Constructing specialist software tools using Rascal

@jurgenvinju
April 24th 2012
@sogyo
Centrum Wiskunde & Informatica (www.cwi.nl)

where programming languages come from (1968)

where the internet started for Europe (1988)

Fundamental research

algorithms, theories, languages, models, tools

High societal relevance

Life Sciences, Energy, Logistics, Data, Software
Centrum Wiskunde & Informatica (www.cwi.nl)

where programming languages come from (1968)

where the internet started for Europe (1988)

Fundamental research

algorithms, theories, languages, models, tools

High societal relevance

Life Sciences, Energy, Logistics, Data, Software
Centrum Wiskunde & Informatica (www.cwi.nl)
where programming languages come from (1968)
where the internet started for Europe (1988)
Fundamental research
algorithms, theories, languages, models, tools
High societal relevance
Life Sciences, Energy, Logistics, Data, Software
Centrum Wiskunde & Informatica (www.cwi.nl)
where programming languages come from (1968)
where the internet started for Europe (1988)
Fundamental research
algorithms, theories, languages, models, tools
High societal relevance
Life Sciences, Energy, Logistics, Data, Software
Disclaimer
Disclaimer

Do try this “at home”
Disclaimer

- Do try this "at home"
- Rascal is under development & evaluation
  - beta quality
  - alpha guarantees
  - ready for proofs-of-concepts and one-offs
Disclaimer

Do try this "at home"

- Rascal is under development & evaluation
  - beta quality
  - alpha guarantees
  - ready for proofs-of-concepts and one-offs
- We use it for our own research
Disclaimer

- Do try this “at home”
- Rascal is under development & evaluation
  - beta quality
  - alpha guarantees
  - ready for proofs-of-concepts and one-offs
- We use it for our own research
- We use it in teaching at the Master Software Evolution (UvA) and Open Universiteit, etc.
Disclaimer

Do try this "at home"

- Rascal is under development & evaluation
  - beta quality
  - alpha guarantees
  - ready for proofs-of-concepts and one-offs
- We use it for our own research
- We use it in teaching at the Master Software Evolution (UvA) and Open Universiteit, etc.
- Today is a not a crash course in Rascal programming

Check out http://tutor.rascal- mpl.org
Why am I here?
Master Software Engineering

Engineering research

CWI Centrum Wiskunde & Informatica

SOGYO

Medium

Why am I here?
CHAOTIC GOOD
Because seeking the truth and blowing shit up go hand in hand
Why?

- Why does CWI:SEN1 invest in a meta-programming language?
- Why does UvA, OU, et al. teach it?

What?

- What is it from a bird’s eye view?
- What is it used for?

First 45 minutes

Never explain yourself to anyone. Because the person who likes you doesn’t need it, and the person who dislikes you won’t believe it...

© dbgreetings.com
Metrics!

- Why build your own metrics?
- How to build your own metrics?
- S.W.O.T. discussion

Refactorings!

- Why build your own refactorings?
- Example: change your design
- S.W.O.T. discussion
We study software systems: their design, their construction and their inevitable evolution.

learning to understand software systems
learning to improve them
focusing on complexity as the primary quality attribute
studying the causes of software complexity
studying solutions to get simpler software

(NASA mission control, apollo 13)
Software is not so difficult to understand, but it is extremely complex.
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)
The source code of "ls"

3894 lines

367 ifs

174 cases
Size does matter

A normal Dutch company may own $3 \times 10^{10}$ 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.

What a about 1M, 100k, 50k? Easy?
It’s all about understanding
It’s all about understanding

- Maintenance represents the major part (50% - 80%) of the cost of ownership of Software.
- Understanding takes much more time than editing.

So we should:

- make simpler code
- make understanding code easier
Size + Complexity -> Tools

Tool

Research

Application
Which tools are used at Sogyo?
Research

Tool

Application
(open-source) tools are for improved **research methods** and improved **transfer** to society
(open-source) tools are for improved research methods and improved valorisation.

