

Let linguistics guide  
software analysis

# *Software Analysis using Natural Language Queries*

Pooja Rani  
Ph.D Student  
Software Composition Group  
University of Bern, Switzerland

# Inspiration

Natural languages like English, German, French etc.

Programming languages like R, Java, Python, C++ etc

Do programming languages shape our mind as the natural languages do?

# Software Analysis

Code snippet of a function

```
distanceBetweenTwoQuestions

|sentenceOne sentenceTwo questionDistanceCalculator sentenceDistance|
sentenceOne := 'The hunter killed the lion'.
sentenceTwo := 'The lion was killed by the hunter'.

questionDistanceCalculator := McQuestionsDistance new questionOne: sentenceOne; questionTwo:sentenceTwo.

sentenceDistance := questionDistanceCalculator calculateDistance.

self
assert: [ sentenceDistance isNumber ]
description: [ 'Distance must only contain digits' ].

^ sentenceDistance
```

# Software Analysis

Code snippet of a function

I, a function, represent an example.

I have four temporary variables named <sentenceOne,sentenceTwo, questionDistanceCalculator, sentenceDistance>

I takes two setnence named <sentenceOne>,<sentenceTwo>.

I use McQuestionsDistance class to calculate distance between two sentences.

I verify the value <sentenceDistance> given by McQuestionsDistance.

I makes sure the distance returned is in the form of digits.

# Software Analysis

Code snippet of a function that calculates the distance between two sentences.

```
distanceBetweenTwoQuestions
```

```
|sentenceOne sentenceTwo questionDistanceCalculator sentenceDistance|
sentenceOne := 'The hunter killed the lion'.
sentenceTwo := 'The lion was killed by the hunter'.

questionDistanceCalculator := McQuestionsDistance new questionOne: sentenceOne; questionTwo:sentenceTwo.

sentenceDistance := questionDistanceCalculator calculateDistance.

self
assert: [ sentenceDistance isNumber ]
description: [ 'Distance must only contain digits' ].

^ sentenceDistance
```

# Software Analysis

Code snippet of a function that calculates the distance between two sentences.

I, a function, represent an example.

I have four temporary variables named <sentenceOne,sentenceTwo, questionDistanceCalculator, sentenceDistance>

I takes two setnence named <sentenceOne>,<sentenceTwo>.

I use McQuestionsDistance class to calculate distance between two sentences.

I verify the value <sentenceDistance> given by McQuestionsDistance.

I makes sure the distance returned is in the form of digits.

# Software Analysis

Code snippet of a function that calculates the distance between two sentences.

```
distanceBetweenTwoQuestions

|sentenceOne sentenceTwo questionDistanceCalculator sentenceDistance|
sentenceOne := 'The hunter killed the lion'.
sentenceTwo := 'The lion was killed by the hunter'.

questionDistanceCalculator := McQuestionsDistance new questionOne: sentenceOne; questionTwo:sentenceTwo.

sentenceDistance := questionDistanceCalculator calculateDistance.

self
assert: [ sentenceDistance isNumber ]
description: [ 'Distance must only contain digits' ].

^ sentenceDistance
```

I, a function, represent an example.  
I have four temporary variables named <sentenceOne,sentenceTwo, questionDistanceCalculator, sentenceDistance>  
I takes two sentence named <sentenceOne>,<sentenceTwo>.  
I use McQuestionsDistance class to calculate distance between two sentences.  
I verify the value <sentenceDistance> given by McQuestionsDistance.  
I makes sure the distance returned is in the form of digits.

The Interesting part is developers most of the time do not write **comments** for their code.

How we are suppose to understand code  
elements without description?

We **search** for code elements in tools. For their examples, occurrences in function or comments.



Search

Remote Search File Search Task Search Git Search Java Search JavaScript Search Plug-in Search

Search string (\* = any string, ? = any character):  
McQuestionDistance

Case sensitive

Search For

Type Method  
Package Constructor  
Module Field

Limit To

All occurrences Declarations  
References Implementors  
Match locations (1 of 15 selected)

Search In

Sources  Required projects  JRE libraries  Application libraries

Scope

Workspace Selected resources Enclosing projects  
Working set:  Choose...

