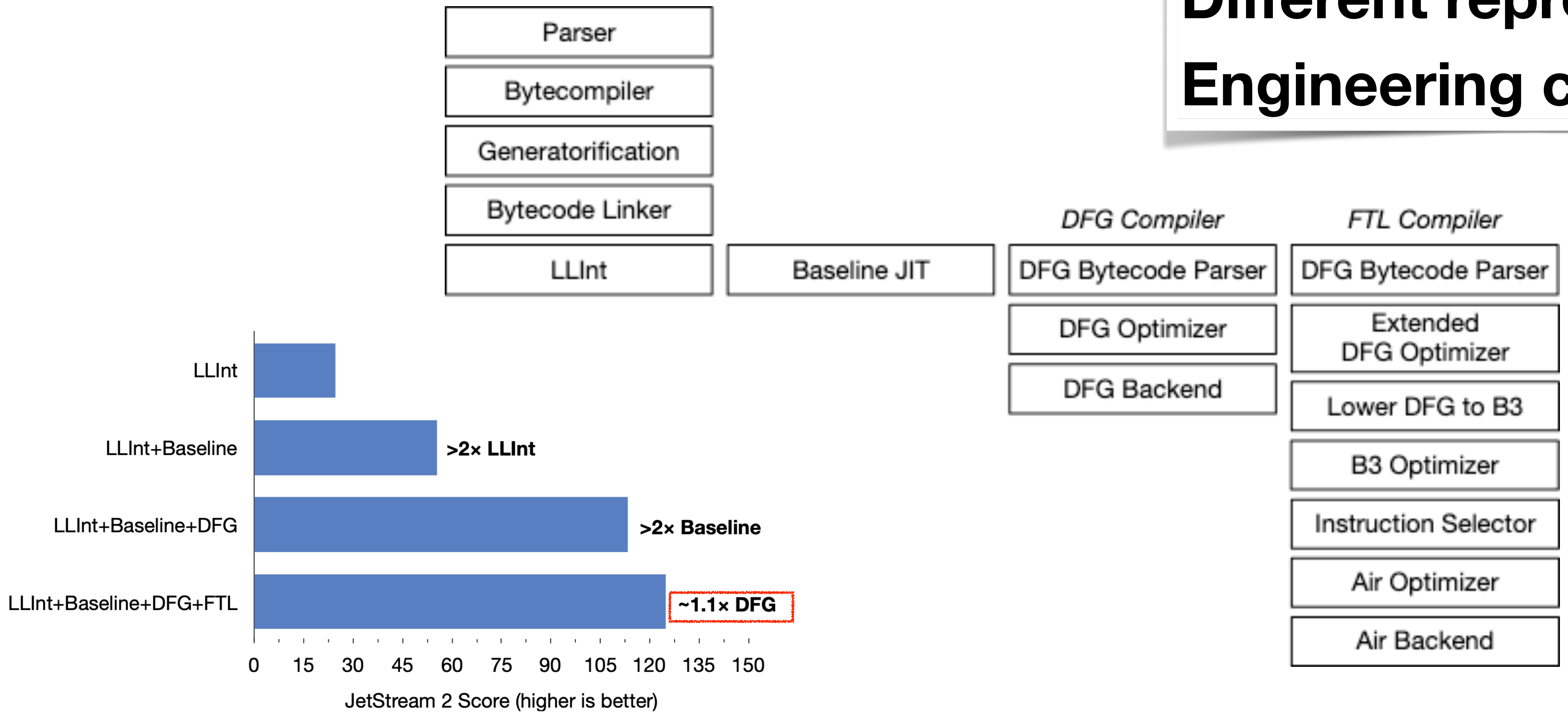
VMs: ***auto-adapt***
systems

Complexity and Cost of VMs

Multiple levels

Different repres

Engineering cos



Parser

Bytecompiler

Generatorification

Bytecode Linker

LLInt

Baseline JIT

DFG Compiler

DFG Bytecode Parser

DFG Optimizer

DFG Backend

FTL Compiler

DFG Bytecode Parser

Extended DFG Optimizer

Lower DFG to B3

B3 Optimizer

Instruction Selector

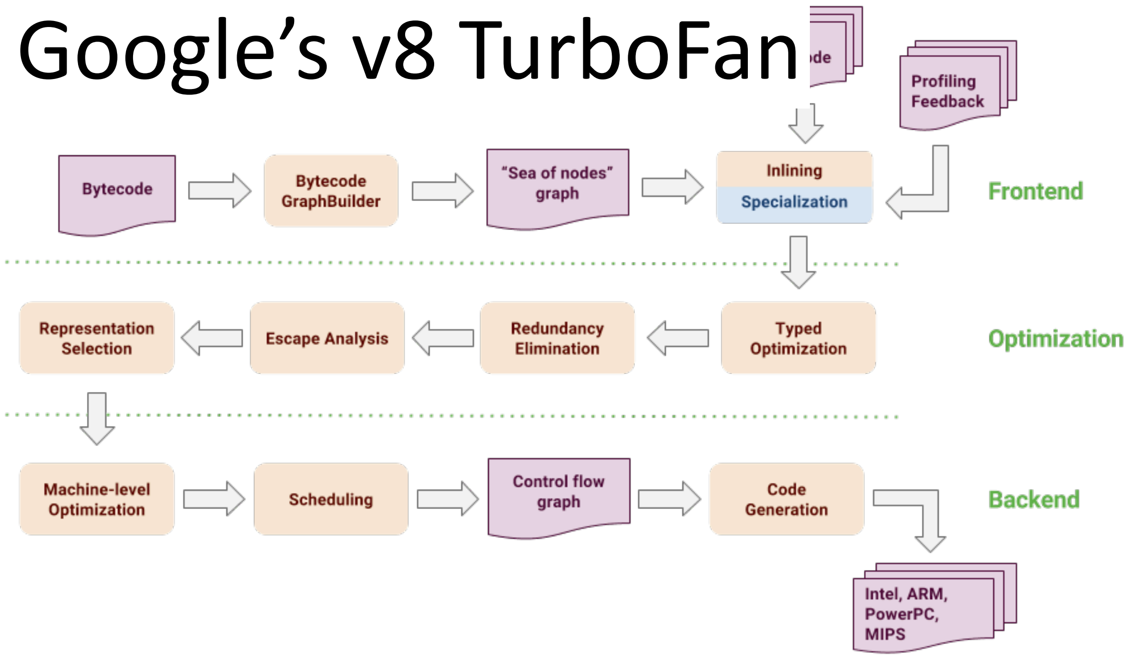
Air Optimizer

Air Backend

https://webkit.org/blog/10308/speculation-in-javascriptcore

Complexity and Cost of VMs (II)

