

Test Design

Analyse et Vérification de Logiciel

Guille Polito — CRIS^tAL — CNRS — RMoD - Inria Lille
guillermo.polito@inria.fr
@guillep

Automated Testing Refreshment

```
SetTest >> testAdd
```

```
| aSet |  
"Context"  
aSet := Set new.
```

```
"Stimulus"  
aSet add: 5.  
aSet add: 5.
```

```
"Check"  
self assert: aSet size equals: 1.
```

Automated Testing Refreshment

SetTest >> testAdd

```
| aSet |  
"Context"  
aSet := Set new.
```

in this context

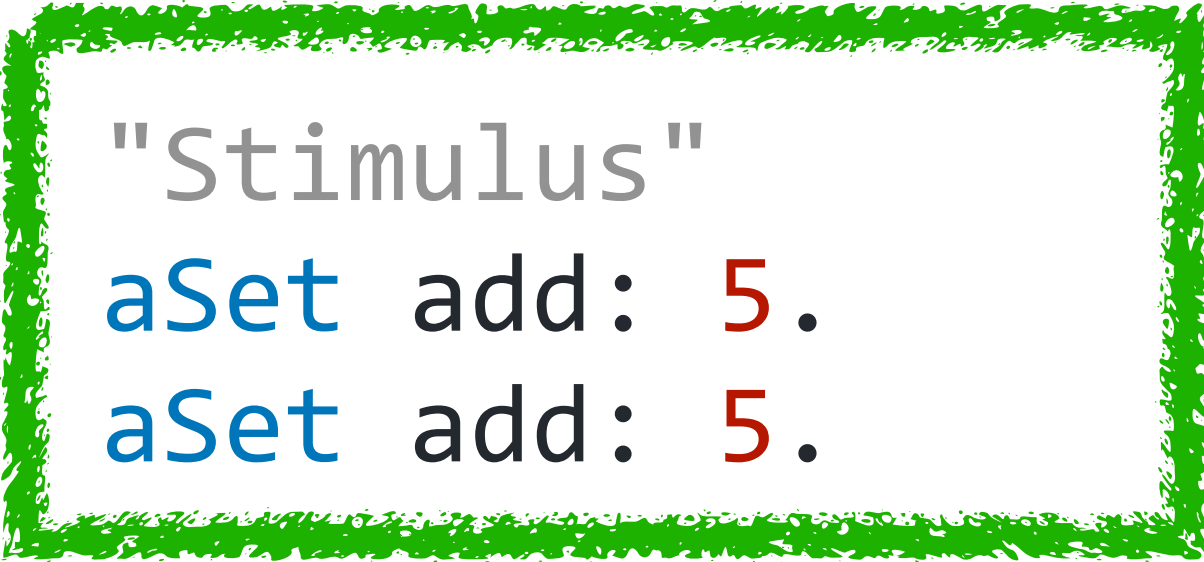
```
"Stimulus"  
aSet add: 5.  
aSet add: 5.
```

```
"Check"  
self assert: aSet size equals: 1.
```

Automated Testing Refreshment

SetTest >> testAdd

```
| aSet |  
"Context"  
aSet := Set new.
```



```
"Stimulus"  
aSet add: 5.  
aSet add: 5.
```

```
"Check"  
self assert: aSet size equals: 1.
```

in this context
when this happens

Automated Testing Refreshment

SetTest >> testAdd

```
| aSet |  
"Context"  
aSet := Set new.
```

```
"Stimulus"  
aSet add: 5.  
aSet add: 5.
```

```
"Check"  
self assert: aSet size equals: 1.
```

in this context
when this happens
then this should happen

Why testing?

Why testing?

- **Increase quality!**
- **Detect regressions:** *Wait, this was working before!*
- **Trust changes:** *I'll refactor/change this piece of critical code...*
- **Murphy's law:** *Anything that can go wrong will go wrong*

Kinds of testing

- **Unit tests:** low level, single-component
- **Integration testing:** how different modules work together
- **Functional testing:** focus on the business requirements of an application
- **Acceptance testing:** verify minimal business requirements
- **Performance testing:** behaviours the system under significant load
- **Smoke testing:** check that the system *does not fail*

What is a good test?

What is a good test?

