CWI Centrum Wiskunde & Informatica SEN1: SWAT

Universiteit van Amsterdam

MSE vrije Universiteit amsterdam

ATEAMS INRIA

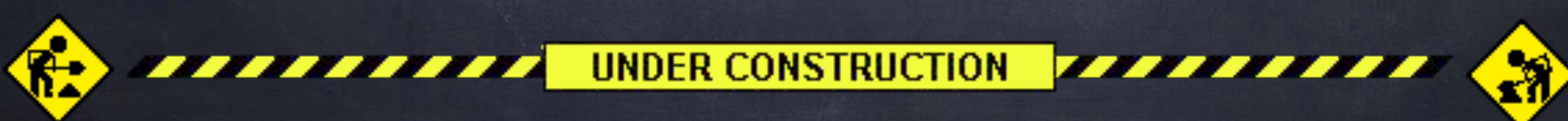
Controlled Experiments in Software Engineering

Jurgen Vinju

TCSA Day, October 28th 2011

This talk is about improving software research

- What is software **engineering**?
    - What is **software**?
    - What are the research **questions**?
    - What are the research **methods?**
- A new **empirical** research method
    - That can **isolate** causes of software quality
    - That **motivates** theoretical research in program semantics



Software engineering:

"The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software, and the study of these approaches." [SWEBOK]

What we have proven and/or have evidence of:

- people trump technology and methodology
- size matters
- many techn[...]pes

Unsatisfactory

- we do not know what **matters** about these recipes
- We do not know which **design choices** are better

Vik Muniz

"The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software, and the study of these approaches."

Solution

Reality

Quality

C0mpl3x17y

Problem

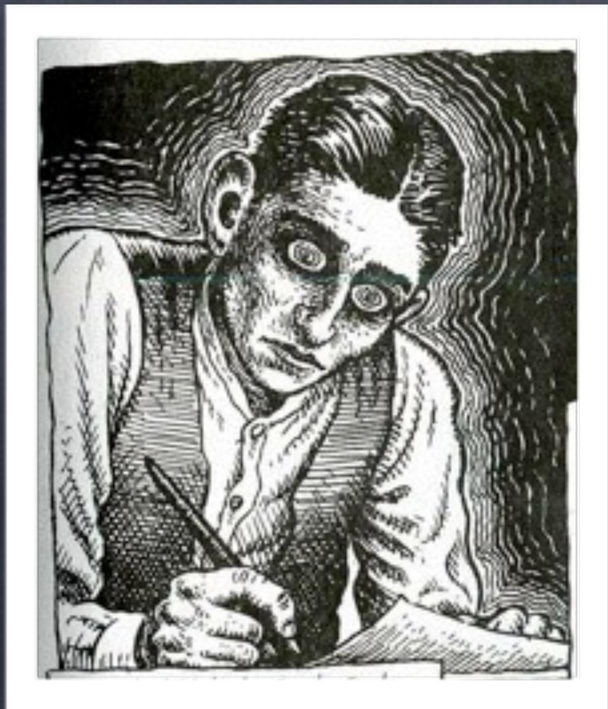
POLITIE

vts Politie Nederland

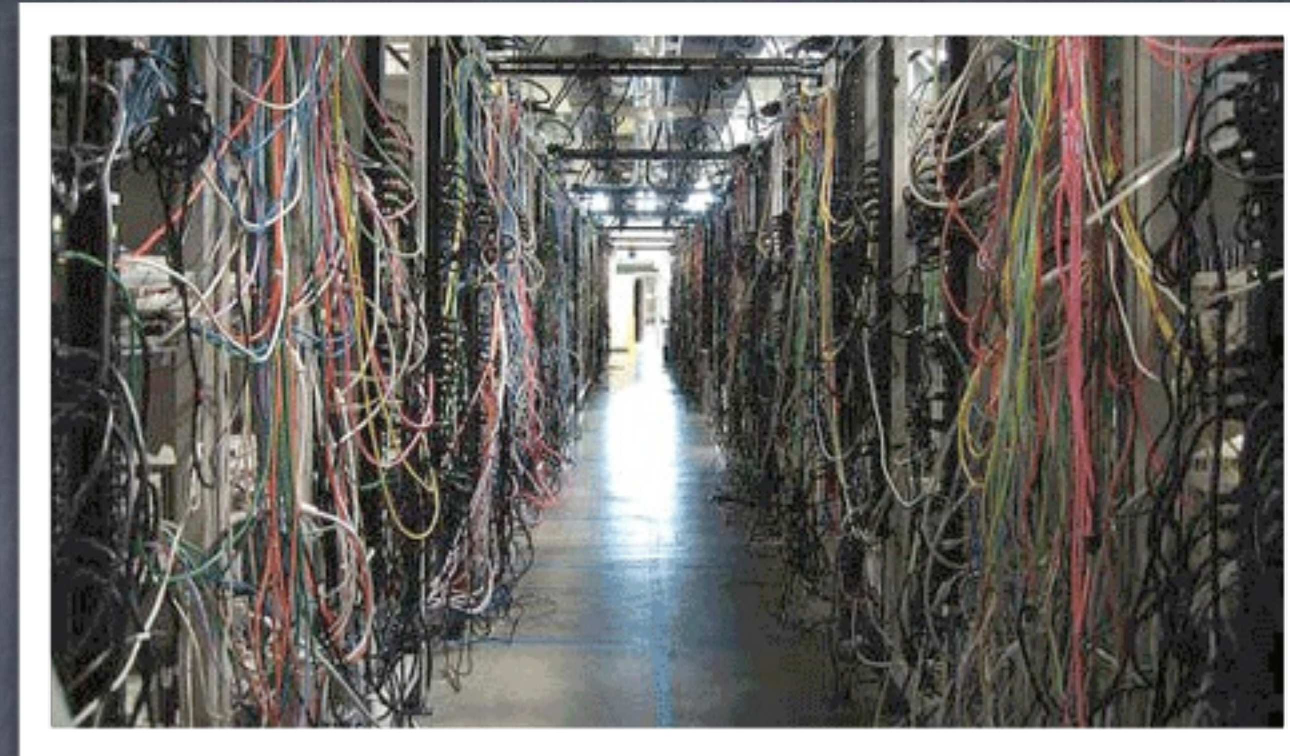
"Beware of bugs in the above code; I have only proved it correct, not tried it." —
Donald E. Knuth to Peter van Emde Boas (1977)

- Theoretical and empirical methods are two sides of the same medal
- Internal & external validity
- Idea & truth
- Elegance & relevance
- *Quality* & C0mpl3x17y

Raphaël - School of Athens



Kafkaesque



We study "software" - large and complex structures of computer instructions, written and read by man, executed by computers

"marked by a senseless, disorienting, often menacing complexity..." (Infoplease.com)

Size does matter

- A normal Dutch company may own 3×10^{10} lines of code - 750,000,000 single column pages.
- It goes a few times around the globe, if printed.
- At 1 minute per page (?) that might take approximately 1427 years to read.
- Ergo, nobody has ever understood it, or will ever fully understand it.

The source code of “ls”

3894 lines

367 ifs

174 cases

Research methods

Example: structured programming
theory: goto's are not needed
practice: goto's are harmful, sometimes
truth: ????

