A case of
Visitor versus Interpreter Pattern

Paul Klint, Mark Hills, Tijs van der Storm,
Jurgen Vinju

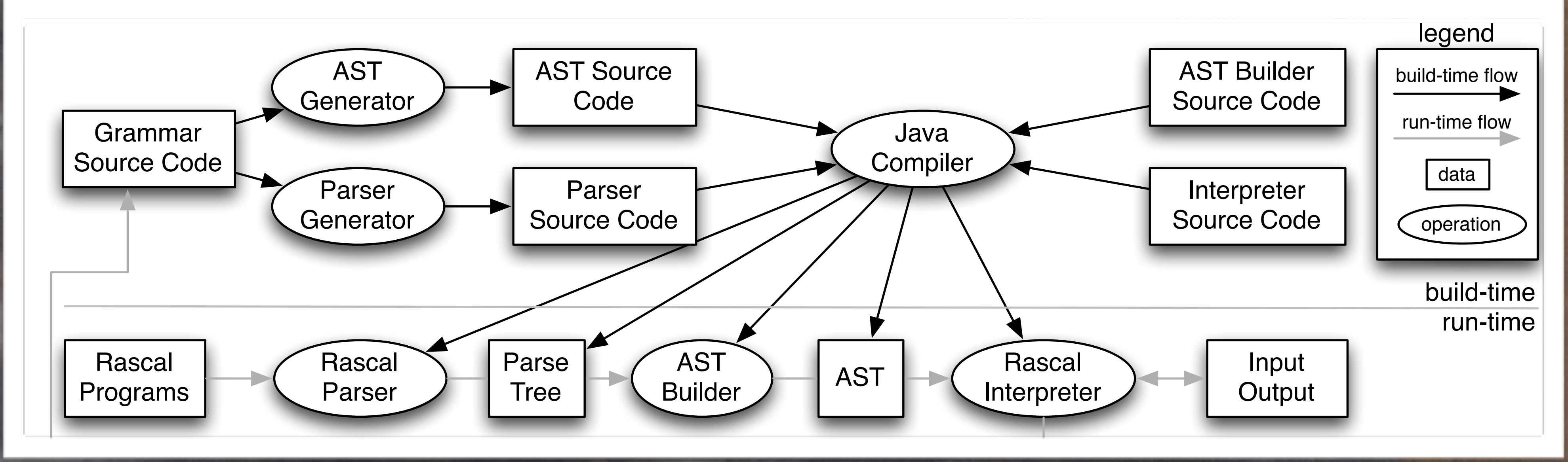
Zürich, June 30th 2011

Why?

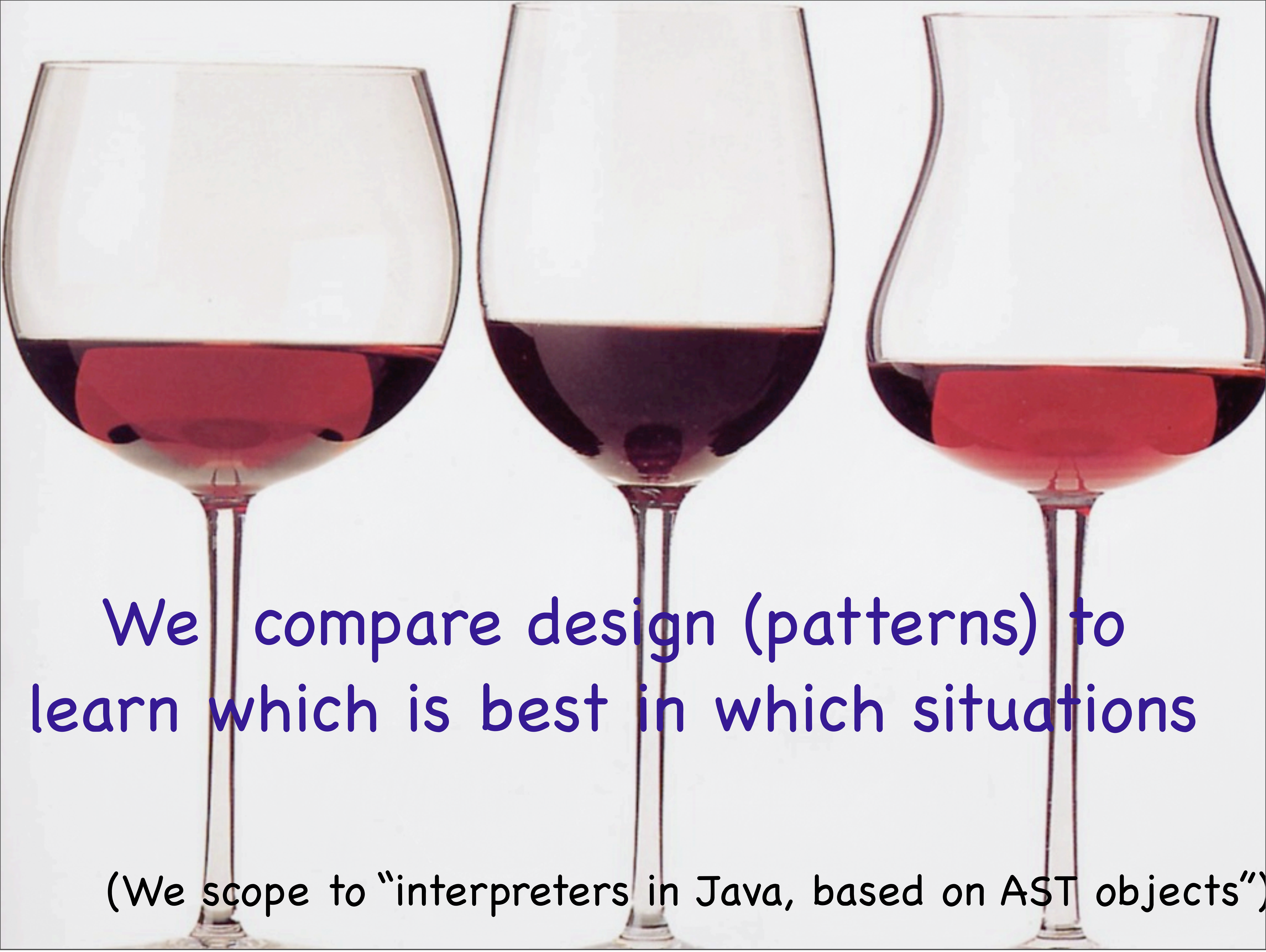
- Why this experiment?
- Why this "laboratory" setup?
- Why trust the conclusions?

"Long Live Incremental Research!"

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

(We scope to “interpreters in Java, based on AST objects”)

Composite Pattern

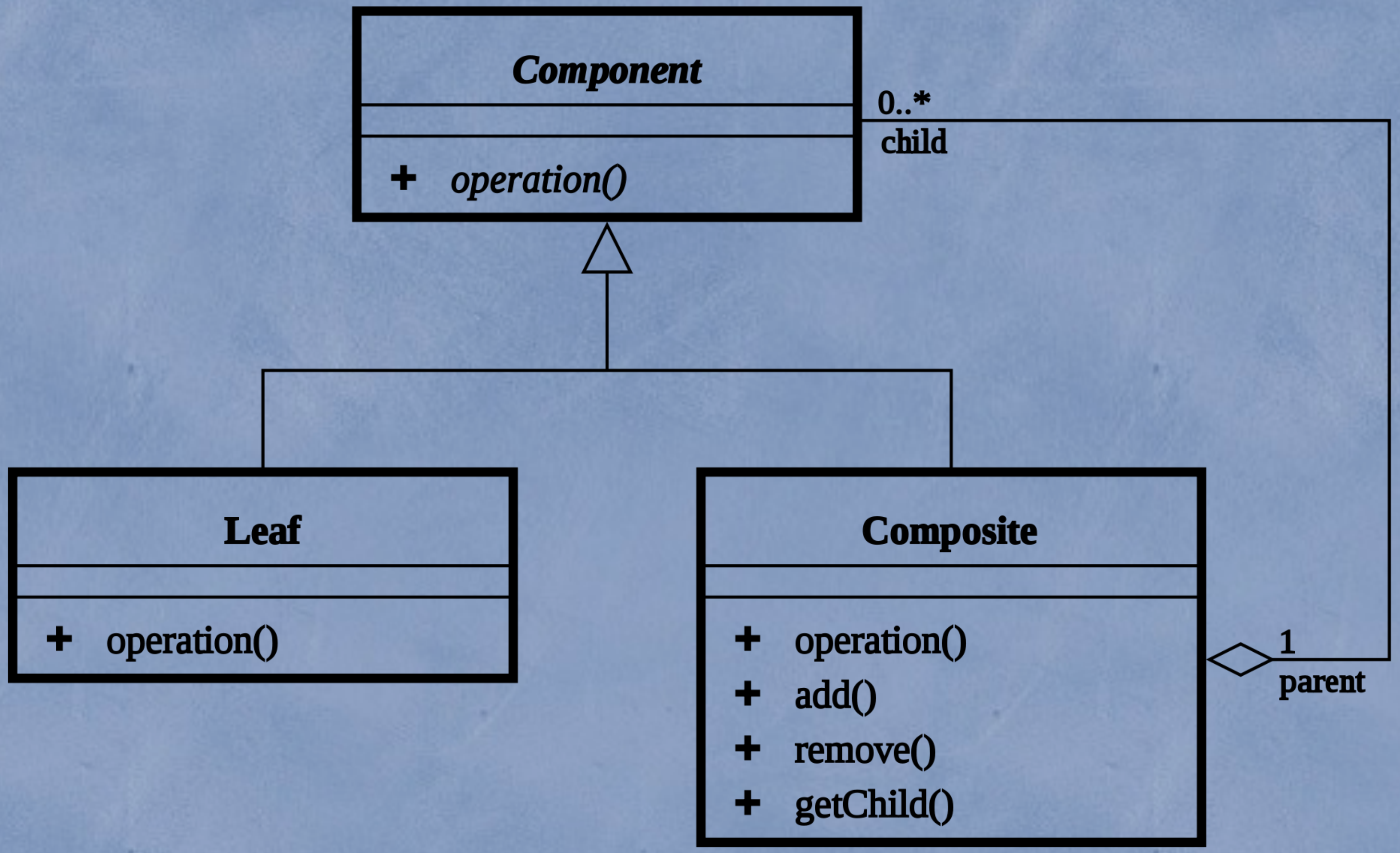


Fig. 2. The Composite Pattern[3]

image from wikipedia.org

Composite Pattern

Statement

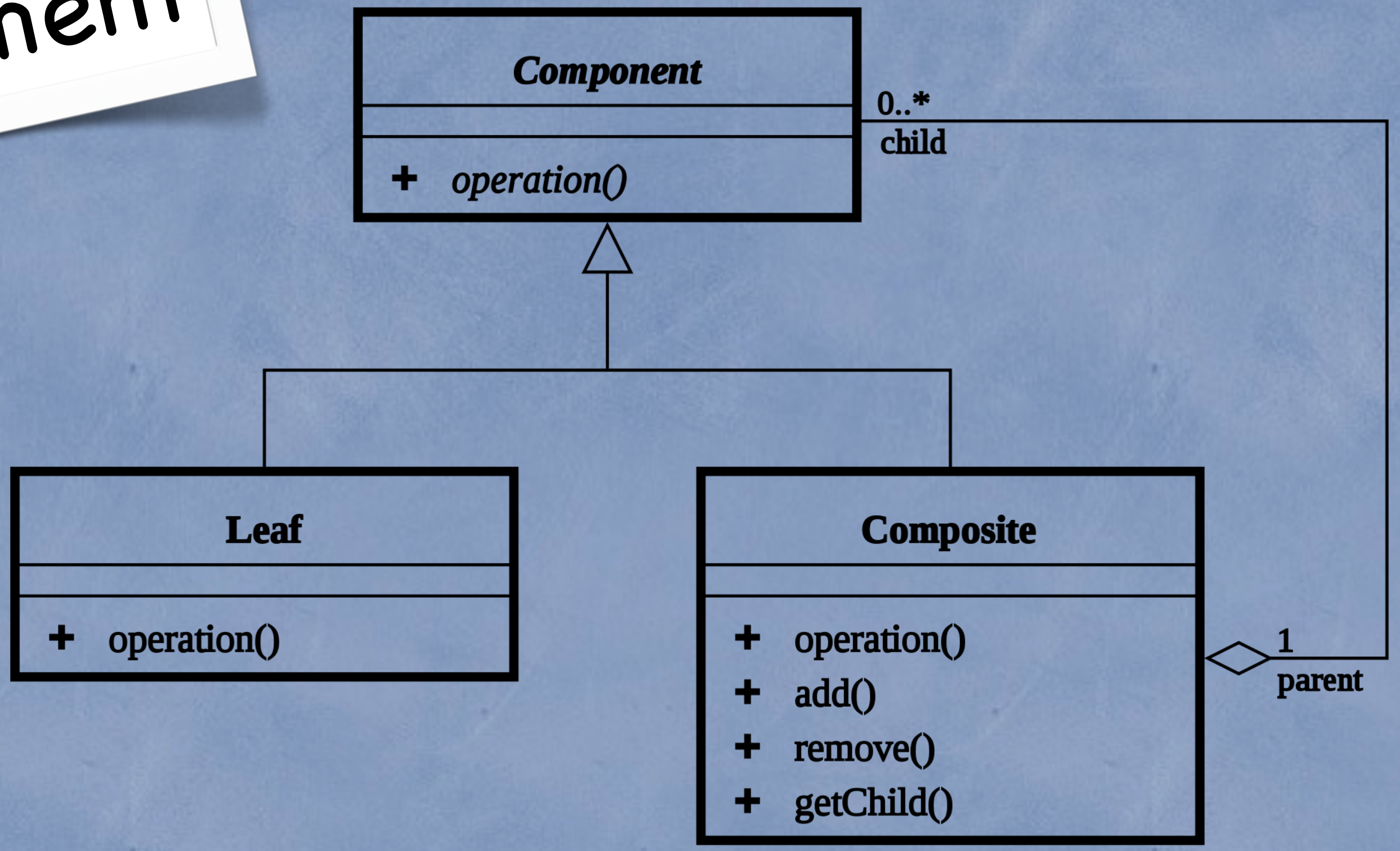


Fig. 2. The Composite Pattern[3]