Google's v8 TurboFan



ode

Profiling Feedback

Bytecode

Bytecode GraphBuilder

"Sea of nodes" graph

Inlining

Specialization

Frontend

Representation Selection

Escape Analysis

Redundancy Elimination

Typed Optimization

Optimization

Machine-level Optimization

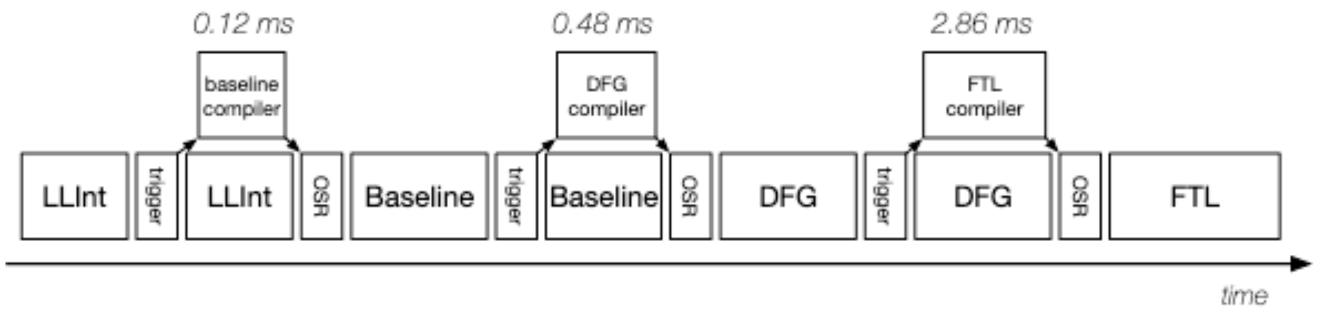
Scheduling

Control flow graph

Code Generation

Backend

Intel, ARM, PowerPC, MIPS



0.12 ms

0.48 ms

2.86 ms

baseline compiler

DFG compiler

FTL compiler

LLInt

trigger

LLInt

OSR

Baseline

trigger

Baseline

OSR

DFG

trigger

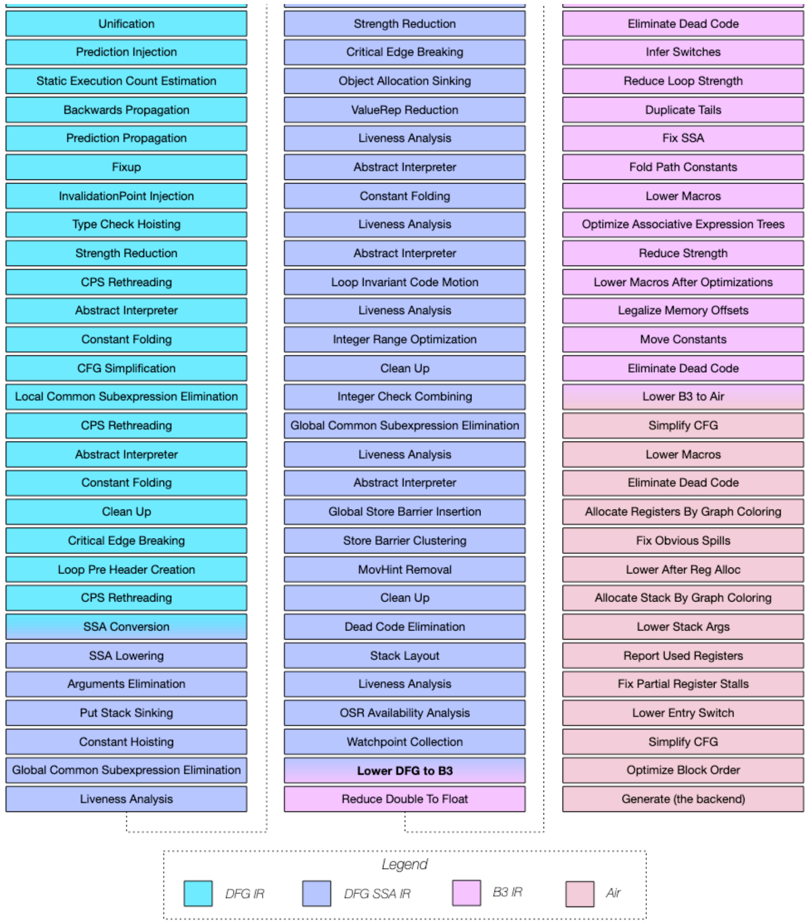
DFG

OSR

FTL

time

Apple's Safari JavascriptCore[2021]



