

Flamel and Other Influences

Gsoc 2013 project and related work

[Home](#) [About](#)

Configuration of Flamel

Posted on September 29, 2013

// *Gofer it*
smalltalkhubUser: 'gisela' project: 'Flamel';
configurationOf: 'Flamel';
loadVersion: #development.

Posted in [Flamel](#), [Pharo](#) | [Leave a comment](#)

Abstraction process

Posted on September 16, 2013

If take a look in the metavariable patterns you can see that it can get as complex as you want.

But, let's focus in a particular token. We end up with a metavariable, but we can see a pattern from simplicity to

Recent posts

Configuration of Flamel
September 29, 2013

Abstraction process September
16, 2013

Define a simple rule
for matching September 6,
2013

Searching in Flamel! August 29,
2013

Just want to match! August 6,
2013

Recent Comments

Gi on A look into Spec

Archives

September 2013 (3)

August 2013 (3)

July 2013 (3)

June 2013 (7)

complexity:

1. Just abstract the token as a metavariable: we do not care if the method contains the exact word we want everything that contains something acting as that.

Example:

```
1 | aVariable size.
```

Maybe we do not care about who is the sender of the message `#size`. In order to do that, we will **abstract** the variable `'aVariable'` as a metavariable:

```
// `object size
```

2. Abstract a list of tokens: we do not care if the method contain not only one element also a lot of elements.

Example:

```
1 | aVariable size.
```

If the pattern is ``object size` we are too specific because we are only matching the send to size whom receiver is an object, if we have something like this:

```
1 | (aCollection select: #even) asOrderedCollection
```

Is not a result, if we want that, we have to specify that we do not care:

```
// `@lotOfStuff size
```

3. Abstract a recursive pattern: still maybe we want matches if the pattern is repeated

```
1 | aVariable size size.
```

If our pattern is: ``object size`, this does not match. If it's:

Categories

Flamel
Pharo
Programming
Rw tool
Uncategorized

Meta

Register
Log in
Entries RSS
Comments RSS
WordPress.com

`@some size we have one match when some = 'aVariable size'.

If we put a recursive pattern:

// *"@some size*

We have two matches: one when some='aVariable size' and another when some= 'aVariable'

4. Abstract a statement: Maybe we do not even care about the hole statement, and for this we should use the dot.

Example:

// *`@.Stat*

We can think that in the abstraction patterns you have a variety in abstraction, since one more concret with more information until one more abstract.

And for that we can conclude that we have a flow during the metavariable abstraction:

// ***NoAbstraction** <=> **Simple** <=> **List** <=> **Recursive** <=> **Statement***

In order to support this in flamel I've added #FlamelAbstractionStep that basically have 3 main responsibilities:

- next step: abstract
- previous step: unAbstract
- pattern: pattern

You can take an expression, obtain the AST representation and then just send to the node the messages abstract and

unAbstract to let the abstraction process to continue, once you finish just use the expression builder to obtain the expression.

Example:

```
1 | testAbstractVariableNodeWritesRightPattern
2 | | searchExpression variable root expression
3 | expression := 'aVariable aMessage'.
4 | root := RBPaser parseExpression: expression
5 | variable := root receiver .
6 | variable abstractOnMatching.
7 | searchExpression := FlamelSearchExpressionBuilder
8 | self assert: searchExpression equals: '`aVariable aMessage`'
```

If we do:

```
1 | variable variable abstractOnMatching.
2 | variable variable abstractOnMatching.
```

We will end up with: ``@aVariable aMessage``.

If you want to get fun annotating nodes I added some protocols to maintain the state in a node:

- `abstractOnMatching`: instead the source code for the node we are interested in the metavariable. Each time you say `abstractOnMatching` you are going up in the matching steps
- `deleteOnMatching`: when we want to match, we should not include the code contained inside the node
- `ignoreOnMatching`: we go to the last step in the matching process
- `unAbstractOnMatching`: we want the previous step for the metavariable

I Also wrote a visitor to generate the matching expression, so in order to obtain the expression, you can evaluate:

```
1 | FlamelSearchExpressionBuilder new searchExpressionBuilder
```

Posted in [Flamel](#), [Pharo](#), [Rw tool](#) | [Leave a comment](#)

Define a simple rule for matching

Posted on September 6, 2013

Ok let's start to understand the rewrite engine.

Let's do it with an example, our goal: Find all the methods that send the consecutive messages: #globals #at:

In our example we want as result:

```
1 findClassesForCategories: aCollection
2 | items |
3 aCollection isEmpty
4 ifTrue: [ ^ self baseClass withAllSubclasses
5 items := aCollection
6 gather: [ :category |
7 ((Smalltalk organization listAtCategoryName
8 collect: [ :each | Smalltalk globals at: each
9 select: [ :each | each includesBehavior: self
10 ^ items asSet
```

But not with this variation:

```
1 findClassesForCategories: aCollection
2 | items |
3 aCollection isEmpty
4 ifTrue: [ ^ self baseClass withAllSubclasses
5 items := aCollection
6 gather: [ :category |
7 ((Smalltalk organization listAtCategoryName
8 collect: [ :each | Smalltalk globals at: each
9 select: [ :each | each includesBehavior: self
10 ^ items asSet
```

Why? because in the second example we send the messages `#globals` (ok we want this!) but then we send `#at:ifAbsent:` (we don't want this)

Ok so... Which object do I send with message to find my "problematics" methods?