image from wikipedia.org

Composite Pattern

Statement

NoOp

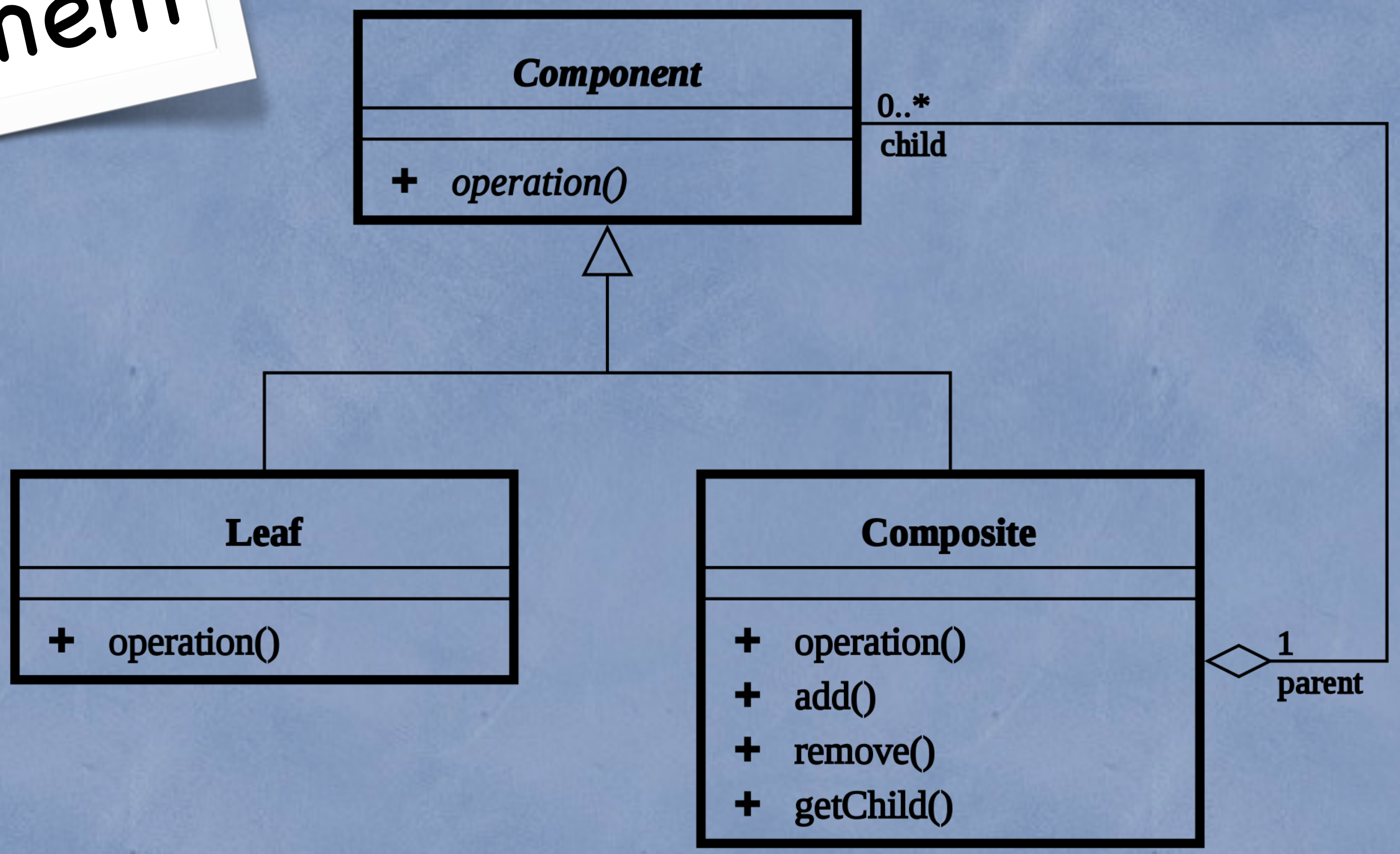


Fig. 2. The Composite Pattern[3]

image from wikipedia.org

Composite Pattern

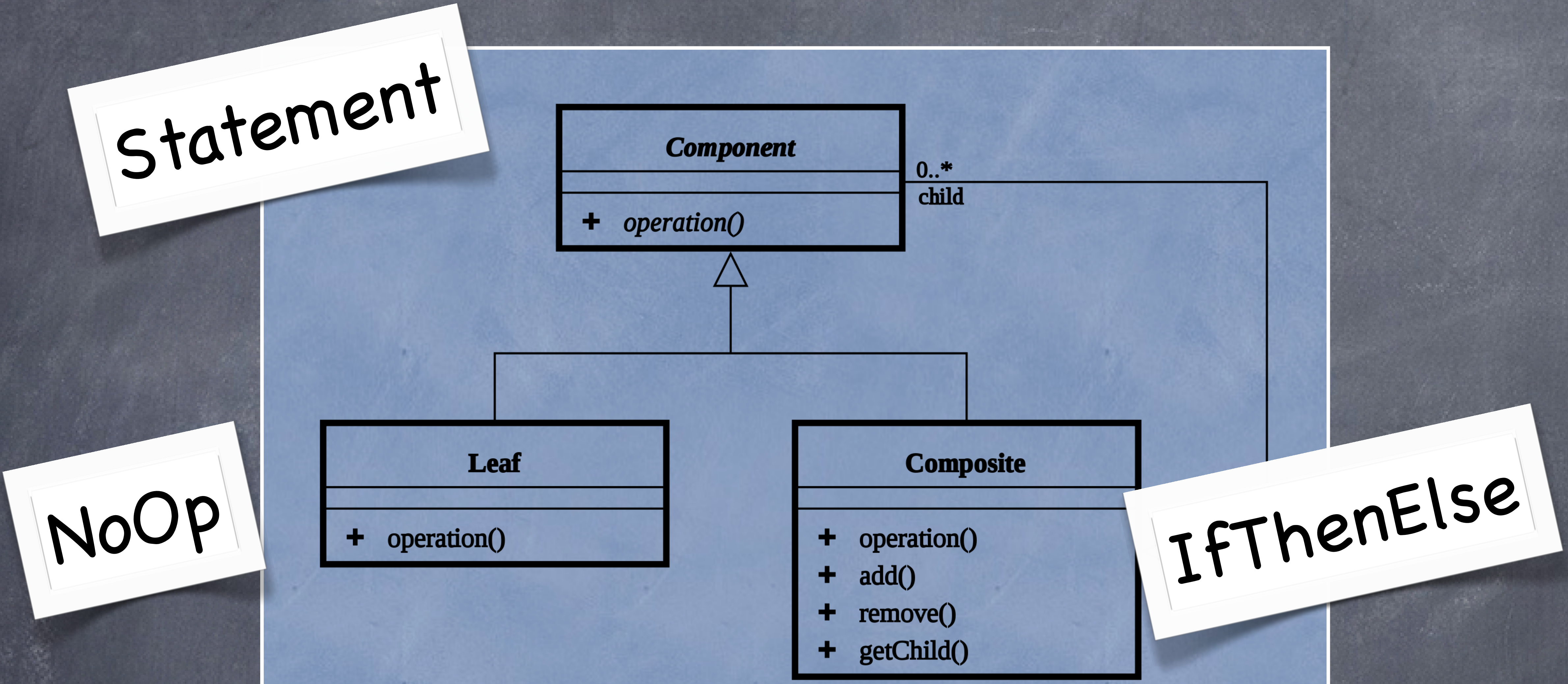


Fig. 2. The Composite Pattern[3]

image from wikipedia.org

Composite Pattern

Statement

Expression

NoOp

IfThenElse

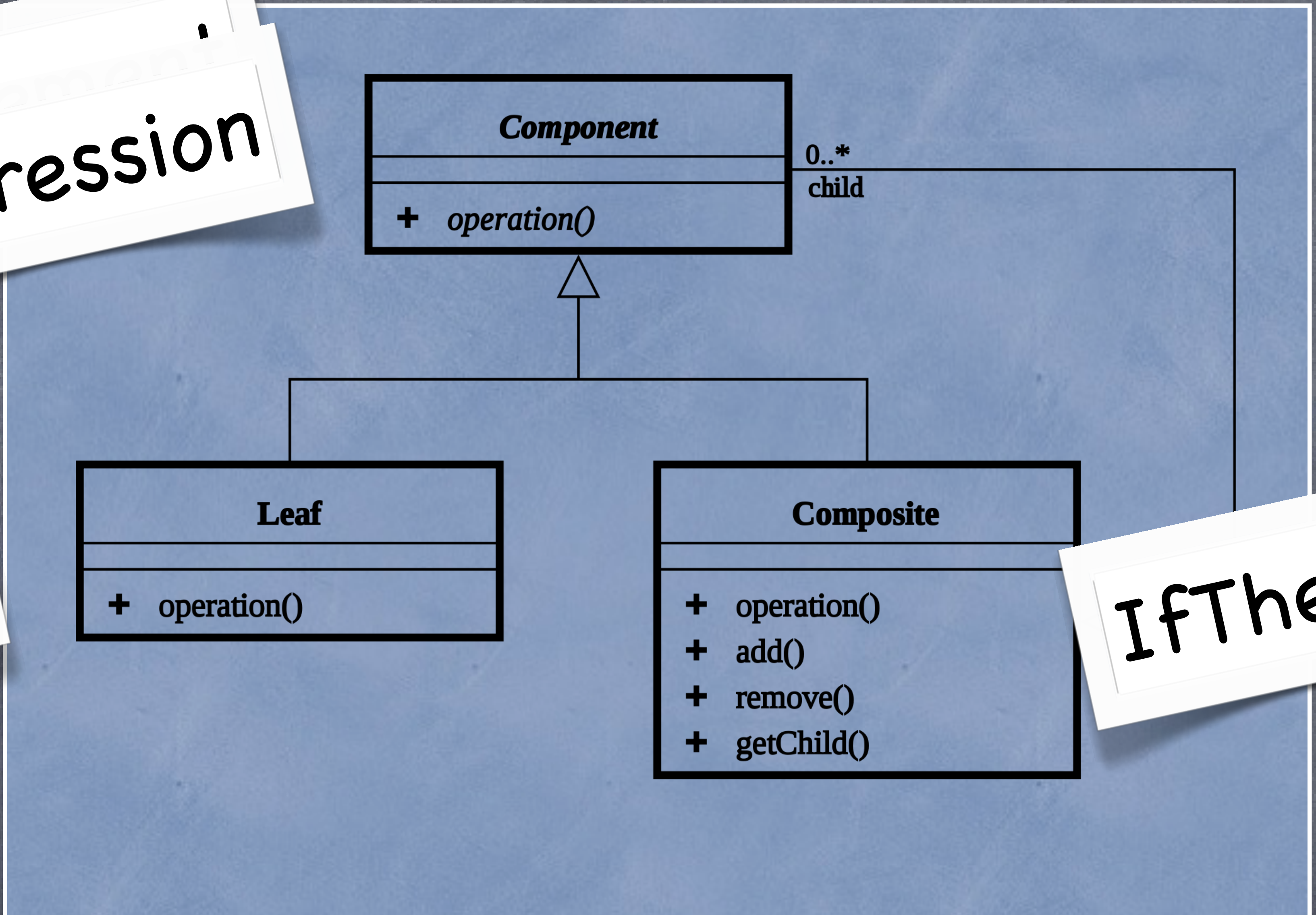


Fig. 2. The Composite Pattern[3]