- Verifiable
- Proven
- Transferable
- Innovative
Meta Software

(Brueghel, Tower of Babel)
3 Meta Software Challenges

1: Diversity

- Code
- Model
- Picture
3 Meta Software Challenges

2: Multi-disciplinary

(Raphael, Parnassus)
3 Meta Software Challenges

3: Precision vs. Efficiency
Ingredients
Ingredients

Integration to tackle multi-disciplinary nature

Familiar notation
IDE integration
Interactive Documentation

Key enablers
Ingredients

- Language parametric
- Generic programming
- Modularity

Programming techniques for dealing with diversity and scale

Integration to tackle multidisciplinary nature

- Relational Calculus
- Term Rewriting
- Syntax definition

Key enablers

- Familiar notation
- IDE integration
- Interactive Documentation
module FileTypes

import experiments::VL::VLCore;
import experiments::VL::Chart;
import viz::VLRender;
import JDT;
import Java;
import Resources;
import IO;
import Set;
import Map;
import Relation;
import Graph;

loc project = project::org.eclipse.imp.pdb.values;

Resource extract0{
    println("reading project ...");
    return extractProject(project);
}

public void main0{
    res = extract0;
    extCnt = 0;
    visit(res) case file(loc l):
        if(l.extension != ")") extCnt[l.extension]0 += 1;
    }

    render(pieChart("File types", extCnt));
}
Rascal tools are typically tens to a few hundred lines of code.
Rascal tools are typically tens to a few hundred lines of code.

Rascal is learned in a day or two by Java or C# programmers.
So why Rascal?

Because we want to understand software and make it simpler and we need tools for that, that deal with real software.
So what is Rascal?

A programming language for manipulating source code and its derivates, for quickly building software analysis, transformation, generation, visualization tools.
So what is Rascal?

A programming language for manipulating source code and its derivates, for quickly building software analysis, transformation, generation, visualization tools.

Questions & Coffee!
There is no substitute for thinking

- By design, Rascal does not think for you
- It is a tool itself that helps you “play” with software
- So, what is a good metric?
- So, when do you build a refactoring?
Software metrics

- Why?
- How?
- Pitfalls?
Why metrics?

“Numbers tell the tale”
Why metrics?

“Numbers tell the tale”

Observing: progress, hot-spots, “quality”
Why metrics?

“Numbers tell the tale”

Observers: progress, hot-spots, “quality”

Relating: cost, deployment, process
Why metrics?

- Observing: progress, hot-spots, “quality”
- Relating: cost, deployment, process
- Predicting: bugs, tests, maintenance cost

“Numbers tell the tale”
Why metrics?

- Observing: progress, hot-spots, “quality”
- Relating: cost, deployment, process
- Predicting: bugs, tests, maintenance cost
- Strategic: business “intelligence”
Why not metrics?

- What are we measuring anyway? The metric...
- Is quality measurable?
- “Lies, damn lies, and statistics”
- Is aggregation sound?
- Value judgments?
- Tunnelvision alert!
How to measure

Source code
Documentation
Tests

Version history
Execution traces
Log files

Extract

Internal Representation

Synthesize

Analyze

Results
How to measure

Source Code → Basic facts → Metrics → Statistics → Report

Source Code → Joined facts → Metrics

Rascal = “one-stop-shop”: all in 20 lines
How to measure

Sources:
- Code

Basic facts
- Metrics
- Statistics

Joined facts

Report

extraction is hard to get right

Rascal = “one-stop-shop”: all in 20 lines

Tuesday, April 24, 12
How to measure

Source Code

Basic facts

Metrics

Statistics

Report

what are we measuring?

extraction is hard to get right

Joined facts

Rascal = “one-stop-shop”: all in 20 lines

Tuesday, April 24, 12
How to measure

Source Code

Basic facts

Metrics

what are we measuring?

Statistics

Report

based on what theory are we aggregating?

Joined facts

extraction is hard to get right

Rascal = “one-stop-shop”: all in 20 lines

Tuesday, April 24, 12
How to measure

Source Code

Basic facts

Metrics

Statistics

Report

what are we measuring?
suggestive visuals

based on what theory are we aggregating?

Joined facts

extraction is hard to get right

Rascal = “one-stop-shop”: all in 20 lines

Tuesday, April 24, 12
Example: Cyclomatic Complexity

- Code with lots of branches, loops and cases is hard to understand.
- Every unique path should be tested and understood.
- McCabe Cyclomatic Complexity is a measure for the number of unique control flow paths through a method/function/procedure.
- High CC is bad, low CC is good.
CC: basic facts

- syntax CompilationUnit = "package" Id ...
- p = parse(#CompilationUnit, |file://myFile.java|)
- a tree for every file in the system
- watch out for double files
- grammars are expensive animals
- regular expressions are tricky animals
CC: metric

(0 | it + 1 | /stat := p, isSplit(stat))

isSplit(s) = s is if || s is while || s is case || s is ...

Easy... but no two tools do the same!

What is in Java, C#, PHP a control flow split?
CC: stats

all = { <cl, cc(cl)> | cl <- classes }

min(all), max(all), avg(all), mod(all)?

threshold(i, rel[T, int] r) = { x | <x,v> <- r, v >= i };

What does an aggregate tell you?

