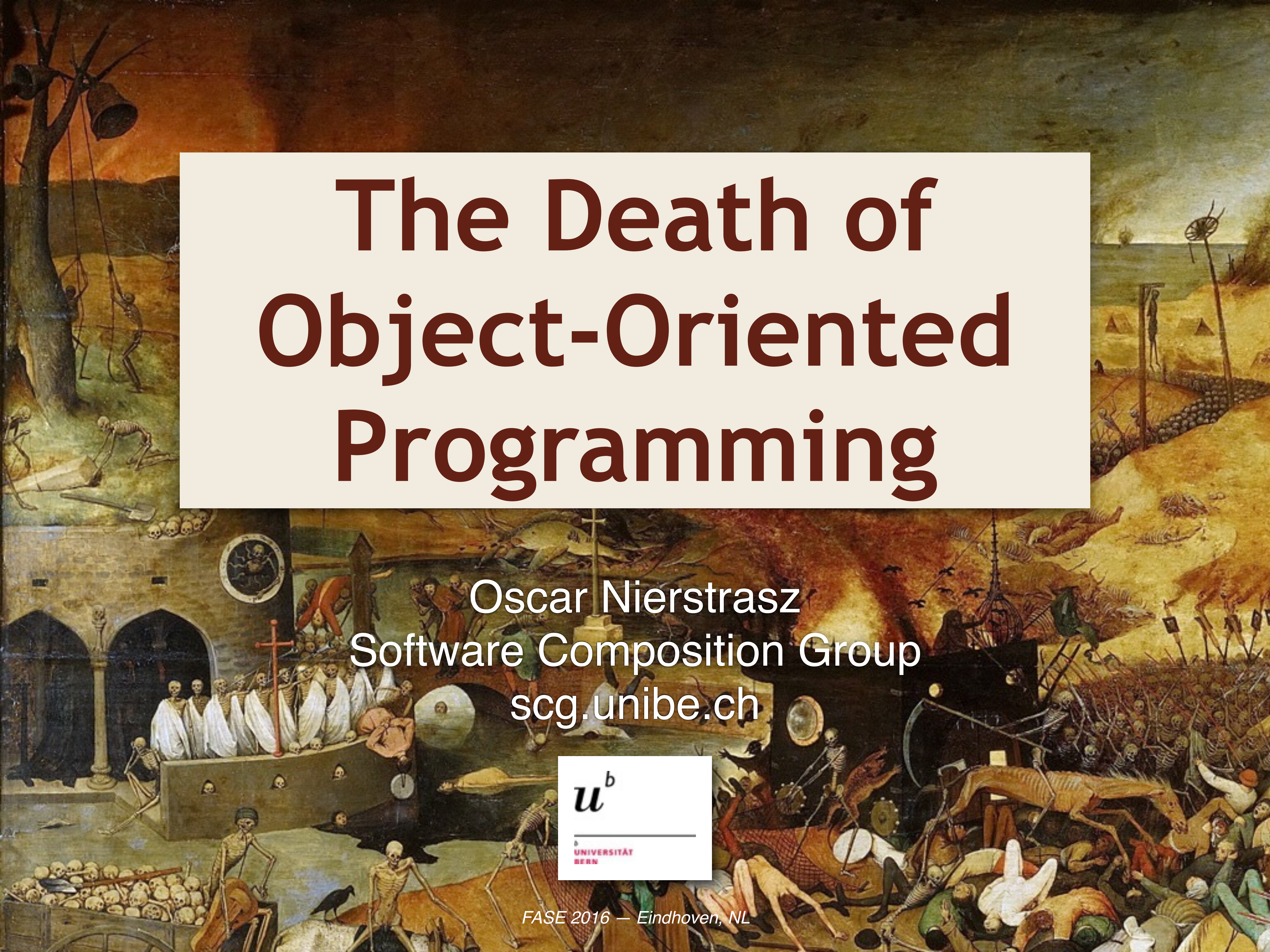


The Death of Object-Oriented Programming

Oscar Nierstrasz
Software Composition Group
scg.unibe.ch

u^b UNIVERSITÄT BERN

FASE 2016 — Eindhoven, NL

Invited talk at FASE 2016, ETAPS 2016: 2-8 April 2016, Eindhoven, The Netherlands.

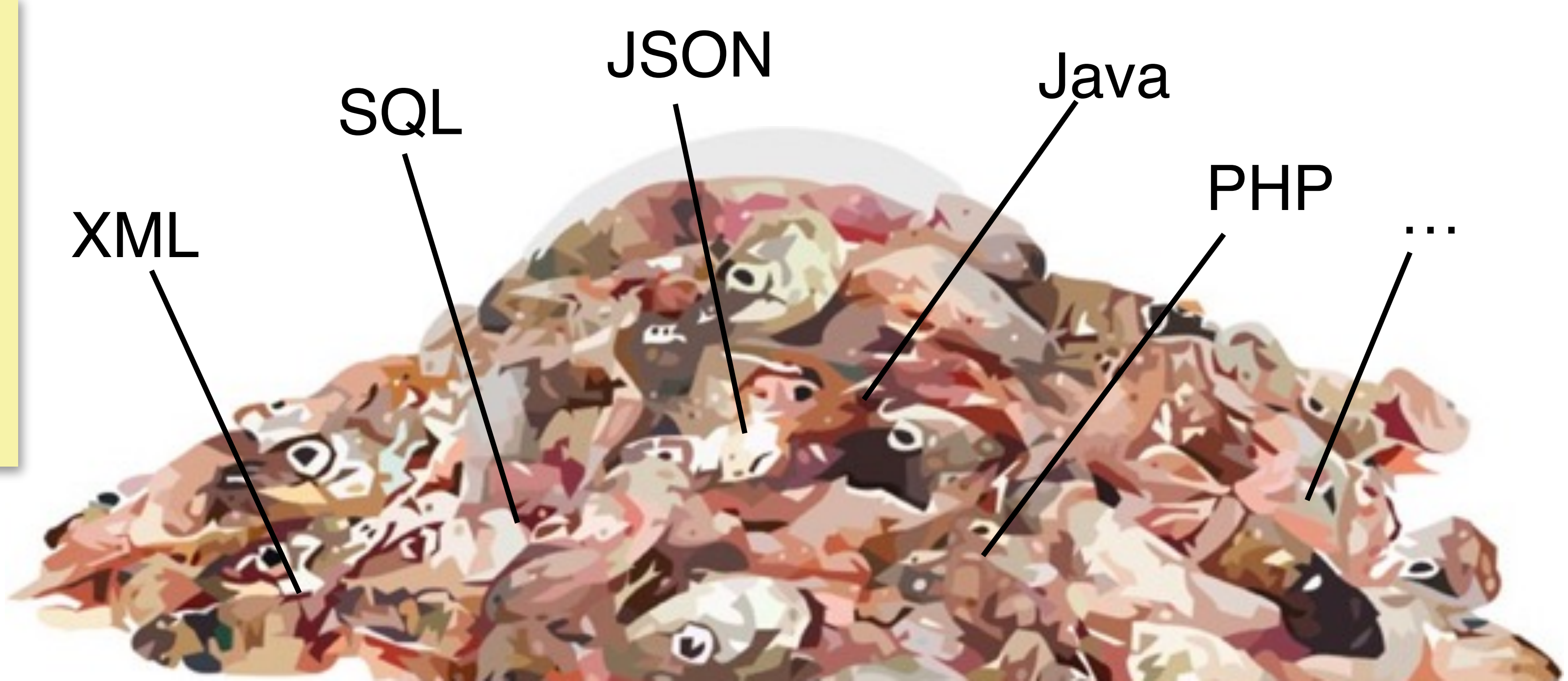
A preprint of the companion paper is available online:

http://scg.unibe.ch/scgbib?query=Nier16a

The trouble with OOP

OOP promised us graceful transition from domain models to implementation

Instead modern applications consist of a heterogeneous pile of different technologies



In the 1980s, one of the selling points of OOP was that object-oriented models could be used consistently from domain modelling, through analysis and design, all the way through to implementation.

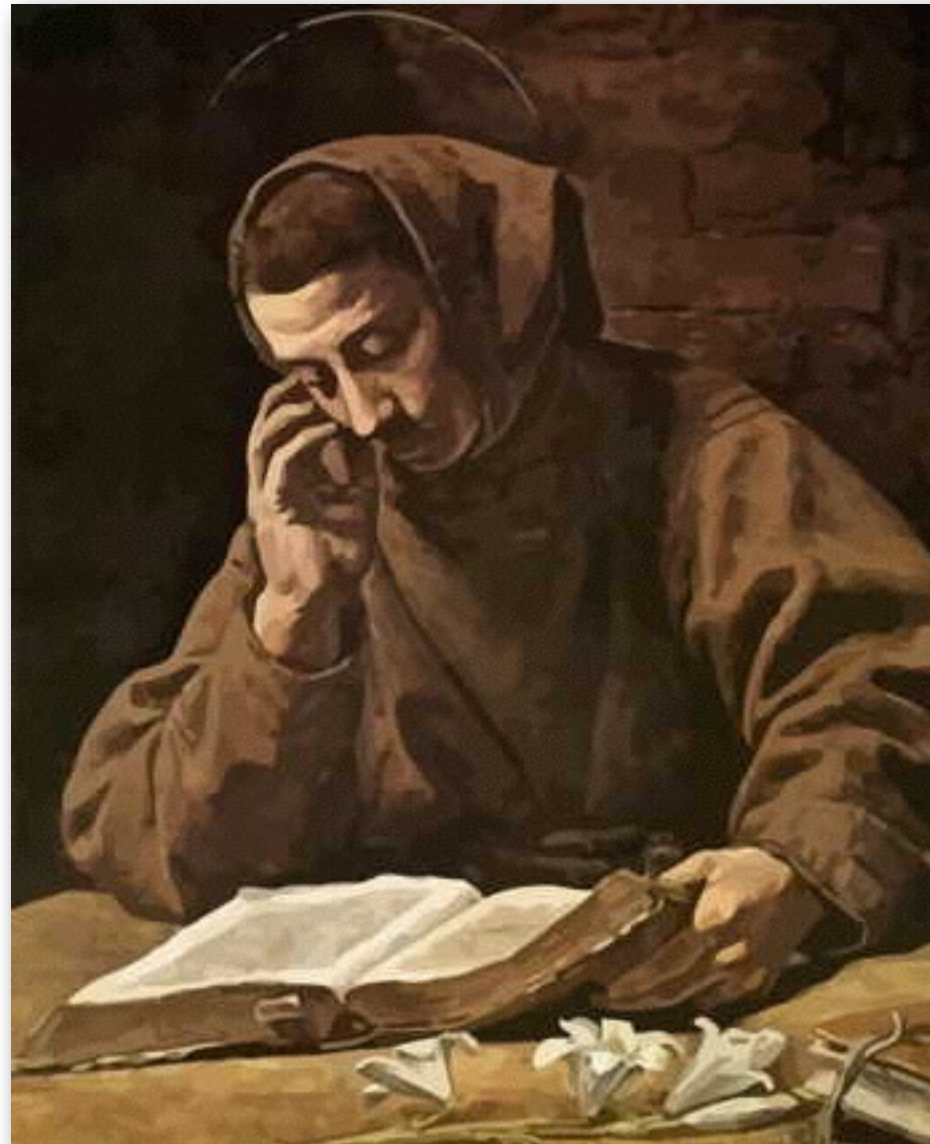
Somewhere along the way this vision has been lost, and now we see modern software systems built from a heterogeneous sludge of different programming languages, configuration languages and domain-specific languages addressing both application and technical domains.

There is a *gap* between Models

and Code

Important aspects of the model are often missing in the code. This makes it harder to make sure that changes are consistent. Architecture and particularly architectural constraints are typically not explicit.