- Prototype an
- Study prog
- Measure s
- Time will te

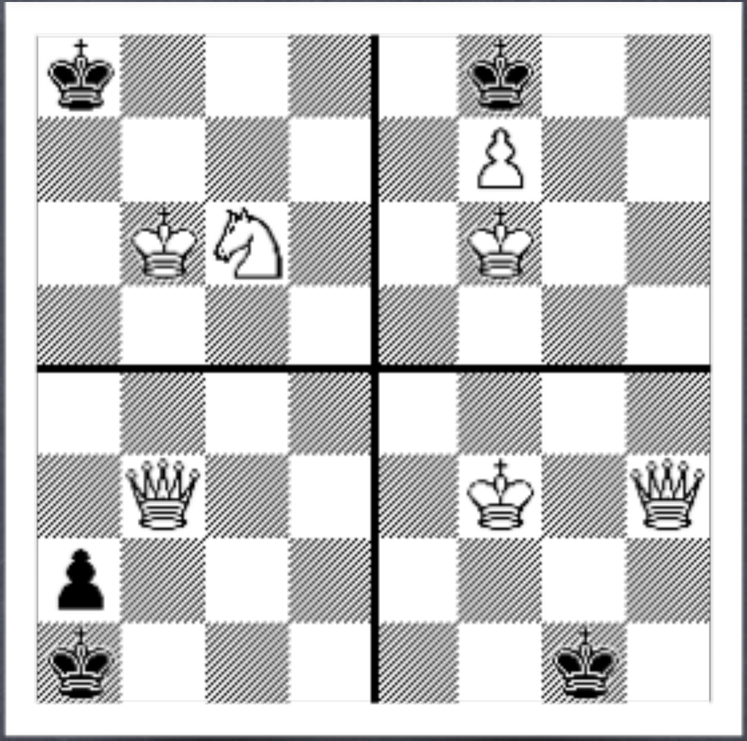
not convi toys

not convi muddy

not convi meaningless

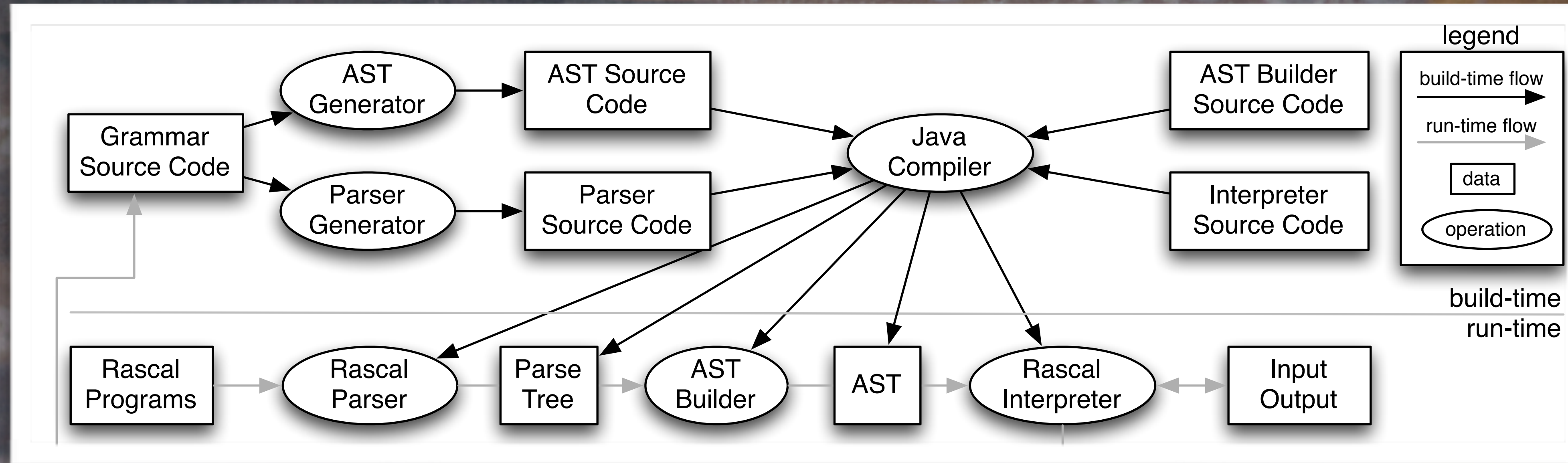
still annoying

Stalemate?



- We need to prove that our ideas work on a relevant scale, but precisely scale is what prevents us from proving anything.
- The challenges are:
  - volume
  - heterogeneity
  - **plurality of factors**

Case:



- Abstract syntax trees (ASTs)
- Operations on ASTs
- 400 concrete classes, 140 abstract classes
- AST classes are generated from a grammar
- Dispatch, dispatch, dispatch
- Evolution of the ± 100 kLOC java code