What does a value over a threshold tell you?
The total CC metric of a class, a system, a package, does NOT MAKE SENSE AT ALL.

It measures something that does not exist (control flow of a class?)

A euro for every tool that computes this aggregate...
CC: distribution?

number of methods

CC metric

Tuesday, April 24, 12
CC: distribution!

number of methods

mean[ingless]

CC metric
How to do aggregate then?

- Software Improvement Group Maintainability model (www.sig.eu)

- “Count the number of lines of code that contribute to methods that have a high mccabe”

- Answers the question: what % of the system is really complex?

- “sum(NCLOC(methods)) / sum(NCLOC(threshold(10, cc(methods))))”
- threshold(10, cc(methods)) may be valueable info to programmers
Works better to start from what you need to know than from what you can measure

You’ll find out when you can’t measure it

It will make sense if you can measure it
Learn from econometrics

- Software metrics can learn from econometrics
- Study distributions, not aggregations
- Study differences not general truths or thresholds
- “Gini, Theil, Hoover, Atkinson”
- Rascal (will) include libraries for all these tools
- There is no substitute for thinking, but visualizing helps!
Rascal gives you

- Tools to make front-ends
- Libraries of front-ends
- Integration with Eclipse IDE
- Visualization
- Queries
- ...

Tuesday, April 24, 12
What else?

- Modeling & Simulation
- Repository mining
- Domain specific languages
- Generating code
- Visualizing code
- Source-to-source transformation
- Refactoring
(Daily Painting: Seascape, Message in a Bottle? painting by artist Nancy Pouche)
Rascal is a domain specific programming language for software tools
Rascal is a domain specific programming language for software tools
Rascal is a domain specific programming language for software tools.

Master internships
Funding opportunities
Visit CWI

(Daily Painting: Seascape, Message in a Bottle? painting by artist Nancy Pouche)
Master internships
Funding opportunities
Visit CWI

Rascal is a domain specific programming language for software tools

http://www.cwi.nl/sen1
http://www.rascal-mpl.org
http://ask.rascal-mpl.org
http://tutor.rascal-mpl.org

mailto:Jurgen.Vinju@cwi.nl
twitter:@jurgenvinju

(Daily Painting: Seascape, Message in a Bottle? painting by artist Nancy Pouche)
A case of Visitor versus Interpreter Pattern

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

- Abstract syntax trees (ASTs)
- Different operations on ASTs
- 400 node types, 140 node type categories
- Dispatch, dispatch, dispatch (case distinction)
- Maintaining the ± 100 kLOC java code is the issue
We compare design (patterns) to learn which is best in which situations
We compare design (patterns) to learn which is simpler
We compare design (patterns) to learn which is faster.
We compare design (patterns) to learn which is easier to change
class, such as Expression or Statement. These contain one or more nested classes that extend the surrounding class for a particular language construct, such as If and While, both contained in and extending Statement, and Addition contained in and extending Expression.

All AST classes also inherit, directly or indirectly, from AbstractAST. AST classes provide access to children by way of getter methods, e.g., If and While have a getConditional<> method.

Fig. 2. The Composite Pattern

Rascal has many AST classes about 140 abstract classes and 400 concrete classes. To facilitate language evolution the code for these classes, along with the Rascal parser, is generated from the Rascal grammar. The AST code generator also creates a Visitor interface IASTVisitor, containing methods for all the node types in the hierarchy, and a default visitor NullASTVisitor. This class prevents us from having to implement a visit method for all AST node types, especially useful when certain algorithms focus on a small subset of nodes. Naturally, each AST node implements the accept<IASTVisitor<T> visitor> method by calling the appropriate visit method. For example, Statement.If contains:

```java
public <T> accept<IASTVisitor<T> v> {
    return v.visitStatementIf<this>;
}
```

Fig. 3. The Visitor Pattern

The desire to generate this code played a significant role in initially deciding to use the Visitor pattern. We wanted to avoid having to manually edit generated code. Using the Visitor pattern, all functionality that operates on the AST nodes can be separated from the generated code. When the Rascal grammar changes, the AST hierarchy is regenerated. Many implementations of IASTVisitor will contain Java compiler errors and warnings because the signature of visit methods will have changed. This is very helpful for locating the code that needs to be changed due to a language change. Most of the visitor classes actually extend...
Fig. 2. The Composite Pattern\textsuperscript{3}

\textbf{Composite Pattern}

![Composite Pattern](http://en.wikipedia.org/wiki/Composite_pattern)

\textbf{Statement}

![Statement](http://en.wikipedia.org/wiki/Visitor_pattern)

\textbf{Fig. 2. The Composite Pattern}\textsuperscript{3}

\textbf{image from wikipedia.org}
Expression or Statement (These contain one or more nested classes that extend the surrounding class for a particular language construct, such as If, While both contained in and extending Statement, and Addition contained in and extending Expression (All AST classes also inherit, directly or indirectly, from AbstractAST (AST classes provide access to children by way of getter methods, e.g., If and While have a getConditional<> method.