Which object?

The answer is very simple a RULE... For each rule you should extend one flavor of rule and define what you want to do.

Sounds easy let's check the rules hierarchy

All of those are abstract classes, so we will take a look into them and choose the one that is better for our case.

- **RBLintRule:** defines the protocol used for execute a rule.

This is a very general class, doesn't define:

- what to do when we are checking a class, for this we have to implement: `#checkClass:`. In our case we don't care about the class, so we can let the default that does nothing with the class
- what to do when we are checking methods, for this we have to implement: `#checkMethod:`.

In our case we should obtain the source code from the method and search in the source code all the variants for the consecutive messages `#globals` `#at:`, an innocent version of this can be:

A way to do it:

```
1 | checkMethod: aContext
2 | (aContext compiledMethod source findSt
3 | ifTrue:["we have a match in aContext s
```

I said that it is an innocent implementation because if we have a method with the code: `"(...) globals "yes a comment here!"` `at: (...)"` or `"(...) globals` `at:`

number” we aren’t causing a match and if we have:
 “(...) globals at: something ifAbsent: (...)” we are and
 we shouldn’t.

Probably we should always use one subclass of
 LintRule with more features implemented otherwise we
 have to extend LintRule to define what to do in case of
 matching, model a result, transforming code (...)

- **RBBasicLintRule:** it’s another abstract class that adds
 a result associated to a rule. We still should redefine
 checkMethod and/or check class but we have a
 proposal for a result:

```
1 | checkMethod: aContext
2 | (aContext compiledMethod source findSt
3 | ifTrue:[result addClass: aContext sele
```

This can be a good extension point, but again... you
 have a lot to implement here.

- **RBBlockLintRule:** it’s an abstract class, that by default
 specifies that the resultClass corresponds to a selector
 environment.
 This class doesn’t add a lot of behavior but we reduce
 errors related to the result handling.
- **RBParseTreeLintRule:** here is where everything gets
 interesting.

The main point in this class is to offer an
 implementation for #checkMethod: the implementation
 motivation is to check methods using ast’s
 representations.

We will have a wanted tree and then we will obtain the
 AST for the checked method, in order to find a match
 we observe if the wanted tree is subtree of the method
 tree.

Also it’s possible to define metavariables where we can
 specify that we want to match with a node type but not

necessary the value of it.

In our example our matching expression will look like:

```
//  “@lotOfStuffBefore globals at:
    “@lotOfStuffAfter
```

With this expresion we are saying:

- “@lotOfStuffBefore: ignore everything before globals (receiver / message send):
- “@lotOfStuffAfter: ignore everything after at: (object / messages)

(For more information about metavariables see my first post)

The solution implemented for the check method introduces new objects:

- **Matcher:** Is the responsable for visit an AST and verify the matches. Is the matcher the one who interprets the matching expressions. First of all we have to specify if we are matching with a method (matchesMethod: aString do: aBlock) or an expreesion (matches: aString do: aBlock) pattern.

We can add more than one matching expression, it's very important to know that every time we add an expression in the matcher inside we are adding a RBSearchRule for that expression.

When the tree rule sends the message executeTree: to the matcher at the end the matchers iterates over all the rules visiting the nodes delegating in the RBSearchRules to perform the match.

It's important to say that the search rules aren't deleted automatically, so, if you want to reuse the object probably you should reset the rule, resetting the matcher.

- **ParseTreeEnvironment:** Is an specialized SelectorEnvironment that makes it possible to detect

the selection interval for an expression inside the method, using the ast matcher

This class is still abstract because we should categorize it, adding a name and initialize the rule with the matching expression patterns.

In functionality is almost the same that before but we have other abstraction level, the result is handled automatically, and we don't have to worry about checking a method or a class only to define the desired matching expression.

- **RBCompositeLintRule:** Is just a composite for rules.
- **RBTransformationRule:** The main idea here is to produce a transformation in the system, for this implements: #checkMethod: in a similar way that RBParseTreeLintRule the difference is that if we find a match we will produce a modification in the code and then we change the method with the new version of the code (recompiling the new method).

In order to solve this the rule adds some objects:

- **RBParseTreeEnvironment:** with all the results, the results are: RBAddMethodChange to track a change in a method
- **RBParseTreeRewriter:** It's a subclass off the matcher (RBParseTreeSearcher) and again the main point is that this is a visitor that works over a method AST changing it depending in the matching and transforming expression.