We compare design (patterns) to learn which is best in which situations

AST instance

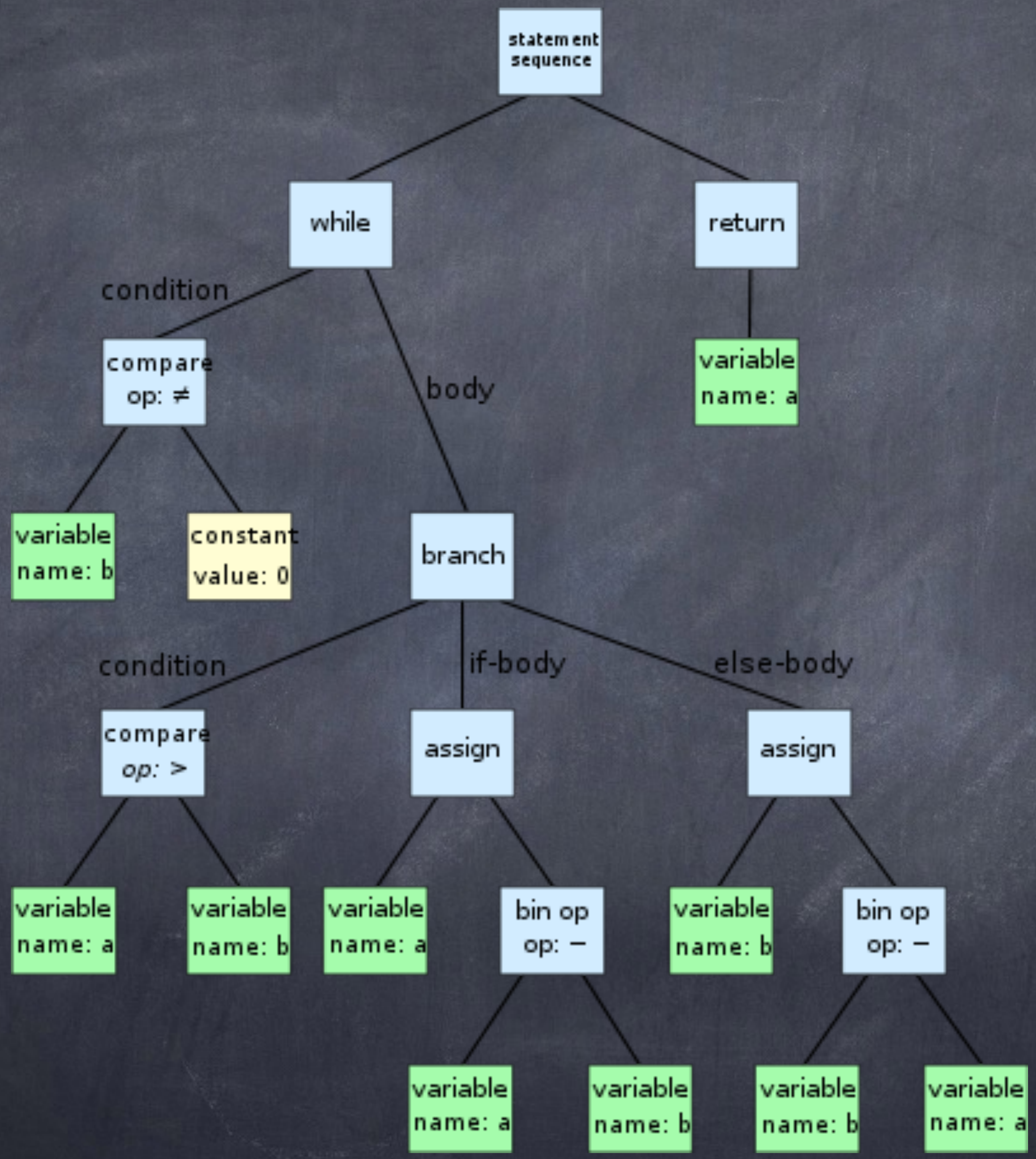


image from wikipedia.org

Composite Pattern

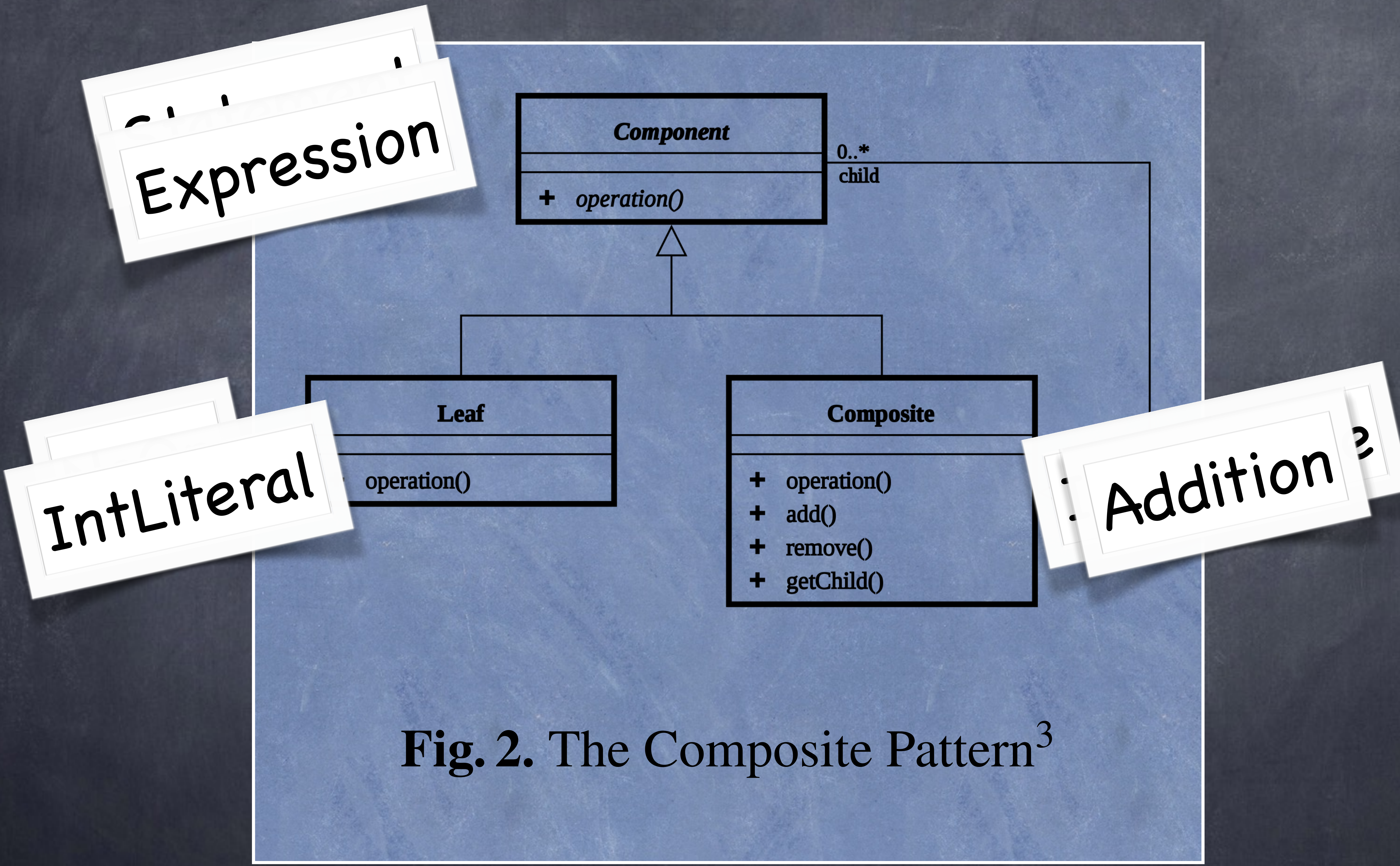


Fig. 2. The Composite Pattern[3]

image from wikipedia.org

Interpreter Pattern

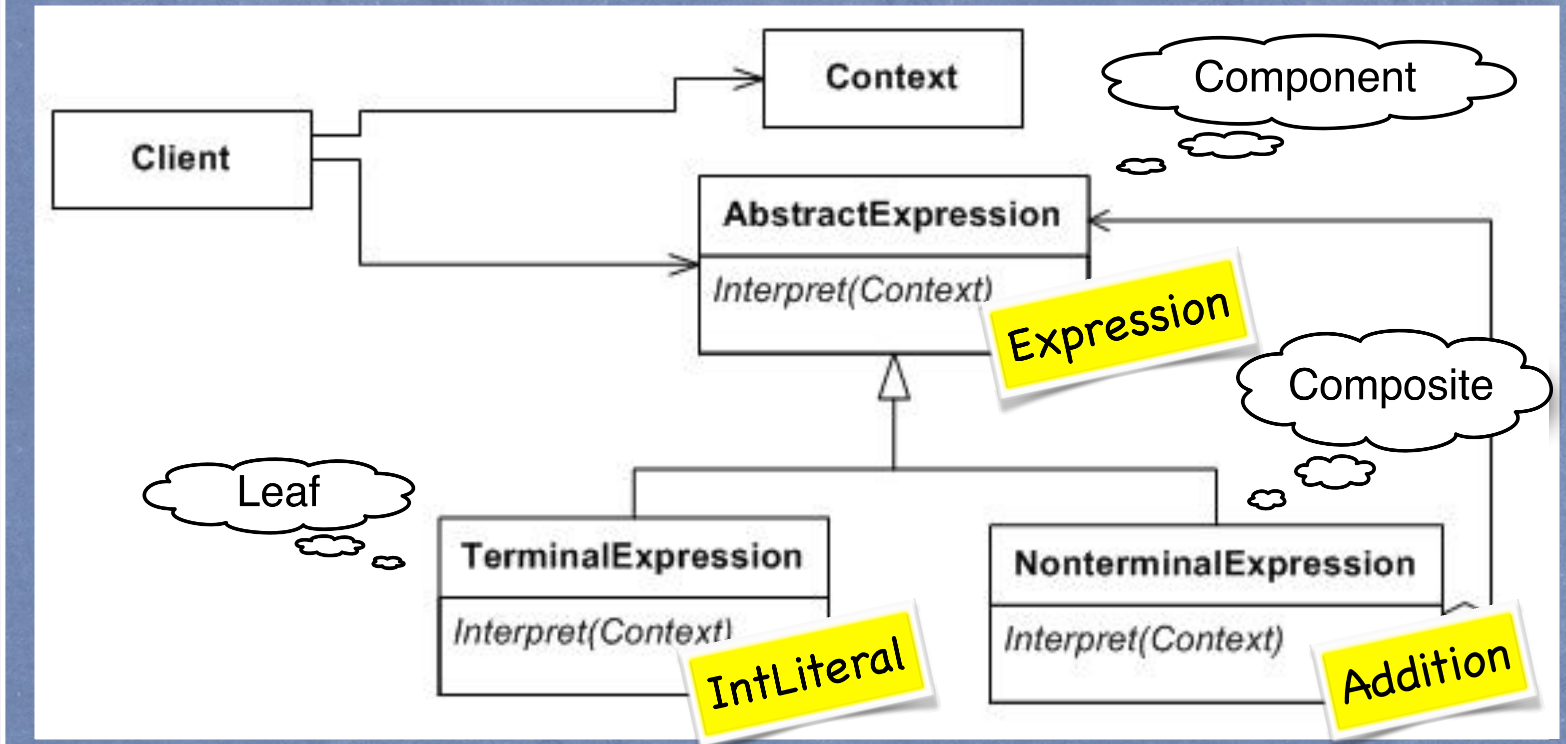


Fig. 4. The Interpreter Pattern with references to Composite (Figure 2).[7]