2.1 Creating and Processing Abstract Syntax Trees

Fig. 2. The Composite Pattern

Rascal has many AST classes about 140 abstract classes and 400 concrete classes (To facilitate language evolution the code for these classes, along with the Rascal parser, is generated from the Rascal grammar (The AST code generator also creates a Visitor interface IASTVisitor, containing methods for all the node types in the hierarchy, and a default visitor NoOp that returns null for every node type (This class prevents us from having to implement a visit method for all AST node types, especially useful when certain algorithms focus on a small subset of nodes (Naturally, each AST node implements the accept<IASTVisitor<T> visitor> method by calling the appropriate visit method (For example, Statement.If contains:

```java
public <T> accept<IASTVisitor<T> v> {
    return v.visitStatementIf<this>;
}
```

Fig. 3. The Visitor Pattern

The desire to generate this code played a significant role in initially deciding to use the Visitor pattern (We wanted to avoid having to manually edit generated code (Using the Visitor pattern, all functionality that operates on the AST nodes can be separated from the generated code (When the Rascal grammar changes, the AST hierarchy is regenerated (Many implementations of IASTVisitor will contain Java compiler errors and warnings because the signature of visit methods will have changed (This is very helpful for locating the code that needs to be changed due to a language change (Most of the visitor classes actually extend NoOp)

![Composite Pattern Image from wikipedia.org](http://en.wikipedia.org/wiki/Composite_pattern)

![Visitor Pattern Image from wikipedia.org](http://en.wikipedia.org/wiki/Visitor_pattern)
class, such as Expression or Statement (These contain one or more nested classes that extend the surrounding class for a particular language construct, such as If, While 4 both contained in and extending Statement 0, and Addition 4 contained in and extending Expression 0 (All AST classes also inherit, directly or indirectly, from AbstractAST (AST classes provide access to children by way of getter methods, e.g.

To facilitate language evolution the code for these classes, along with the Rascal parser, is generated from the Rascal grammar (The AST code generator also creates a Visitor interface IASTVisitor 0, containing methods for all the node types in the hierarchy, and a default visitor NullASTVisitor 0 (This class prevents us from having to implement a visit method for all AST node types, especially useful when certain algorithms focus on a small subset of nodes (Naturally, each AST node implements the accept<IASTVisitor<T> visitor> method by calling the appropriate visit method (For example, Statement.If contains:

```java
public <T> accept<IASTVisitor<T> v> {
    return v.visitStatementIf<this>;  
}
```

The desire to generate this code played a significant role in initially deciding to use the Visitor pattern (We wanted to avoid having to manually edit generated code (Using the Visitor pattern, all functionality that operates on the AST nodes can be separated from the generated code (When the Rascal grammar changes, the AST hierarchy is regenerated (Many implementations of IASTVisitor will contain Java compiler errors and warnings because the signature of visit methods will have changed (This is very helpful for locating the code that needs to be changed due to a language change (Most of the visitor classes actually extend

---

![Composite Pattern](http://en.wikipedia.org/wiki/Composite_pattern)

![Visitor Pattern](http://en.wikipedia.org/wiki/Visitor_pattern)

---

Fig. 2. The Composite Pattern³
Fig. 2. The Composite Pattern

---

**Composite Pattern**

```
Component
+ operation()
0..* child

Leaf
+ operation()

Composite
+ operation()
+ add()
+ remove()
+ getChild()
```

---

*Fig. 2. The Composite Pattern*[^3]

[^3]: Image from [wikipedia.org](http://en.wikipedia.org/wiki/Composite_pattern)
class, such as Expression or Statement (These contain one or more nested classes that extend the surrounding class for a particular language construct, such as If, While, and Addition contained in and extending Statement, and Addition contained in and extending Expression. All AST classes also inherit, directly or indirectly, from AbstractAST (AST classes provide access to children by way of getter methods, etc.). If and While have a getConditional<> method.