Unification

Prediction Injection

Static Execution Count Estimation

Backwards Propagation

Prediction Propagation

Fixup

InvalidationPoint Injection

Type Check Hoisting

Strength Reduction

CPS Rethreading

Abstract Interpreter

Constant Folding

CFG Simplification

Local Common Subexpression Elimination

CPS Rethreading

Abstract Interpreter

Constant Folding

Clean Up

Critical Edge Breaking

Loop Pre Header Creation

CPS Rethreading

SSA Conversion

SSA Lowering

Arguments Elimination

Put Stack Sinking

Constant Hoisting

Global Common Subexpression Elimination

Liveness Analysis

Strength Reduction

Critical Edge Breaking

Object Allocation Sinking

ValueRep Reduction

Liveness Analysis

Abstract Interpreter

Constant Folding

Liveness Analysis

Abstract Interpreter

Loop Invariant Code Motion

Liveness Analysis

Integer Range Optimization

Clean Up

Integer Check Combining

Global Common Subexpression Elimination

Liveness Analysis

Abstract Interpreter

Global Store Barrier Insertion

Store Barrier Clustering

MovHint Removal

Clean Up

Dead Code Elimination

Stack Layout

Liveness Analysis

OSR Availability Analysis

Watchpoint Collection

Lower DFG to B3

Reduce Double To Float

Eliminate Dead Code

Infer Switches

Reduce Loop Strength

Duplicate Tails

Fix SSA

Fold Path Constants

Lower Macros

Optimize Associative Expression Trees

Reduce Strength

Lower Macros After Optimizations

Legalize Memory Offsets

Move Constants

Eliminate Dead Code

Lower B3 to Air

Simplify CFG

Lower Macros

Eliminate Dead Code

Allocate Registers By Graph Coloring

Fix Obvious Spills

Lower After Reg Alloc

Allocate Stack By Graph Coloring

Lower Stack Args

Report Used Registers

Fix Partial Register Stalls

Lower Entry Switch

Simplify CFG

Optimize Block Order

Generate (the backend)

Legend

DFG IR

DFG SSA IR

B3 IR

Air

https://webkit.org/blog/10308/speculation-in-javascriptcore/
https://ponyfoo.com/articles/an-introduction-to-speculative-optimization-in-v8

Managed Execution

Remarkable Challenges

- What are ***optimal*** organisations of multi-tier engines?
    - Combining interpreters with ***many levels*** of optimising compilers
- What is a ***better/minimal runtime*** support for developer ***tooling***?
    - Better debugging support
    - Runtime (speed, energy…) profiling
    - Benchmark automatic generation

Runtime Binary Translation

Remarkable Challenges

VMs are ***auto-adaptive*** systems

- How can runtime-compilers ***better speculate*** on application behaviour?
    - Speculate **on** more than types
    - Speculate **for** more than speed
- How can we improve the efficiency of ***cold code***?
    - Better interpreter optimisations
    - Low overhead binary translators

Managed Memory

Remarkable Challenges

- How can ***managed memory adapt*** to memory consumption patterns?
    - Scalability to ***multi-TB*** heaps
    - Automatically memory re-organisation
    - Reduce pauses
    - Support for modern hardware (e.g., non-volatile memories)