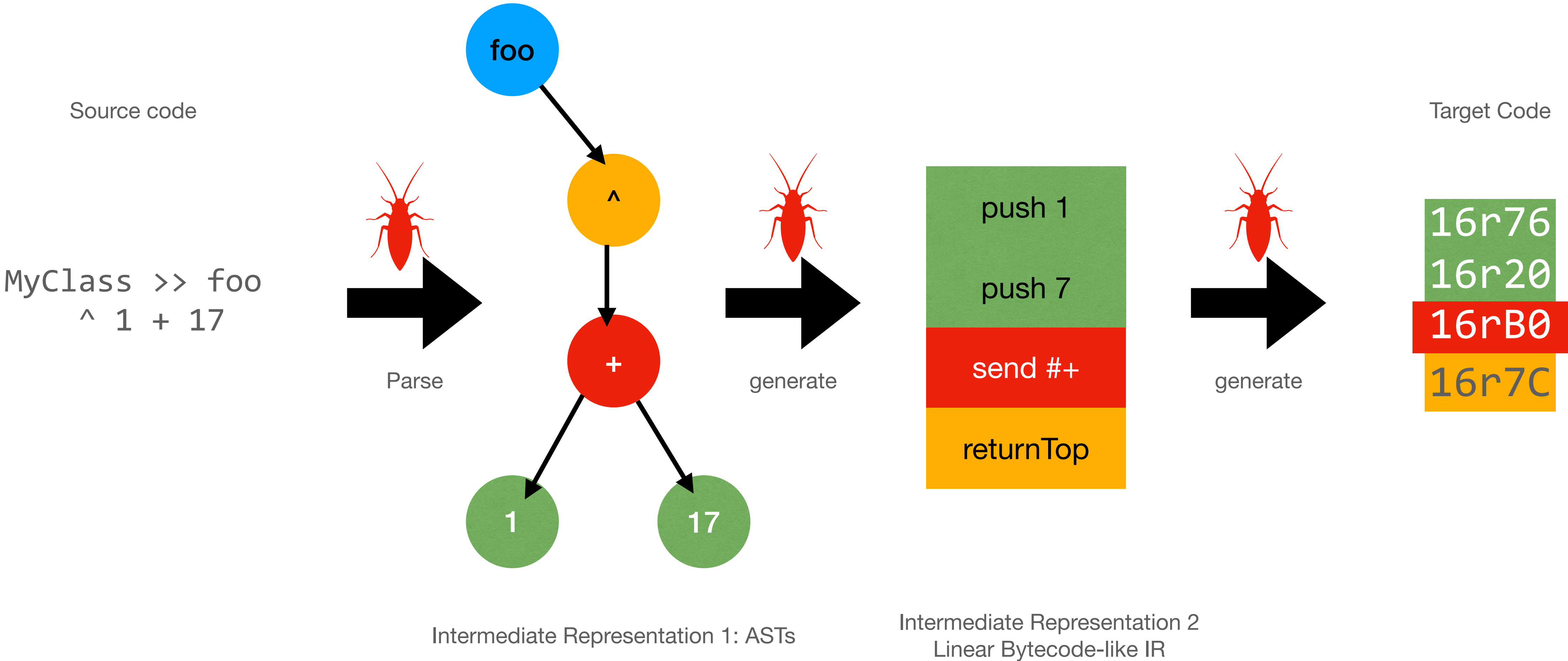
“A good test is a test that catches bugs”

- me

Tests that catch bugs

- check extreme cases (e.g. null, 0, empty, bigger than the collection...)
- check complex cases (e.g. exceptions)
- check different execution paths

Case Study 1: Compiler



Compiler test example

```
testPushConstantZeroBytecodePushesASmallIntegerZero
```

```
self compile: [ compiler genPushConstantZeroBytecode ].  
self runGeneratedCode.
```

```
self assert: self popAddress equals: (memory integerObjectOf: 0)
```

Compiler test example

Insights: Black box testing

Black box testing

=> depend only on **observable behaviour**

=> **reusable** in different backends

=> more **resistant to changes** in the implementation

```
testPushConstantZeroBytecodePushesASmallIntegerZero
```

```
self compile: [ compiler genPushConstantZeroBytecode ].  
self runGeneratedCode.
```

```
self assert: self popAddress equals: (memory integerObjectOf: 0)
```

