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 technological and methodological recipes

...we do not know what matters about these recipes

...We do not know which design choices are better.

Unsatisfactory

Vik Muniz
“The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software, and the study of these approaches.”
Theoretical and empirical methods are two sides of the same medal

Internal & external validity

Idea & truth

Elegance & relevance

Quality & Complexity

“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)
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 $3 \times 10^{10}$ 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.
The source code of “ls”

3894 lines

367 ifs

174 cases
Research methods

Prototype and demonstrate it

Study programmers

Measure source code

Time will tell

Example: structured programming

**theory:** goto’s are not needed

**practice:** goto’s are harmful, sometimes

**truth:** ????

- not convincing
- toys
- not convincing
- muddy
- not convincing
- 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

```
statement sequence
  while
    condition
      compare op: !=
        variable name: b
        constant value: 0
    body
    branch
      condition
        compare op: >
          variable name: a
          variable name: b
      if-body
        assign
          bin op op: =
            variable name: a
            variable name: b
        assign
          bin op op: =
            variable name: b
            variable name: b
      else-body
        assign
          bin op op: =
            variable name: a
            variable name: b
```

image from wikipedia.org
Fig. 2. The Composite Pattern

The Composite Pattern

Composite Pattern

Component

Leaf

Composite

+ operation()
+ add()
+ remove()
+ getChild()

Expression

IntLiteral

Addition

image from wikipedia.org
Interpreter Pattern

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, the accept method 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 ; however, the details of constructing the refactoring are out of the scope of the current paper. They can instead be found online [00]. The benefits of an automated approach are reproducing target code makes it easy to replay the refactoring during experimentation, while also allowing others to literally replicate the experiment.

---

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

---

Image from wikipedia.org created by Jing Guo Yao and licensed under the Creative Commons Attribution-ShareAlike 4.0 License.
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. 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.

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:

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 Visitor Pattern

![Visitor Pattern](http://en.wikipedia.org/wiki/Visitor_pattern)
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 entire algorithms
- Interpreter encapsulates language constructs
- Visitor’s dynamic indirection is less complex
- 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.
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.
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?

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

Unfortunately, we neither understand nor trust these models.
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
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 3.3 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.

Rascal & JDT to implement Visitor to Interpreter refactoring
“Complexity of Maintenance”

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

Precise definitions in [TOOLS2011]

Inspired by “Measuring Software Flexibility” by Mens & Eden, IEE Software 2006
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 would be comparable. We argue that this knowledge is equally needed to be able to replicate the scenarios and their measurement. If shorter but otherwise plausible meta programs are defined, this might invalidate our analysis. Naturally, our conclusions about maintainability and runtime performance may be invalidated. The dimension of parallel collaborative development—as enabled by a modular architecture—is one aspect 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 be hard to understand. Fortunately, in the case of Visitor vs. Interpreter the method bodies are practically equivalent in complexity on both sides. We do not refer to any intrinsic details of the syntax and semantics of Rascal.

It is unknown what the effect of not having these tools would be on the case of the Rascal framework. We do expect that if the current study were replicated on different AST constructs, the results might have an unpredictable impact on our results. In terms of construct validity, we hope to have provided enough detail for the reader to understand that a programmer needs of the particular program before she can decide what to look for and what to change. We argue that this knowledge is equally needed to be able to replicate the scenarios and their measurement. If shorter but otherwise plausible meta programs are defined, this might invalidate our analysis. Naturally, our conclusions about maintainability and runtime performance may be invalidated. The dimension of parallel collaborative development—as enabled by a modular architecture—is one aspect 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 be hard to understand. Fortunately, in the case of Visitor vs. Interpreter the method bodies are practically equivalent in complexity on both sides. We do not refer to any intrinsic details of the syntax and semantics of Rascal.

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

<table>
<thead>
<tr>
<th>S</th>
<th>Visitor</th>
<th>(COM)</th>
<th>Interpreter</th>
<th>(COM)</th>
<th>Vis. &gt; Int.</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>(c_{i_1}^{11}(g^2a)^2)</td>
<td>(18)</td>
<td>(m^2b(ef^2)^3(ga)^2)</td>
<td>(16)</td>
<td>yes</td>
</tr>
<tr>
<td>S1(N)</td>
<td>(c_{i_1}^{11}(g^2a)^2)</td>
<td>(14 + 2N)</td>
<td>(m^2b(ef^2)^3(ga)^2)</td>
<td>(16)</td>
<td>yes</td>
</tr>
<tr>
<td>S1'(N,2)</td>
<td>(c_{i_1}^{11}(g^2a)^2)</td>
<td>(14 + 2N)</td>
<td>(m^2b(ef^2)^3(ga)^2)</td>
<td>(16)</td>
<td>yes</td>
</tr>
<tr>
<td>S1'(N,M)</td>
<td>(c_{i_1}^{11}(g^2a)^2)</td>
<td>(14 + 2N)</td>
<td>(m^2b(ef^2)^3(ga)^2)</td>
<td>(16)</td>
<td>yes</td>
</tr>
<tr>
<td>S2</td>
<td>(i_2g^3iga)</td>
<td>(8)</td>
<td>(i_2g^3iga)</td>
<td>(14)</td>
<td>no</td>
</tr>
<tr>
<td>S3</td>
<td>(d_{g^5}eg_{c_5}(g^2a(eea)^4h(ga)^3)</td>
<td>(43)</td>
<td>(d_{g^5}eg_{c_5}(g^2a(eea)^4h(ga)^3)</td>
<td>(83)</td>
<td>no</td>
</tr>
<tr>
<td>S3'</td>
<td>(d_{g^5}eg_{c_5}(g^2a(eea)^4h(ga)^3)</td>
<td>(70)</td>
<td>(d_{g^5}eg_{c_5}(g^2a(eea)^4h(ga)^3)</td>
<td>(83)</td>
<td>no</td>
</tr>
</tbody>
</table>

break-even at \(N = 14\)
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.
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