image from wikipedia.org

Visitor Pattern

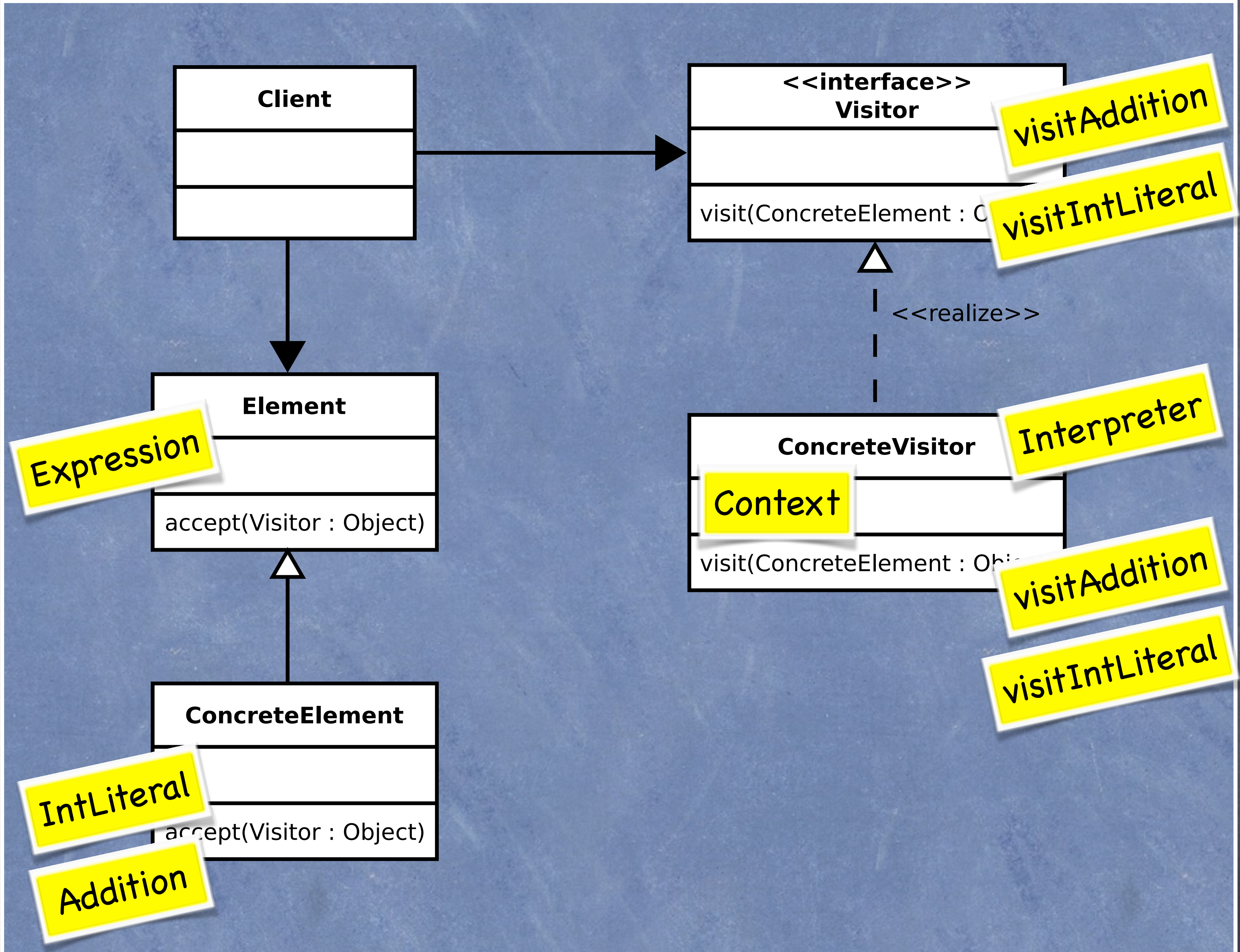


Fig. 3. The Visitor Pattern[4]

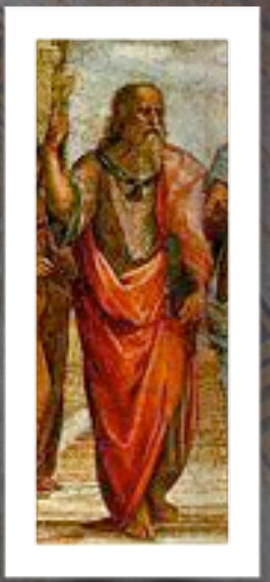
image from wikipedia.org

Visitor design pattern and the **Interpreter** design pattern are functionally inter-changeable

But, they are different in **non-functional** properties

And, these **emergent** properties tend to be difficult to predict

Theoretical Observations



- Visitor is conceptually more complex
  - Interpreter is only a small extension of composite
- Visitor encapsulates ...
  - Interp... encapsulates language constructs
- Visitor's d... dynamic indirection
  - Interpreter has less dynamic dispatch

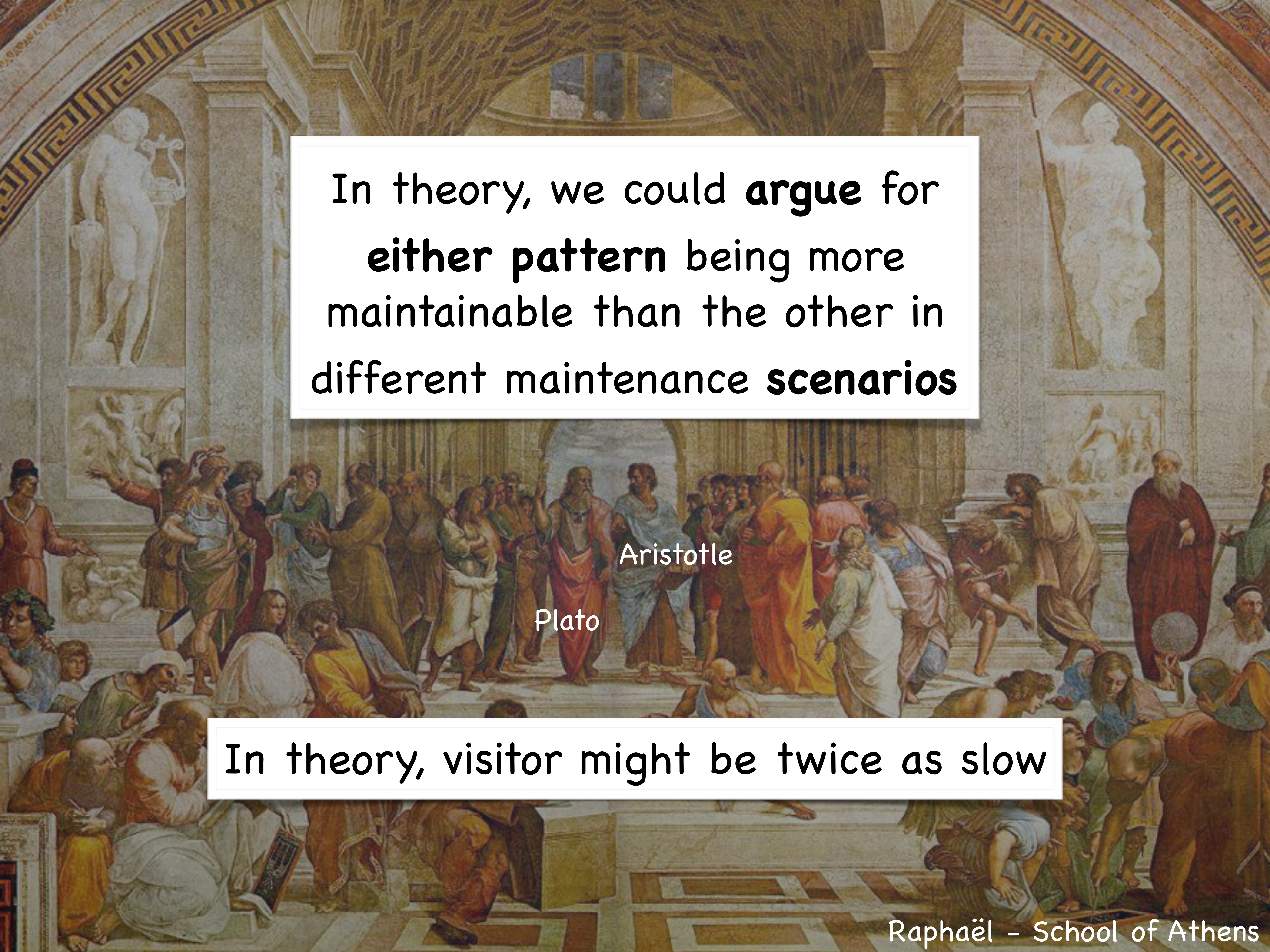
Harder to maintain, right?

Easy for adding algorithm, hard for adding new language construct, right?

Slower, right?

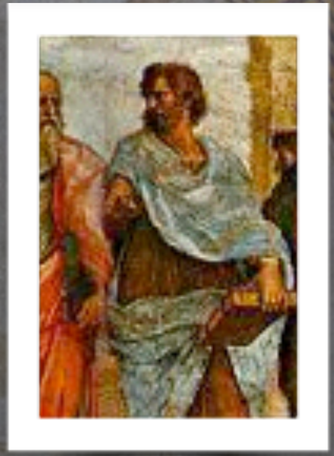
In theory, we could **argue** for **either pattern** being more maintainable than the other in different maintenance **scenarios**

In theory, visitor might be twice as slow



Raphaël - School of Athens

Empirical Observations



- Visitor-based interpreter is complex
  - Many visitors classes
  - Main interpreter is a “God class”
- Interpreter should run faster than this

Why this experiment?