Compiler test example

Insights: Cross-compile / Cross-execute



<http://www.unicorn-engine.org>

Use a machine simulator

=> **hardware independent:** test and debug in any machine any backend

=> **parametrizable tests** run the same test with multiple backends

```
testPushConstantZeroBytecodePushesASmallIntegerZero
```

```
self compile: [ compiler genPushConstantZeroBytecode ].
```

```
self runGeneratedCode.
```

```
self assert: self popAddress equals: (memory integerObjectOf: 0)
```

Compiler test example

Insights: Start Small

- First: The **simplest** test you can write for the **simplest** functionality
- Second: The **next simplest** test you can write for the **next simplest** functionality

=> The first focus is in understanding **how to** better write the **tests**

testPushConstantZeroBytecodePushesASmallIntegerZero

```
self compile: [ compiler genPushConstantZeroBytecode ].  
self runGeneratedCode.
```

```
self assert: self popAddress equals: (memory integerObjectOf: 0)
```


Compiler test example

Insights: Invest in infrastructure

- Refactor
- Clean
- Create Reusable Components

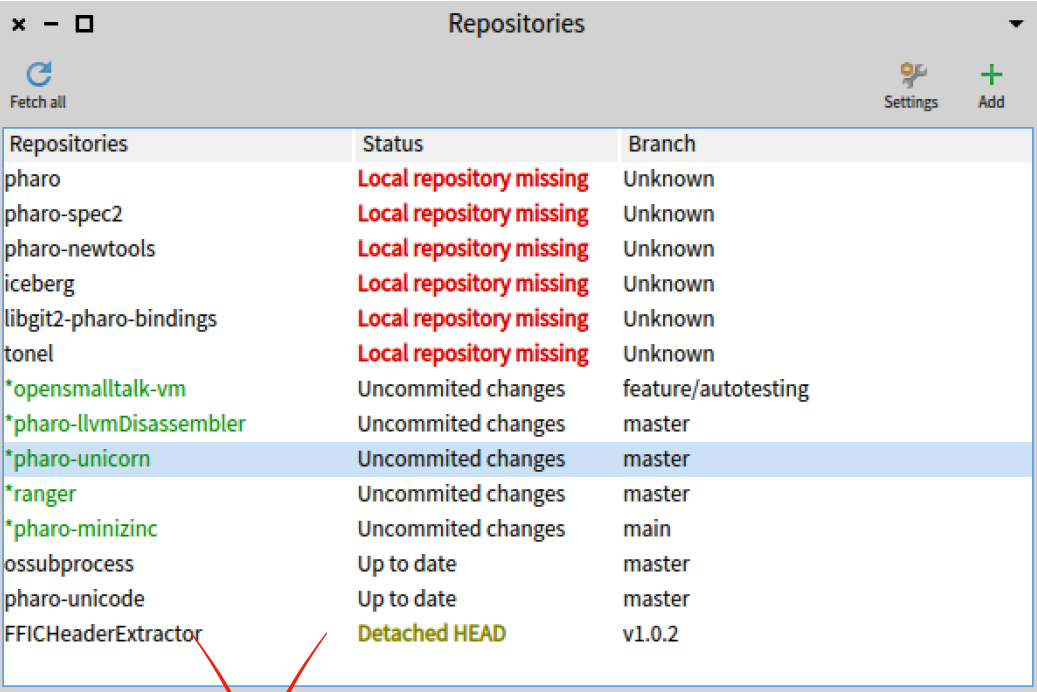
testPushConstantZeroBytecodePushesASmallIntegerZero

```
self compile: [ compiler genPushConstantZeroBytecode ].
```

```
self runGeneratedCode.
```

```
self assert: self popAddress equals: (memory integerObjectOf: 0)
```