The transforming expression also works with metavariable and usually we use the metavariables defined in the matching expression to specify the transformation.

Resuming:

1. Before start you should choose if you want to:
 - perform a search
 - do a match
2. Implement your rules because all of them are abstract, probably you will end up using the Tree rules because are more automatics and powerfull than to basics.
3. When you are defining your rule:
 - give a name for it
 - define if you will use a method or an expression type pattern
 - write your patterns and add them to the rule
 - if you are in a search rule to the matcher
 - if you are in a transformation rule `rewriteRule`
 - define what to do with a result
4. run your rule
5. use your results
6. if you want to reuse it reset your rule, again if it's a search rule, reset the matcher, if it's a transformation rule the `rewriteRule`.

In our example:

- we want to match, so let's create an object that extends: `RBParseTreeLintRule`:

```

1 | RBParseTreeLintRule subclass: #SearchGlobal:
2 |   instanceVariableNames: ''
3 |   classVariableNames: ''
4 |   poolDictionaries: ''
5 |   category: 'Blog-example'
```

- we have to implement the abstract methods:

```

1 | name
2 | ^ 'Find all potential wrong usage in with g'
```

- I want to match an expression type because I do not care about the rest of the method, I want everything that contains the messages: `#globals` `#at:`, we also

have to say what to do with the matching node in this example I will open an inspector:

```
1 initialize
2 super initialize.
3 self matcher
4 matches: ``@lotOfStuffBefore globals at: `
5 do: [:theMatch :theOwner | theMatch inspect
```

- now we should run the rule:

WARNING: this can take a time because you will check the hole system

```
1 SearchGlobalsAtUsage new run.
```

To avoid this you can restrict the environment for your rule, an example:

```
1 rule := SearchGlobalsAtUsage new.
2 environment := RBClassEnvironment class: Re:
3 RBSmalllintChecker runRule: rule onEnvironm
```

If you have matches then you will see the inspector.

So, as we can see this is quite complex, and in the sinopsis you can see that before doing anything you have to make too many decisions, the idea behind Flamel is to make it easier.

The equivalent code (with a restricted environment)

using Flamel for all this is:

```
1 FlamelMatchAndTransformRule new
2 matchingExpression: ``@lotOfStuffBefore g:
3 scope: environment;
4 run;
5 result
```

If you evaluate that and inspect it you can search your results



// *I think this is quite cool to replace all that code (with class creation included) with 5 lines.*

And this was all for understanding a little bit the rewrite engine and see Flamel in action for today

Posted in [Flamel](#), [Pharo](#), [Programming](#), [Rw tool](#) | [Leave a comment](#)

Searching in Flamel!

Posted on August 29, 2013

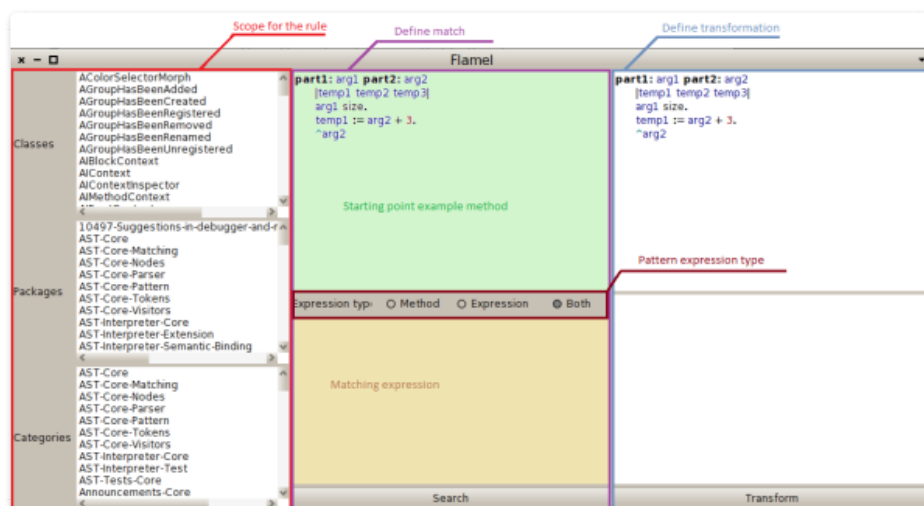
I want to show a proposal for using the matching expression builder.

The idea is to keep it simple, so if you think this is very complex, let me know!

First of all open Flamel-UI, in a workspace:

1 | **Flamel** **new** openWithSpec.