- Is the difference between Interpreter and Visitor **causing** a part of these two problems, or not at all?

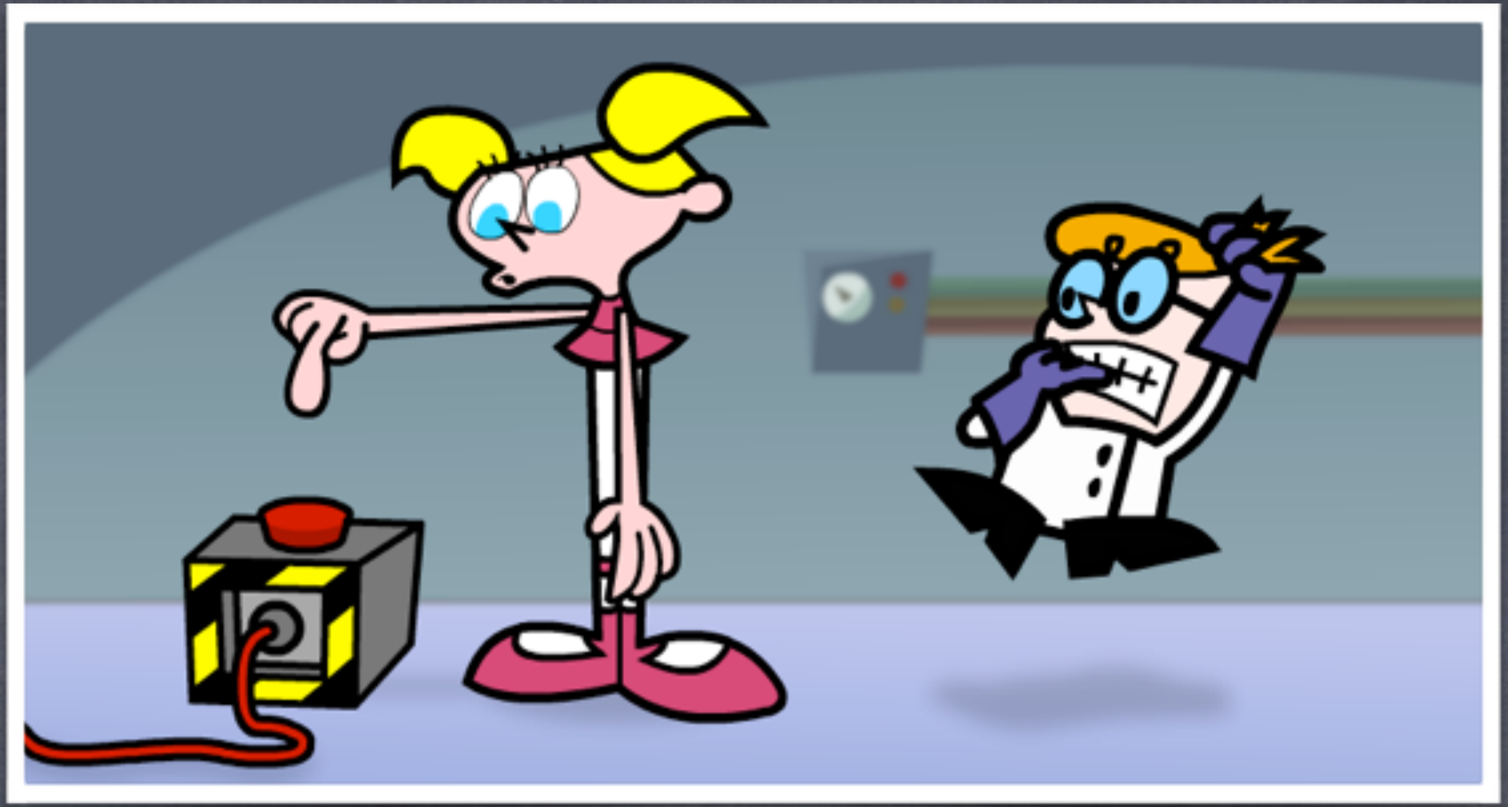
- How does one answer such a question?

Why this lab setup?

Observing software "in the wild"

- In reality, there exist **no two different versions** of the same interpreter
- In reality, there are **many other factors** influencing maintenance and efficiency other than this design choice
- Reality is perhaps easy to see, but it is **very hard to understand**

Lab Experiment



- In a lab we may **isolate** a factor
- In the lab we may **focus** on the effect
- In the lab we can observe **causality** more directly

Possible lab experiments

- Source code metrics for maintainability
- Construction of Cognitive Models
- New method based on "Evolution complexity"

Source Code Metrics are (perhaps) good for observing reality **statistically**, but not for observing implications of design choices

Maintainability Index I&II

Maintenance Complexity Metric

SIG maintainability model

Computing and aggregating metrics values, **independent** of maintenance scenario, predicting long-term expectations on maintenance costs

If validated and calibrated these make sense on huge long-lived systems, but they say **nothing about the next maintenance scenario** applied to the system

What about using **Cognitive Models** of understanding the source code then?