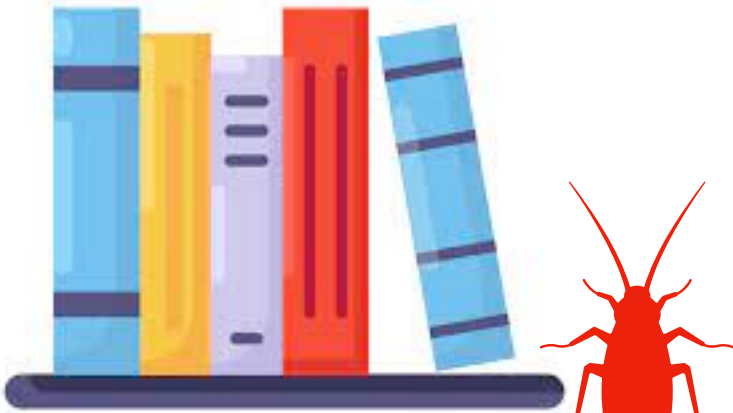
Case Study 2: Git Client



Repositories	Status	Branch
pharo	Local repository missing	Unknown
pharo-spec2	Local repository missing	Unknown
pharo-newtools	Local repository missing	Unknown
iceberg	Local repository missing	Unknown
libgit2-pharo-bindings	Local repository missing	Unknown
tonel	Local repository missing	Unknown
*opensmalltalk-vm	Uncommitted changes	feature/autotesting
*pharo-llvmDisassembler	Uncommitted changes	master
*pharo-unicorn	Uncommitted changes	master
*ranger	Uncommitted changes	master
*pharo-minizinc	Uncommitted changes	main
ossubprocess	Up to date	master
pharo-unicode	Up to date	master
FFICHeaderExtractor	Detached HEAD	v1.0.2

Branch

Commit

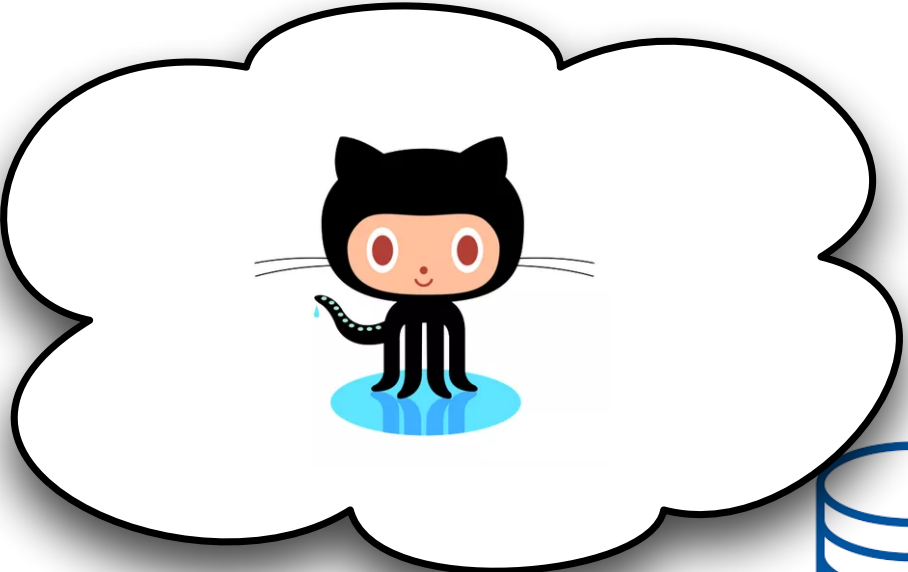


LibGit

Git Client

Git Repository

Remote repositories

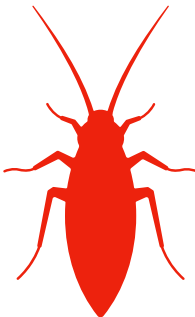


Working Copy

What you see

Index

A “hidden”
staging area
(sort of)