Fig. 2. The Composite Pattern

![Composite Pattern](http://en.wikipedia.org/wiki/Composite_pattern)

Rascal has many AST classes—about 140 abstract classes and 400 concrete classes. To facilitate language evolution, the code for these classes, along with the Rascal parser, is generated from the Rascal grammar. The AST code generator also creates a Visitor interface, IASTVisitor, containing methods for all the node types in the hierarchy, and a default visitor, NullASTVisitor. This class prevents us from having to implement a visit method for all AST node types, especially useful when certain algorithms focus on a small subset of nodes. Naturally, each AST node implements the accept<IASTVisitor<T> visitor> method by calling the appropriate visit method. For example, Statement.If contains:

```java
public <T> accept<IASTVisitor<T> v> {
  return v.visitStatementIf<this>;
}
```

Fig. 3. The Visitor Pattern

The desire to generate this code played a significant role in initially deciding to use the Visitor pattern. We wanted to avoid having to manually edit generated code. Using the Visitor pattern, all functionality that operates on the AST nodes can be separated from the generated code. When the Rascal grammar changes, the AST hierarchy is regenerated. Many implementations of IASTVisitor will contain Java compiler errors and warnings because the signature of visit methods will have changed. This is very helpful for locating the code that needs to be changed due to a language change. Most of the visitor classes actually extend

![Visitor Pattern](http://en.wikipedia.org/wiki/Visitor_pattern)

image from wikipedia.org
class, such as `Expression` or `Statement` (These contain one or more nested classes that extend the surrounding class for a particular language construct, such as `If`, `While`) both contained in and extending `Statement` (All AST classes also inherit, directly or indirectly, from `AbstractAST`).

To facilitate language evolution, the code for these classes, along with the Rascal parser, is generated from the Rascal grammar. The AST code generator also creates a Visitor interface (`IASTVisitor`), containing methods for all the node types in the hierarchy, and a default visitor (`NullASTVisitor`) that returns null for every node type.

The desire to generate this code played a significant role in initially deciding to use the Visitor pattern. We wanted to avoid having to manually edit generated code. Using the Visitor pattern, all functionality that operates on the AST nodes can be separated from the generated code. When the Rascal grammar changes, the AST hierarchy is regenerated. Many implementations of `IASTVisitor` will contain Java compiler errors and warnings because the signature of visit methods will have changed. This is very helpful for locating the code that needs to be changed due to a language change. Most of the visitor classes actually extend `NoOp`.

Fig. 2. The Composite Pattern

![Composite Pattern](http://en.wikipedia.org/wiki/Composite_pattern)

![Visitor Pattern](http://en.wikipedia.org/wiki/Visitor_pattern)
NullASTVisitor though which is why it is important that each method they override is tagged with the @Override tag. Note that the class used to construct ASTs at runtime uses reflection to map parse tree nodes into the appropriate AST classes. Hence this code does not have to change when we change the grammar of the Rascal language.

2.2 A Comparison with the Interpreter Pattern

Fig. 4. The Interpreter Pattern with references to Composite (Figure 2).
Though which is why it is important that each method they override is tagged with the `@Override` tag.

Note that the class used to construct ASTs at runtime uses reflection to map parse tree nodes into the appropriate AST classes. Hence, this code does not have to change when we change the grammar of the Rascal language.

### 2.2 A Comparison with the Interpreter Pattern

Considering that our design already employs the Composite pattern, the difference in design complexity between the Visitor and Interpreter patterns is striking. The Composite pattern contains all the elements for the Interpreter pattern: abstract classes that are instantiated by concrete ones—only an interpret method needs to be added to all relevant classes. So rather than having to add new concepts, such as a Visitor interface and `NullASTVisitor`, the Interpreter pattern builds on the existing infrastructure of Composite and reuses it. Also, by adding more interpret methods, varying either the name or the static type, it is possible to reuse the Interpreter design pattern again and again without having to add additional classes. However, as a consequence, understanding each algorithm as a whole is now complicated by the fact that the methods implementing it are scattered over different AST classes. Additionally, there is the risk that methods contributing to different algorithms get tangled because a single AST class may have to manage the combined state required for all implemented algorithms.

The experiments discussed in Section help make this tradeoff between separation of concerns and complexity more concrete.

### 2.3 Refactoring from Visitor to Interpreter using Rascal

We constructed an automated refactoring tool for transforming Visitor classes to Interpreter methods. It is the key to our research method—see Figure.