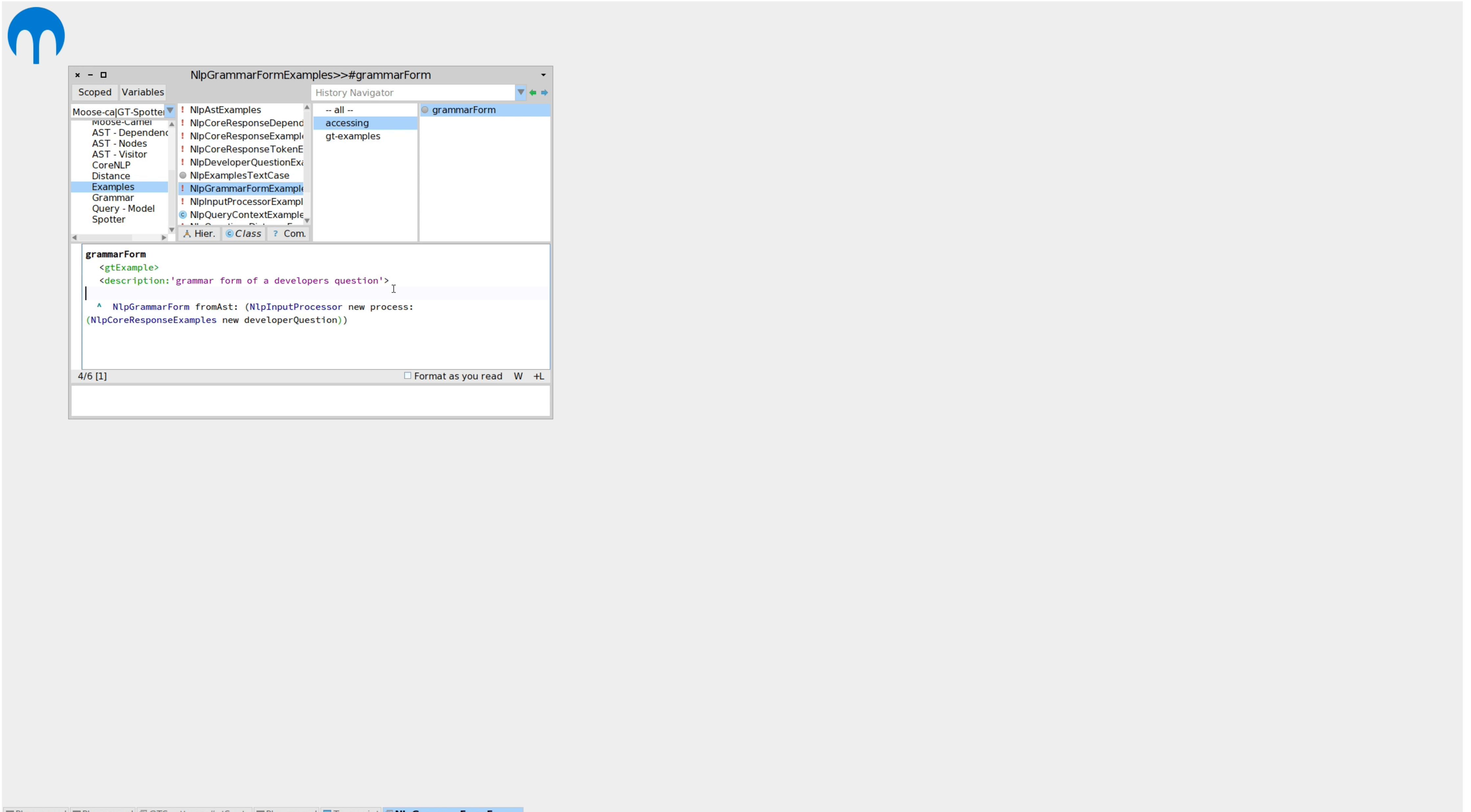
Customize... Cancel Search

Not everyone can use these tools.

Natural language interface for analysis tools

Communication barrier for new developers

# Search in English



The screenshot shows the Moose IDE interface with the following details:

- Title Bar:** NlpGrammarFormExamples>>#grammarForm
- Scope Bar:** Scoped Variables
- Left Navigator:** Moose-ca|GT-Spotter, AST - Examples (highlighted), Grammar, Query - Model, Spotter.
- History Navigator:** grammarForm (highlighted)
- Central Editor:** grammarForm code:

```
grammarForm
<gtExample>
<description:'grammar form of a developers question'>
|
^ NlpGrammarForm fromAst: (NlpInputProcessor new process:
(NlpCoreResponseExamples new developerQuestion))
```
- Status Bar:** 4/6 [1], Format as you read, W, +L

The Interesting part is developers most of the time do not write **comments** for their code.

The Interesting part is developers  
write **comments** for their code.

When they write, how will you take maximum information our of those comments?

# How developers write comments

a MccComment (Comment of BrStencil) ×

Comment Raw \_GT Meta

From the Oxford Dictionary:  
Stencil - a thin sheet of card, plastic, or metal with a pattern or letters cut out of it, used to produce the cut design on the surface below by the application of ink or paint through the holes.

In Brick I play a role of an abstract factory to create predefined objects. The most common use case for a stencil is to produce widgets, hence create visual elements. Stencils are intended to be reusable.

The idea behind a stencil as concept is to help developers reduce the use of block closures used with scripting API. The reason for that is the fact that block closures can not be recompiled while in a debugger, which makes system less debuggable and live programmable.