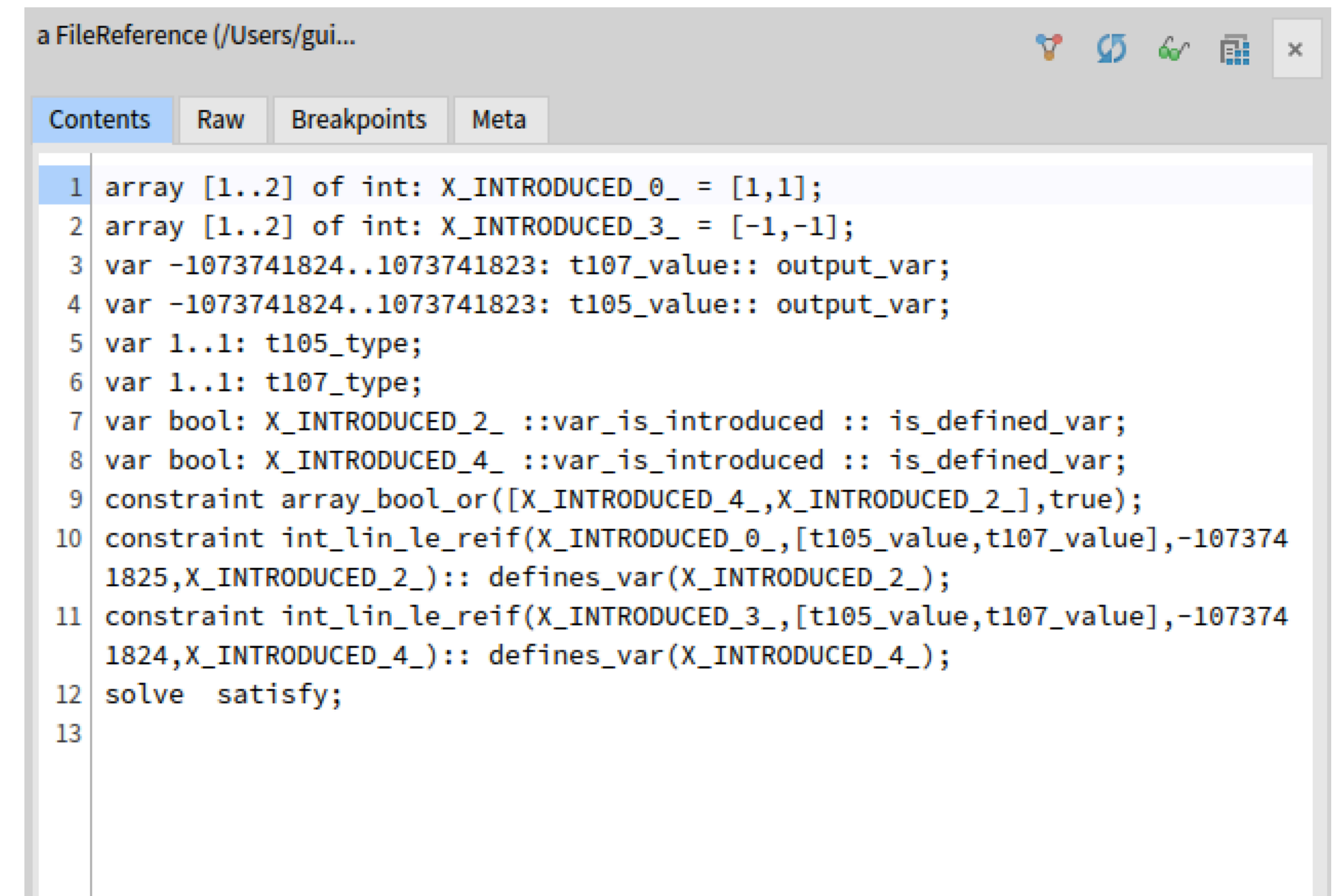
Repository

Where
commits are



Case Study 3: UI Plugins

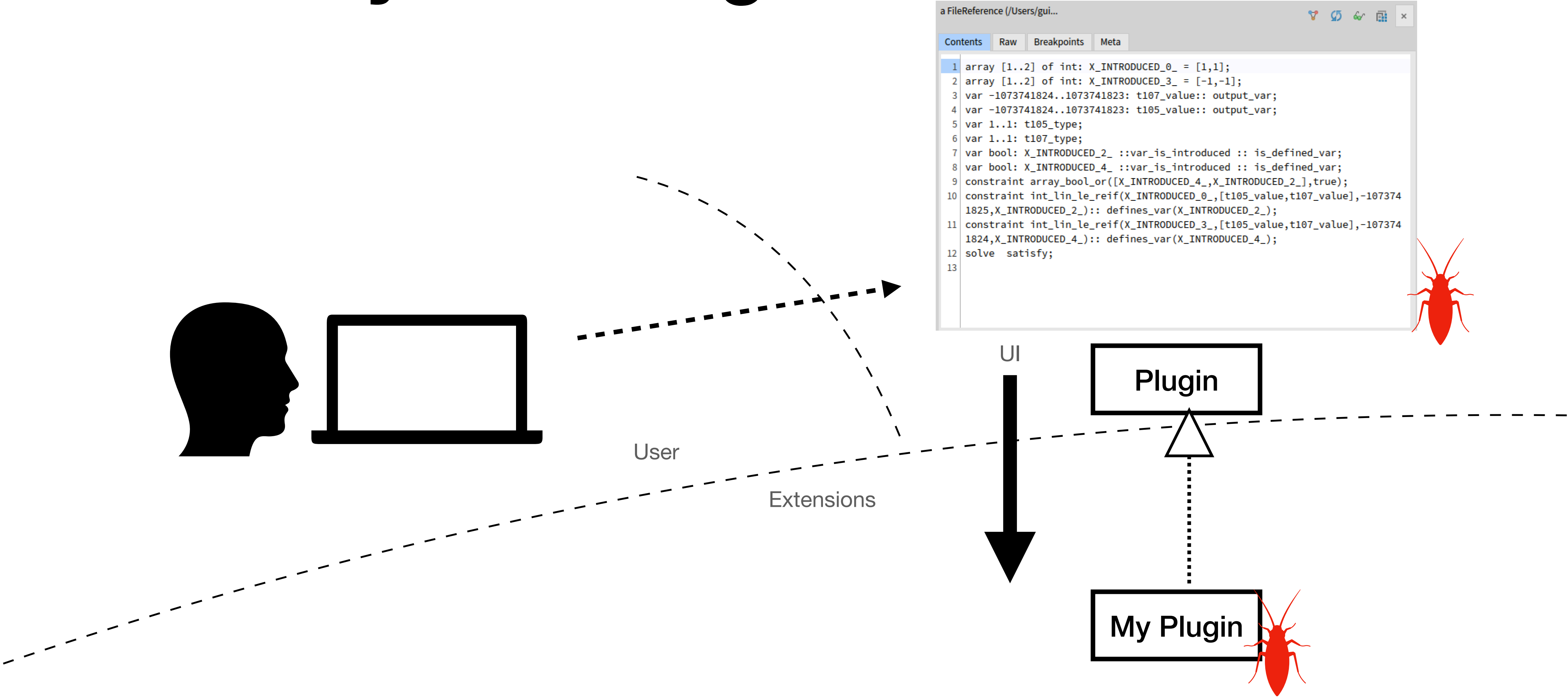
```
File >> gtInspectorContentsIn: composite
<gtInspectorPresentationOrder: 5>
composite text
  title: 'Contents';
  format: #asText;
  display: [ ... ]
```



The screenshot shows a code editor window titled "a FileReference (/Users/gui...". The window has tabs for "Contents", "Raw", "Breakpoints", and "Meta". The "Contents" tab is active, displaying a list of constraints and variable definitions. The code is as follows:

```
1 array [1..2] of int: X INTRODUCED_0_ = [1,1];
2 array [1..2] of int: X INTRODUCED_3_ = [-1,-1];
3 var -1073741824..1073741823: t107_value:: output_var;
4 var -1073741824..1073741823: t105_value:: output_var;
5 var 1..1: t105_type;
6 var 1..1: t107_type;
7 var bool: X INTRODUCED_2_ ::var_is_introduced :: is_defined_var;
8 var bool: X INTRODUCED_4_ ::var_is_introduced :: is_defined_var;
9 constraint array_bool_or([X INTRODUCED_4_,X INTRODUCED_2_],true);
10 constraint int_lin_le_reif(X INTRODUCED_0_,[t105_value,t107_value],-107374
1825,X INTRODUCED_2_):: defines_var(X INTRODUCED_2_);
11 constraint int_lin_le_reif(X INTRODUCED_3_,[t105_value,t107_value],-107374
1824,X INTRODUCED_4_):: defines_var(X INTRODUCED_4_);
12 solve satisfy;
13
```

Case Study 3: UI Plugins



Tests that are challenging to write

- non-deterministic behavior
- user-interactions
- external interactions

• SD GIF



Conclusion

- *Anything that can go wrong will go wrong!*
- *Tests give freedom to change without fear*
- *Testing Guidelines*
 - *Start simple, then “what happens if...?”*
 - *Make tests resistant to implementation changes*
 - *Tests are code!*