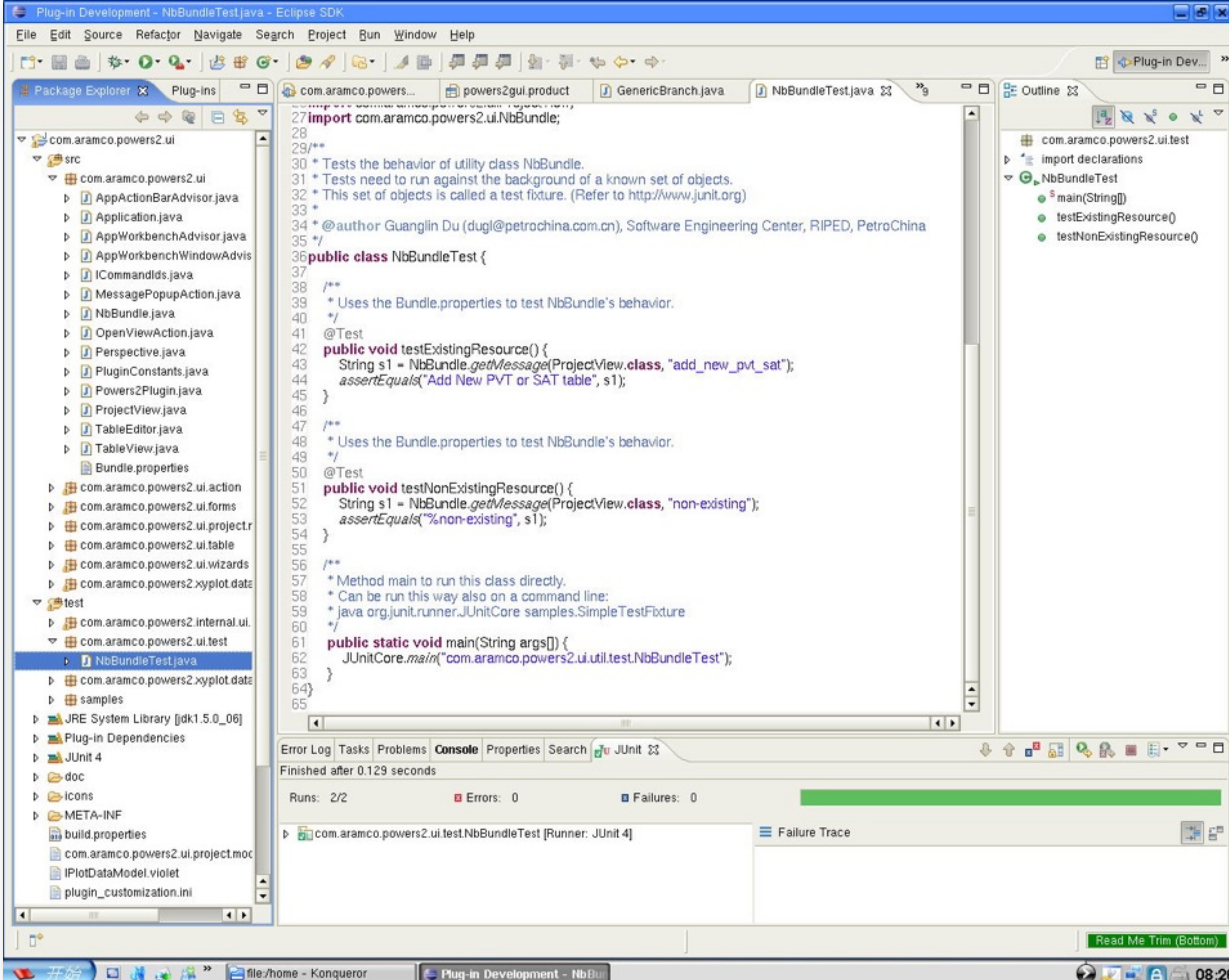
The programming language may get in the way — boilerplate code can obfuscate intent. Dependencies are often hidden, so it can be unclear what will be the impact of a change. Furthermore, the development context and project history is not part of the code at all.



Developers spend more time reading than writing code

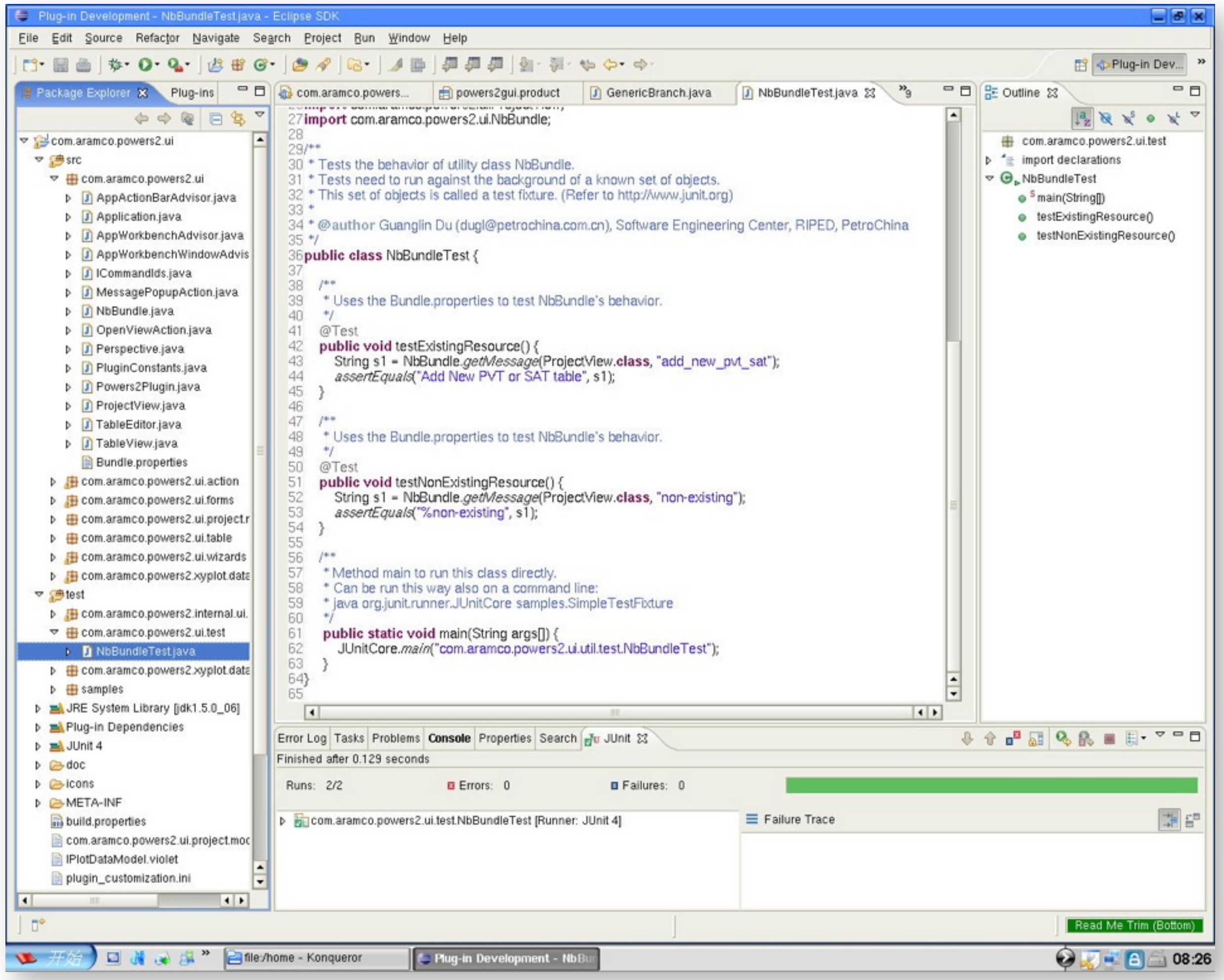
Especially with OO, where the code does not reflect run time behaviour well, more time is spent reading than writing, to understand code and to understand impact of change.

IDEs do not well support this reading activity, since they focus on PL concepts, like editing, compiling and debugging, not architectural constraints, or user features.



Plug-in Development - NbBundleTest.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window Help
Plug-in Dev...
Package Explorer
Plug-ins
com.aramco.powers2.ui
src
com.aramco.powers2.ui
AppActionBarAdvisor.java
Application.java
AppWorkbenchAdvisor.java
AppWorkbenchWindowAdvis
ICommandIds.java
MessagePopupAction.java
NbBundle.java
OpenViewAction.java
Perspective.java
PluginConstants.java
Powers2Plugin.java
ProjectView.java
TableEditor.java
TableView.java
Bundle.properties
com.aramco.powers2.ui.action
com.aramco.powers2.ui.forms
com.aramco.powers2.ui.project.r
com.aramco.powers2.ui.table
com.aramco.powers2.ui.wizards
com.aramco.powers2.xyplot.data
test
com.aramco.powers2.internal.ui.
com.aramco.powers2.ui.test
NbBundleTest.java
com.aramco.powers2.xyplot.data
samples
JRE System Library [jdk1.5.0_06]
Plug-in Dependencies
JUnit 4
doc
icons
META-INF
build.properties
com.aramco.powers2.ui.project.moc
IPlotDataModel.violet
plugin_customization.ini
com.aramco.powers...
powers2gui.product
GenericBranch.java
NbBundleTest.java
27 import com.aramco.powers2.ui.NbBundle;
28
29 /**
30 * Tests the behavior of utility class NbBundle.
31 * Tests need to run against the background of a known set of objects.
32 * This set of objects is called a test fixture. (Refer to http://www.junit.org)
33 *
34 * @author Guanglin Du (dugl@petrochina.com.cn), Software Engineering Center, RIPED, PetroChina
35 */
36 public class NbBundleTest {
37
38 /**
39 * Uses the Bundle.properties to test NbBundle's behavior.
40 */
41 @Test
42 public void testExistingResource() {
43 String s1 = NbBundle.getMessage(ProjectView.class, "add_new_pvt_sat");
44 assertEquals("Add New PVT or SAT table", s1);
45 }
46
47 /**
48 * Uses the Bundle.properties to test NbBundle's behavior.
49 */
50 @Test
51 public void testNonExistingResource() {
52 String s1 = NbBundle.getMessage(ProjectView.class, "non-existing");
53 assertEquals("%non-existing", s1);
54 }
55
56 /**
57 * Method main to run this class directly.
58 * Can be run this way also on a command line:
59 * java org.junit.runner.JUnitCore samples.SimpleTestFixture
60 */
61 public static void main(String args[]) {
62 JUnitCore.main("com.aramco.powers2.ui.util.test.NbBundleTest");
63 }
64 }
65
Outline
com.aramco.powers2.ui.test
import declarations
NbBundleTest
main(String[])
testExistingResource()
testNonExistingResource()
Error Log Tasks Problems Console Properties Search JUnit
Finished after 0.129 seconds
Runs: 2/2
Errors: 0
Failures: 0
com.aramco.powers2.ui.test.NbBundleTest [Runner: JUnit 4]
Failure Trace
Read Me Trim (Bottom)
开始
file:/home - Konqueror
Plug-in Development - NbBu
08:26

Yet mainstream IDEs are basically glorified text editors



Can you guess from this view the application domain of the code?

IDEs offer only general-purpose tools for editing and managing code, and are typically unaware of the application domain.

Software inevitably changes …

But our programming languages and development tools and methods pretend the world is frozen!

Few, if any mechanisms *enable* change

Types, modules, namespaces all assume a frozen, unchanging snapshot of the world. Mainstream programming languages offer no specific mechanisms to enable software evolution.

(Deprecation limits the effects of change, but does not especially enable it.)

Outlook: Programming is Modeling

Instead of having disconnected models and code, or even transformations between models and code, we should consider code as *being* the models of concern.

Roadmap

Bring Models Closer to Code

Link code to the ecosystem

Exploit domain models in the IDE

Bring Models Closer to Code

What exactly is “the OO paradigm”?

The OO Paradigm is commonly (mis-)represented as:

programs = objects + messages

Or even:

programs = objects + classes + inheritance

Although technically correct, this misses the point.

OOP was invented (by Nygaard and Dahl) in the early 60s out of a need to program real-world *simulations*. The mechanisms of objects, messages, classes and inheritance realised in the Simula language (an extension of Algol) enabled them to develop the simulations they wanted. Only later did programmers realise that simulation — as a paradigm — was more generally useful in software engineering.

"Design your own paradigm"

Object-oriented programming is really about *designing your own paradigm*. You decide what domain abstractions are important for your application, and you use them to build your system.

Every OO program is a simulation of a virtual world, in which the objects you have imagined interact to realize some specific goals.