image from wikipedia.org

Composite Pattern

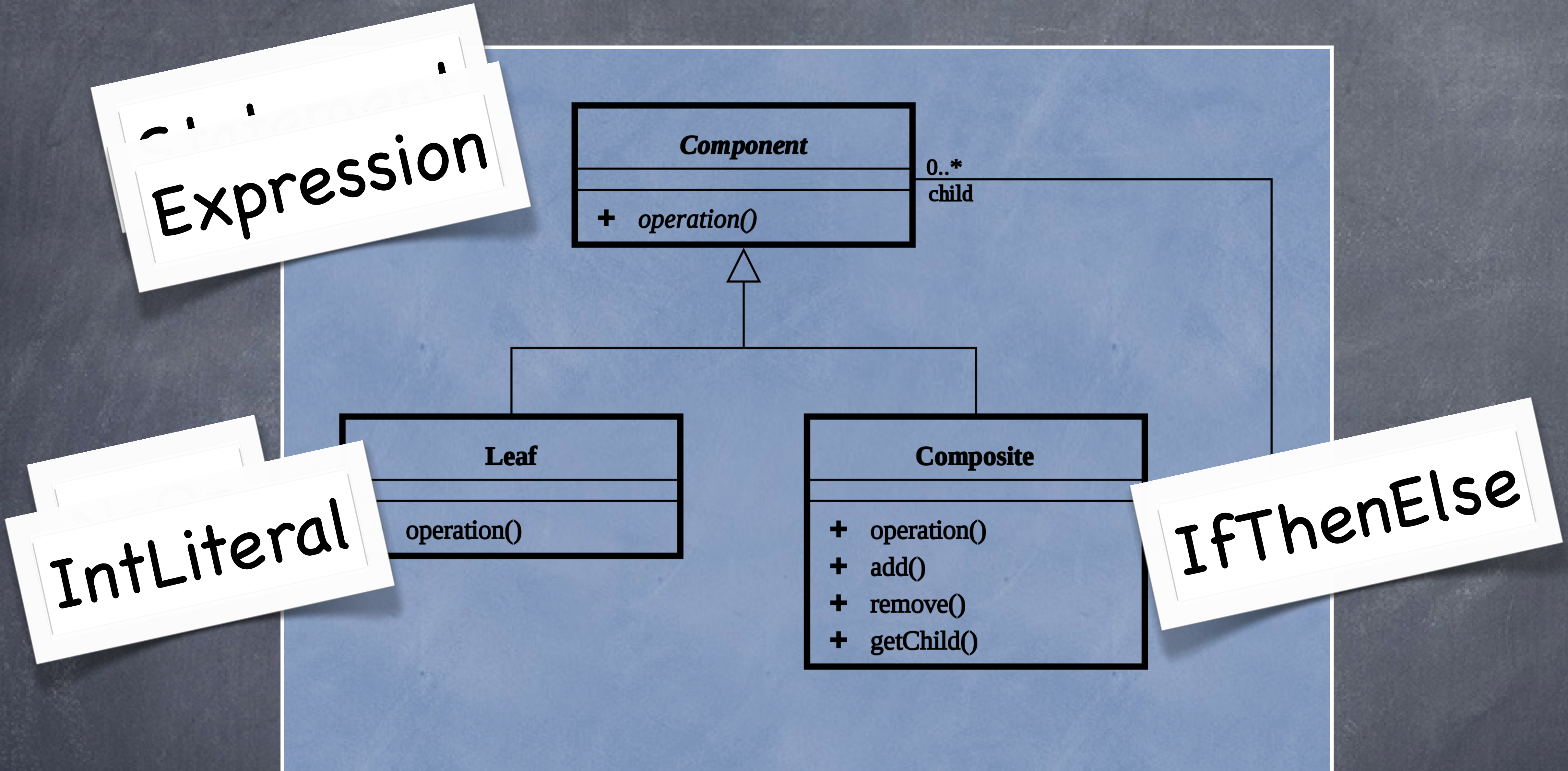


Fig. 2. The Composite Pattern[3]

image from wikipedia.org

Composite Pattern

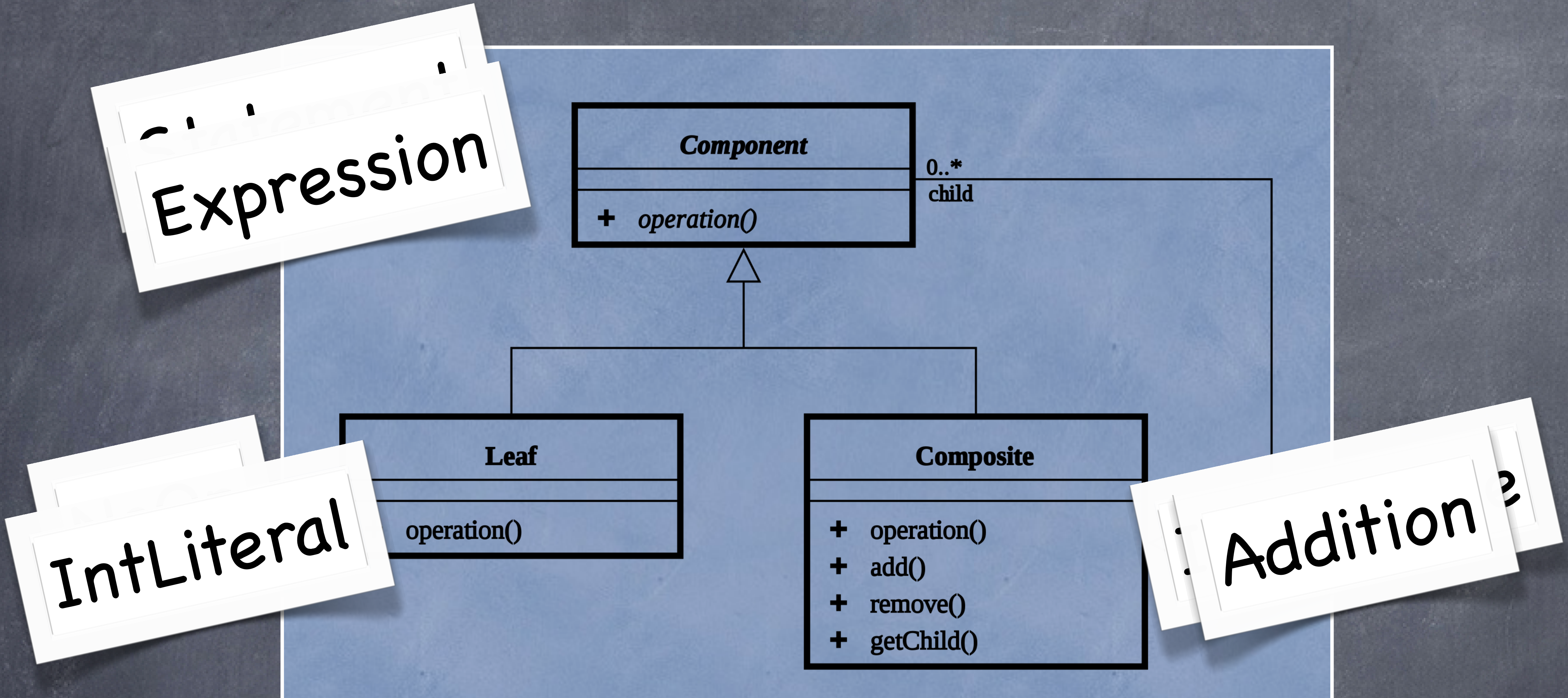


Fig. 2. The Composite Pattern[3]

image from wikipedia.org

AST instance

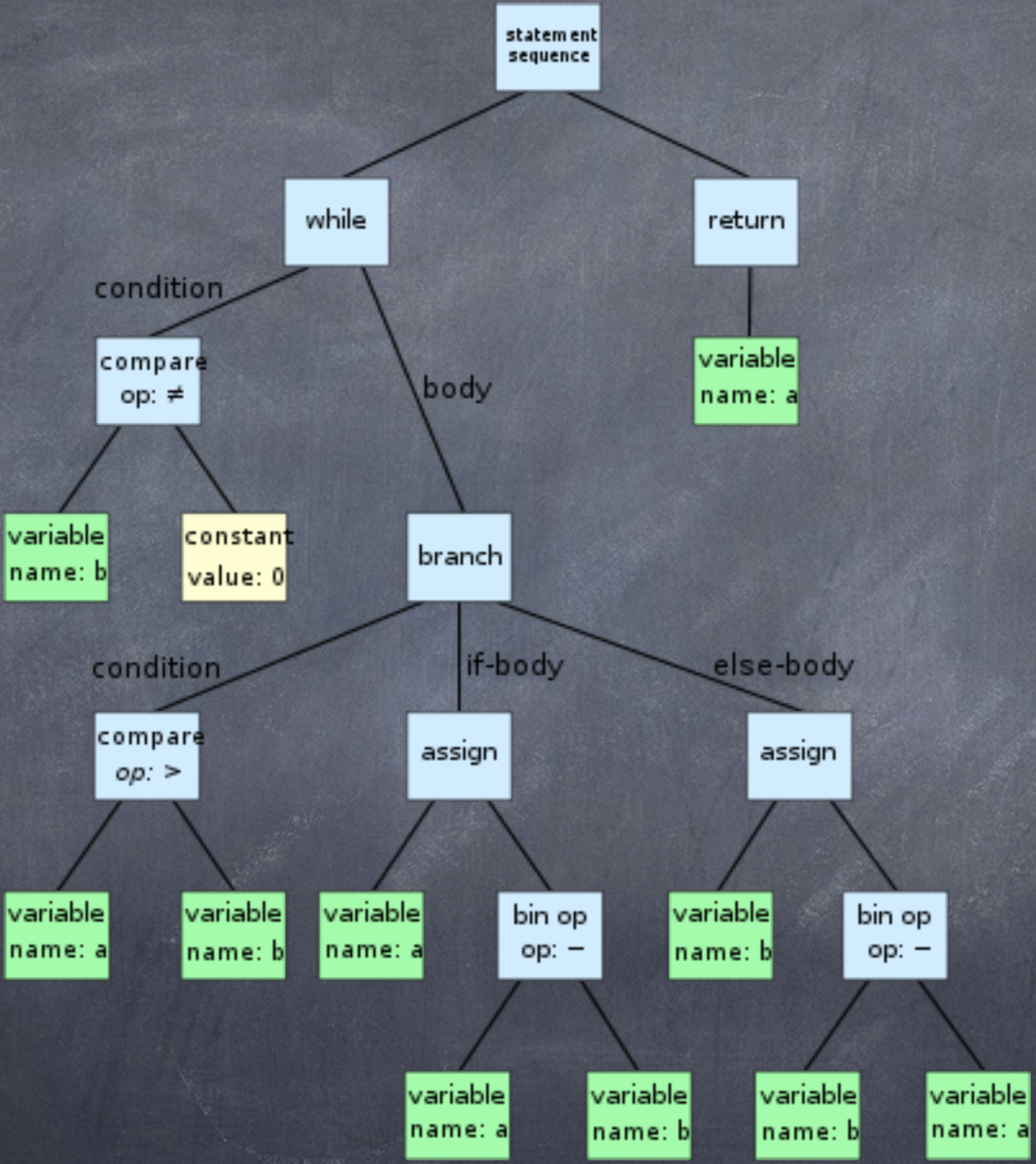


image from wikipedia.org

Interpreter Pattern

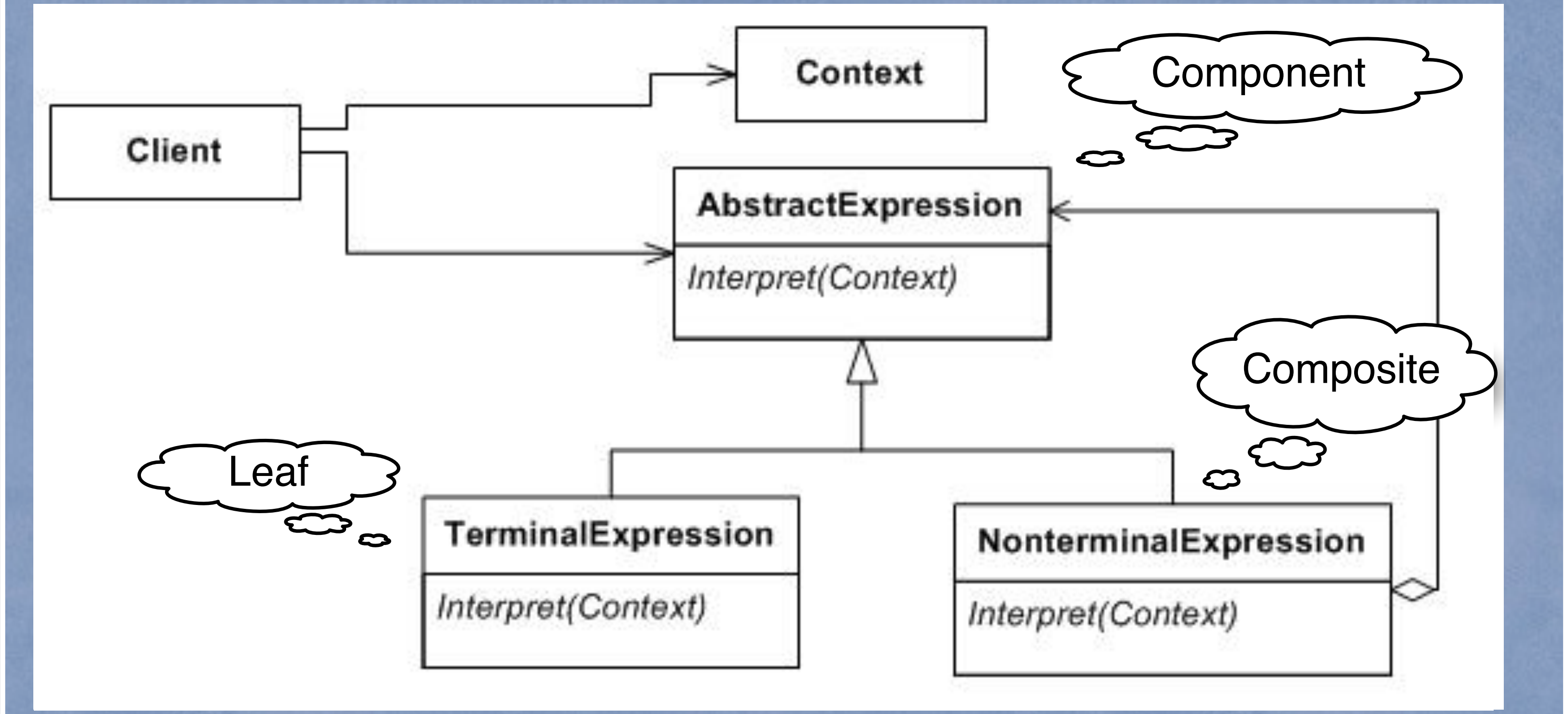


Fig. 4. The Interpreter Pattern with references to Composite (Figure 2).[7]