For those non familiarized with the project, you will see something like this:





A short summary:

- *Scope for the rule*: Here we restrict the scopes for the rule, for those familiarized with the RW-Engine behind this we have the RBEEnvironment objects.
- *Define match*: The idea here is to help to define the matching rule, for this we have:
- *Starting point example method*: We choose a method to use as an example for the pattern, you can change for the method you want, and you have more options available (just, right click and see)
- *Pattern expression type*: 3 options here:
 - Method: The pattern corresponds to a method
 - Expression: The pattern corresponds to an expression, so we don't have the selector definition plus the optional temporary variables definition
 - Both: Basically, try to search the matching expression as a method and as an expression
- *Matching expression*: The pattern we will search
- *Define transformation*: The same idea for the matching but in this case to help you with the transformation

How to use Flamel in 5 step

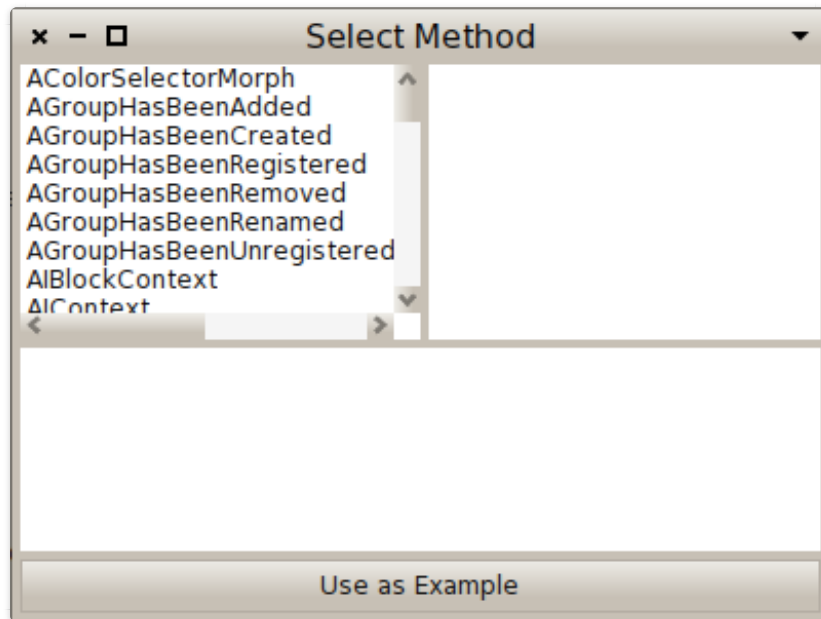
Choose the scope for your rule

1.  *if we don't clarify this we will look in the hole image and this takes time*
2. Select your base example: You can use the default example or... choose an example method that is better for you

 *The example will help you to define the pattern*

For change the example, right click in the Starting point example method and choose: "Change example", you

will see a window like this:



Where you can choose the class and method you desire to use as an example.

3. Choose the Pattern expression type: If you are not sure, just let the both and we will try to do our best, but if you really know if you want to define a pattern with a method syntax or not (an expression) choose the best option.
4. Define the matching expression: For this we use the right-click menu options to help us.
We can start choosing “Reset expression” and then we will see the same code in the example as a pattern
Then we use the menu to modify the expression
5. Perform the search just clicking in the Search button.

And then see your results!

Posted in [Flamel](#) | [Leave a comment](#)

Just want to match!

Posted on August 6, 2013



My face inspecting the empty result set

It took me many hours (and too much coffee) to realized all of this and I want to share with you some tips to help you define the matching pattern you **really** want to aply

- Pay attention to the dots! ``sel "@.Statements1. self subclassResponsibility. "@.Statements2` is very different to: ``sel "@.Statements1 self subclassResponsibility. "@.Statements2`
- Matching the selectors is not a simple task
- Is not the same matching an expression that matching a method one line can change everything!

If you say:

```
1 </pre>
2 matcher <strong>matchesMethod:</strong> aMa
```

You are particullary saying that aMatchingExpression should

be parsed doing:

RBParser parseRewriteMethod: aMatchingExpression

instead **of**

RBParser parseRewriteExpression: aMatchingExpression

And this mean that:

1. Your string is well formed, if you have a syntactic error you will see the window saying that the expression is not correct
2. Your string has a method structure
3. Your metavariables will play as a message send or as a variable depending on the context

A fast example, imagine this expression:

// *`anObject size.*

Let's play we are the a cool parser and someone told us hey... this is a string, but you should interpret it as a mehod... so:

`anObject = the selector so I *want all the **unary** methods*

size = a **variable named 'size'** because it can't be a parameter because my message is unary, it's not a variable definition it has to be a variable.

// *So as a cool parser I say that I will match with all the unary methods that the body is just an unique sentence that contains a variable called 'size'*

Now, imagine someone told us this is an expression... so:

`anObject = the object, I do not know nothing, only that here goes an unique object

size = before I had an object so here I should have an invocation for the message #size.