*Image from [http://en.wikipedia.org/wiki/Interpreter_pattern](http://en.wikipedia.org/wiki/Interpreter_pattern) created by Jing Guo Yao and licensed under the Creative Commons Attribution-ShareAlike 4.0 International License.*

---

**Fig. 4.** The Interpreter Pattern with references to Composite (Figure 2).
class, such as Expression or Statement (These contain one or more nested classes that extend the surrounding class for a particular language construct, such as If, While both contained in and extending Statement, and Addition contained in and extending Expression. All AST classes also inherit, directly or indirectly, from AbstractAST (AST classes provide access to children by way of getter methods, especially useful when certain algorithms focus on a small subset of nodes. Naturally, each AST node implements the accept<IASTVisitor<T> visitor> method by calling the appropriate visit method. For example, Statement.If contains:

```java
public <T> accept<IASTVisitor<T> v> {
  return v.visitStatementIf<this>;
}
```

Fig. 3. The Visitor Pattern

![Visitor Pattern image from wikipedia.org](http://en.wikipedia.org/wiki/Visitor_pattern)
class, such as `Expression` or `Statement`. These contain one or more nested classes that extend the surrounding class for a particular language construct, such as `If` and `While`. Both are contained in and extending `Statement`, and `Addition` is contained in and extending `Expression`.

All AST classes also inherit, directly or indirectly, from `AbstractAST`. AST classes provide access to children by way of getter methods, e.g., `If` and `While` have a `getConditional<>()` method.

### 2.1 Creating and Processing Abstract Syntax Trees

Fig. 2. The Composite Pattern

Rascal has many AST classes, about 140 abstract classes and 400 concrete classes. To facilitate language evolution, the code for these classes, along with the Rascal parser, is generated from the Rascal grammar. The AST code generator also creates a Visitor interface containing methods for all the node types in the hierarchy, and a default visitor that returns null for every node type. This class prevents us from having to implement a visit method for all AST node types, especially useful when certain algorithms focus on a small subset of nodes. Naturally, each AST node implements the `accept<IASTVisitor<T>>` visitor method by calling the appropriate visit method. For example, `Statement.If` contains:

```java
public <T> accept<IASTVisitor<T>> v> {
    return v.visitStatementIf<this>;
}
```

Fig. 3. The Visitor Pattern

The desire to generate this code played a significant role in initially deciding to use the Visitor pattern. We wanted to avoid having to manually edit generated code. Using the Visitor pattern, all functionality that operates on the AST nodes can be separated from the generated code. When the Rascal grammar changes, the AST hierarchy is regenerated. Many implementations of `IASTVisitor` will contain Java compiler errors and warnings because the signature of visit methods will have changed. This is very helpful for locating the code that needs to be changed due to a language change. Most of the visitor classes actually extend...
class, such as Expression or Statement (These contain one or more nested classes that extend the surrounding class for a particular language construct, such as If, While both contained in and extending Statement, and Addition contained in and extending Expression (All AST classes also inherit, directly or indirectly, from AbstractAST (AST classes provide access to children by way of getter methods, e.g., If and While have a getConditional<> method (2.1 Creating and Processing Abstract Syntax Trees

Fig. 2. The Composite Pattern

Rascal has many AST classes about 140 abstract classes and 400 concrete classes (To facilitate language evolution the code for these classes, along with the Rascal parser, is generated from the Rascal grammar (The AST code generator also creates a Visitor interface (IASTVisitor, containing methods for all the node types in the hierarchy, and a default visitor (NullASTVisitor (This class prevents us from having to implement a visit method for all AST node types, especially useful when certain algorithms focus on a small subset of nodes (Naturally, each AST node implements the accept<IASTVisitor<T> visitor> method by calling the appropriate visit method (For example, Statement.If contains:

```java
public <T> accept<IASTVisitor<T> v> {
    return v.visitStatementIf<this>;
}
```

Fig. 3. The Visitor Pattern

The desire to generate this code played a significant role in initially deciding to use the Visitor pattern (We wanted to avoid having to manually edit generated code (Using the Visitor pattern, all functionality that operates on the AST nodes can be separated from the generated code (When the Rascal grammar changes, the AST hierarchy is regenerated (Many implementations of IASTVisitor will contain Java compiler errors and warnings because the signature of visit methods will have changed (This is very helpful for locating the code that needs to be changed due to a language change (Most of the visitor classes actually extend


class, such as Expression or Statement (These contain one or more nested classes that extend the surrounding class for a particular language construct, such as If, While both contained in and extending Statement, and Addition contained in and extending Expression (All AST classes also inherit, directly or indirectly, from AbstractAST (AST classes provide access to children by way of getter methods, e.g., If and While have a getConditional<> method.