image from wikipedia.org

Interpreter Pattern

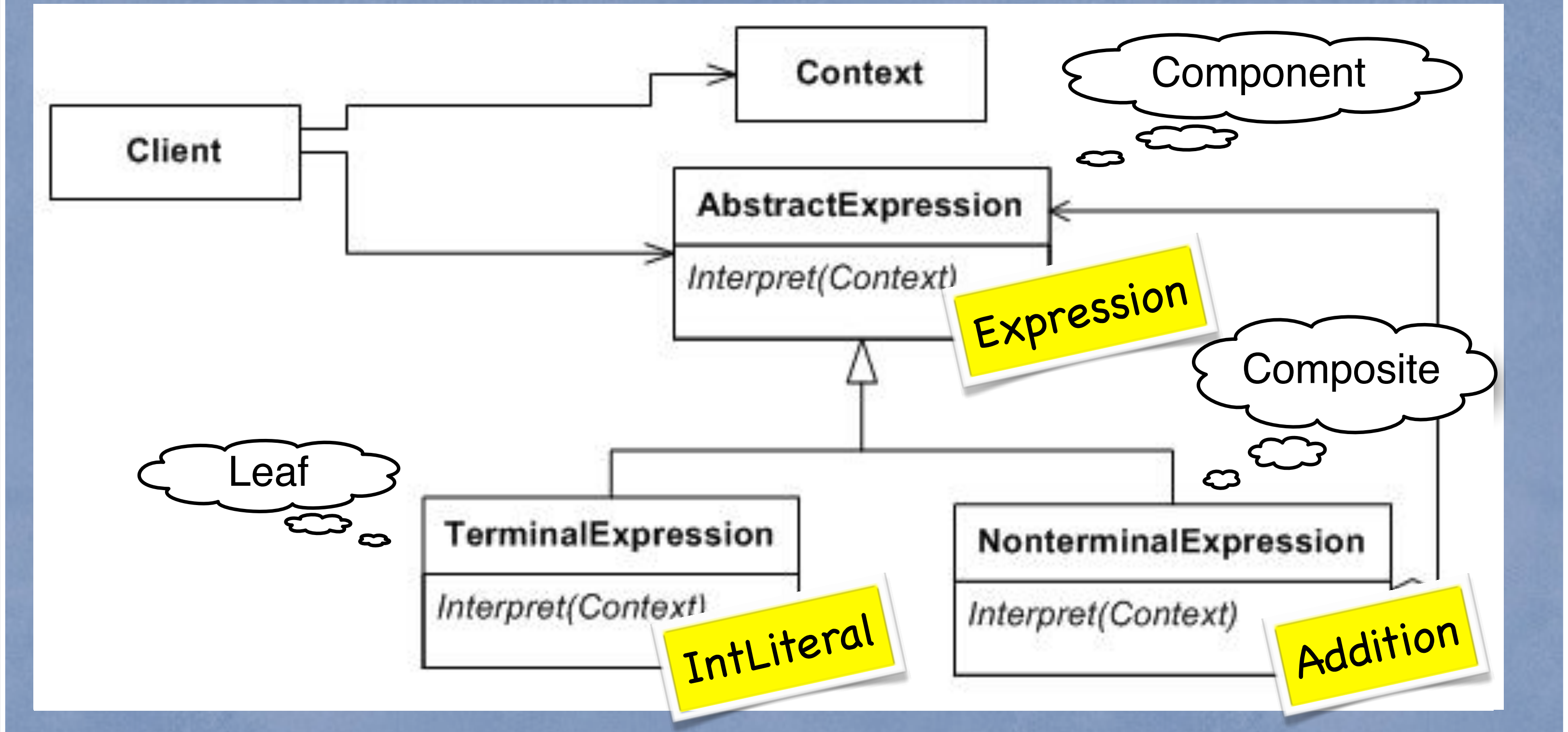


Fig. 4. The Interpreter Pattern with references to Composite (Figure 2).[7]

image from wikipedia.org

Visitor Pattern

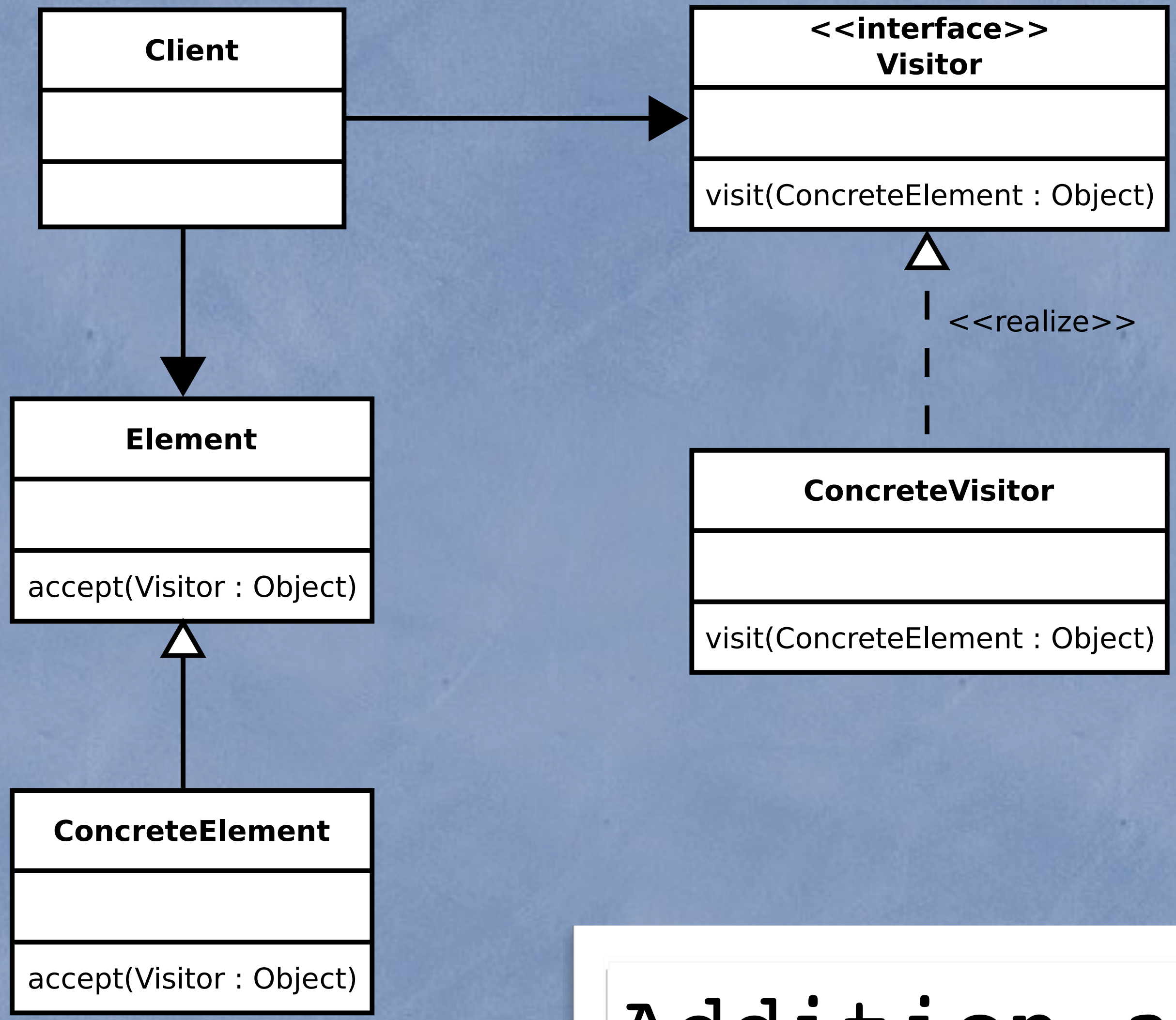


Fig. 3. The V

```
Addition accept(Visitor v) {
  return v.visitAddition(this);
}
```

image from wikipedia.org

Visitor Pattern

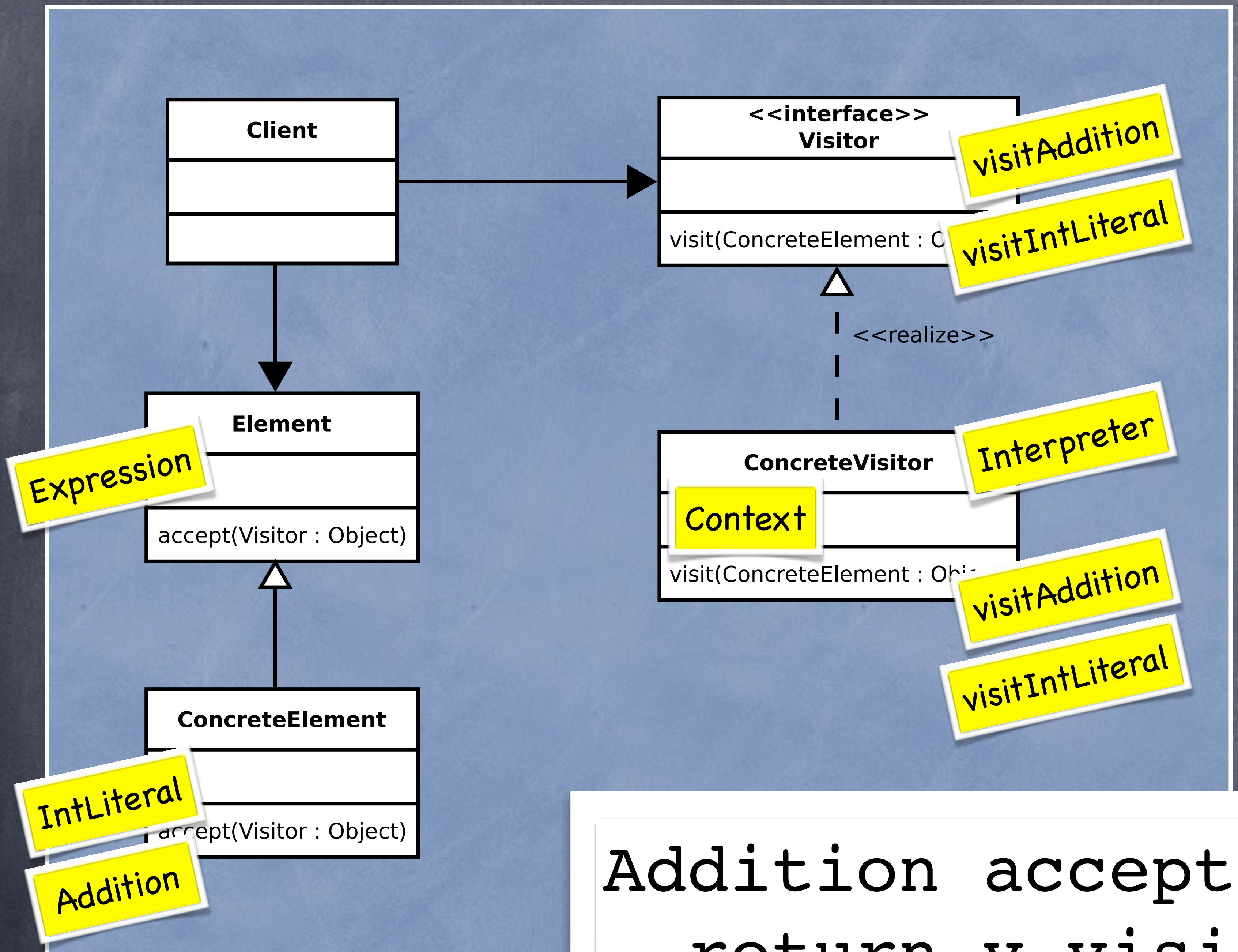


Fig. 3. The V

```
Addition accept(Visitor v) {
  return v.visitAddition(this);
}
```

image from wikipedia.org

Monday, December 10, 12

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

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

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

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

Theoretical Observations

- Visitor is conceptually more complex
  - Interpreter is only a small extension of composite

Theoretical Observations

- Visitor is conceptually more complex
  - Interpreter is only a small extension of composite
- Visitor encapsulates entire algorithms
  - Interpreter encapsulates language constructs

Theoretical Observations

- Visitor is conceptually more complex
  - Interpreter is only a small extension of composite
- Visitor encapsulates entire algorithms
  - Interpreter encapsulates language constructs
- Visitor's decoupling implies **dynamic** indirection
  - Interpreter has less dynamic dispatch

Theoretical Observations

- Visitor is conceptually more complex
  - Interpreter is only a small extension of composite

Harder to maintain, right?