“ So as a cool parser I say that I will match with all the methods the expressions that sends the message size

- If you look for a particular message send you should care about the structure:
 1. Do you want all the senders? for defining the expression is not the same *object messageToFind* that *object message anotherMessage messageToFind moreMessages*
 2. It can be inside a block?
 3. It can be invoked as a symbol? *object perform: #messageToFind*
 4. It can be in any part of method? (the first sentence? after that can be more message sends?)
- The matcher uses **pattern matching**! You give an alias and when it matches then is bounded and that's

This pattern:

```
`sel "@.Statements1. self messageToFind.
"@.Statements1
```

Does not match with:

```
1 | example
2 | self oneMessage.
3 | self messageToFind.
4 | self otherMessage.
```

Because we use the same variable name! **Statements1** and in the matching it bounded to the statement **self oneMessage** that is NOT identical to **self otherMessage** so we don't have the match.

But! It should match with:

```
1 | example
2 | self oneMessage.
3 | self messageToFind.
4 | self oneMessage.
```

But this feature is really cool if you want to search repeated code in the same method...

- If you are matching inside a method, does the method defines temporary variables?
- Remember that when you parse if the variable definition is empty it gets ignored but if you do not put it in your pattern and the method defines a temp you are excluding it.

An example

```
1 | example
2 | |aTempVar|
3 | self messageToFind.
```

matches with method pattern:

```
`sel |`@vars| `object messageToFind
```

but does NOT match with:

```
`sel `object messageToFind
```

In the end I can understand that all this “tinny” details are very important... but if you want to match a common case it can be really ugly and it’s VERY easy to do it wrong.

But do not worry too much one big goal for Flamel it’s to offer a simple API to avoid all this common errors.

Posted in [Pharo](#), [Programming](#), [Rw tool](#) | [Leave a comment](#)

Developing under develop

Posted on August 5, 2013

I think that this had happend to a lot of smalltalkers with the time... After a while you just get used to do some tasks that aren't very intuitive for a newcommer and started to feel "natural". Yes... the classic user that get used to a system...

And these days I've been watching the frustation of someone that is starting the path and trying to help I just realize that I have a lot of tips and tricks that I follow unconcious and decided to make an entry here maybe this help and more people with more expirience would share their tips.

Starting

Before start your coding you have to setup your environment, an one choose is which image? Wich version? I strongly suggest the lastest but maybe you should keep reading before choose.

Why does the title say developing under develop?

Because when I start a new project in pharo I want to use the lastest features, Pharo is getting really cool and has improved a lot and to use all those feautres I need the lastest image.

Other reason is that while you are programming your system you are testing Pharo also and time to time you find some problems and correct them are not so hard, so it pushes you to improve Pharo itself!

So in order to have all the benefits you have to use the latest image (under develop) and with that you should have concience in the consecuencies and take some considerations.

Mental preparation

Ok you are in a system under developing, IS NOT PERFECT but **it is in the path to be** one step near to the perfection and to achive that needs your help. Basically applies to the system the same rule that applies to your development:

- The system may change: some apis can change and maybe you are an user of that API. This is not bad! Probably it's a refactor to improve the code, so you get benefit because your code will be more expressive.
- The system can have bugs! Yes as I said is not perfect and it's not on purpose.
- Sometimes the latest image have problems, yes we would love to be perfect but sometimes there are some nasty side effects and some functionality that stop working, but to find it sometimes you have to release and wait for feedback and you are a great candidate to give that feedback.
- The debugger does not mean that everything is lost, your first approach should NOT be close the debugger and say abandon abandon!
- Get attach to an image is not a great idea... Your "latest" image is not there for stay with you for a long time, you have chosen to use under develop image,

this means that your image will change (in the happiness path but change anyway) and deliberated decide to attach with an old particular version from a changing image has the worst of two worlds: Now you are not in an stable but you are stuck in a version that nobody is using and by consequence you don't have the new fratures/fixes

- The tests are your friends 😊 (As usual!)

Tips and tricks!

If you are new at Pharo y strongly recomend to take a look in [Mariano Peck's](#) blog particullary the post called: [Pharo tips and tricks](#). The shortcuts are really cool and from your previous experience maybe you would say... I can't do that only with my keyboard... but the true is that probably you can do it! Just take a look in [Key mappings](#)

Mitigate the errors

Some tips when you are developing:

- Use fresh images, if you are in lastest Pharo a week with the same image is really A LOT
- Allways use a repository to share your code, I strongly recommend [smalltalkhub](#)
- If your project has a complex setup invest some time making a configuration
- Do not make a lot of actions before commit, small and numerous commit is too much easier than one big commit
- Reverting the changes is not so easy. There are a lot of effor in this topic and is comming, but is not ready! So, try to keep it simple
- If you find extrange behavior send email to the user mailing list: pharo-users@lists.pharo.org nobody will

judge you for asking!