Fig. 2. The Composite Pattern

Rascal has many AST classes about 140 abstract classes and 400 concrete classes. To facilitate language evolution the code for these classes, along with the Rascal parser, is generated from the Rascal grammar (The AST code generator also creates a Visitor interface IASTVisitor, containing methods for all the node types in the hierarchy, and a default visitor NullASTVisitor. This class prevents us from having to implement a visit method for all AST node types, especially useful when certain algorithms focus on a small subset of nodes. Naturally, each AST node implements the accept<IASTVisitor<T> visitor> method by calling the appropriate visit method. For example, Statement.If contains:

```java
public <T> accept<IASTVisitor<T> v> {
    return v.visitStatementIf<this>;
}
```

Fig. 3. The Visitor Pattern

The desire to generate this code played a significant role in initially deciding to use the Visitor pattern. We wanted to avoid having to manually edit generated code. Using the Visitor pattern, all functionality that operates on the AST nodes can be separated from the generated code. When the Rascal grammar changes, the AST hierarchy is regenerated. Many implementations of IASTVisitor will contain Java compiler errors and warnings because the signature of visit methods will have changed. This is very helpful for locating the code that needs to be changed due to a language change. Most of the visitor classes actually extend IASTVisitor.
Visitor design pattern and the Interpreter design pattern are functionally interchangeable.
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.
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
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
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

Harder to maintain, right?
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
- Visitor encapsulates entire algorithms
- Interpreter encapsulates language constructs
- Visitor’s decoupling implies 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.
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?

Tuesday, April 24, 12
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
Our Lab Setup

- Refactoring to get two versions
- Applying realistic maintenance scenarios
- Observing differences
A “refactoring” is an automated source-to-source program transformation that guarantees run-time semantics to be preserved.
Isolating the variable

Fig. 5. Comparative framework for observing differences in maintainability.

In our case study: version $n$ is the Rascal interpreter based on the Visitor pattern and version $m$ is the version of the Rascal interpreter based on the Interpreter pattern. The details of this automated refactoring are not relevant for the present analysis, but it is important to note that it is semantics preserving. The maintainability of both versions is now compared by designing a number of maintenance scenarios and applying them to both versions. For each maintenance scenario we do the following:

- Perform the maintenance scenario manually.
- Create an abstract description of this activity by expressing it as meta-program.
- Compare the computational complexity of the meta-programs needed to carry out the maintenance scenario for versions $n$ and $m$.

This allows us to objectively calculate the complexity of the scenarios as applied to the two versions while at the same time pinpointing exact causes of the differences.

Results produced by this framework can be replicated by anybody given the source code of the two versions: a precise description of the meta programs and the scenarios; and a precise description of the complexity analysis. In Section (a), we define a "virtual machine for maintenance" that provides the foundation for our current comparison.

3.3 Alternative Methods to Measure Maintainability

Our framework tries to abstract from the human programmer that actually carries out the maintenance tasks. This makes it easier to replicate our results. Alternative ways of studying maintenance do focus on human beings: like programmer observation and using models of cognition. Statistical observation of the efficiency of a group of programmers while doing maintenance tasks can be done to summarize the effects of differences between design patterns. However, such an "expensive" study cannot explain the causes of these effects, while our method can. The use of cognitive modeling can also shed light on the causes of complexity. With this method one explicitly constructs a representation of the knowledge that a human being is using while analyzing and modifying source code. Complexity measures for such representations exist as well and have been used to study understandability of programming in different kinds of languages. We have not opted for this approach because such detailed cognitive models are difficult to construct.
Isolating the variable

**System Version n**
- Meta Program A
- System Version n + 1
  - Complexity Analysis
  - Complexity of A
  - Cause Analysis
  - Refactoring
  - Maintenance Scenario
  - Complexity Analysis
  - Complexity of B
  - Comparison
  - System Version m
  - Meta Program B
  - System Version m + 1

**Key enabler**

Rascal to implement Visitor to Interpreter refactoring

---

Fig. 5. Comparative framework for observing differences in maintainability.
Isolating the variable

Fig. 5. Comparative framework for observing differences in maintainability

- Perform the maintenance scenario manually
- Create an abstract description of this activity by expressing it as meta-program
- Compare the computational complexity of the meta-programs needed to carry out the maintenance scenario for versions $n$ and $m$

This allows us to objectively calculate the complexity of the scenarios as applied to the two versions while at the same time pinpointing exact causes of the differences.

Results produced by this framework can be replicated by anybody given the source code of the two versions: a precise description of the meta programs and the scenarios: and a precise description of the complexity analysis. In Section ( ) we define a “virtual machine for maintenance” that provides the foundation for our current comparison.