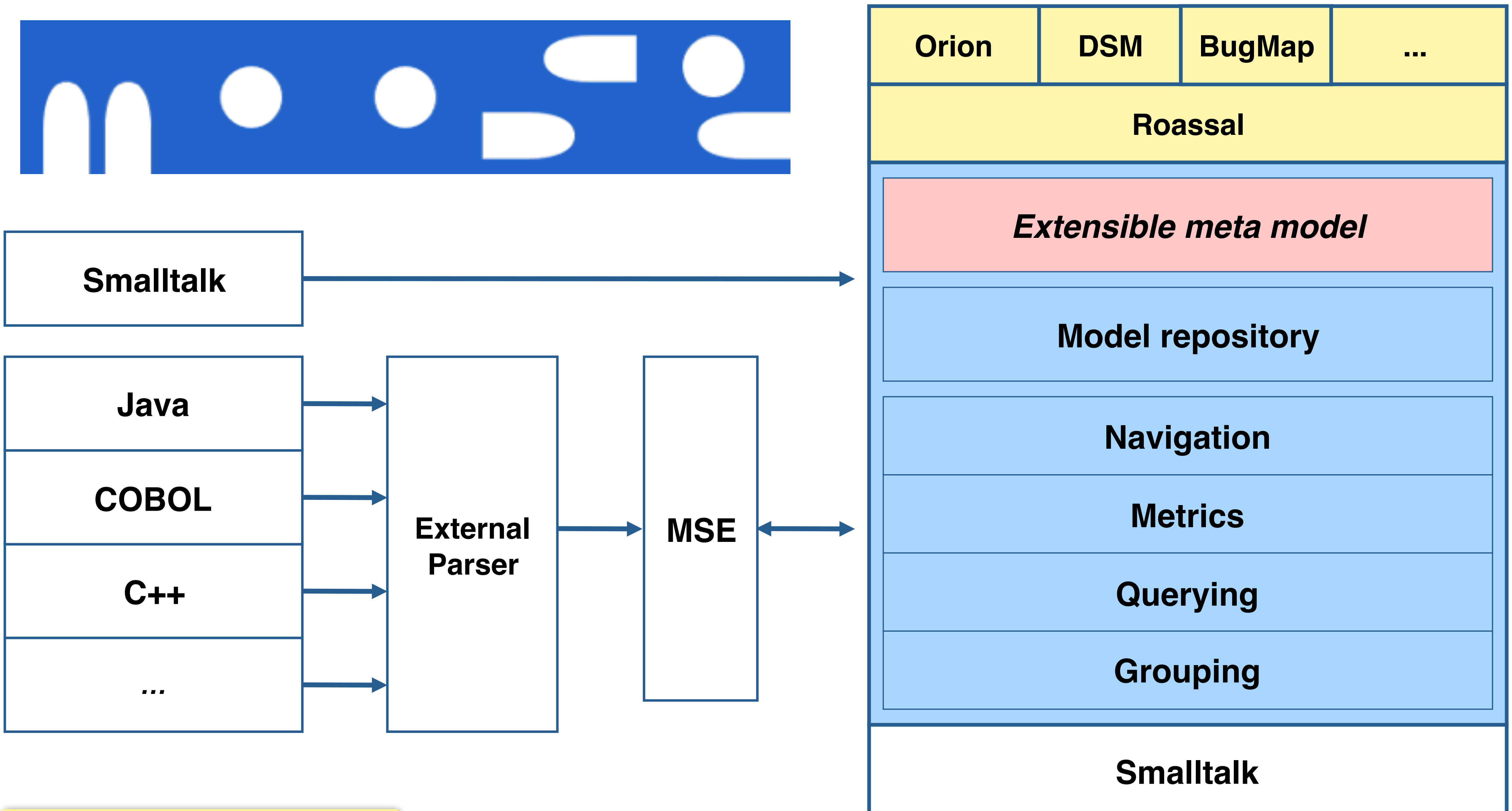
For an OOPL to succeed as a modelling language, (code) models should be *queryable* and *manipulable*

What would this mean in practice?

Developers continuously ask questions about the code they work with, but don't have good tools to formulate these questions.

If programming languages are to succeed as modeling languages, the models they are used to construct must be comprehensible, analyzable, queryable and manipulable.

Moose is a platform for software and data analysis

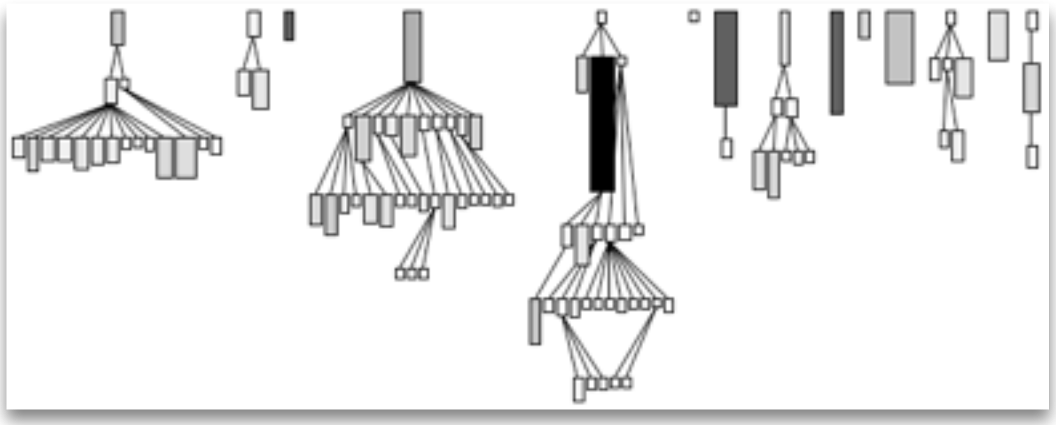


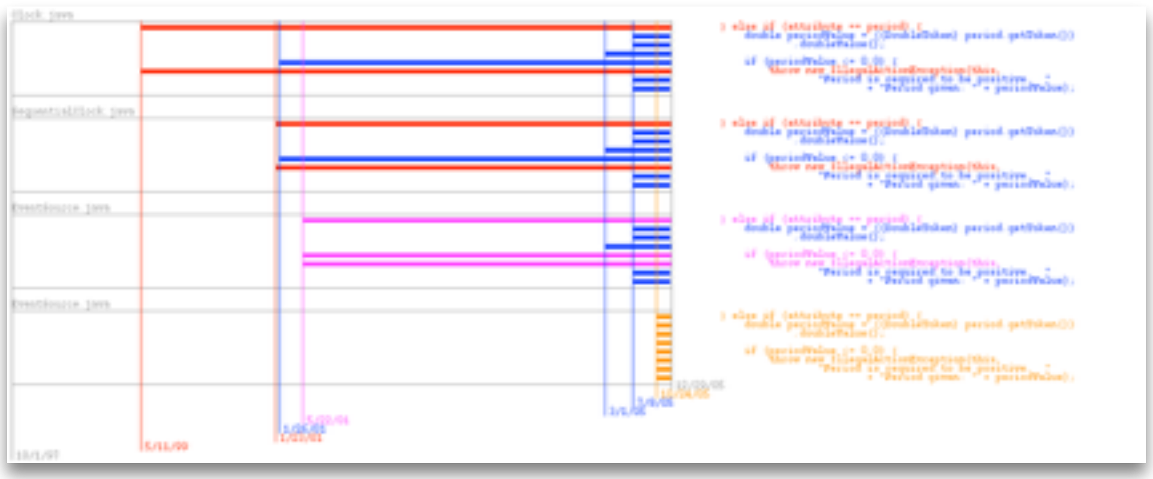
www.moosetechnology.org

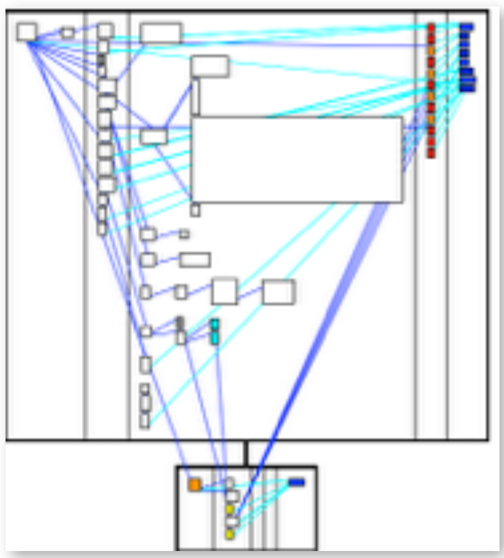
Moose is a platform for modeling software artefacts to enable software analysis. Moose offers a number of core features to navigate models, query them and analyze them. Numerous analysis and visualization tools have been developed on top of Moose.

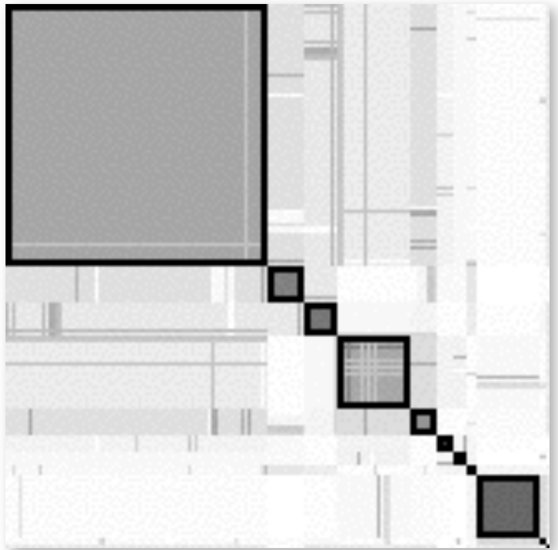
Moose has been developed for well over a decade. It is the work of dozens of researchers, and has been the basis of numerous academic and industrial projects.

www.moosetechnology.org

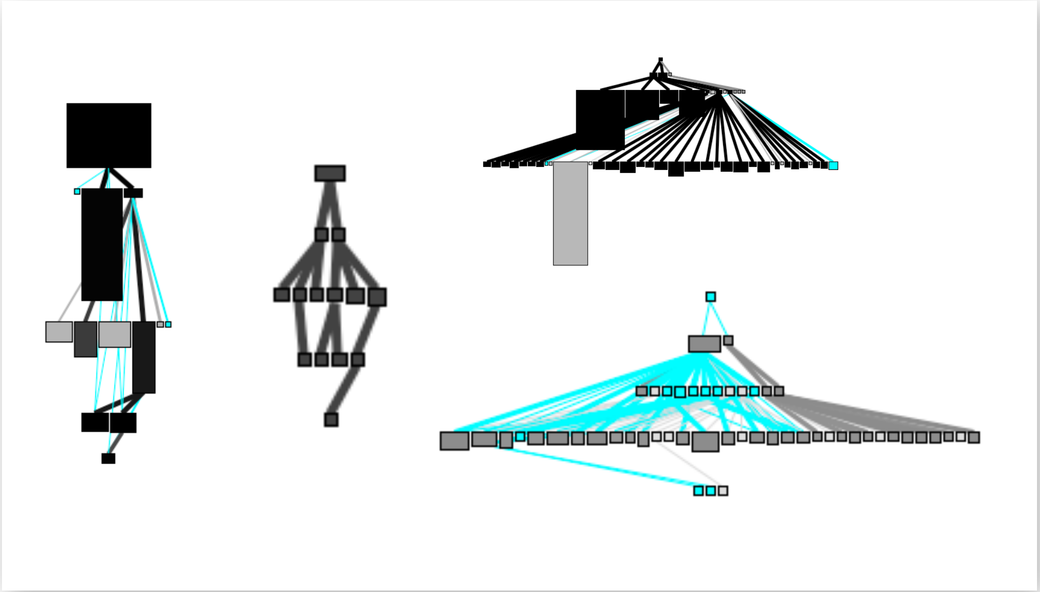


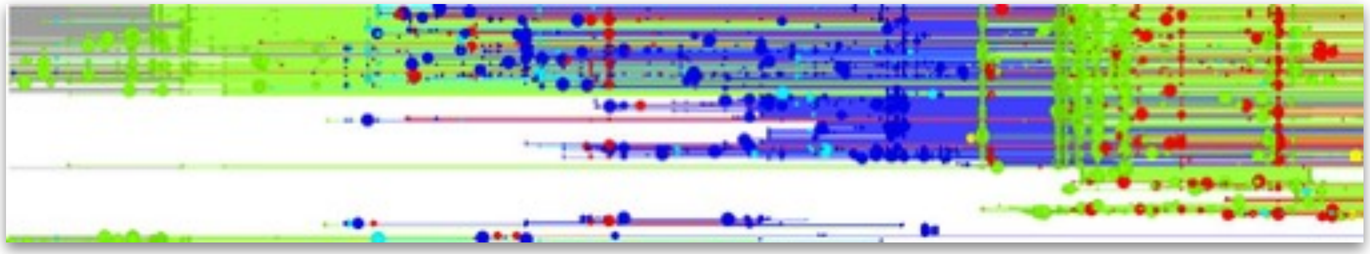







menu
util
menu
browser
macos
search
installer
options
gui
help
bsh
syntax
buffer
print
textarea
xml
asm