- Language barrier is not so hard with a tolerating community, Pharo community is full of non native-english speaker and is more important to improve Pharo than writing poetry, just ask and try to make that understandable (look... if I'm writing this blog with my horrible english, you shouldn't care at all)
- If you can reproduce a problem open an issue in the [issue tracker](#)
- Save your image regularly, if you have worked for a while and you have uncommitted changes save your image regularly!
- Before killing your image because "does not respond" you have the resource to interrupt the process just press **alt + '.'** or **cmd + '.'**
- **Contribute and give feedback**, it's a way to keep growing

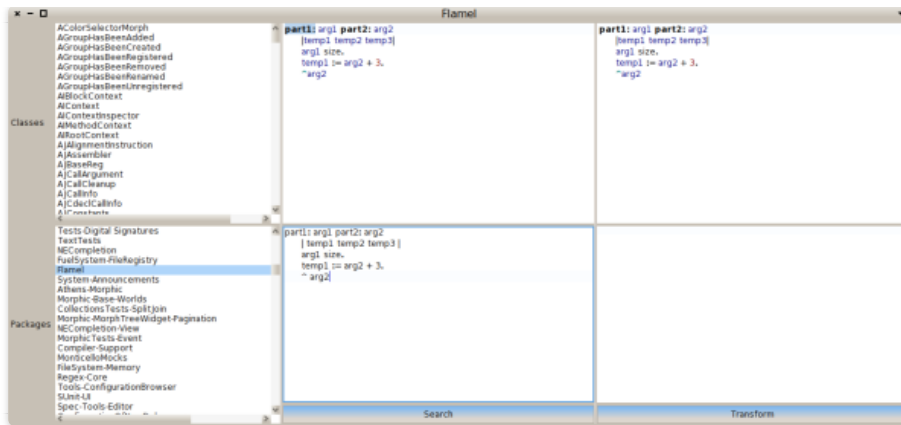
I'm sure we have a lot more, but at least is a beginning!

Posted in [Pharo](#), [Programming](#) | Tagged [Pharo](#) | [Leave a comment](#)

Flamel is growing!

Posted on July 31, 2013

One image it's a great way to show the current status:



A quick view

As you can see we still have (a lot of) work to do... But we can:

- Choose the scope where we run the run (right now a class/es or a package/packages)
- Enter the expresion for matching and transforming
- Search the matches
- Apply the entire rule (matching expresion + transforming)
- We have some actions implemented for the menus (same code, containing code, defining a variable)
- Inspect the result for a Search or Transformation action

But still now I have some thing that I observed: The rewrite engine its in fact very cool, and surely does everything we want to do.

But it's very complex you have to choose the matcher , modify the pattern (is an expression? is a method? you have multiple options? is about an argument?), set the environment (do you want to run where? each Environment is represented but wait! Which matcher are you using? Because not all have the feature, and they are not all polimorfic when you want to match. And of course do not even talk about the regular expressions and the usual doubts we have (with temps? with N arguments? previous statements? and don't forget the dot!)),

and... before starting... do you want to match? or do you want to transform? You have to choose!

And usually you don't have all the information to choose that in an early moment, so what I've done it's to implement a rule that delegates in existing components but takes the decisions for you... By example: Do you want to transform? Or just match? That depends... do we have a transformation expression? If that is true we want to search and replace.

```
1 | testMatchAnUndefinedExpressionNotFind
2 | | rule |
3 | rule := FlamelMatchAndTransformRule new.
4 | rule matchingExpression: 'testFlamelFlamel'
5 | rule scope: (RBClassEnvironment class: Flamel)
6 | rule run.
7 | self assert: (rule result isEmpty )
```

I want to change that to be something like:

```
1 | testMatchAnUndefinedExpressionNotFind
2 | | rule |
3 | rule := FlamelMatchAndTransformRule new.
4 | rule matchingExpression: 'testFlamelFlamelN'
5 | rule applyToClass: FlamelSearchMock.
6 | rule run.
7 | self assert: (rule result isEmpty )
```

So, I have lot of work!

Some known bugs, and some unknown you can check the todo list in a Trello

dashboard: <https://trello.com/b/XqfJGqeB/flamel>

It would be great to improve the UI because it's quite "rustic".

And my plan for this week it's:

- Implement result views ==> with this one we have a first real prototype for the tool because we can: filter the scope, enter a matching/transforming expression and understand the results

- Implement actions for transforming menu

And as allways if you want to try download from
[Smalltalkhub!](http://smalltalkhub.com/mc/gisela/Flamel/main) <http://smalltalkhub.com/mc/gisela/Flamel/main>.

To open the browser just:

1 | `Flamel new openWithSpec`

All the comments that you have are very wellcome, please
don't hestiate in comment or send me an email directly to me:
giseladecuzzi@gmail.com.

Have a great week and nice coding!

Posted in [Flamel](#), [Pharo](#) | [Leave a comment](#)

Still true

Posted on July 30, 2013

*54. Beware of the Turing tar-pit in which everything is possible
but nothing of interest is easy.*

—Alan Perlis, *Epigrams on Programming*

Posted in [Flamel](#), [Rw tool](#) | [Leave a comment](#)

Developer's tool in Pharo

Posted on July 1, 2013

I've finished an internship in RMOD team in Inria and I was doing some retrospective in the work I've done.