The only essential Stencil API is ===#create== method, developers should implement it in their stencil subclasses. Note, that ===#create== has zero arguments which by default makes Stencil stateless. It is also possible to implement a statefull Stencil. Let us formalise the both types:

Stateless Stencil - A stencil that does not have any parameters and the result of #create method is always identical. Two stateless stencils are equal then and only then when their type is the same.

Stateful Stencil - A parametrizable stencil whose result of a #create method depends on configuration. Two stencils of the same type configured differently are not considered to be equal.

While subclassing is being the preferable way of creating Stencils we still want to provide the support of block closures that play a role of a stencil. Any Valuable object can be converted to so the ValuableStencil by sending ===#asStencil== conversion method to the object that implements it.

For example the following code creates a Stateless stencil that creates a simple visual element out of BlockClosure:

```
[[[
| aStencil anElement |
aStencil := [ BIElement new background: Color gray ] asStencil.
anElement := aStencil create.
]]]
```

a MccComment (Comment of BrToggleEnabledEvent) ×

Comment Raw \_GT Meta

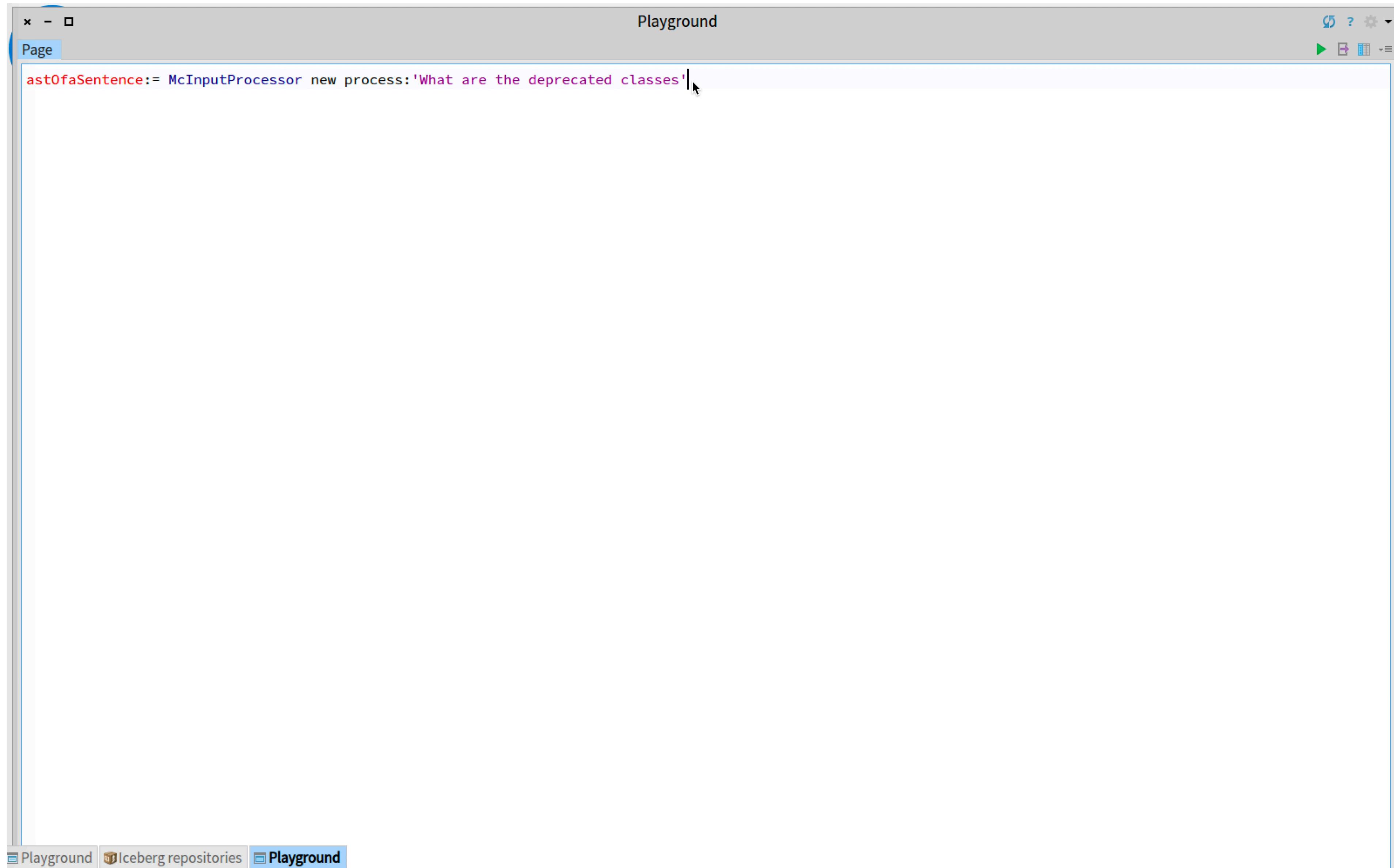
I am sent when switch is enabled

# Challenges



- Developers do not write detailed comments
- The text data is mixed of programming and natural language
- Applying linguistics to software artifacts
- Semantic tools are not ready for software engineering domain