Programmer

Scenario

Source

abstract

Cognitive Model

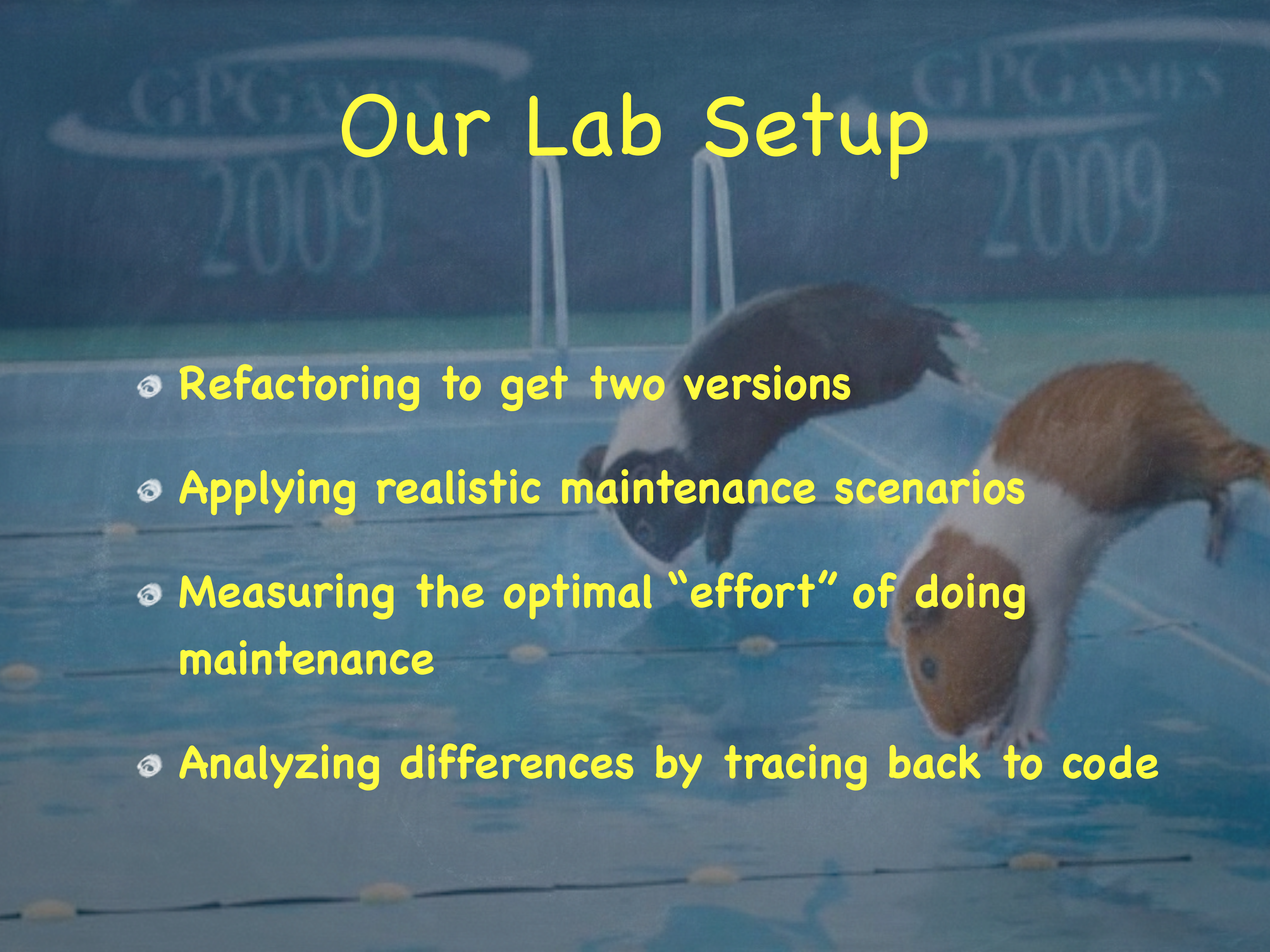
measure

Analyze

conclude

Unfortunately, we neither understand nor trust these models

IDE + source code + human => very complex models of cognition



Our Lab Setup

- Refactoring to get two versions
- Applying realistic maintenance scenarios
- Measuring the optimal "effort" of doing maintenance
- Analyzing differences by tracing back to code

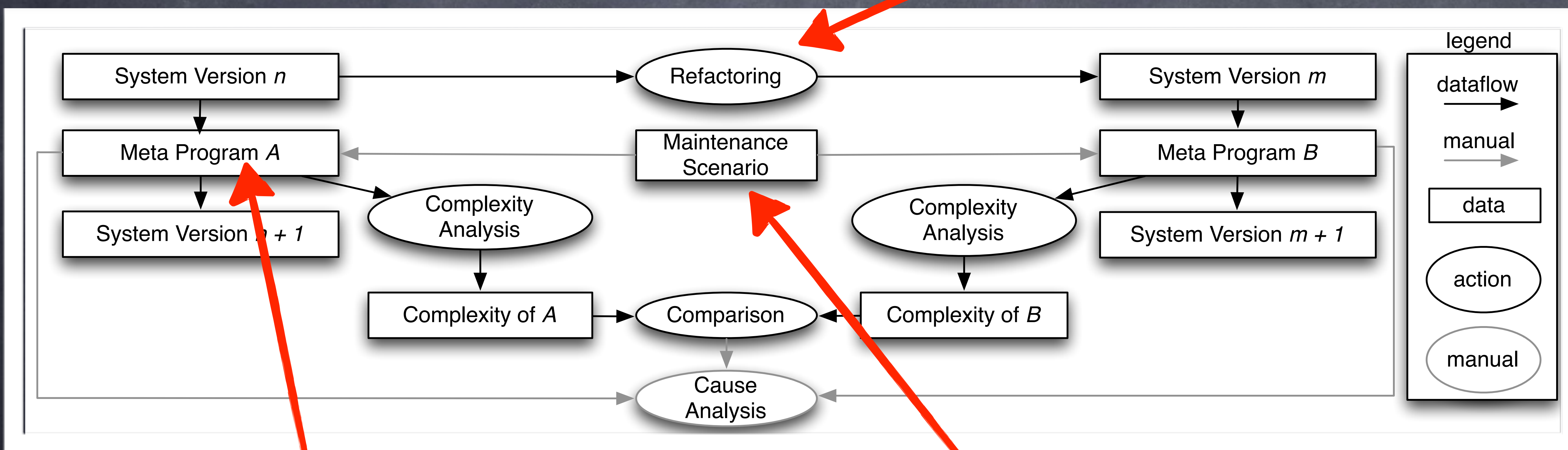
A "refactoring" is an automated source-to-source program transformation that **guarantees** run-time **semantics** to be preserved.

The application of a refactorings is intended to improve quality of source code without too much manual labor.

Refactorings are a way to mitigate complexity

Isolating the variable

Key enabler

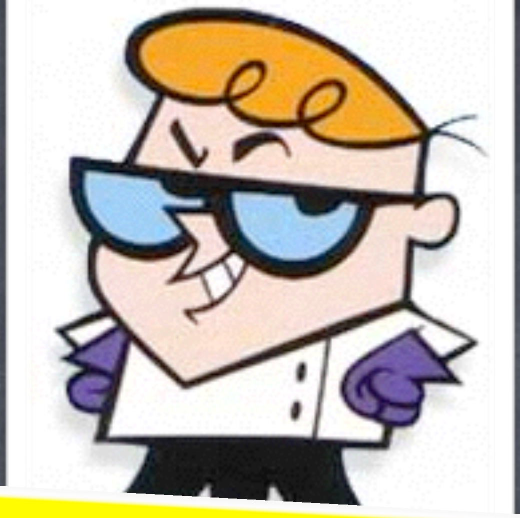


Traceability

Manual labor

Rascal & JDT to implement Visitor to Interpreter **refactoring**

"Complexity of Maintenance"