The idea for my internship was to improve the developer experience while using Pharo [<http://www.pharo-project.org/>], (quite challenging!), so in order to do this I was mainly focused on 4 points:

- Suggestions
- Code Navigation
- Syntax Coloring
- Class Definition representation

Suggestions

While we are coding we usually want to apply actions depending on the element we are writing/seeing, for example if it's a variable we may want to rename it. But in order to do this, we have big menus to find what we want, usually with lot of options that don't apply.

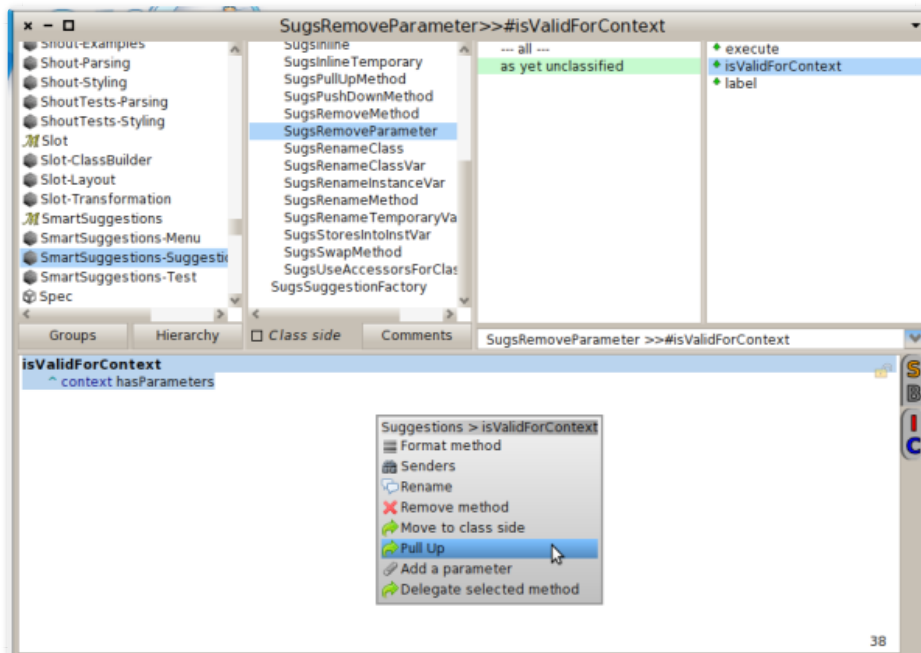
The idea in smart suggestions is that based on the context offer only the relevant options.

We use the current AST to do this through RBPParser (parseFaultyMethod;parseFaultyExpresion;) and Opal Semantic Analysis.

We choose RBPParser because we can parse faulty expressions, with this feature we can offer suggestions while the user types the method.

Opal is the new compiler integrated in Pharo, it's a very clean implementation and we think that will replace the old compiler, so we didn't want to couple suggestions with an old compiler. The semantic analysis helps us to complete information form a code, because there is information that we can not know from the code without the context, by example: if we have a temporary variable or an instance one.

With the best AST node for the selection we have the available suggestions and with the semantic analysis we go one level deeper for the variables getting to know the nature: temporary, instance or class to refine the suggestions.



If you want to define a new suggestion you only have to annotate a method with the scope where you would like to offer the suggestion and return an object respecting the **SugsSuggestions** API.

For returning the object you can extend `SugsSuggestion` and redefine the desired methods, you can see various examples with the current implementation in the `SmartSuggestions-Suggestion` package.

Also you can instantiate a generic suggestion: with `SugsSuggestion class>>#for:named:icon:` where the first parameter is a block with the action to execute and receives a valid `SugsAbstractContext`.

The available scopes are:

- `<assignmentCommand>`
- `<classCommand>`
- `<classVarCommand>`

- <instVarCommand>
- <literalCommand>
- <messageCommand>
- <methodCommand>
- <sourceCodeCommand>
- <tempVarCommand>
- <undeclaredVarCommand>
- <globalCommand>

Here we have an example defining a Dummy action in

DummyAction class >>

```

1  newFormattingSuggestion
2  <globalCommand>
3  ^ SugsSuggestion
4  for: [ :context | context formatSourceCode
5  named: 'My global formater'
6  icon: Smalltalk ui icons smallFindIcon

```

Navigation

Sometimes while browsing code we think in programming terms instead of text, for example we think in a messageSend or a statement instead of word, spaces or symbols.

The idea is to use context information and let the programmer navigate the code thinking in those terms.

In order to do this we find the best ast node through the RBPParser and we offer navigations in different directions:

- Parent: The node that contains the selected one. For example if we have the code 'aNumber between: anotherNumber' and we are selecting the variable anotherNumber if we navigate to the parent we will go to the message send.
- Sibling: The node in the same level as the selected node. For example in a temporary variables definition: '| one two three |' if we are in the variable one we can go to the siblings two or three.
- Child: Node contained by the selected node. For example if

we are in a message send: 'aNumber between: anotherNumber' we will go the parameter anotherNumber.