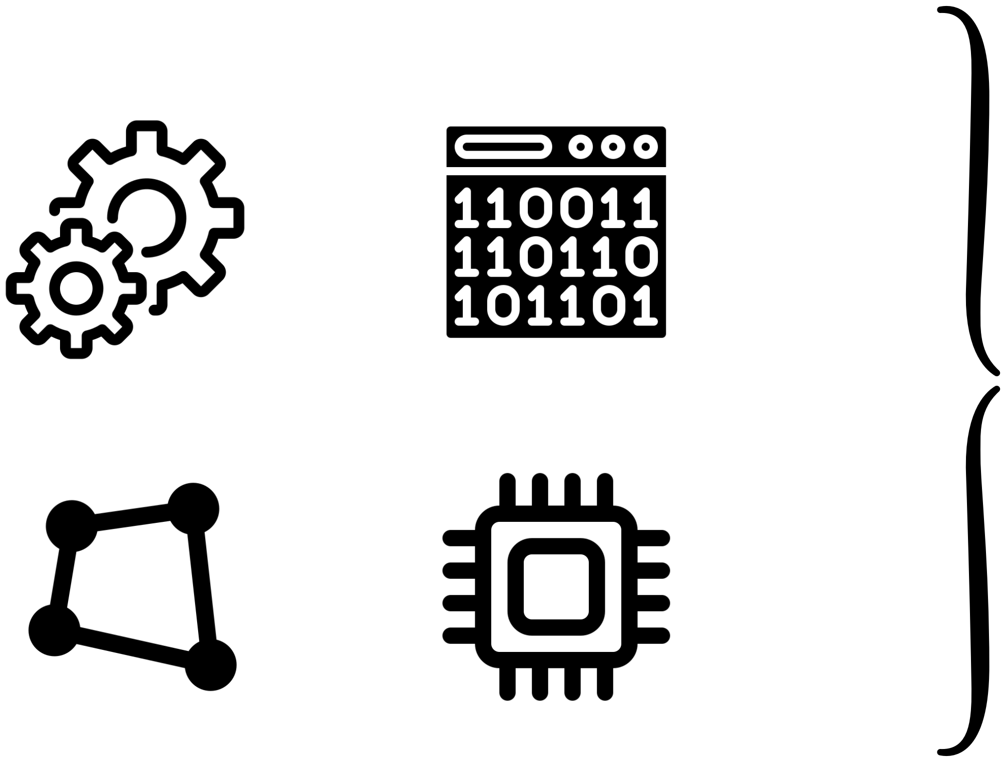
Hardware/System Interaction

Remarkable Challenges

- How can modern VMs exploit ***hardware-software co-design***?
- Automatic deport computation to dedicated hardware
    - GPU
    - FPGA
    - Extensible ISAs (e.g., RISC-V)

Cross-Cutting Challenges

(And Contradictory Challenges!)



- Energy Consumption
- Execution Speed
- Security
- Correctness