- Visitor encapsulates entire algorithms
  - Interpreter encapsulates language constructs
- Visitor's decoupling implies **dynamic** indirection
  - Interpreter has less dynamic dispatch

Theoretical Observations

- Visitor is conceptually more complex
  - Interpreter is only a small extension of composite

Harder to maintain, right?

- Visitor encapsulates
  - Interpreter encapsulates language constructs

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

- Visitor's decoupling implies **dynamic** indirection
  - Interpreter has less dynamic dispatch

Theoretical Observations

- Visitor is conceptually more complex

Harder to maintain, right?

  - Interpreter is only a small extension of composite

- Visitor encapsulates

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

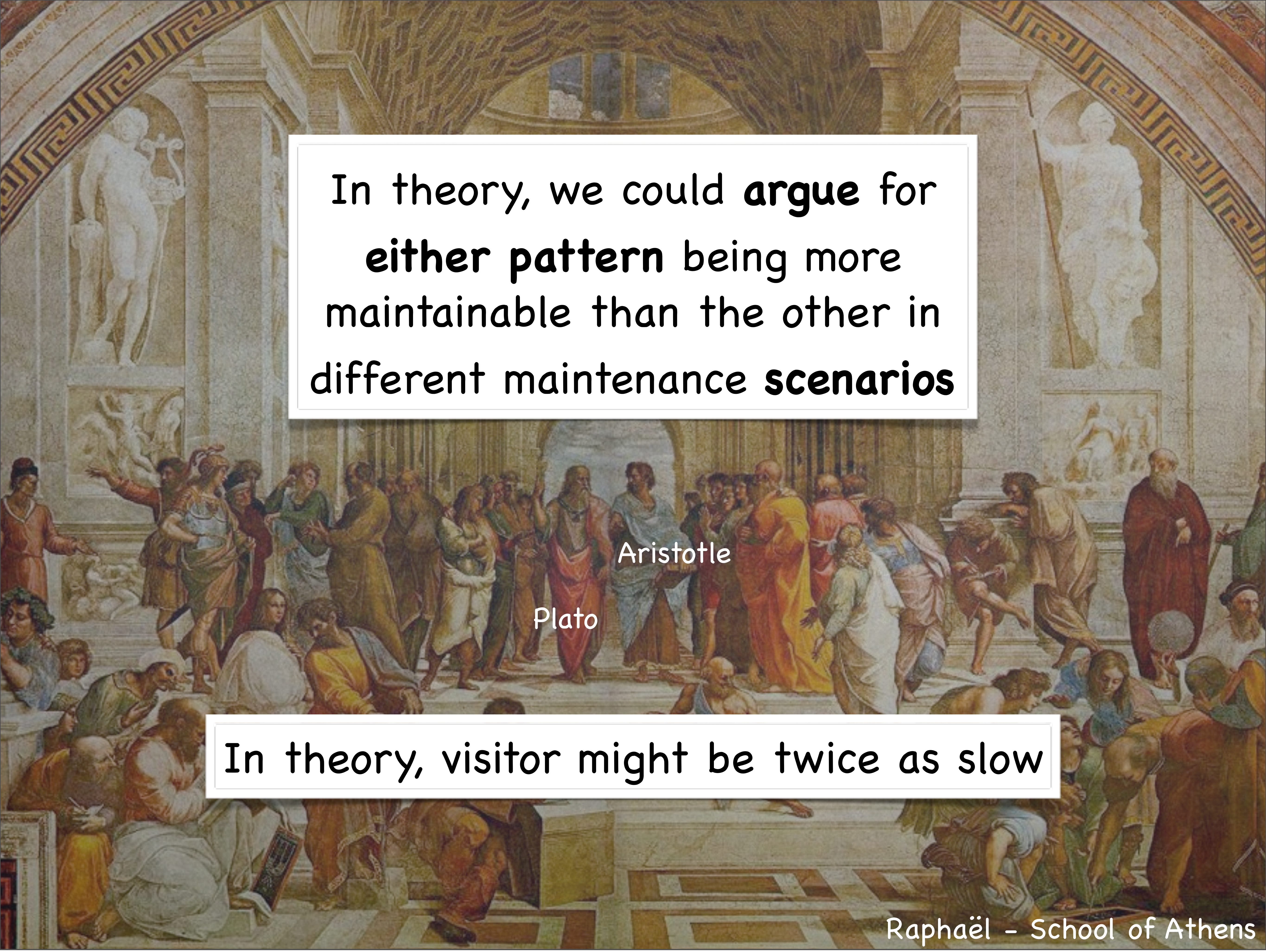
  - Interpreter encapsulates language constructs

- Visitor's dec amic indirection

Slower, right?

  - Interpreter has less dynamic dispatch

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?

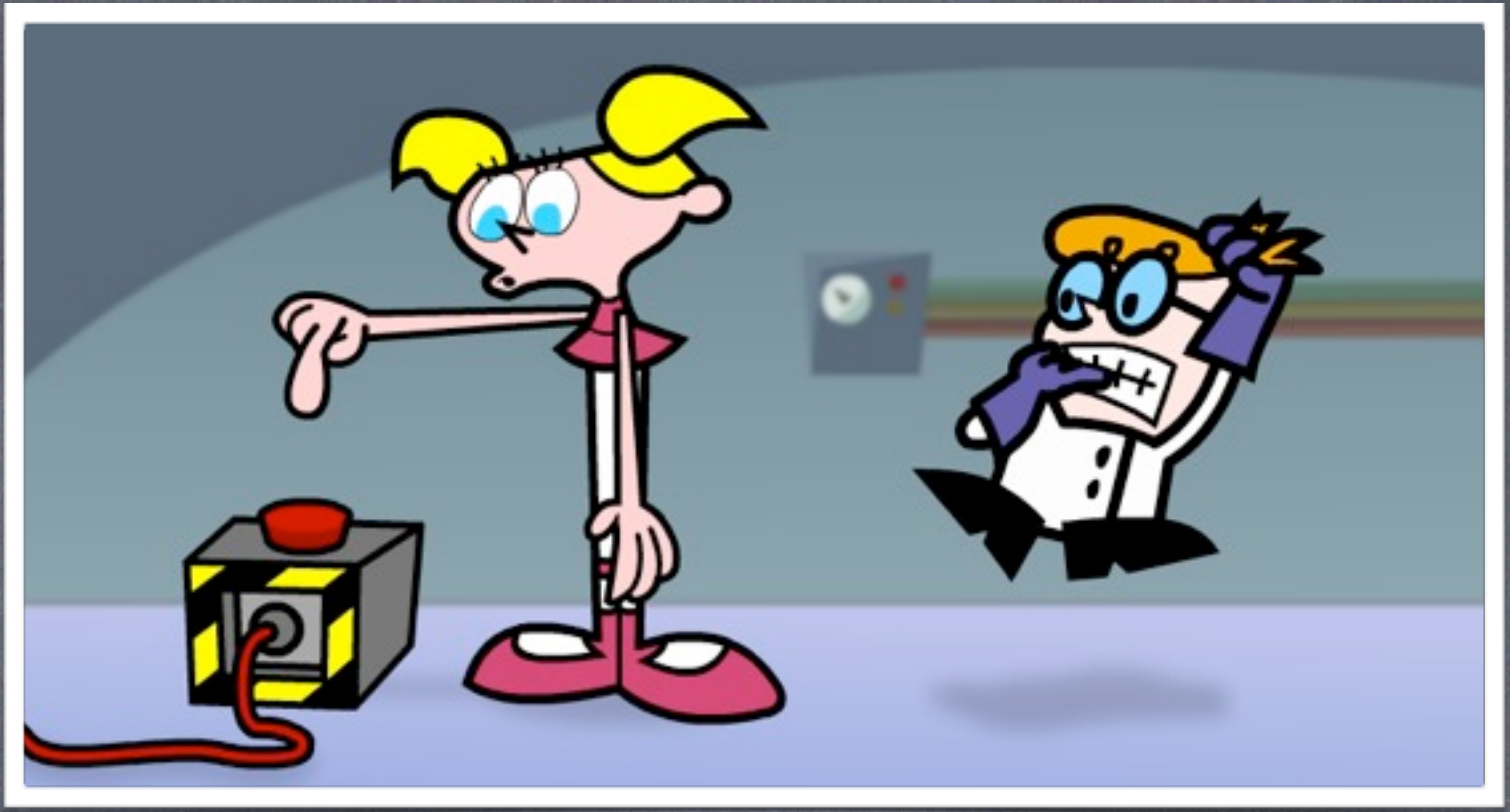
- How does one answer such a question?

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

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

Maintainability Index I&II

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

Maintainability Index I&II

SIG maintainability model

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

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

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

Maintainability Index I&II

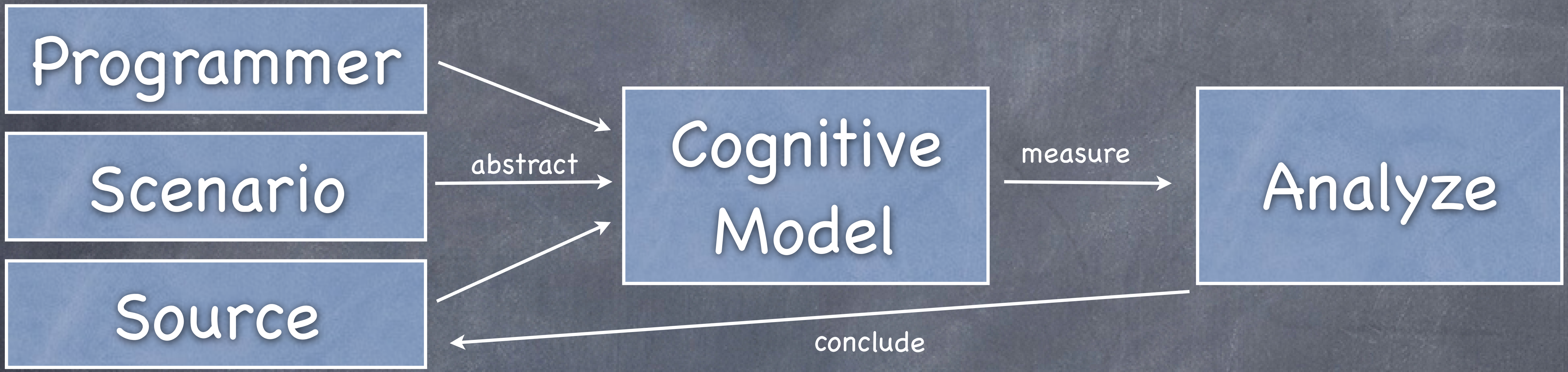
Maintenance Complexity Metric

SIG maintainability model

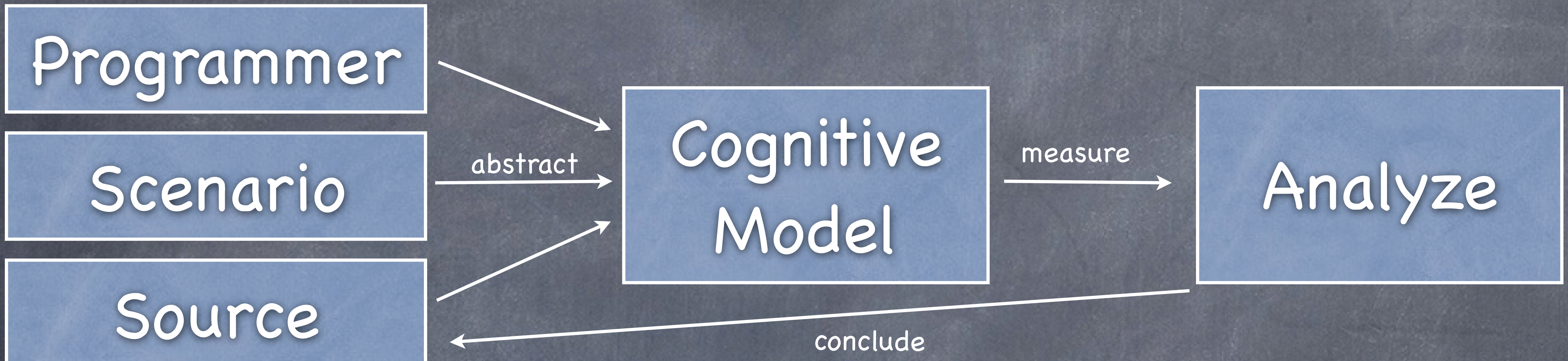
Computing and aggregating metrics values, **independent** of maintenance scenario, predicting long-term expectations on maintenance costs

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?