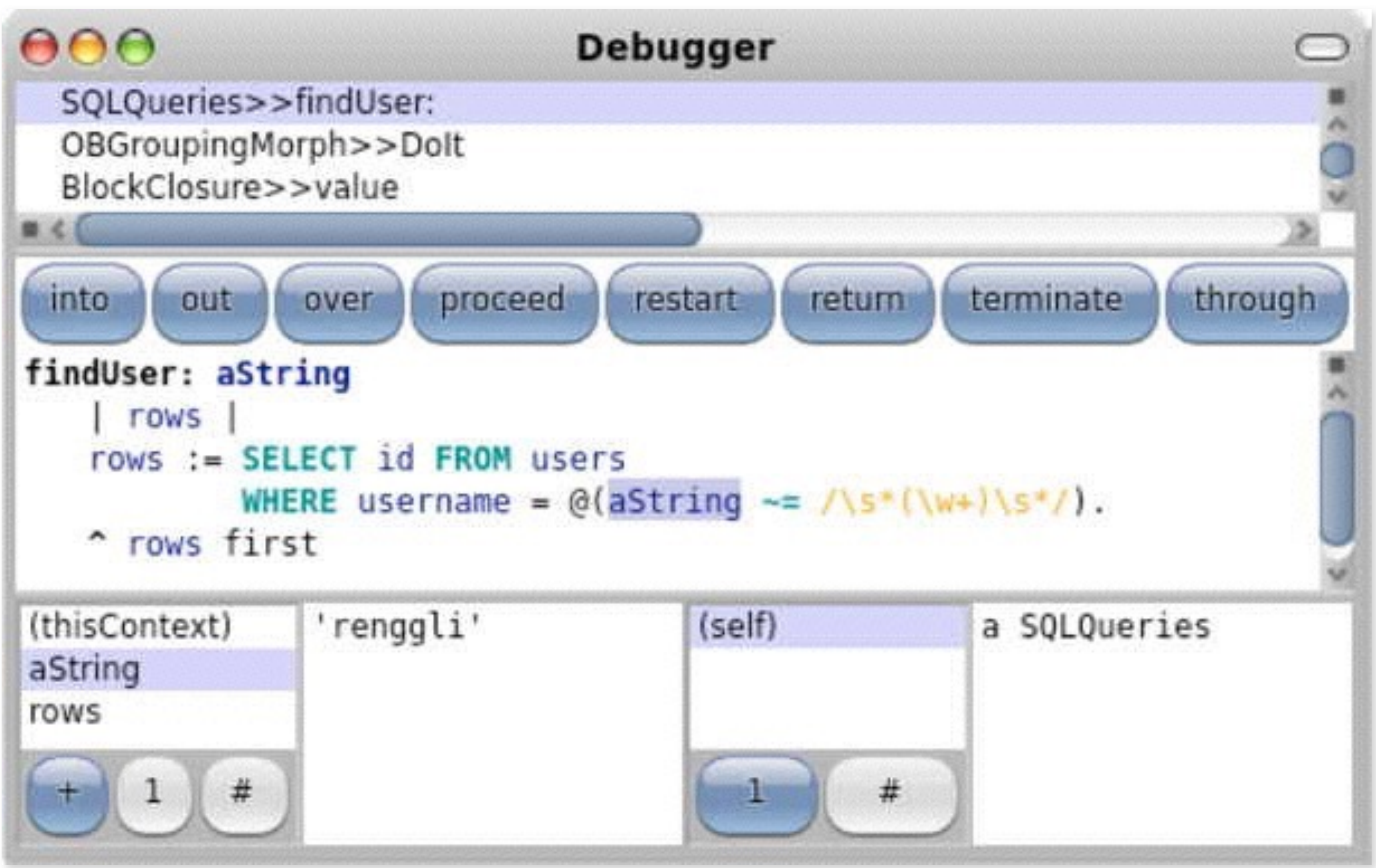


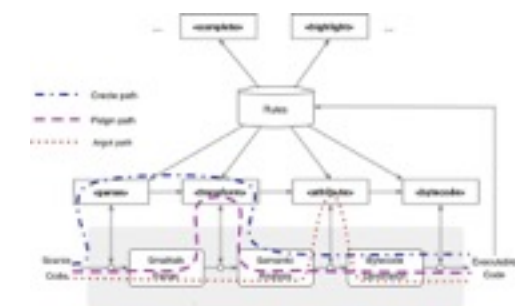
The figure shows the following visualisations:

- First row: System complexity (class hierarchy decorated with metrics) - Clone evolution view
- Second row: Class blueprint (shows relationships between methods and attributes within a class) - Topic Correlation Matrix - Distribution Map (for topics spread over classes in packages)
- Third row: Hierarchy Evolution view (shows histories of classes) - Ownership Map (shows ownership of artefacts over time)

Although Moose is a powerful and expressive platform, it still requires that models be imported from a code base. The close integration of the development environment and analysis tools is still missing.

How to make tools understand DSLs?





Renggli et al., ***Embedding Languages without Breaking Tools***. *ECOOP 2010*

Domain-specific languages help to maintain the link between models and code.

Unfortunately such language extensions typically do not play well with the IDE.

Here we see SQL and regexes as extensions to Smalltalk, with syntax highlighting integrated into the development tools.

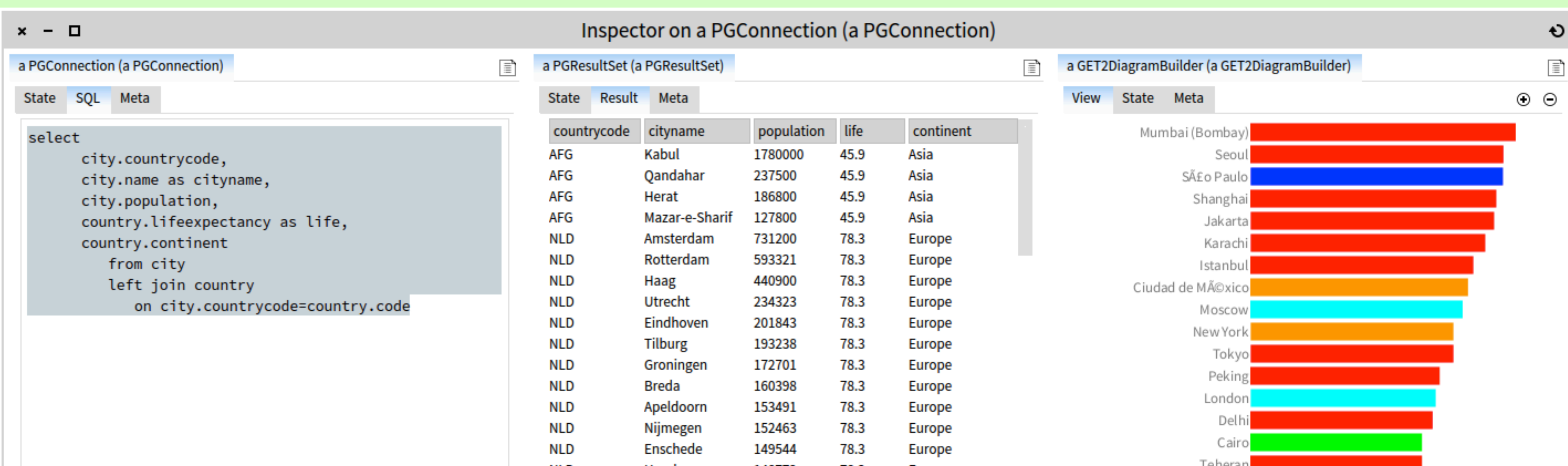
Renggli et al., Embedding Languages without Breaking Tools. ECOOP 2010

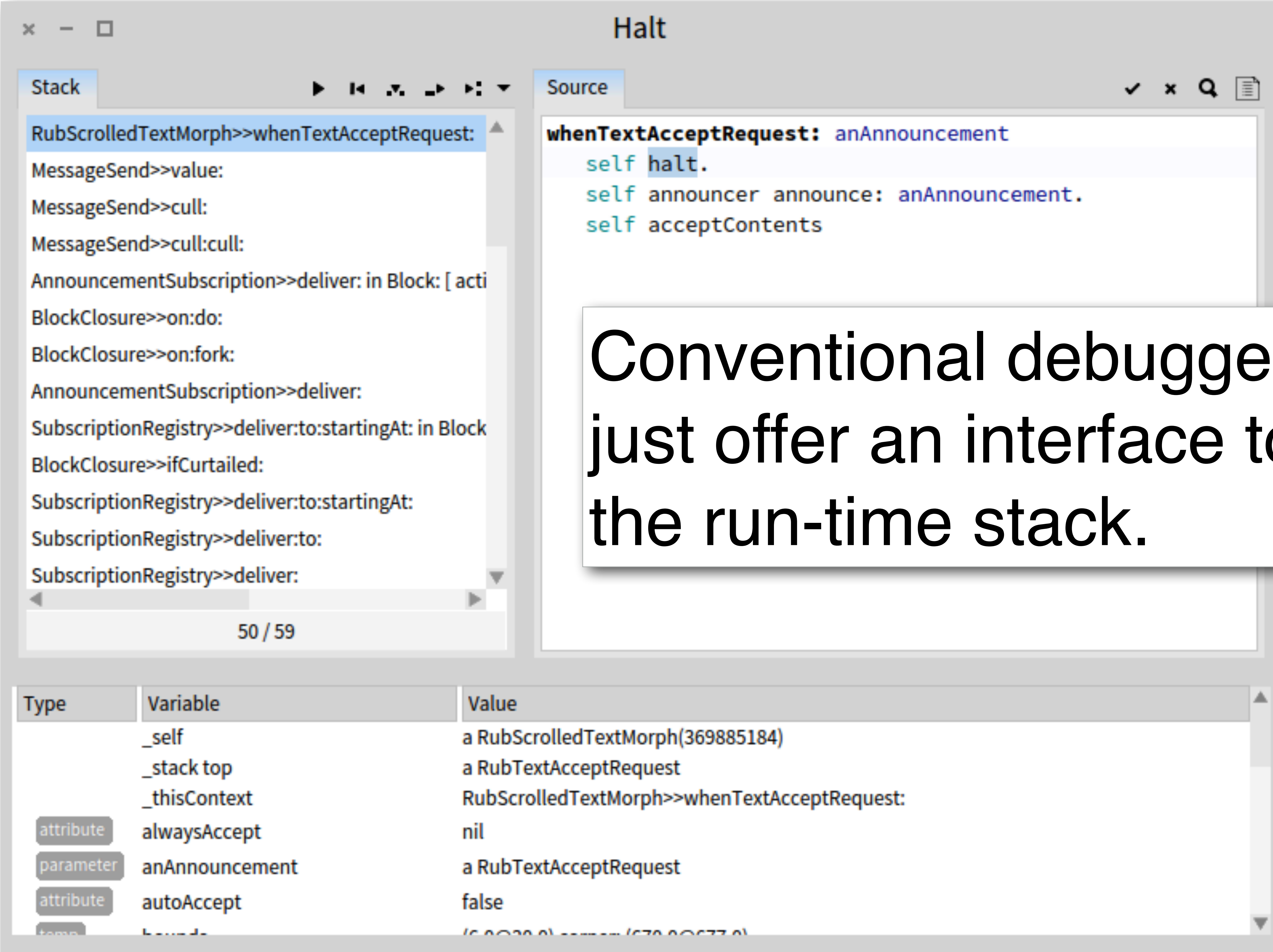
Outlook: models = code

Rather than modeling code, we need the code to *be* the model. (This Lego town is both a model of a town, and it *is* a toy town at the same time.)

Bertrand Meyer says he was long puzzled by the fascination with modeling notations and CASE tools, until he realized one day their attraction: *"Bubbles and arrows don't crash."*

Exploit domain models in the IDE



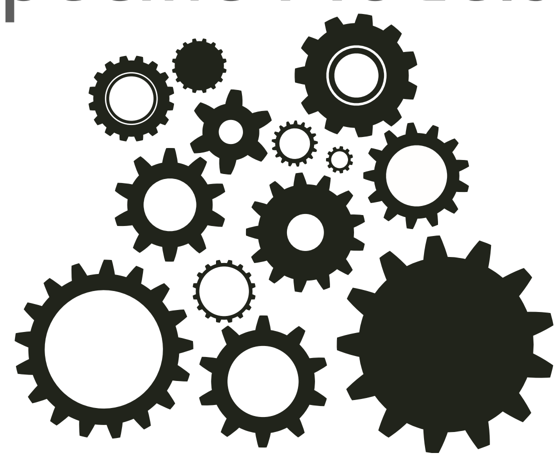


Conventional debuggers just offer an interface to the run-time stack.

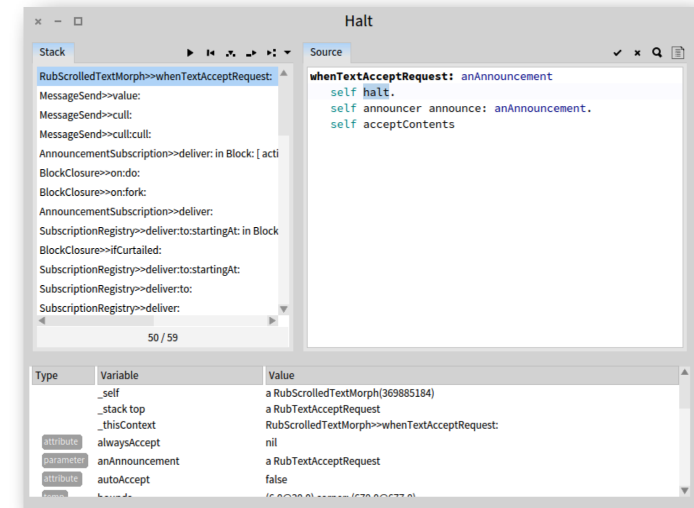
The debugger is just one example of a classical IDE tool that knows nothing about your specific application domain. It just offers generic functionality that often does not fit well the needs of a particular domain.

