Domain-Specific Tooling

Oscar Nierstrasz
Software Composition Group
scg.unibe.ch
Agile Software Assessment
Developers spend more time reading than writing code
There is a gap between Models and Code
The architecture

... is not in the code
Specialized analyses require custom tools
Moose is a platform for modeling software artifacts to enable software analysis. Moose has been developed for well over a decade. It is the work of dozens of researchers, and has been the basis of numerous academic and industrial projects.
System complexity - Clone evolution view
Class blueprint - Topic Correlation Matrix - Distribution Map
for topics spread over classes in packages
Hierarchy Evolution view - Ownership Map
Mondrian Demo

Demo: visualizing name cohesion within packages

Agile Modeling
Moose is a powerful tool once we have a model …

Nierstrasz et al. The Story of Moose. ESEC/FSE 2005. DOI: 10.1145/1095430.1081707
The key bottleneck to assessment is creating a suitable model for analysis. If a tool does not already exist, it can take days, weeks or months to parse source files and generate models.
Developing a parser for a new language is a big challenge. Parsers may be hard to scavenge from existing tools. Not only source code, but other sources of information, like bug reports and emails can be invaluable for model building. Few projects today are built using a single language. Often a GPL is mixed with scripting languages, or even home-brewed DSLs.
Ideas

Grammar Stealing

Stealing an existing tool

Recycling Trees

Evolutionary Grammar Generation

Hooking into an existing tool

Parsing by Example
Agile Modeling Lifecycle

1. Build a coarse model
2. Refine the model
3. Build a custom analysis
Idea: use island grammars to extract coarse models

class Shape
  int x;
  int y;

  method draw() ... end
end

method main() ... end

'class' ID
  (method / . {avoid})*
'end'

method?

.method

. {avoid}
Problem: island grammars lead to shipwrecks

```ruby
class Shape
  method
  end
Tweaking island grammars till they work is not an option …

'class' ID
  (method / !'end' !method)*
'end'
method?
```
A Bounded Sea searches for an island in a bounded scope

'class' ID (~method~)* 'end'

method?

Jan Kurs, et al. Bounded Seas: Island Parsing Without Shipwrecks. SLE 2014. DOI: 10.1007/978-3-319-11245-9_4
Architectural Monitoring
Challenge

“What will my code change impact?”

Large software systems are so complex that one can never be sure until integration whether certain changes can have catastrophic effects at a distance.
Ideas: Tracking Software Architecture; exploiting Big Software Data
Problems

SA is not in the code

Diverse views of SA

The IDE focuses on code
Uncovering “Software Architecture in the Wild”

Architecture monitoring (beyond dependencies)
What is SA in the Wild?

Survey on Architectural Constraints

In the following survey you will be asked to estimate the relevance of several architectural constraints. At each step you will be presented with:

- an architectural concern: an interest which pertains to the system’s development, its operation or any other aspects that are critical or otherwise important to one or more stakeholders.
- some constraint examples related to the specified architectural concern.
- questions related to constraints associated to the specified architectural concern.

The evaluation is subjective and should be based on your personal professional experience. Before expressing your evaluation, please make sure you carefully read and understand the examples given for each constraint. All information obtained from this study will be kept anonymous.

Motivation

The architecture of a software system consists of design constraints that guarantee non-functional properties, such as ease of evolution, good run-time performance, and rapid build times. Architecture is rarely explicit in code and alignment between intended and actual architecture is often very difficult to validate and requires consistent human effort. There have been several attempts to address this problem (e.g. component dependencies: JDepend, Sotagraph, Structure 101; performance: JMeter) but there are still many aspects that remain ignored. Our goal is to develop new ways for expressing and verifying architectural constraints. We envision a set of tools that help identifying implementation solutions that break specified architectural requirements. This questionnaire will help us to identify the most relevant and commonly specified aspects of an architecture. This will provide a solid starting point for our research project.

Thanks for your collaboration.

Impact of SA constraints

<table>
<thead>
<tr>
<th>constraint</th>
<th>Impact (1-5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>availability</td>
<td>4.2</td>
</tr>
<tr>
<td>response-time</td>
<td>4.0</td>
</tr>
<tr>
<td>authorization</td>
<td>3.9</td>
</tr>
<tr>
<td>authentication</td>
<td>3.6</td>
</tr>
<tr>
<td>communication</td>
<td>3.4</td>
</tr>
<tr>
<td>throughput</td>
<td>3.4</td>
</tr>
<tr>
<td>signature</td>
<td>3.4</td>
</tr>
<tr>
<td>software infrastructure</td>
<td>3.3</td>
</tr>
<tr>
<td>data integrity</td>
<td>3.3</td>
</tr>
<tr>
<td>recoverability</td>
<td>3.1</td>
</tr>
<tr>
<td>dependencies</td>
<td>3.1</td>
</tr>
<tr>
<td>visual design</td>
<td>3.0</td>
</tr>
<tr>
<td>data retention policy</td>
<td>3.0</td>
</tr>
<tr>
<td>hardware infrastructure</td>
<td>2.9</td>
</tr>
<tr>
<td>system behavior</td>
<td>2.9</td>
</tr>
<tr>
<td>data structure</td>
<td>2.9</td>
</tr>
<tr>
<td>event handling</td>
<td>2.9</td>
</tr>
<tr>
<td>code metrics</td>
<td>2.7</td>
</tr>
<tr>
<td>meta-annotation</td>
<td>2.6</td>
</tr>
<tr>
<td>naming conventions</td>
<td>2.6</td>
</tr>
<tr>
<td>file location</td>
<td>2.5</td>
</tr>
<tr>
<td>accessibility</td>
<td>2.5</td>
</tr>
<tr>
<td>software update</td>
<td>2.2</td>
</tr>
</tbody>
</table>
Automated Validation is not Prevalent