- Modularity

Cross-Cutting Challenges

Selected Challenges

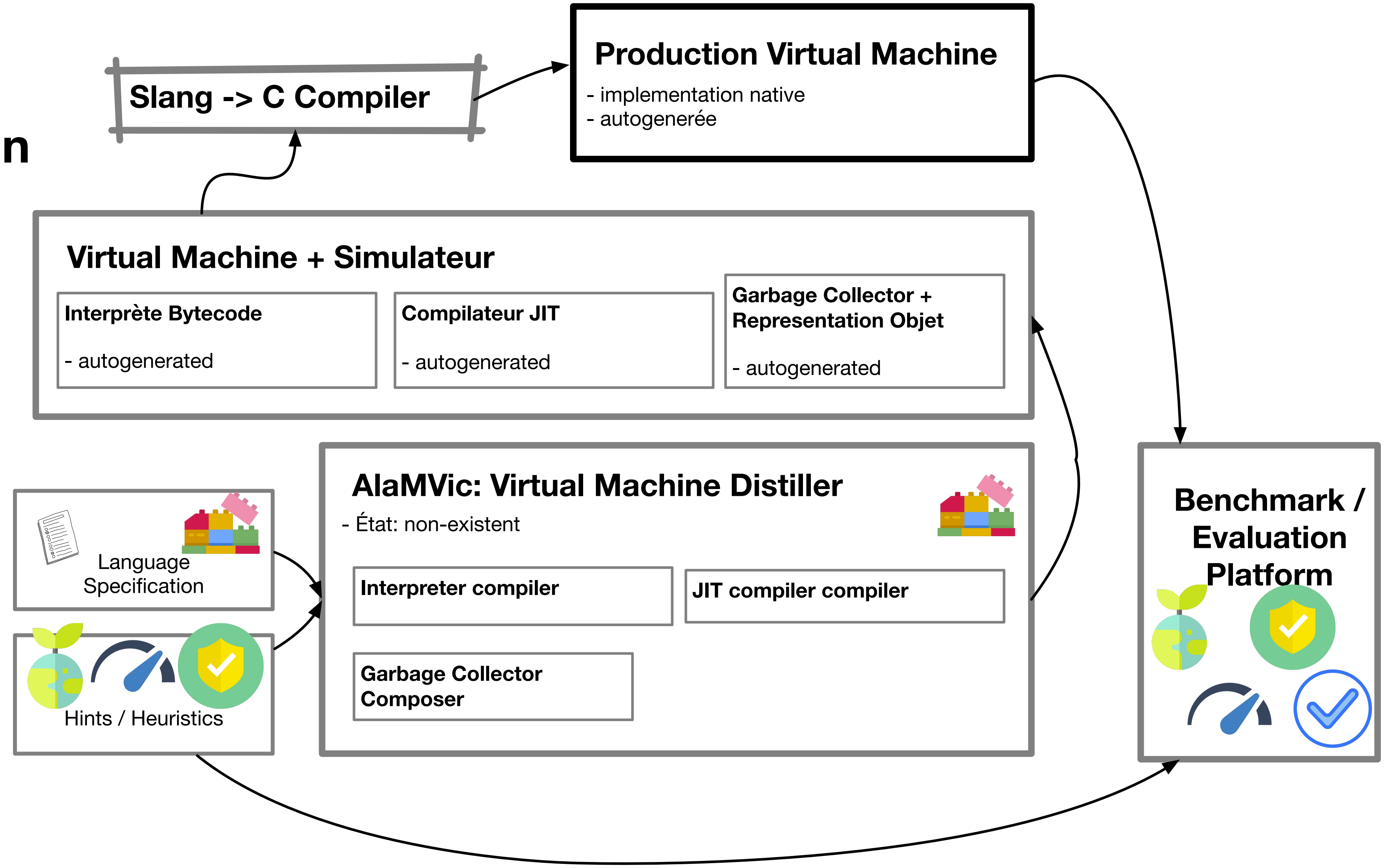
- **Security threats** of multi-tier execution engines
- Speculative runtime compilation for **frugal systems**
- **Profile-guided** detection of application parallelisation opportunities
- **Securing** VMs through **dedicated** hardware
- Minimising **energy impact** of garbage collection algorithms