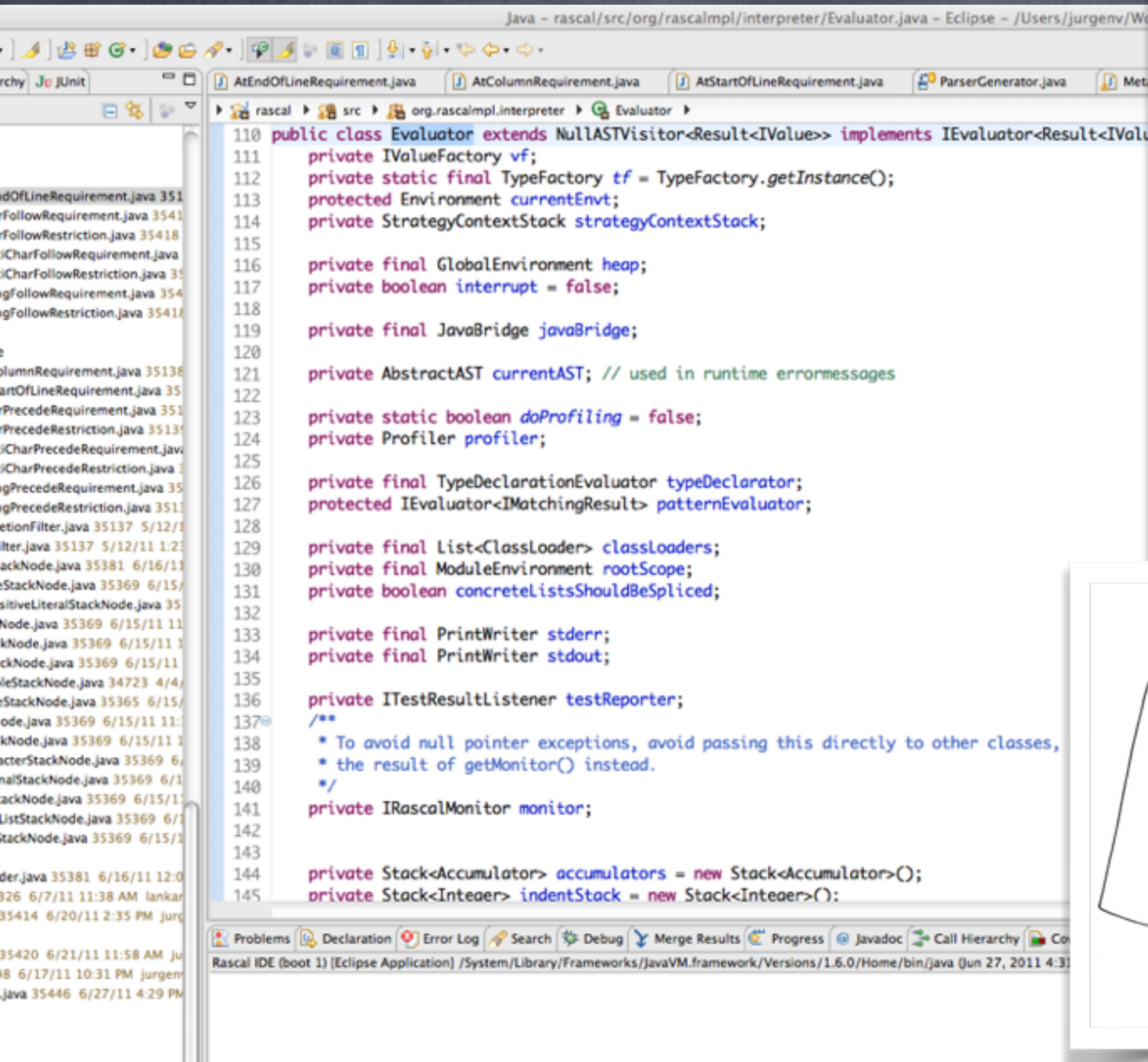


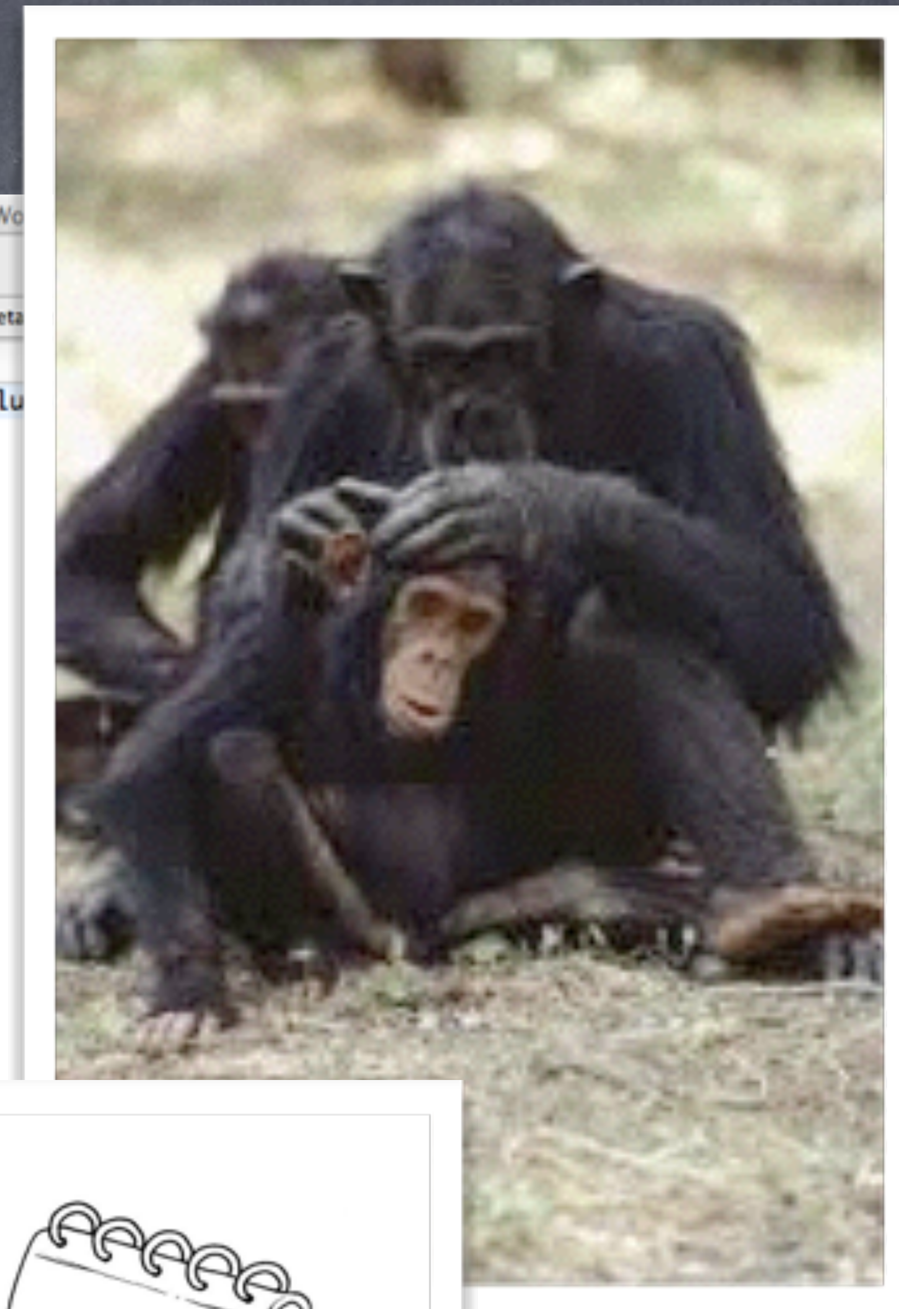
Precise definitions in [TOOLS2011]

- Maintainability = Understandability + Modifiability
- Complexity of a maintenance scenario is =
  - #steps to learn facts about a Program +
  - #steps to modify the Program
- Reify steps as a "Meta Program" that operates the IDE

Inspired by "Measuring Software Flexibility" by Mens & Eden, IEE Software 2006

Collecting data





Results

S	**Visitor**	(Com)	**Interpreter**	(Com)	**Vis.>Int.**
S1	$ci^{11}(g^2a)^2)$	(18)	$m^2b(ef^2)^3(ga)^2$	(16)	yes
S1(N)	$ci^{11}(g^Na)^2)$	$(14+2N)$	$m^Nb(ef^N)^3(ga)^N$	$(4+6N)$	if $N \leq 2$
S1'(N,2)	$ci^{11}(g^Na)^2)$	$(14+2N)$	$m^N(ga)^N$	$(3N)$	if $N \leq 14$
S1'(N,M)	$ci^{9+M}(g^Na)^M)$	$(10+NM+2M)$	$m^N(ga)^{MN}$	$(N+2MN)$	if $N \leq \frac{2M+10}{M+1}$
S2	i^2g^3iga	(8)	$i^2g^3gaig^3aiga$	(14)	no
S3	$dg^5egcg^{15}g^2a(eea)^4i^2h(ga)^3$	(43)	$d(ig)^2a(iga)^{15}(ig)^3gai$ $(ig^2)a(igg)^2anigaih(ga)^3$	(83)	no
S3'	$d(ga)^5egac(ga)^{15}(ga)^2$ $(eea)^4i^2h(ga)^3$	(70)	$d(ig)^2a(iga)^{15}(ig)^3gai$ $(ig^2)a(igg)^2anigaih(ga)^3$	(83)	no
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]36)	no
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]3)	yes

...n c... ...ble 1)

steps to add N constructs to **Visitor**
14 + 2N

steps to add N constructs to **Interpreter**
3N

break-even at N = 14

Why trust this?

- **Construct** validity: are all aspects of maintainability observable in this experiment?

> other factors may still dominate, but that is why we compare two equivalent systems

- **Internal** validity: did you really do … job possible in all scenarios?

> there is no proof of that - we invite you to reproduce or invalidate the results

- **External** validity: does this say anything about the next interpreter I write in Java? The next maintenance? What if I d… use Eclipse? What if <blablabla>?

> we do **not** know

Summary of case

- We used **Rascal** to build a **refactoring** tool
- to **isolate** the difference between **Visitor** & **Interpreter**
- and using the "**Complexity of Maintenance**" method
- we found that **Visitor is better***

*given the scope of the experiment

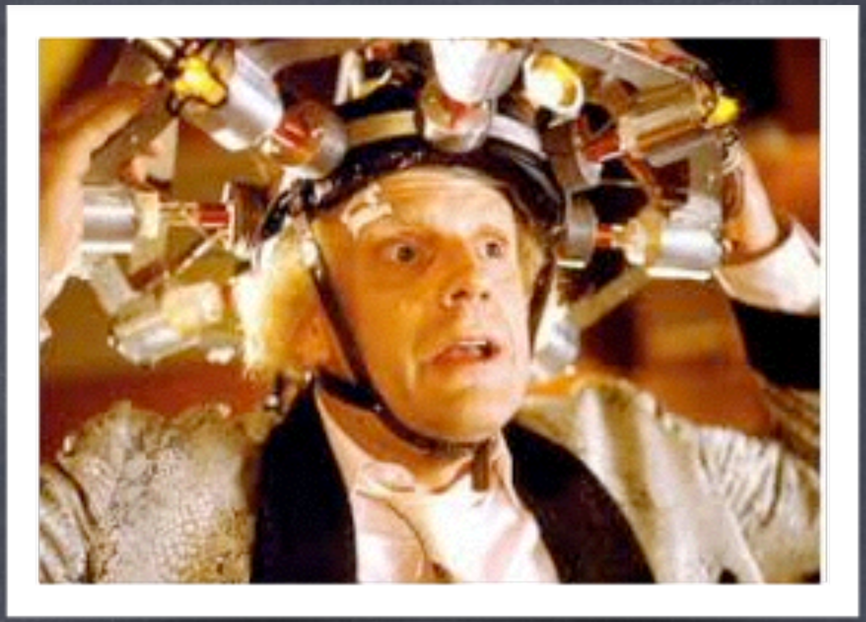
From threats to questions

- **Theoretical**: how to prove semantics preservation for these types of transformations for real programming languages?
- **Empirical**: how to validate that our maintainability complexity measure makes sense?