Moldable Tools

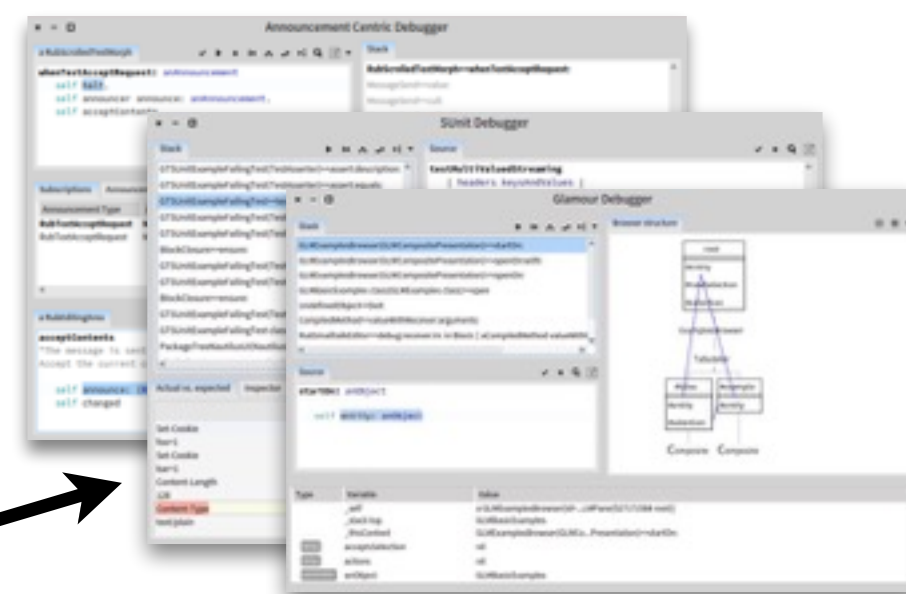
Specific Models



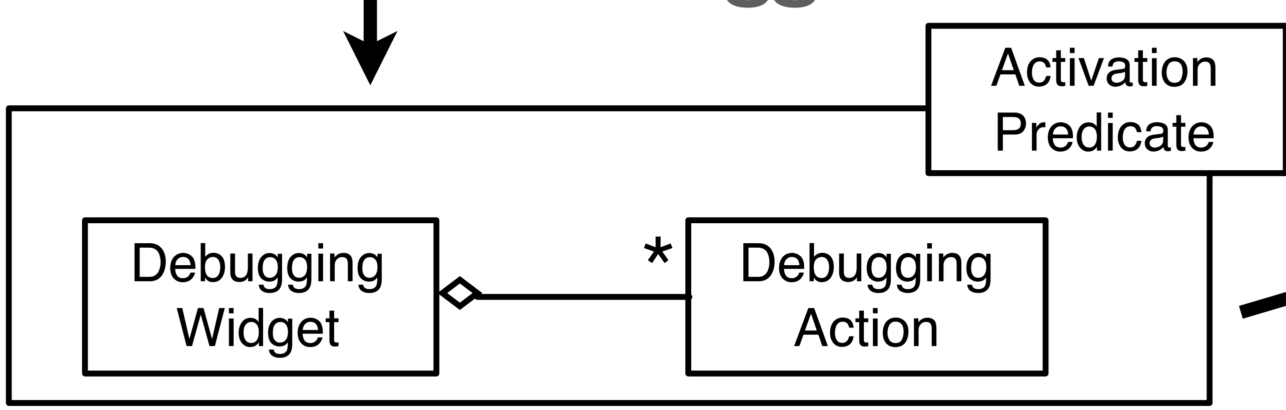
Generic Debugger



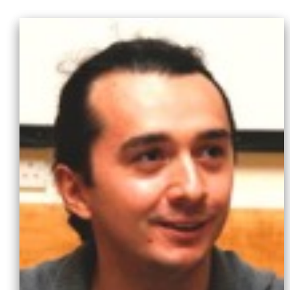
Domain-specific Debuggers



The Moldable Debugger



Andrei Chis et al. **The Moldable Debugger: A Framework for Developing Domain-Specific Debuggers.** SLE 2014.

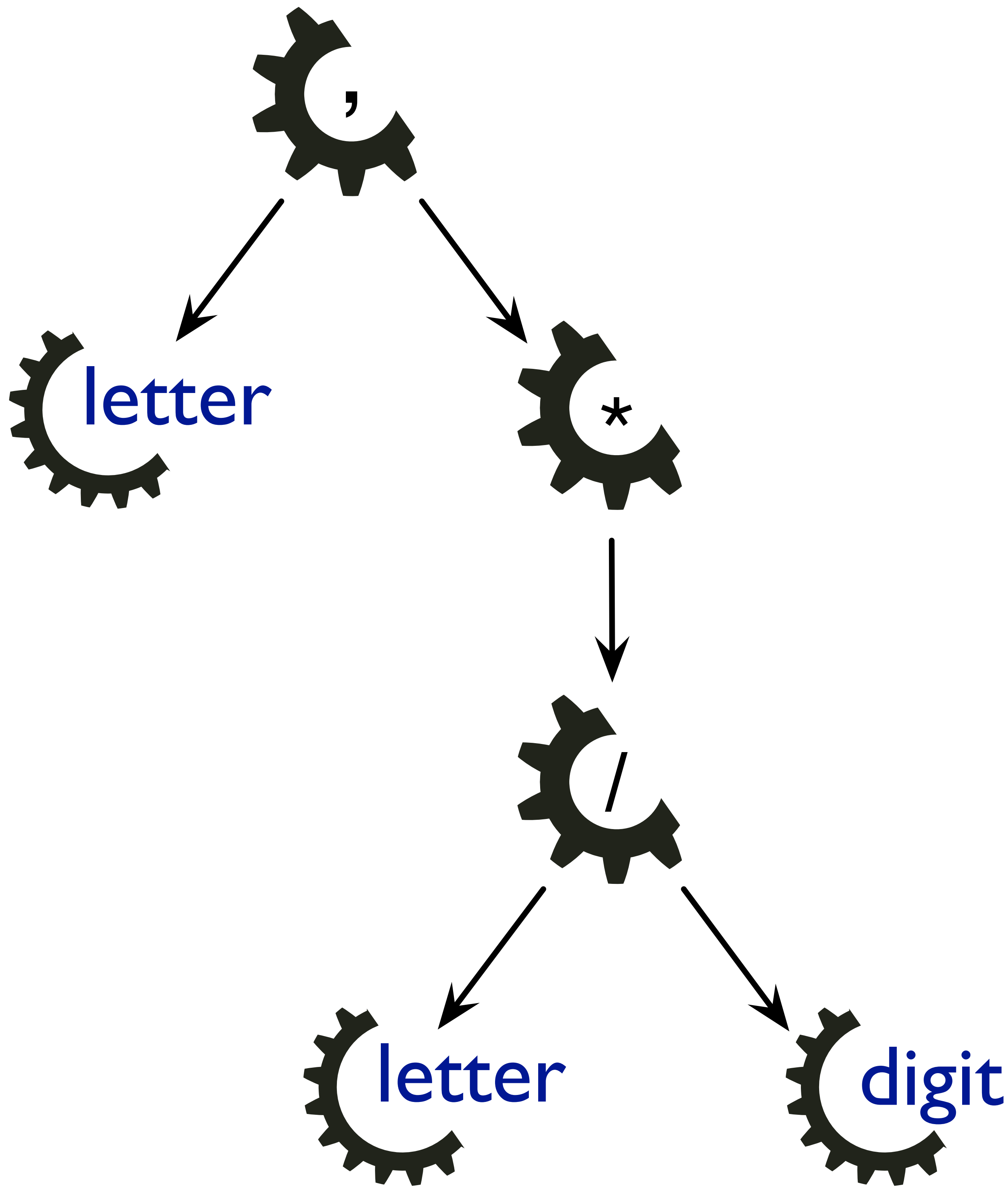


Classical development tools like browsers, debuggers and inspectors are generic and do not address the needs of specific domains.

The Moldable debugger can be easily adapted to different domains, such as event-driven computation, GUI construction and parser generation.

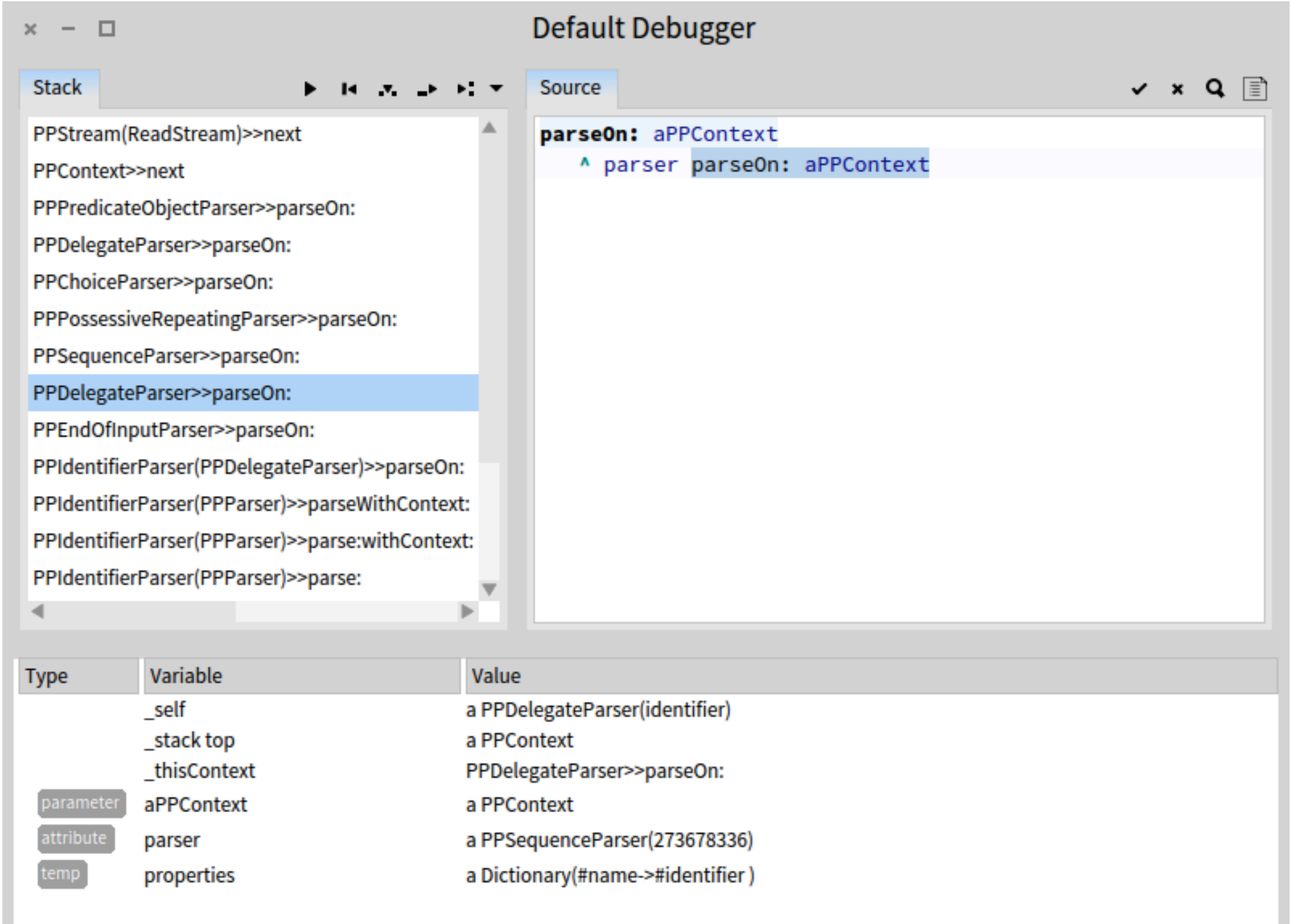
Andrei Chis et al. The Moldable Debugger: A Framework for Developing Domain-Specific Debuggers. SLE 2014.