Selected Software Engineering Challenges

- **Automatic** detection of **performance** regressions
- **Automatic validation** of multi-tier execution engines
- Minimising the **construction cost** of efficient JIT compilers

AlaMVic: a generative approach

- Compiler generation
- Exchangeable components
- Optimization heuristics
- Open exploratory platform



Slang -> C Compiler

Production Virtual Machine

- implementation native
- autogenerée

Virtual Machine + Simulateur

Interprète Bytecode

- autogenerated

Compilateur JIT

- autogenerated

Garbage Collector + Representation Objet

- autogenerated

Language Specification

Hints / Heuristics

AlaMVic: Virtual Machine Distiller

- État: non-existent

Interpreter compiler

JIT compiler compiler

Garbage Collector Composer

Benchmark / Evaluation Platform

Early RMOD achievements

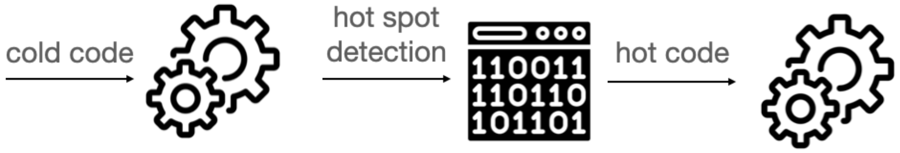
Dev side of things

- JIT for Apple M1, Windows, Raspberry ARM 64bits in production
- Helping ENSTA Bretagne to develop a Risc-V JIT
- Streamlining transpilation/compilation chain
- Taking advantage of VM tests [MPLR paper]
- Some productivity enhancer tools (Unicorn simulator, assembly browser, interactive CFG navigation,…)