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



Programmer

Scenario

Source

abstract

Cognitive Model

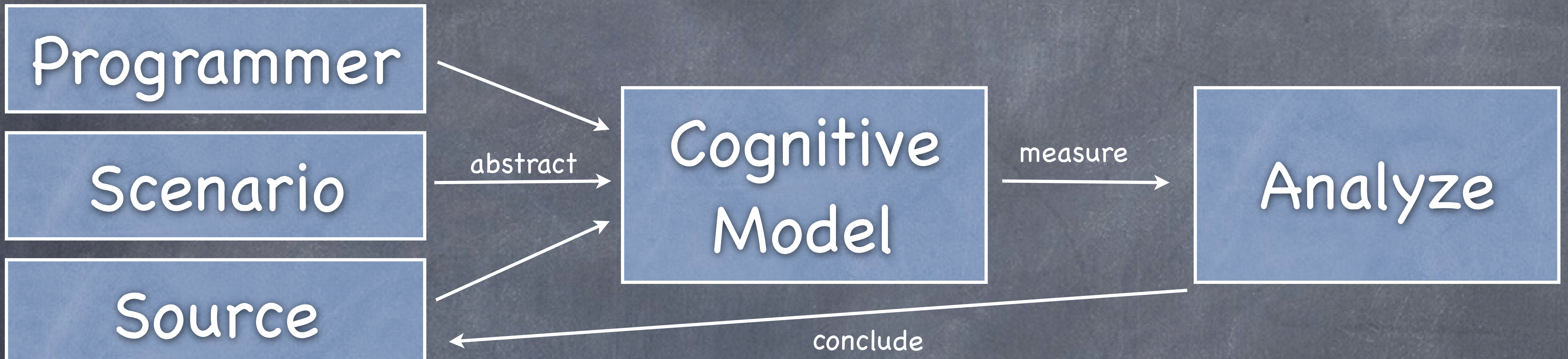
measure

Analyze

conclude

Unfortunately, we neither understand nor trust these models

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



Programmer

Scenario

Source

abstract

Cognitive Model

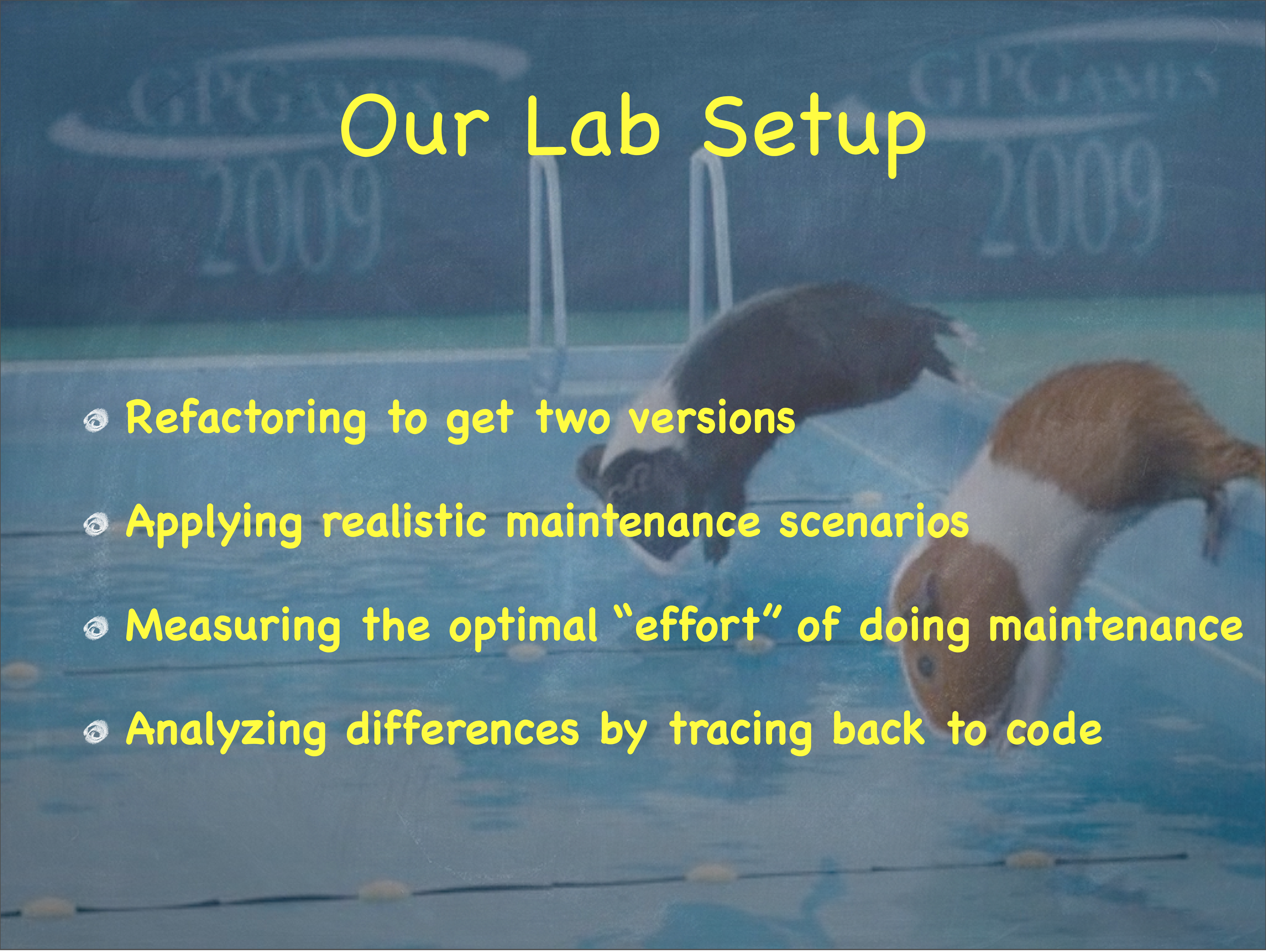
measure

Analyze

conclude

Unfortunately, we neither understand nor trust these models

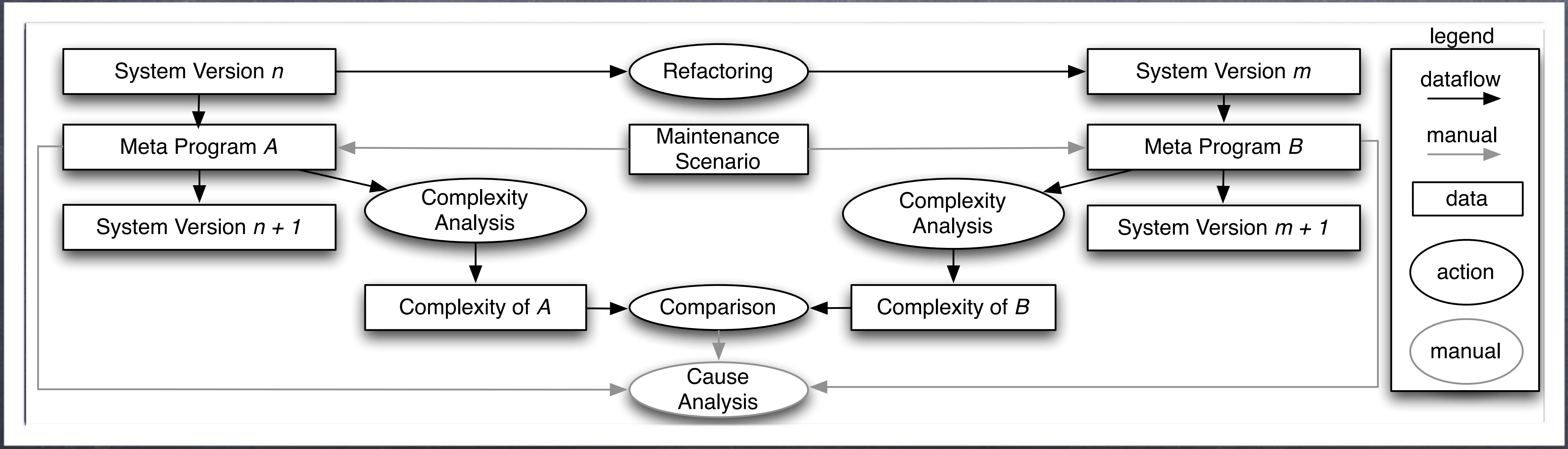
IDE + source code + human => very complex models of cognition



Our Lab Setup

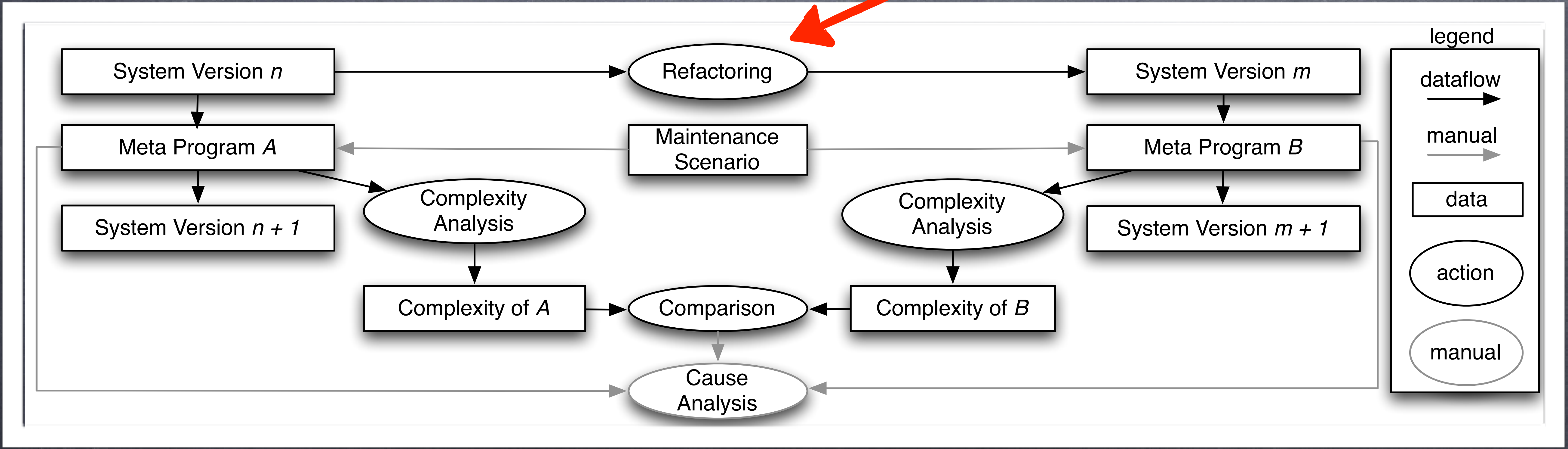
- Refactoring to get two versions
- Applying realistic maintenance scenarios
- Measuring the optimal “effort” of doing maintenance
- Analyzing differences by tracing back to code