PetitParser

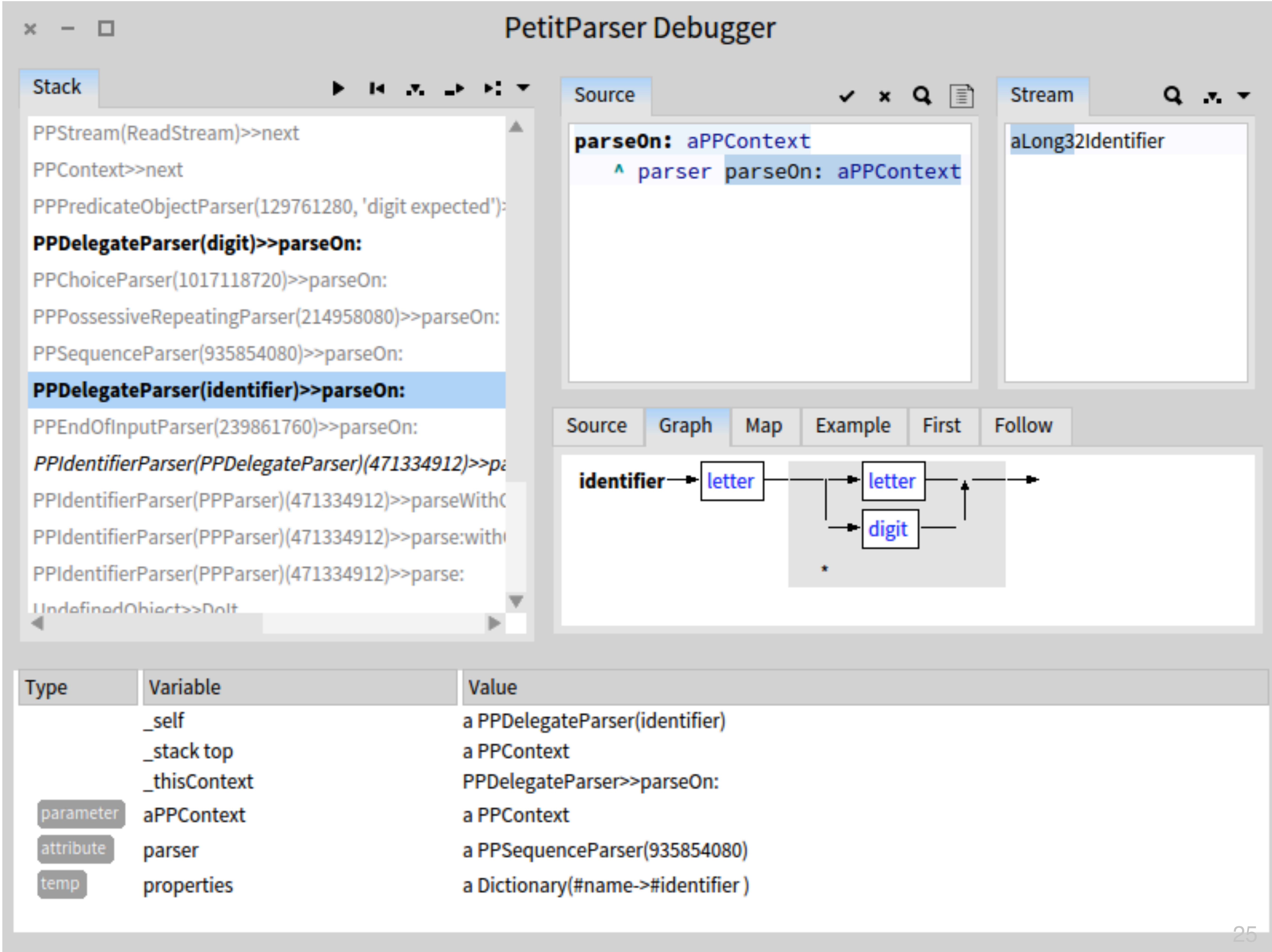


identifier
letter , (letter / digit) *

PetitParser is a PEG-based framework for developing parsers composed of objects.

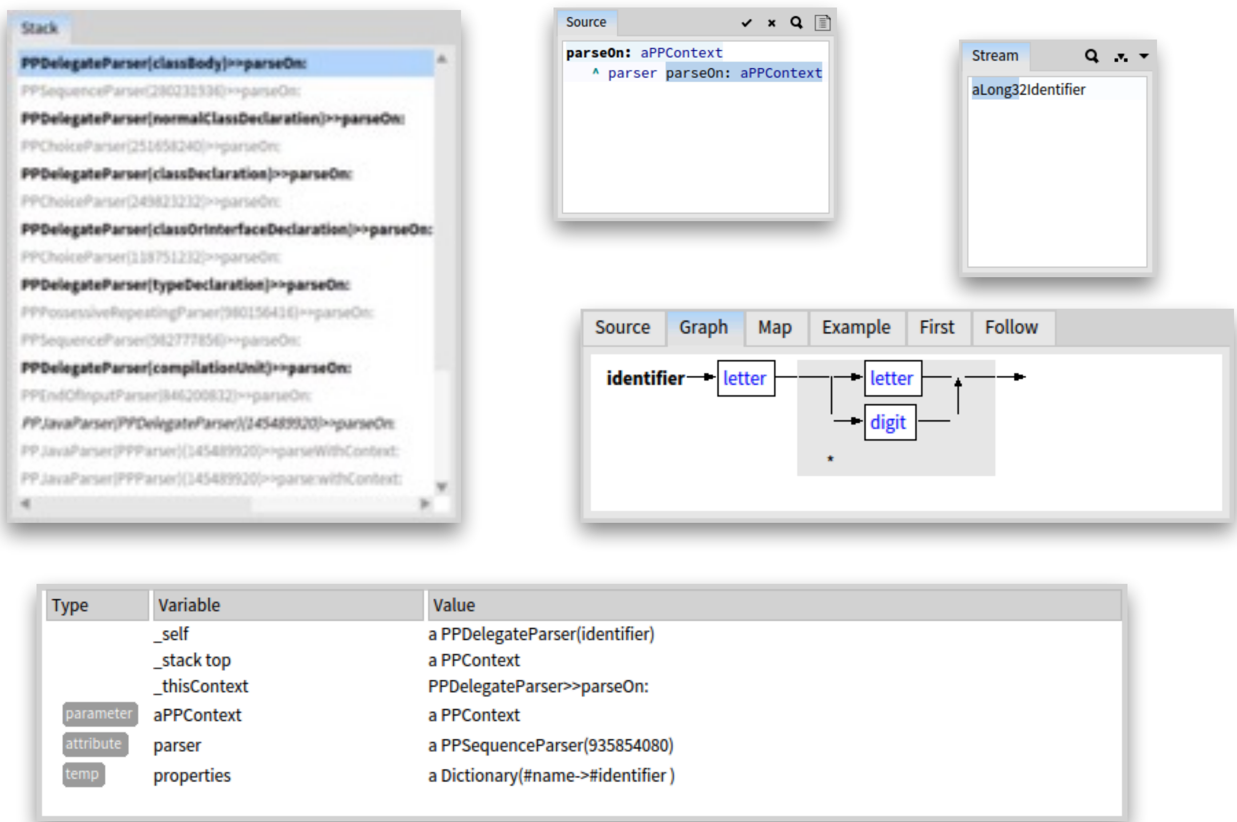

Default Debugger
Stack
PPStream(ReadStream)>>next
PPContext>>next
PPPredicateObjectParser>>parseOn:
PPDelegateParser>>parseOn:
PPChoiceParser>>parseOn:
PPPossessiveRepeatingParser>>parseOn:
PPSequenceParser>>parseOn:
PPDelegateParser>>parseOn:
PPEndOfInputParser>>parseOn:
PPIdentifierParser(PPDelegateParser)>>parseOn:
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:
parameter
aPPContext
a PPContext
attribute
parser
a PPSequenceParser(273678336)
temp
properties
a Dictionary(#name->#identifier)

A conventional debugger knows nothing about the parsing domain. Here we see the Pharo Smalltalk debugger with a view of the run-time stack at the left, the source code of the selected method at right, and the currently accessible local variables at the bottom.


PetitParser Debugger
Stack
PPStream(ReadStream)>>next
PPContext>>next
PPPredicateObjectParser(129761280, 'digit expected')
PPDelegateParser(digit)>>parseOn:
PPChoiceParser(1017118720)>>parseOn:
PPPossessiveRepeatingParser(214958080)>>parseOn:
PPSequenceParser(935854080)>>parseOn:
PPDelegateParser(identifier)>>parseOn:
PPEndOfInputParser(239861760)>>parseOn:
PPIdentifierParser(PPDelegateParser)(471334912)>>pa
PPIdentifierParser(PPParser)(471334912)>>parseWith
PPIdentifierParser(PPParser)(471334912)>>parse:with
PPIdentifierParser(PPParser)(471334912)>>parse:
Source
parseOn: aPPContext
^ parser parseOn: aPPContext
Stream
aLong32Identifier
Source
Graph
Map
Example
First
Follow
identifier
letter
letter
digit
*
Type
Variable
Value
_self
a PPDelegateParser(identifier)
_stack top
a PPContext
_thisContext
PPDelegateParser>>parseOn:
parameter
aPPContext
a PPContext
attribute
parser
a PPSequenceParser(935854080)
temp
properties
a Dictionary(#name->#identifier)

A moldable PetitParser debugger knows which objects are parsers, knows where we are in the input, and can show us which parser object is currently active. Instead of being forced to laboriously step through methods to find what we are looking for, we can step directly to the next grammar rule of interest.

Debugging widgets



Debugging actions

Next parser

Next production

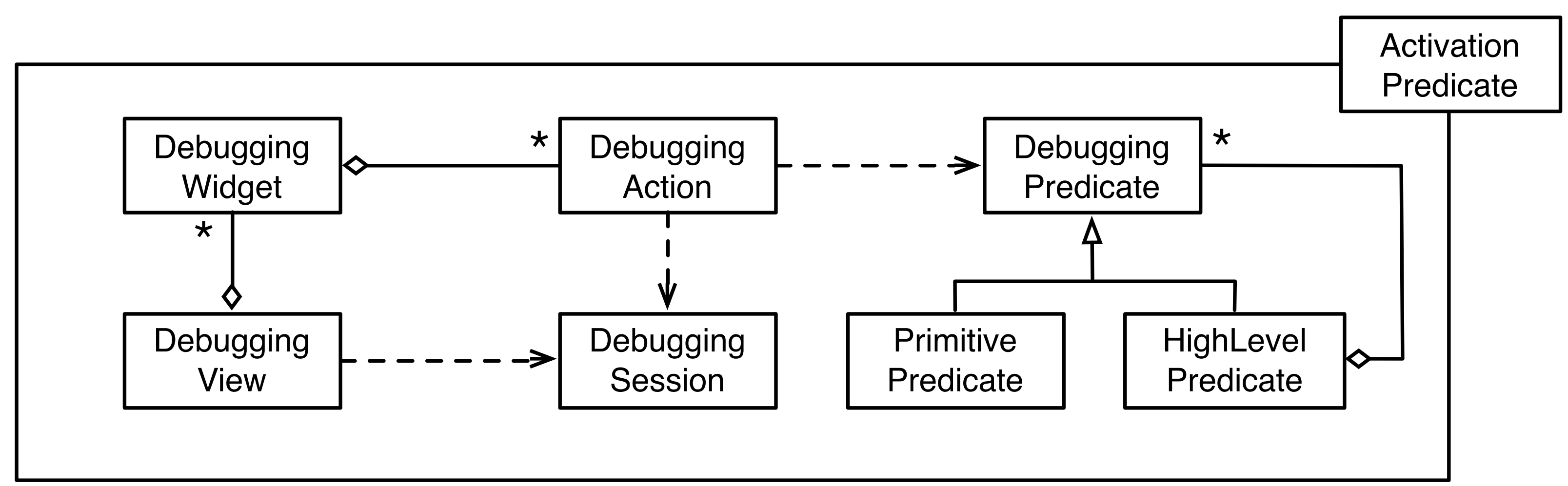
Production(aproduction)

Next failure

Stream position(anInteger)

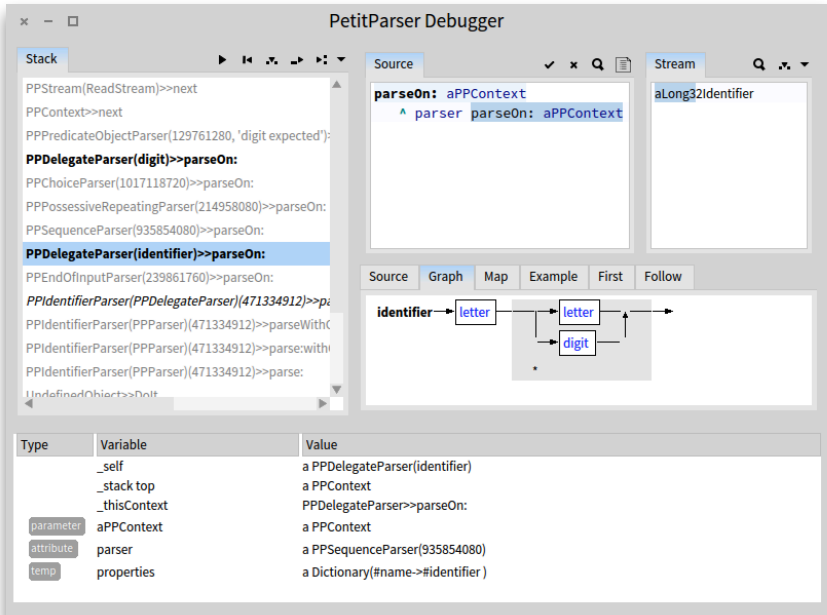
Stream position changed

Domain-specific extensions are composed from debugging widgets and actions, and triggered by contextual debugging predicates.

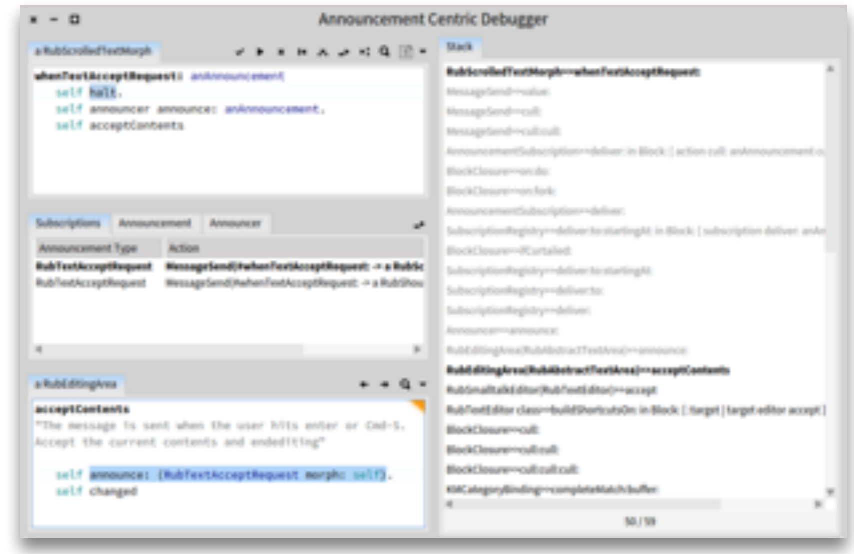


Moldable debuggers are built up from debugging widgets and debugging actions. The moldable debugger uses activation predicates to know which debuggers can currently be activated, allowing the developer to switch between debuggers without starting a new session.