3.3 Alternative Methods to Measure Maintainability

Our framework tries to abstract from the human programmer that actually carries out the maintenance tasks. This makes it easier to replicate our results. Alternative ways of studying maintenance do focus on human beings: like programmer observation and using models of cognition.

Statistical observation of the efficiency of a group of programmers while doing maintenance tasks can be done to summarize the effects of differences between design patterns. However, such an expensive study can not explain the causes of these effects: while our method can. The use of cognitive modeling can also shed light on the causes of complexity. With this method one explicitly constructs a representation of the knowledge that a human being is using while analyzing and modifying source code.

Complexity measures for such representations exist as well and have been used to study understandability of programming in different kinds of languages. We have not opted for this approach because such detailed cognitive models are difficult to construct.
Isolating the variable

**Key enabler**

**Traceability**

**Manual labor**

**Rascal to implement Visitor to Interpreter refactoring**

Fig. 5. Comparative framework for observing differences in maintainability

In our case study: version \( n \) is the Rascal interpreter based on the Visitor pattern and version \( m \) is the version of the Rascal interpreter based on the Interpreter pattern. The details of this automated refactoring are not relevant for the present analysis, but it is important to note that it is semantics preserving. The maintainability of both versions is now compared by designing a number of maintenance scenarios and applying them to both versions. For each maintenance scenario we do the following:

1. Perform the maintenance scenario manually.
2. Create an abstract description of this activity by expressing it as a meta-program.
3. Compare the computational complexity of the meta-programs needed to carry out the maintenance scenario for versions \( n \) and \( m \).

This allows us to objectively calculate the complexity of the scenarios as applied to the two versions while at the same time pinpointing exact causes of the differences. Results produced by this framework can be replicated by anybody given the source code of the two versions: a precise description of the meta-programs and the scenarios: and a precise description of the complexity analysis.

In Section 3.3 Alternative Methods to Measure Maintainability, our framework tries to abstract from the human programmer that actually carries out the maintenance tasks. This makes it easier to replicate our results. Alternative ways of studying maintenance do focus on human beings: like programmer observation and using models of cognition. Statistical observation of the efficiency of a group of programmers while doing maintenance tasks can be done to summarize the effects of differences between design patterns. However, such an expensive study can not explain the causes of these effects, while our method can. The use of cognitive modeling can also shed light on the causes of complexity. With this method one explicitly constructs a representation of the knowledge that a human being is using while analyzing and modifying source code. Complexity measures for such representations exist as well and have been used to study understandability of programming in different kinds of languages. We have not opted for this approach because such detailed cognitive models are difficult to construct.
Collecting data
We now focus on the effect on runtime efficiency of moving from Visitor to Interpreter. This expectation is motivated by the fact that the scenarios above are designed to highlight different aspects of performance and to represent typical Rascal usage scenarios. The impact is measured using four programs designed both to highlight different aspects of performance and to represent typical Rascal usage scenarios: Efficiency, construct validity, and external validity.

Finally, if other quality attributes enter the scene or other refactorings are applied, our results might be invalidated. The dimension of one parallel collaborative development—as enabled by a modular architecture—is also open to discussion. This might invalidate our analysis. Naturally, our interpretation of the causes of differences is also open to discussion.

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

- **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>?

Other factors may still dominate, but that is why we compare two equivalent systems.
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>?

---

other factors may still dominate, but that is why we compare two equivalent systems.there is no proof of that – we invite you to reproduce or invalidate the results.
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>?

Other factors may still dominate, but that is why we compare two equivalent systems. There is no proof of that — we invite you to reproduce or invalidate the results we do not know.
The role of Rascal

- Integration with Eclipse Java front-end
- Relations and trees to model abstract facts about Java
- Pattern matching and visit to analyze these models checking conditions
- Templates to generate new code
- < two man-weeks of work
- Opportunity to do it again, and again, and again!
“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)

There is no lack of theories in software engineering...

There is a lack of good experimental research methods that can (in)validate them
(Daily Painting: Seascape, Message in a Bottle? painting by artist Nancy Pouche)
Rascal is a domain specific programming language for software tools

(Daily Painting: Seascape, Message in a Bottle? painting by artist Nancy Pouche)
Rascal is a domain specific programming language for software tools.
Rascal is a domain specific programming language for software tools

Master internships
Funding opportunities
Visit CWI
Rascal is a domain specific programming language for software tools

Master internships
Funding opportunities
Visit CWI

mailto:Jurgen.Vinju@cwi.nl
twitter:@jurgenvinju

(Daily Painting: Seascape, Message in a Bottle? painting by artist Nancy Pouche)