Isolating the variable



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

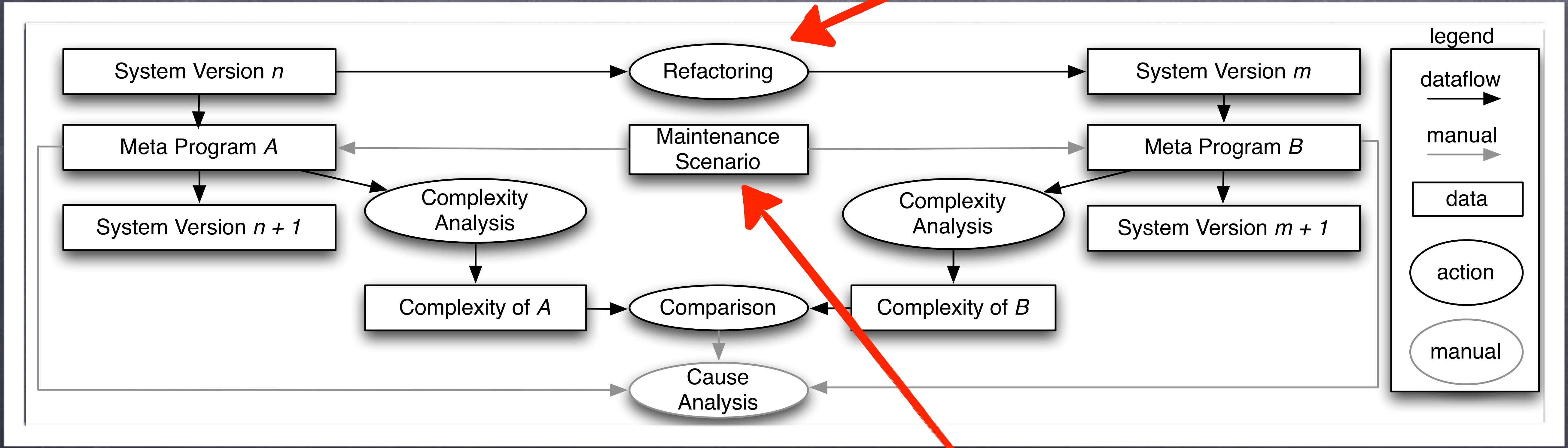
Isolating the variable



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

Isolating the variable

Key enabler

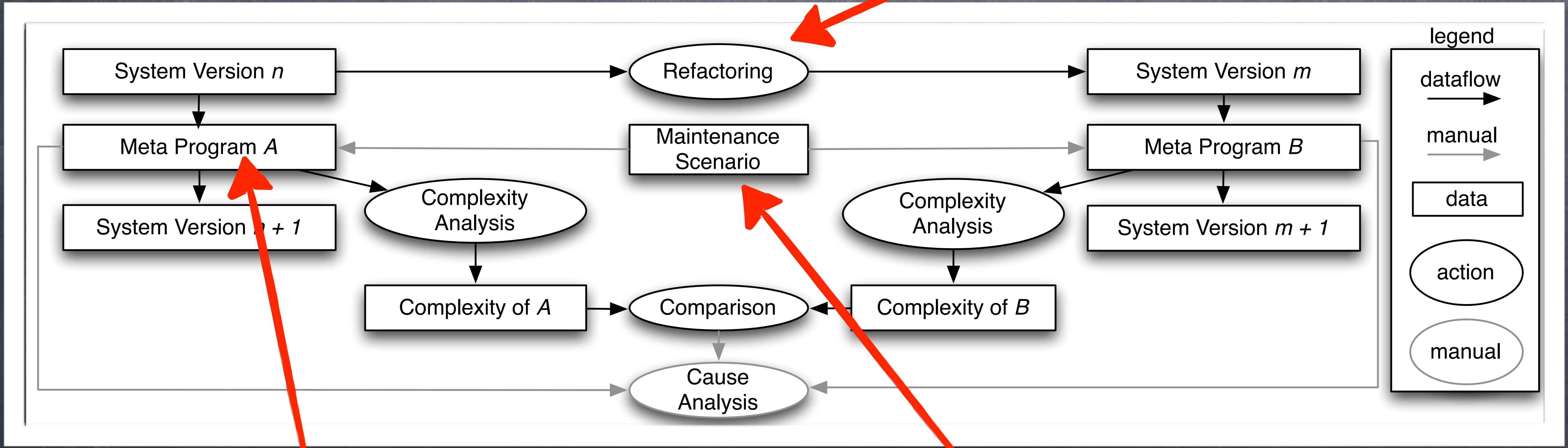


Manual labor

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

Isolating the variable

Key enabler

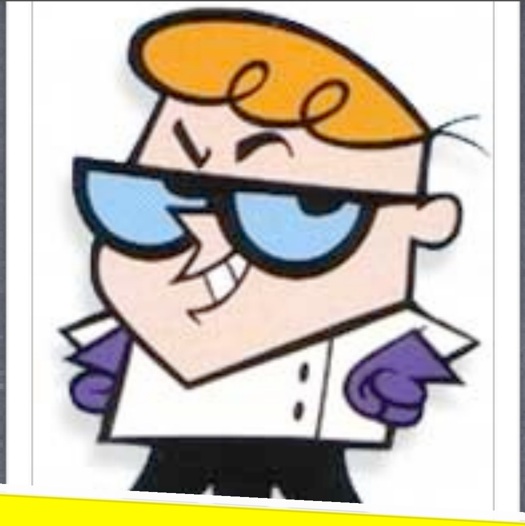


Traceability

Manual labor

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

“Complexity of Maintenance”

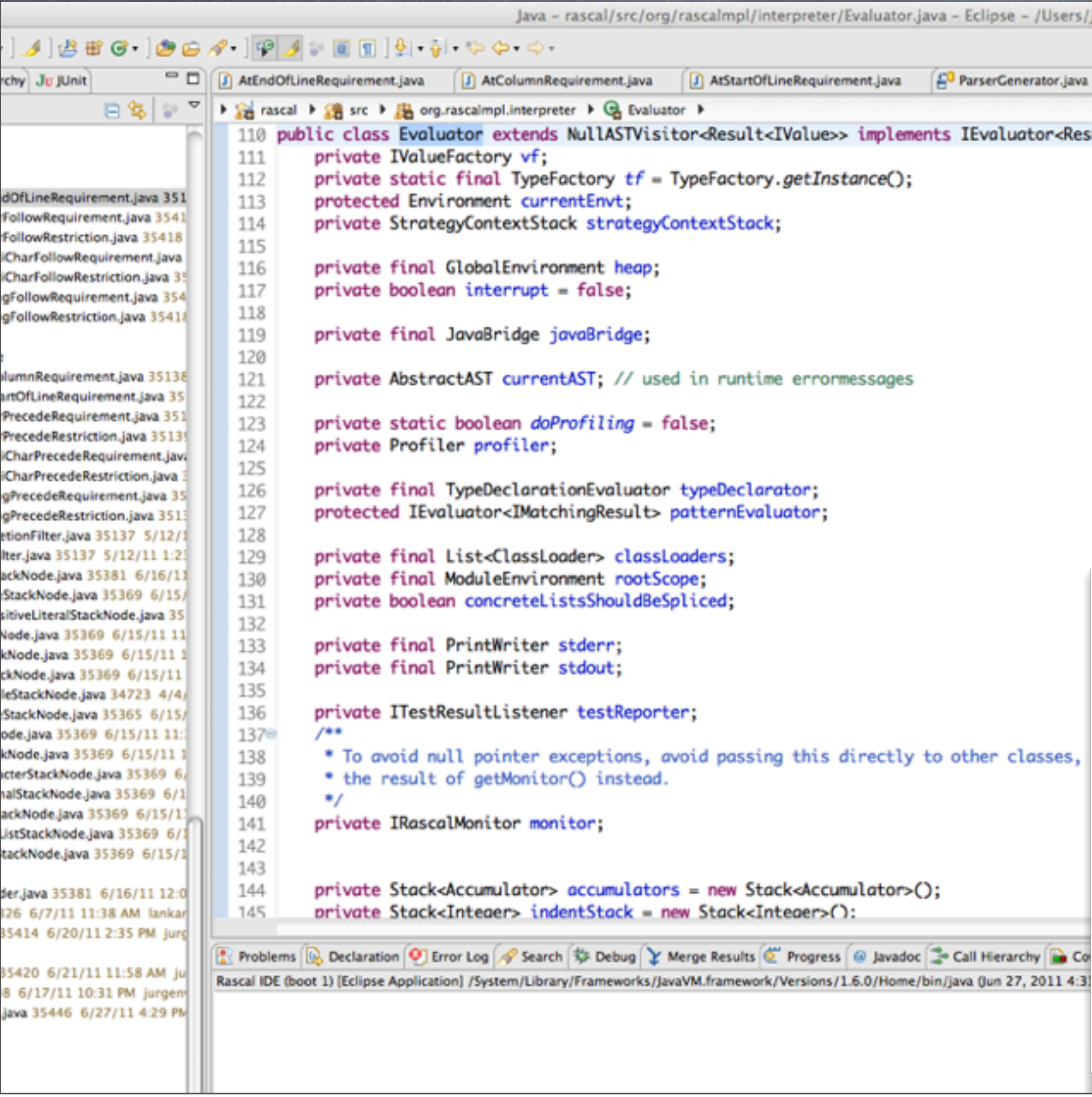


Precise definitions in the paper

- 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

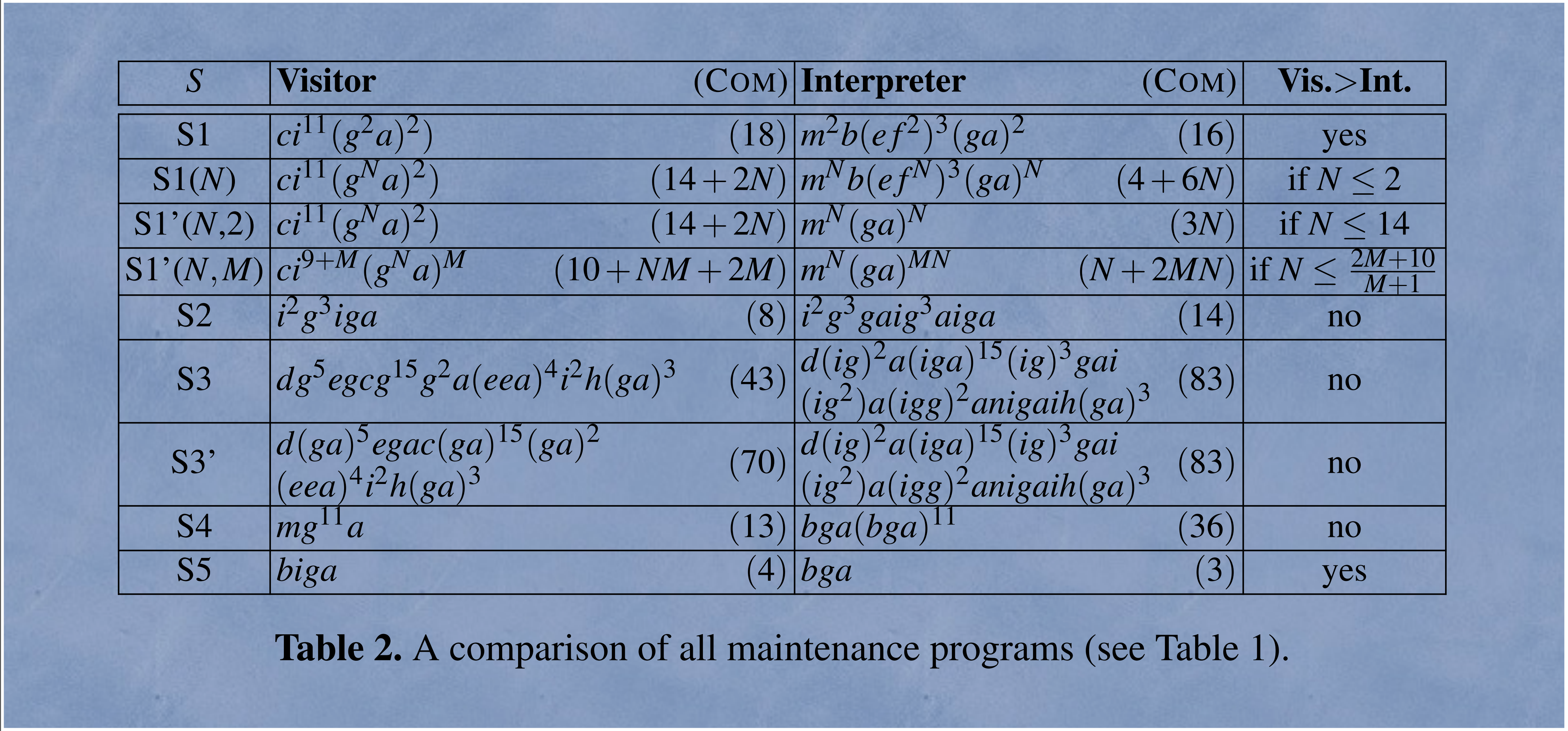

Java – rascal/src/org/rascalmpl/interpreter/Evaluator.java – Eclipse – /Users/jurgenv/Wo
JUnit
AtEndOfLineRequirement.java
AtColumnRequirement.java
AtStartOfLineRequirement.java
ParserGenerator.java
Meta
rascal
src
org.rascalmpl.interpreter
Evaluator
Problems
Declaration
Error Log
Search
Debug
Merge Results
Progress
Javadoc
Call Hierarchy

```java
public class Evaluator extends NullASTVisitor<Result<IValue>> implements IEvaluator<Result<IValu
	private IValueFactory vf;
	private static final TypeFactory tf = TypeFactory.getInstance();
	protected Environment currentEnvt;
	private StrategyContextStack strategyContextStack;

	private final GlobalEnvironment heap;
	private boolean interrupt = false;

	private final JavaBridge javaBridge;

	private AbstractAST currentAST; // used in runtime errormessages

	private static boolean doProfiling = false;
	private Profiler profiler;

	private final TypeDeclarationEvaluator typeDeclarator;
	protected IEvaluator<IMatchingResult> patternEvaluator;

	private final List<ClassLoader> classLoaders;
	private final ModuleEnvironment rootScope;
	private boolean concreteListsShouldBeSpliced;

	private final PrintWriter stderr;
	private final PrintWriter stdout;

	private ITestResultListener testReporter;
	/**
	 * To avoid null pointer exceptions, avoid passing this directly to other classes,
	 * the result of getMonitor() instead.
	 */
	private IRascalMonitor monitor;


	private Stack<Accumulator> accumulators = new Stack<Accumulator>();
	private Stack<Integer> indentStack = new Stack<Integer>();
```

```
Rascal IDE (boot 1) [Eclipse Application] /System/Library/Frameworks/JavaVM.framework/Versions/1.6.0/Home/bin/java (Jun 27, 2011 4:3
```