Early RMOD achievements

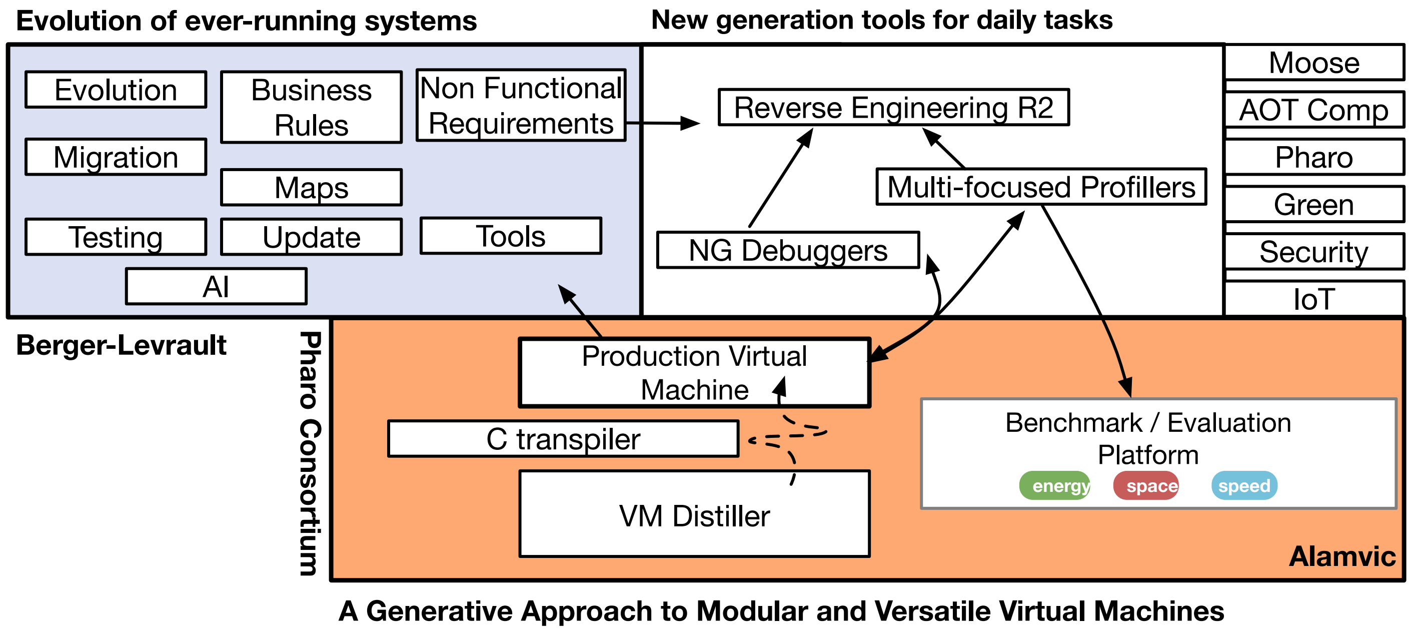
Research side

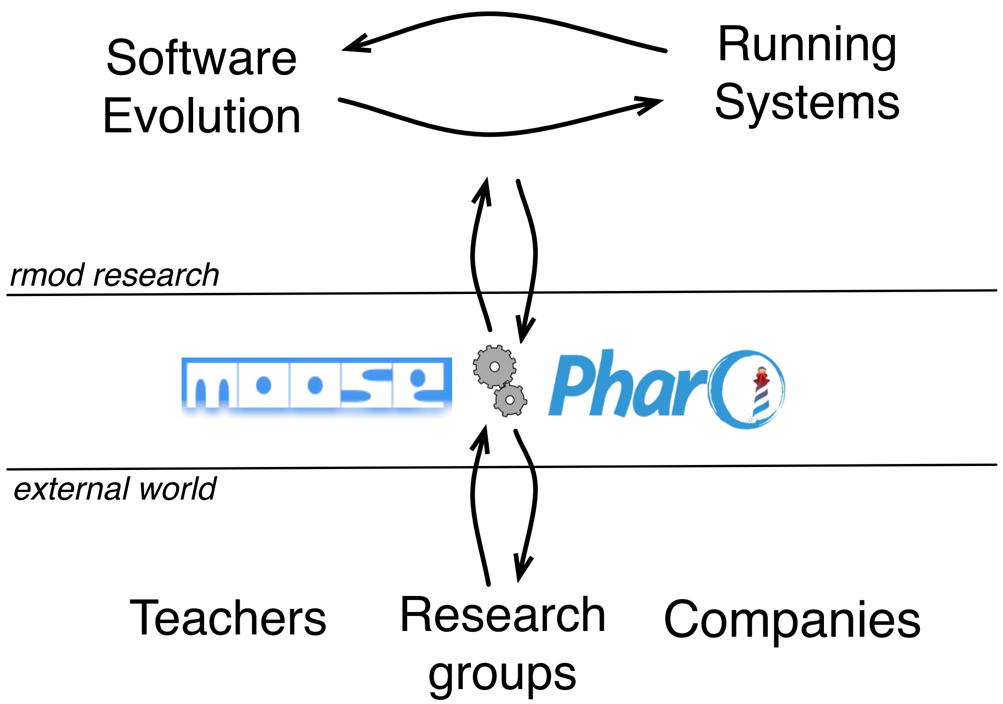
- RQ: ***static*** code fall through reorganisation is it worth ? (alternative to Pettis-Hansen BB reordering)



- Reducing the load of manual code (~100 bytecodes, ~300 primitives)
    - RQ1: Are interpreted and compiled code equivalent? Concolic + differential testing
    - RQ2: Can we remove manual compiled code? Abstract interpreter for compiled code generation (underway)

RMod





Pharo Consortium

Pharo Consortium