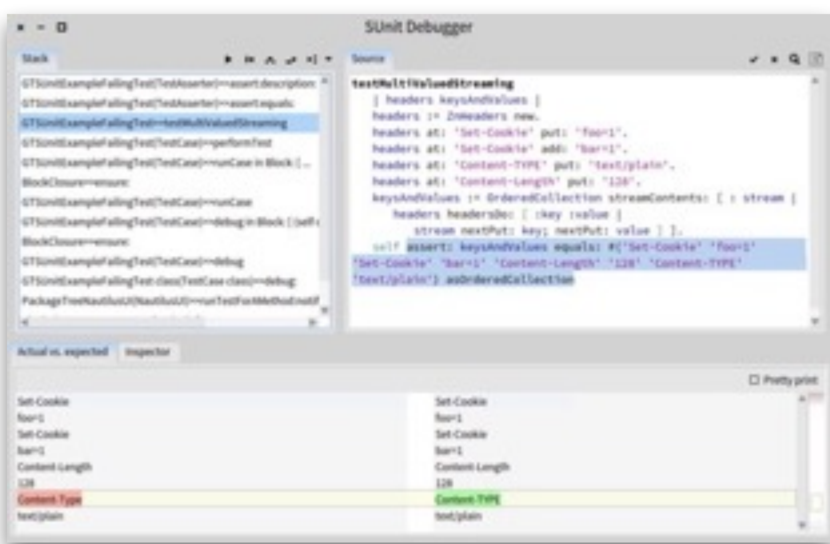
Petit Parser



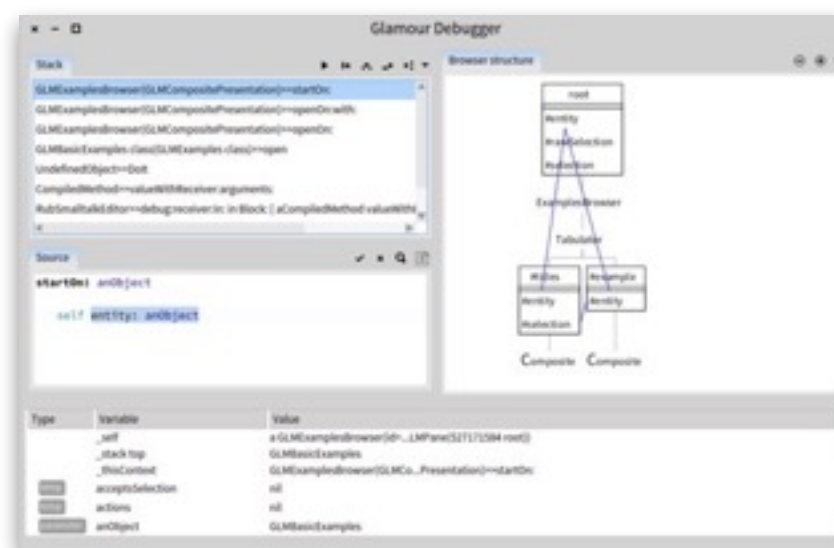
Events



SUnit



Glamour



Moldable debuggers have been built for several different domains already. The event-based debugger supports event-driven programming (which does not map well to a stack).

The SUnit debugger knows about and supports the notion of tests.

The Glamour debugger knows about the domain of flow-based model browsers.

New debuggers are cheap

	Session	Operations	View	Total
Base model	800	700	-	1500
Default Debugger	-	100	400	500
Announcements	200	50	200	450
Petit Parser	100	300	200	600
Glamour	150	100	50	300
SUnit	100	-	50	150

Although some expertise is required to build a new debugger, the development effort for a new debugger is tiny.

Outlook: domain-aware IDEs

We have been exploring how to apply the ideas behind the moldable debugger to other domains, such as object *inspection* (the moldable inspector) and *querying* (the "moldable spotter").

In the long run, we imagine a complete development environment that is easy to adapt (mold) to various technical and application domains with low effort.

Link code to the ecosystem

The architecture

... is not in the code

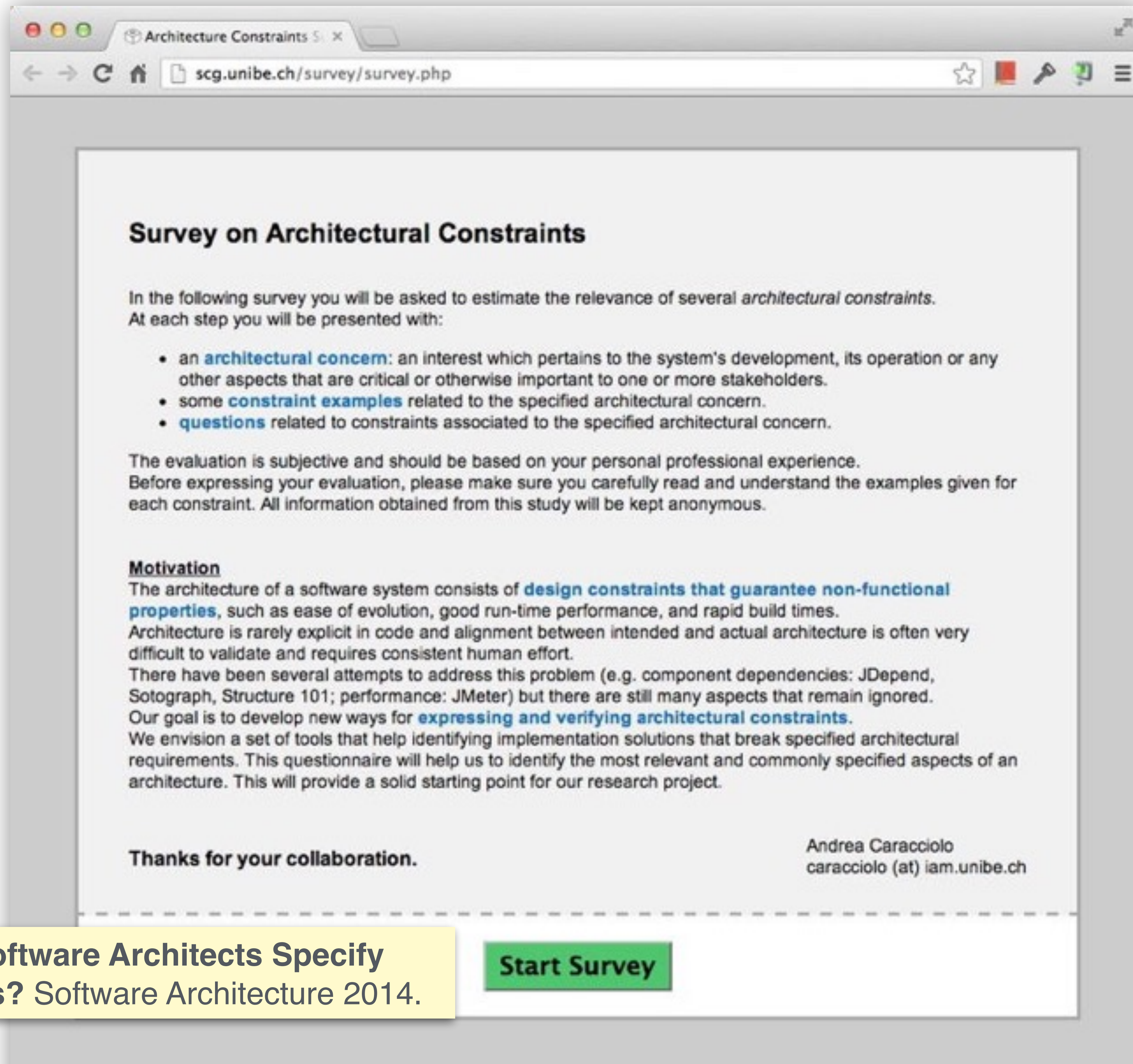
Although the architecture is one of the most important artifacts of a system to understand, it is not easily recoverable from code. This is because: (1) a system may have many architectures (eg layered and thin client), (2) there are many kinds of architecture static, run-time, build etc), (3) PLs do not offer any support to encode architectural constraints aside from coarse structuring and interfaces.

"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.

We somehow need to establish the link between the code and the (hidden) architecture.

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, Sotograph, 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.

Andrea Caracciolo
caracciolo (at) iam.unibe.ch

Start Survey

Andrea Caracciolo, et al. **How Do Software Architects Specify and Validate Quality Requirements?** Software Architecture 2014.

The theory seems to suggest that SA is mainly about structure and dependencies. Our experience with actual projects suggested that the truth might be different.

We carried out a couple of empirical studies, first a qualitative one to understand what is SA in the wild, and then a second, quantitative one to see to what extent various kinds of constraints appear in practice.

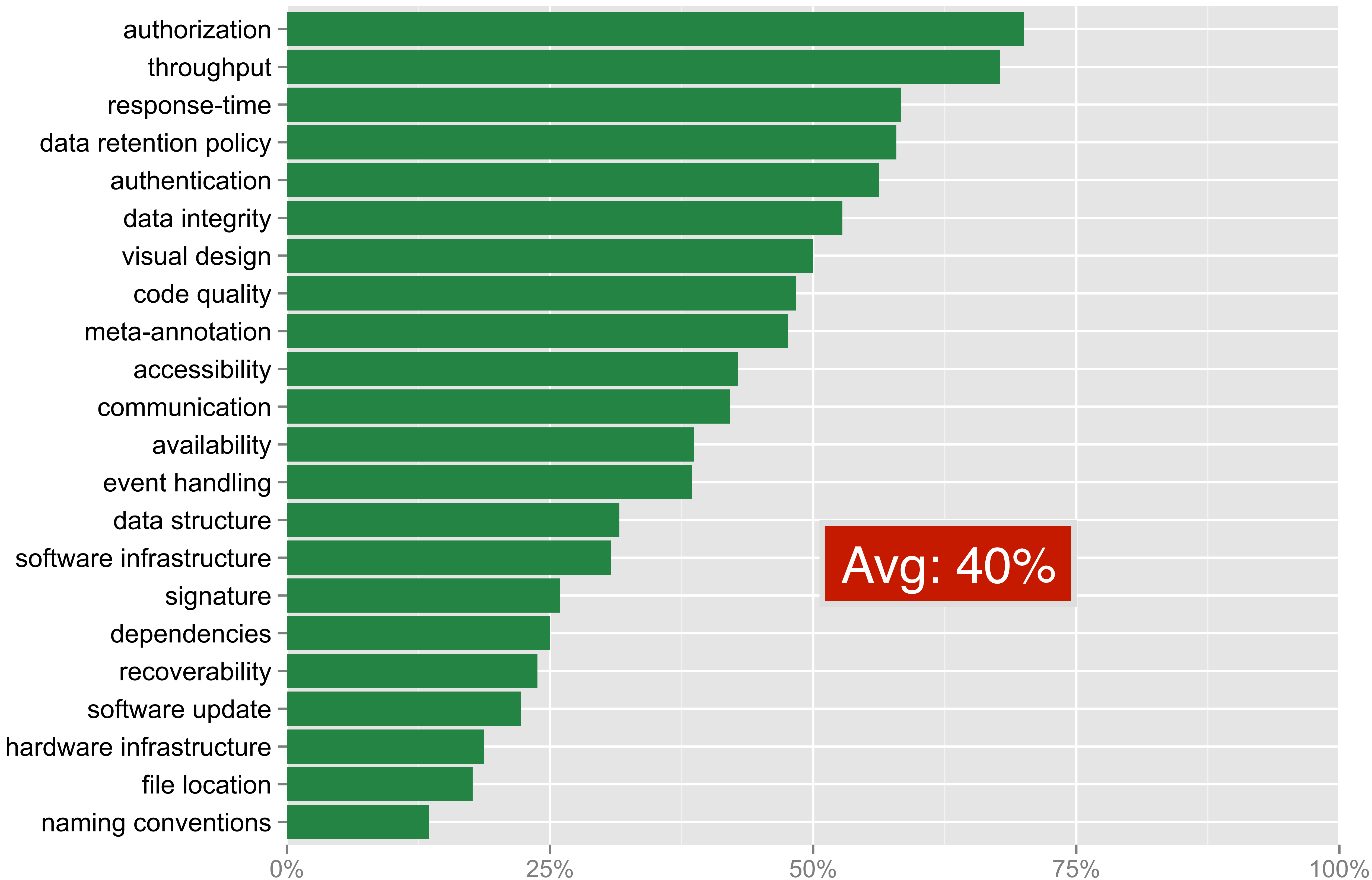
Andrea Caracciolo, et al. "How Do Software Architects Specify and Validate Quality Requirements?" Software Architecture 2014.