Results

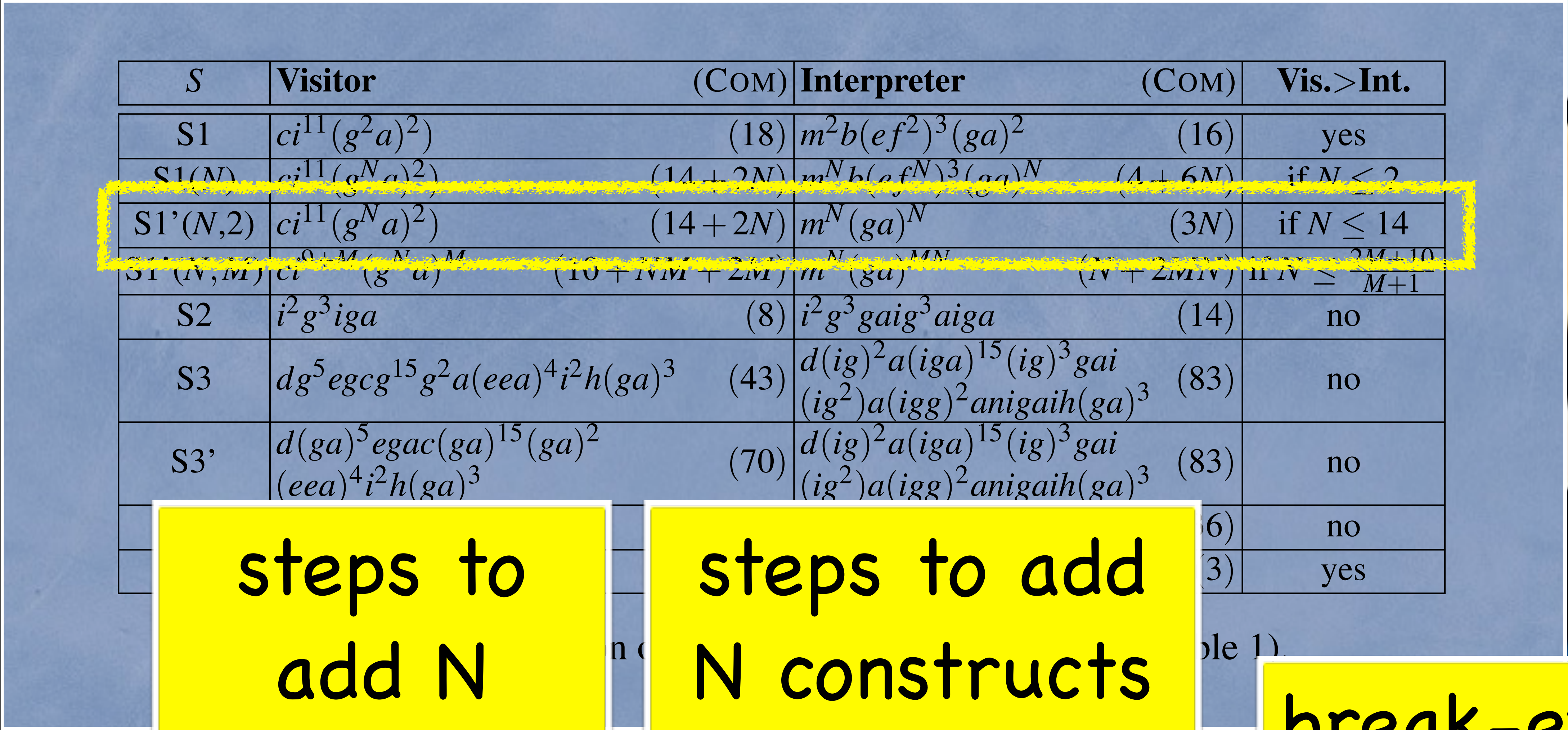
Results



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
S4	$mg^{11}a$	(13)	$bga(bga)^{11}$	(36)	no
S5	$biga$	(4)	bga	(3)	yes

Table 2. A comparison of all maintenance programs (see Table 1).

Results



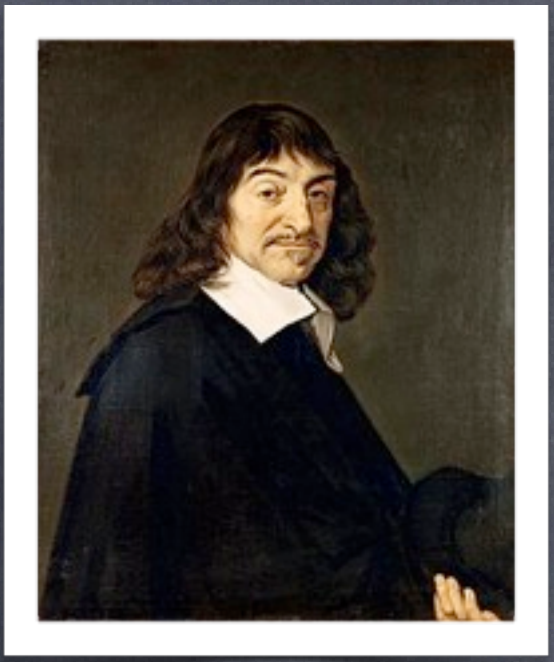
S	Visitor	(COM)	Interpreter	(COM)	Vis.>Int.
S1	$ci^{11}(g^2a)^2)$	(18)	$m^2b(ef^2)^3(ga)^2$	(16)	yes
S1(N)	$ci^{11}(g^Na)^2)$	$(14+2N)$	$m^Nb(ef^N)^3(ga)^N$	$(4+6N)$	if $N \leq 2$
S1'(N,2)	$ci^{11}(g^Na)^2)$	$(14+2N)$	$m^N(ga)^N$	$(3N)$	if $N \leq 14$
S1'(N,M)	$ci^{9+M}(g^Na)^M$	$(10+NM+2M)$	$m^N(ga)^{MN}$	$(N+2MN)$	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
				6)	no
				3)	yes

n c … ble 1)

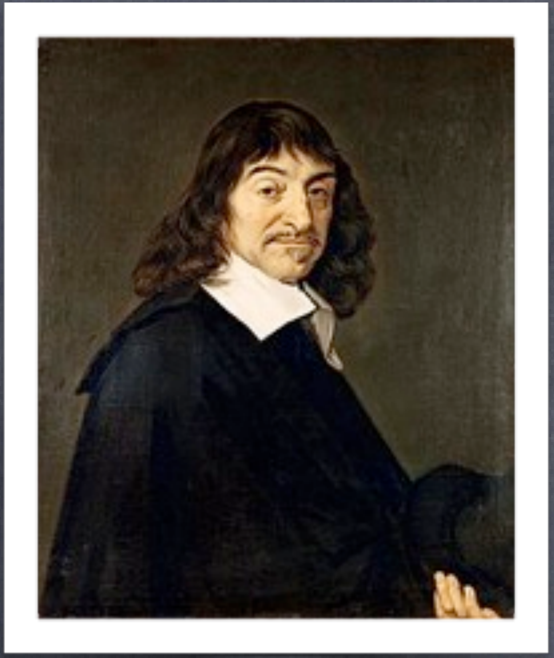
steps to add N constructs to **Visitor**
14 + 2N

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

break-even at N = 14

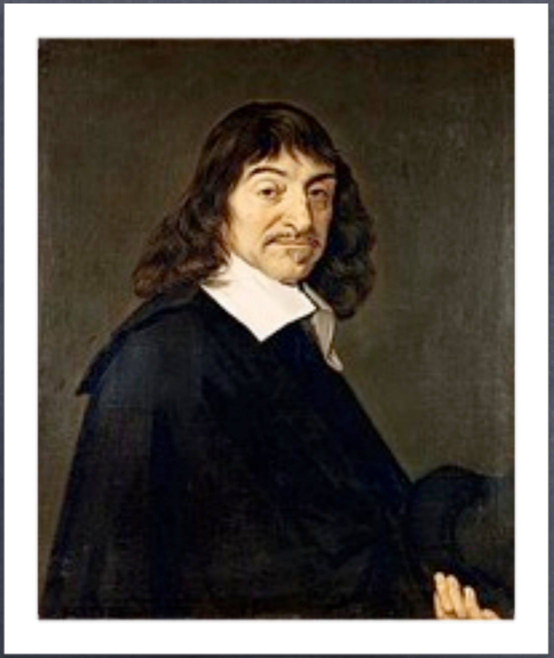


Why trust this?



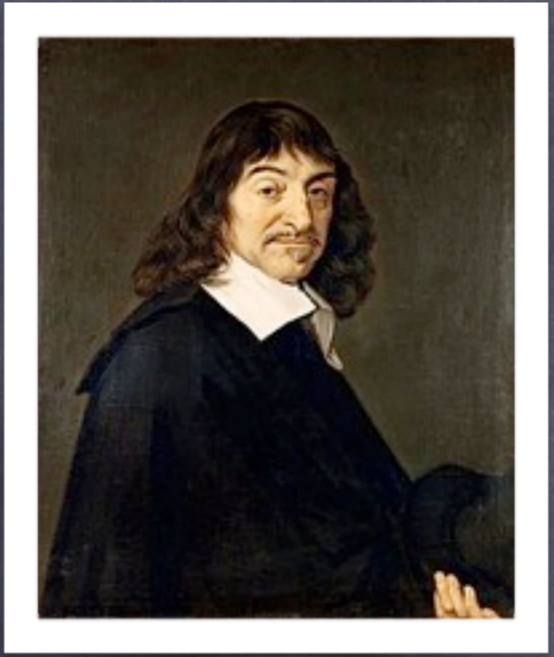
Why trust this?

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



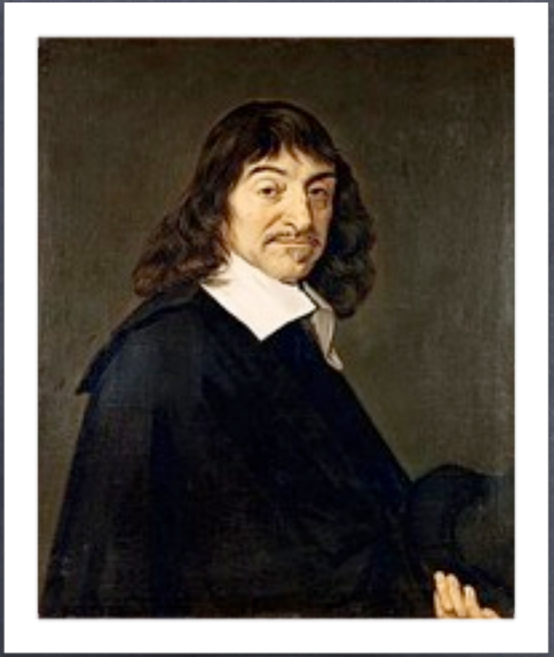
Why trust this?

- **Construct** validity: are all aspects of maintainability observable in this experiment?
- **Internal** validity: did you really do the best job possible in all scenarios?



Why trust this?

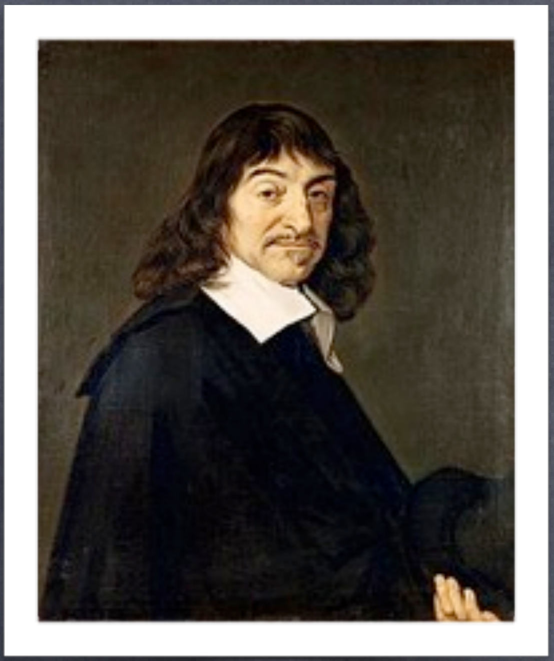
- **Construct** validity: are all aspects of maintainability observable in this experiment?
- **Internal** validity: did you really do the best job possible in all scenarios?
- **External** validity: does this say anything about the next interpreter I write in Java? The next maintenance? What if I don't use Eclipse? What if <blablabla>?



Why trust this?

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

- **Construct** validity: are all aspects o maintainability observable in this experiment?
- **Internal** validity: did you really do the best job possible in all scenarios?
- **External** validity: does this say anything about the next interpreter I write in Java? The next maintenance? What if I don't use Eclipse? What if <blablabla>?



Why trust this?

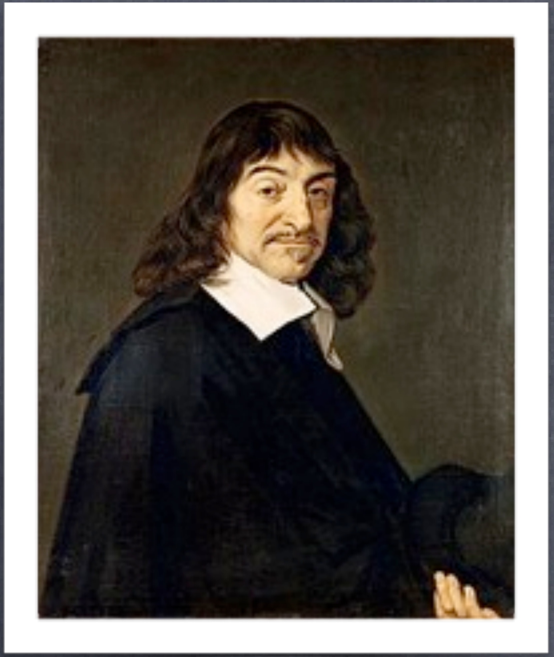
- **Construct** validity: are all aspects o maintainability observable in this experiment?

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

- **Internal** validity: did you really do t 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 don't use Eclipse? What if <blablabla>?



Why trust this?

- **Construct** validity: are all aspects o 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 t 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 don't use Eclipse? What if <blablabla>?

we do **not** know

Summary

*given the scope of the experiment

CWI Centrum Wiskunde & Informatica

Summary

- We used **Rascal** to build a **refactoring** tool

*given the scope of the experiment

CWI Centrum Wiskunde & Informatica

Summary

- We used **Rascal** to build a **refactoring** tool
- to **isolate** the difference between **Visitor** & **Interpreter**

*given the scope of the experiment

Summary

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

*given the scope of the experiment

Summary

- 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

CWI Centrum Wiskunde & Informatica

http://www.rascal-mpl.org

Summary

- 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

http://www.rascal-mpl.org

Summary

Feedback on paper and research method more than welcome!

- 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

CWI Centrum Wiskunde & Informatica