# Analyze structure of a sentence



The screenshot shows a software interface titled "Playground". The main window is labeled "Page" and contains the following code:

```
ast0faSentence:= McInputProcessor new process:'What are the deprecated classes'|
```

The code is written in a syntax highlighting editor. The variable name "ast0faSentence" is in red, followed by a colon and a space. The assignment operator "=" is in blue. The class name "McInputProcessor" is in purple, followed by a space and the method name "new" in blue. The argument to the "process:" method is in pink, enclosed in single quotes. The entire argument string is "What are the deprecated classes|". A cursor arrow is positioned at the end of the argument string. The interface has a standard window title bar with close, minimize, and maximize buttons. The bottom of the window shows tabs: "Playground" (which is selected), "Iceberg repositories", and another "Playground" tab.

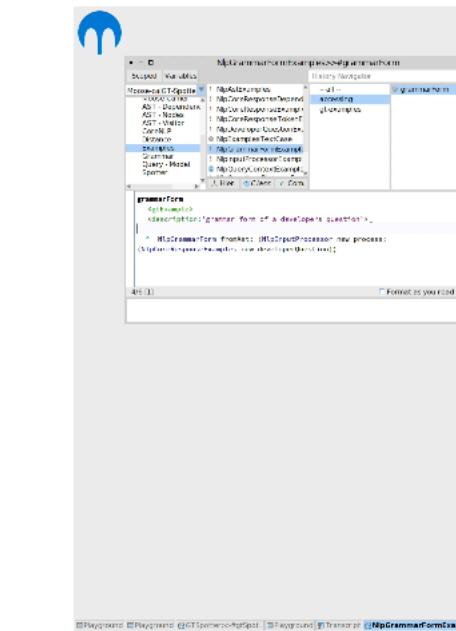
# Summary

## Software Analysis

Code snippet of a function that calculates the distance between two sentences.

```
distanceBetweenQuestions
[sentenceOne sentenceDistanceCalculator sentenceDistance]
sentenceOne := "The huter killed the lion".
sentenceDistanceCalculator := "The lion killed the huter".
questionDistanceCalculator := ObjetionDistance new questionOne: sentenceOne; questionTwo: sentenceTwo.
sentenceDistance := questionDistanceCalculator calculateDistance.
self
assert: [ sentenceDistance valueNumber ]
description: [ distance must only contain digits ].
```

## Search in English



14

The Interesting part is developers most of the time do not write **comments** for their code.

8

## How developers write comments

```
a McCComment [Comment of BrStencil] >
Comment Raw _GT Meta
From the Oxford Dictionary:
Stencil - a thin sheet of card, plastic, or metal with a pattern or letters cut out of it, used to produce the cut design on the surface below by the application of ink or paint through the holes.

In Brick I play a role of an abstract factory to create predefined objects. The most common use case for a stencil is to produce widgets, hence create visual elements. Stencils are intended to be reusable.

The idea behind a stencil is to help developers reduce the use of block closures used with scripting API. The reason for that is the fact that block closures can not be recompiled while in a debugger, which makes system less debugable and live programmable.

The only essential Stencil API is ==>create== method, developers should implement it in their stencil subclasses. Note, that ==>create== has zero arguments which by default makes Stencil stateless. It is also possible to implement a statefull Stencil. Let us formalise the both types.

Stateless Stencil - A stencil that does not have any parameters and the result of >create method is always identical. Two stateless stencils are equal then and only then when their type is the same.

Stateful Stencil - A parametrizable stencil whose result of a >create method depends on configuration. Two stencils of the same type configured differently are not considered to be equal.

While refactoring is being the preferable way of creating stencils we still want to provide the support of block closures that play a role of a stencil. Any valuable object can be converted to be the ValueableStencil by sending ==>#asStencil== conversion method to the object that implements it.

For example the following code creates a Stateless stencil that creates a simple visual element out of BlockClosure:
[[| aStencil anElement |
anElement := BILlement new background: Color gray | aStencil.
anElement > aStencil create.
]]]
```

18

## Challenges



## Feedback

What kinds of linguistic analysis we can do in a software?

What is the linguistic pattern of developers in different programming language?

What are the social aspects of a software?

Is there any English linguistic expert here, I would like to meet.

22

# Feedback

What kinds of linguistic analysis we can do on a software?

What are the linguistic patterns of developers in different programming language?

What are the social aspects of a natural language text present in a software?

Is there any English linguistic expert here, I would like to meet.