- authorization
- throughput
- response-time
- data retention policy
- authentication
- data integrity
- visual design
- code quality
- meta-annotation
- accessibility
- communication
- availability
- event handling
- data structure
- software infrastructure
- signature
- dependencies
- recoverability
- software update
- hardware infrastructure
- file location
- naming conventions

Avg: 40%
Architectural Rules

**Naming Conventions**

"Repository interfaces can only declare methods named `find..()`"

**Dependencies**

"Only Service classes are allowed to throw `AppException`"

**Performance**

"The rendering operation has to be completed in less than 4ms"
Rule Validation

Limited functionality
Poor usability
Dicto — a unified ADSL

Dicto Rules

... 

MyService : Website with url="http://www.abc.com/api"

MyService must HandleLoadFrom("10 users")

MyService cannot HaveResponseTimeLessThan("1000 ms")

MyService can only HandleSOAPMessages()

...
Periodic Validation
Rule Examples

Website response time
Website load testing

Dependencies

Code clones

Deadlock freeness

File Content
grep
Moldable Tools
Challenge

Build a new assessment tool in ten minutes

Custom analyses require custom tools. Building a tool should be as easy as writing a query in SQL or a form-based interface.
Problems

What tools do developers really need?

What is a unifying meta-model for tool construction?

What are appropriate meta-tools?
Analyze developer needs (!)

“Moldable” Tools (not just plug-ins)
Conventional debuggers just offer an interface to the run-time stack.
Specific Models

Mind the abstraction gap

Generic Debugger

Domain-specific Debuggers

The Moldable Debugger

Debugging Widget

* Debugging Action

Activation Predicate

Andrei Chis et al. The Moldable Debugger: A Framework for Developing Domain-Specific Debuggers. SLE 2014. DOI: 10.1007/978-3-319-11245-9_6
PetitParser

identifier
letter , (letter / digit) *

letter

digit
IdentifierParser new
parse: 'aLong32Identifier'
Default Debugger

Stack

PPStream(ReadStream) >> next
PPContext >> next
PPPredicateObjectParser >> parseOn:
PPDelegateParser >> parseOn:
PPChoiceParser >> parseOn:
PPPossessiveRepeatingParser >> parseOn:
PPSequenceParser >> parseOn:
PPDelegateParser >> parseOn:
PPEndOfInputParser >> parseOn:
PPIdentifierParser(PPDelegateParser) >> parseOn:
PPIdentifierParser(PPParser) >> parseWithContext:
PPIdentifierParser(PPParser) >> parse:withContext:
PPIdentifierParser(PPParser) >> parse:

Source

parseOn: aPPContext
  parser parseOn: aPPContext

Type | Variable | Value
--- | --- | ---
_self | | a PPDelegateParser(identifier)
_stack top | | a PPContext
_thisContext | | PPDelegateParser >> parseOn:
parser | | a PPContext
properties | | a PPSequenceParser(273678336)

43
Domain specific-extensions
Debugging widgets

Debugging actions

Next parser

Next production

Production(production)

Next failure

Stream position(anInteger)

Stream position changed
New debuggers are cheap

<table>
<thead>
<tr>
<th></th>
<th>Session</th>
<th>Operations</th>
<th>View</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base model</td>
<td>800</td>
<td>700</td>
<td>-</td>
<td>1500</td>
</tr>
<tr>
<td>Default Debugger</td>
<td>-</td>
<td>100</td>
<td>400</td>
<td>500</td>
</tr>
<tr>
<td>Announcements</td>
<td>200</td>
<td>50</td>
<td>200</td>
<td>450</td>
</tr>
<tr>
<td>Petit Parser</td>
<td>100</td>
<td>300</td>
<td>200</td>
<td>600</td>
</tr>
<tr>
<td>Glamour</td>
<td>150</td>
<td>100</td>
<td>50</td>
<td>300</td>
</tr>
<tr>
<td>SUnit</td>
<td>100</td>
<td>-</td>
<td>50</td>
<td>150</td>
</tr>
</tbody>
</table>
The Moldable Inspector
Conclusion

Current IDEs offer developers primitive support for software assessment

*Developers need support for agile modeling, architectural monitoring and moldable tools*