Impact of SA constraints

constraint	Impact (1-5)
availability	4.2
response-time	4.0
authorization	3.9
authentication	3.6
communication	3.4
throughput	3.4
signature	3.4
software infrastructure	3.3
data integrity	3.3
recoverability	3.1
dependencies	3.1
visual design	3.0
data retention policy	3.0
hardware infrastructure	2.9
system behavior	2.9
data structure	2.9
event handling	2.9
code metrics	2.7
meta-annotation	2.6
naming conventions	2.6
file location	2.5
accessibility	2.5
software update	2.2

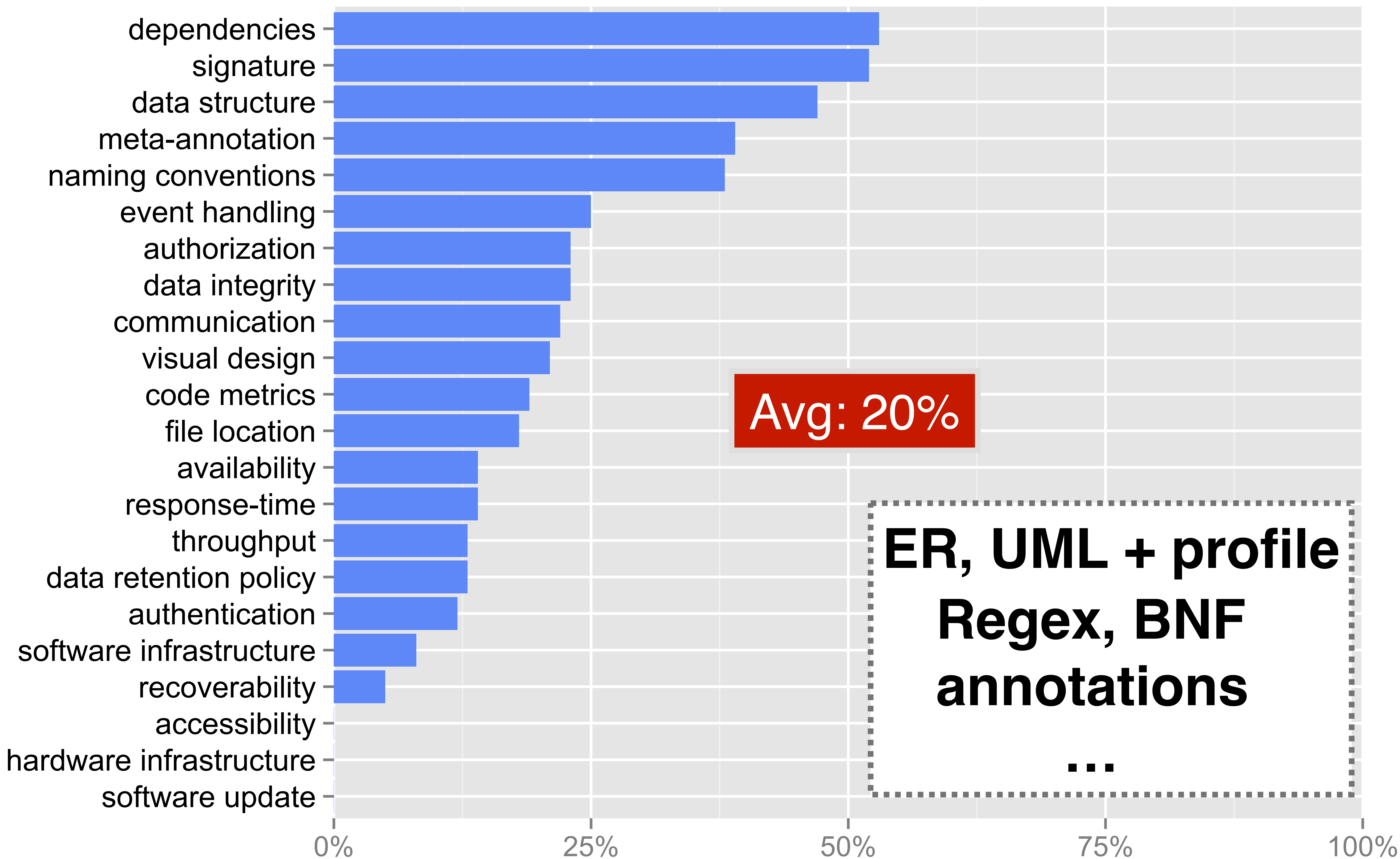
In the quantitative study we asked developers how important different kinds of architectural constraints were for their projects. Interestingly, in the top ten, there were significantly more user constraints, like availability (in green) than developer constraints (in blue). Dependencies were only halfway down the list.

Automated Validation is not Prevalent



Quality requirements are only checked 40% of the time.

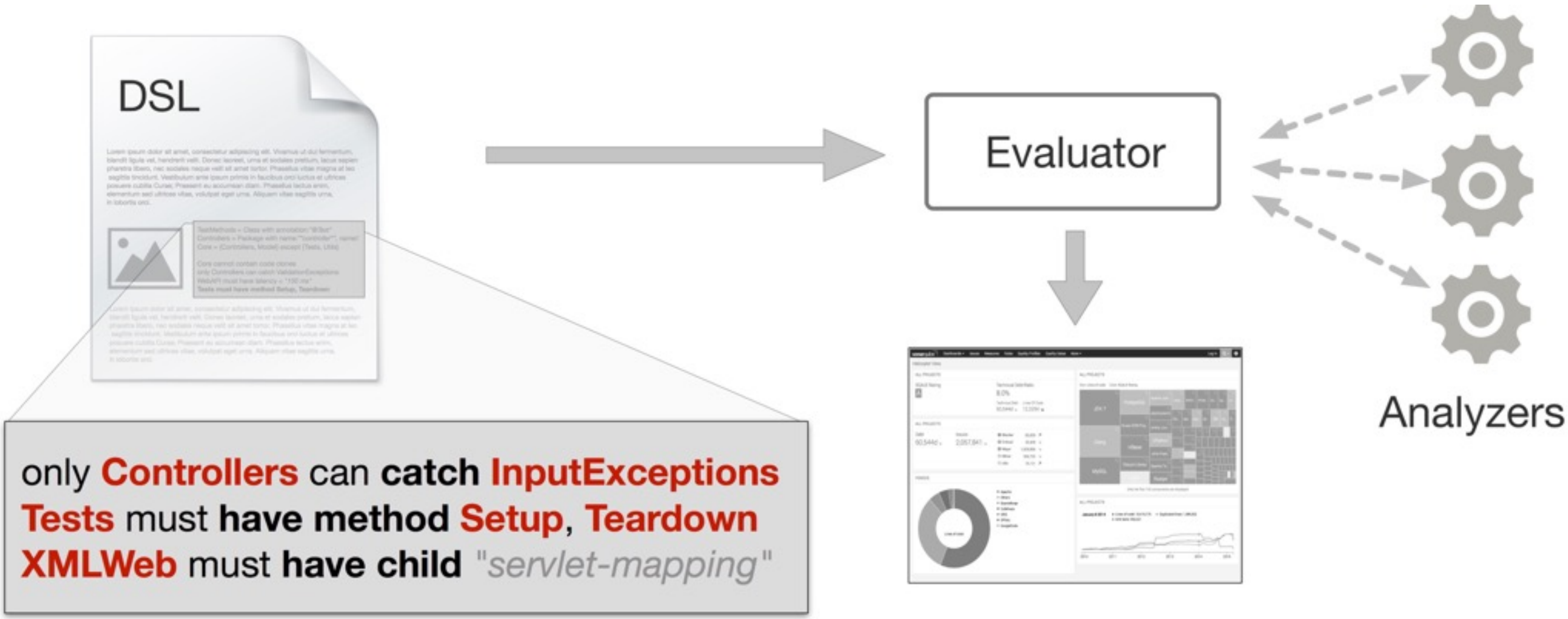
Formalization is not Prevalent



On average QRs are formally specified only 20 % of the time. Practitioners use different formalisms: from UML+profile to regex.

One of the key problems is usability. Where tools exist, functionality is limited and usability is poor. A host of different notations are needed to use these tools.

Dicto — a unified ADSL



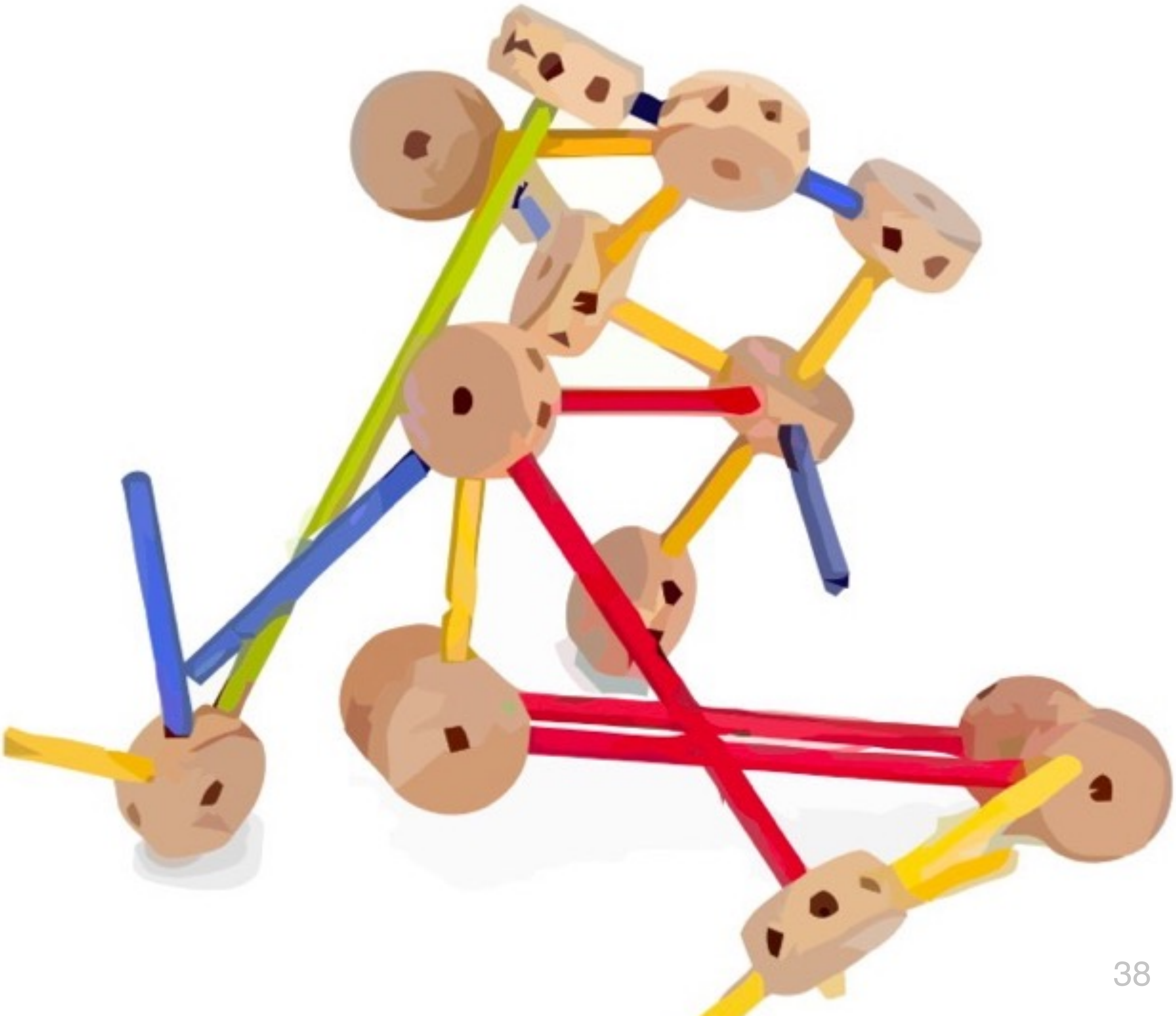
Andrea Caracciolo, et al. **Dicto: A Unified DSL for Testing Architectural Rules.** ECSAW '14.

Dicto offers a unified specification language as a front end to various tools. A generic DSL captures the basic structure of most architectural constraints. The language is adapted to different needs, and is used to generate the actual specification needed as input to a given tool.

The tool has been applied to a variety of domains and has been validated in a number of industrial case studies.

Andrea Caracciolo, et al. "Dicto: A Unified DSL for Testing Architectural Rules." ECSAW '14.

Outlook: link the code to its environment



Linking code to architecture is just one example.

Conclusion

Outlook: Programming is Modeling

Outlook: link the code to its environment

Outlook: domain-aware IDEs

Outlook: models = code