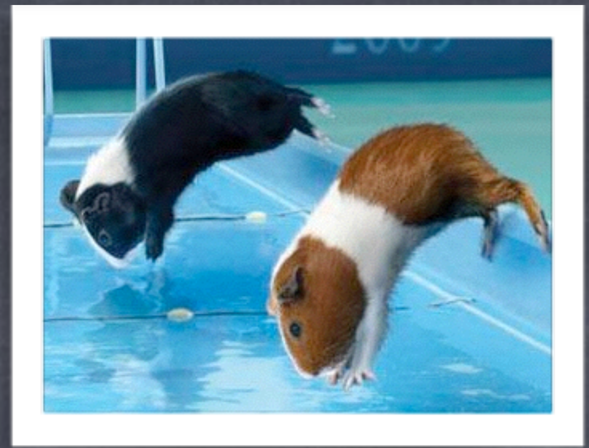
Semantics preserving

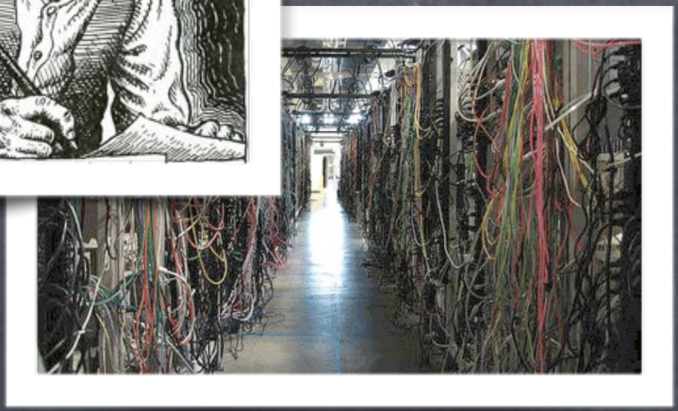
- Problems:
  - Programming languages are ridiculously complex
  - There are ridiculously many languages
- Possible answers:
  - Abstract semantics [Veerman (CFG), Vu (PGA)]
  - Formal specification of refactorings [Tip, DeMoor]

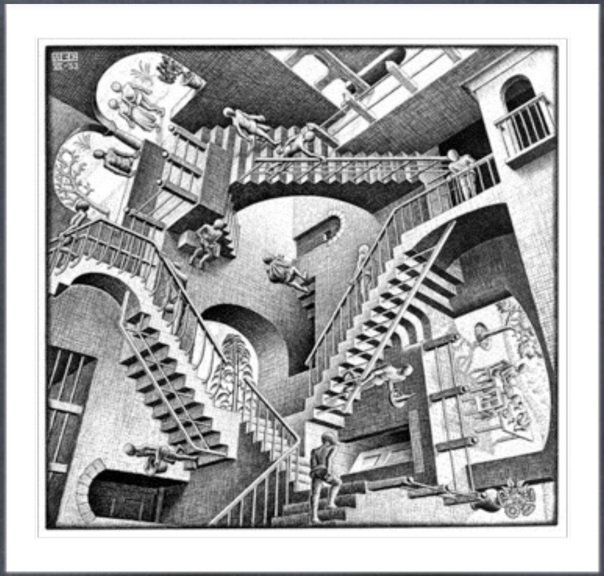
The future

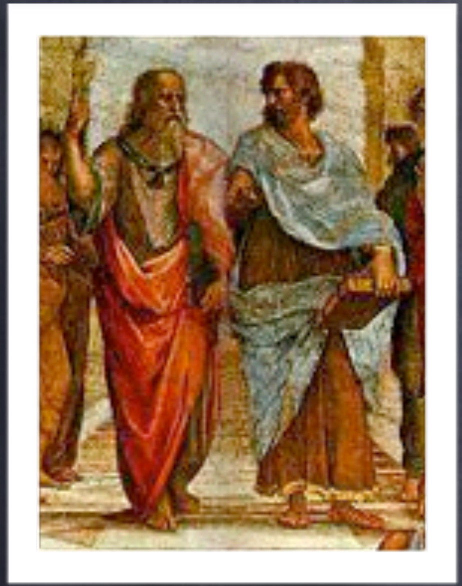


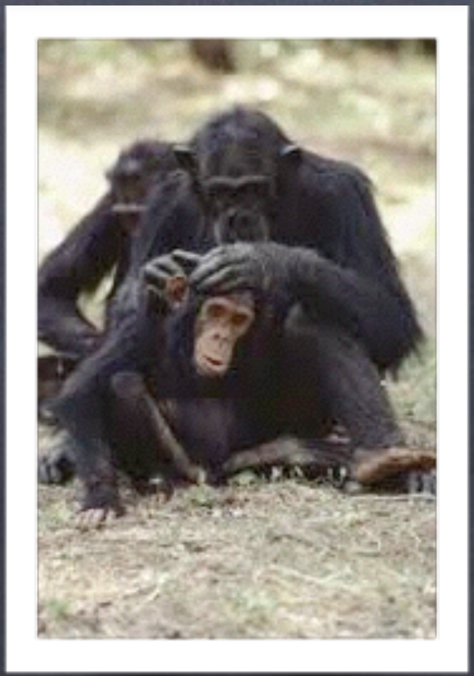
- Do many more of such “isolation” experiments
  - Study theory of refactoring
  - Prototype relevant (lab) tools
  - Find out what matters in software engineering
- Cases: exceptions, parallelism, dynamic dispatch, immutability, … ad infinitum



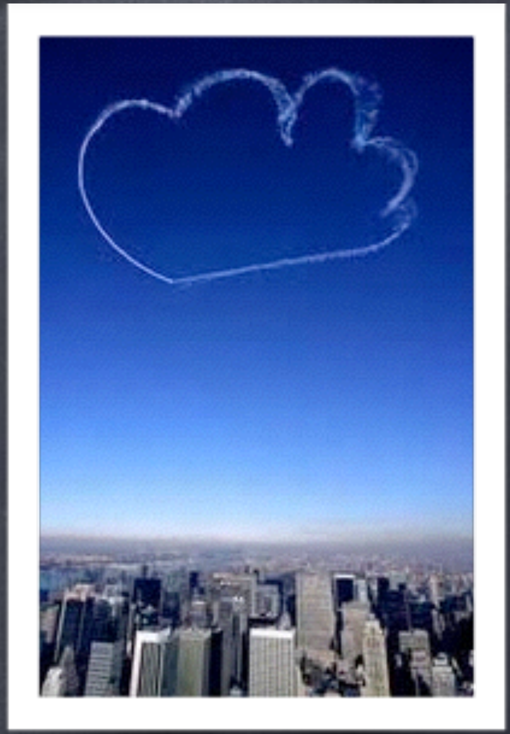


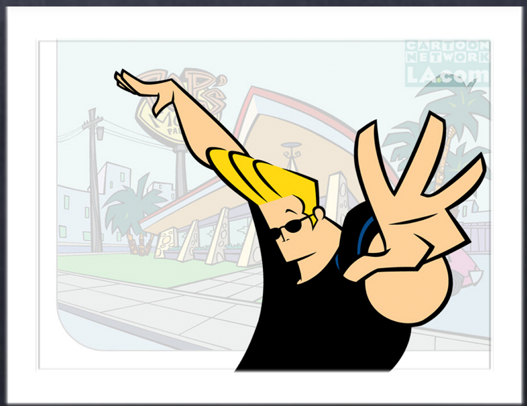






Questions?





UNDER CONSTRUCTION

http://www.rascal-mpl.org