It's very easy to see how the Suggestions + Navigation work together, and how with not too much effort you can improve a lot.

If you want to activate the Node navigation using Command (or Control if unix) + Arrows go to the System Settings and activate: AST Navigation.

ASTColoring

We want to color the syntax we are writing having the most information possible, in order to be able of select the scope where we are or show information associated to that piece of code.

In order to do that we use the AST and the semantic analysis (we need the semantic analysis because we want to show different kinds of variables with different colors, like undeclared variables), through the RBPParser (parseFaultyMethod:/parseFaultyExpression:) to obtain the AST representation. The implementation it's very simple because we can define a new Visitor to which we delegate the coloring algorithm and once we define the coloring from each syntax representation we just visit the tree: ast acceptVisitor: "visitorImplementation".

To enable the syntax coloring activate in System Settings: Enable AST based coloring.

Class Definition

Include a representation modeling the arguments and the message as 1st class object. About this point I have written a whole blog post.

Summary

After working with the AST I can say that is a very powerful tool and also is very easy to use. The visitor model is great and very flexible. But we must be careful because we can easily

add too many responsibilities there and complicate the design. Changing the AST structure has an obvious impact in the visitor implementors, and in the current Pharo implementation this can be complicated because we have some interesting users of the AST (like the new Compiler), and if we make a mistake it is very easy to break the image, so if we are going to do some experiments with this it is better to change the compiler first.

Obtaining an AST representation is not so hard to do when we have valid code, but when we are trying to find the best representation for a faulty expression things get nasty, and if we want to colour the text while we are writing we need this feature.

But the main point of everything is the re-affirmation that Pharo is a great tool for doing experiments, the power of a live environment and reflectivity give us an edge over other environments.

With not so much effort we can implement some tools that improve a lot the programming experience a lot, and everyone can do this!

Posted in [Pharo](#), [Programming](#) | Tagged [Pharo](#) | [Leave a comment](#)

A look into Spec

Posted on June 26, 2013

Today I'm going to write about implementing the UI.

Since I want to have a nice UI in Pharo I choose **Spec** to build it, because it's an on going project and there are lot of effort there, also I want to try it 😊 .

Searching I found some tutorials and demos but some were quite old and not fully compatible with current version of Spec, so I will do a brief tutorial.

■ ■ *Spec was developed by Benjamin Van Ryseghem, you can find more information in: <https://pharo.fogbugz.com/default.asp?spec.5.5.2>*

Anyway I think that the best place to take a look is in the examples that comes under Spec in **Spec-Examples-PolyWidgets** (mainly) and **Spec-Examples-Widgets**.

Doing the minimun for opening a Window

We want an empty window... but our empty window, so... how?

1. We are going to create a class extending ComposableModel

■ ■ *ComposableModel subclass: #TestSpec
instanceVariableNames: ”
classVariableNames: ”
poolDictionaries: ”
category: ‘Flamel-UI’*

2. We have to implement 2 methods

2.1. **initializeWidgets** which responsability is to setup the components we will use (right now nothing)

■ ■ *initializeWidgets
“nothing to do here”*

2.2 **defaultSpec** (class side) which returns a Layout saying which components will be render. There is a number of different layouts implemented: SpecColumnLayout, SpecRowLayout, and in the class methods of SpecLayout you

can find a lot more. We will choose a very easy one:

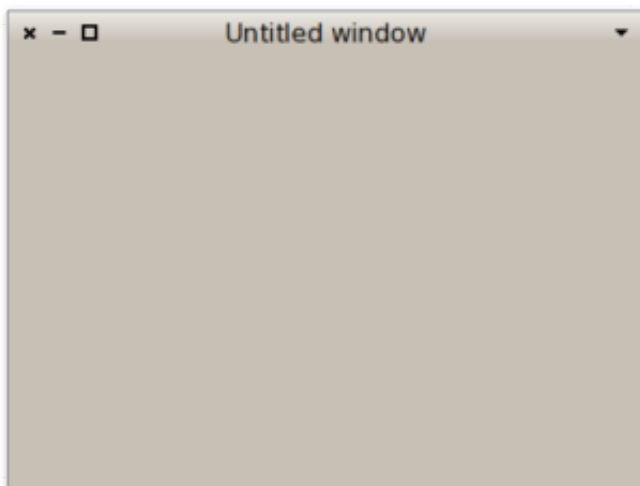
```
// defaultSpec
<spec>
^ SpecLayout composed
```

In fact you can name the method as you want, only need to be annotated with the **<spec> pragma**.

3. Open the window, we evaluate in a workspace:

```
// TestSpec new openWithSpec
```

And we obtain:



Nice... but quite empty... let's put something inside!

Adding a component

1. Choose the widget to add, you can see some in **Spec-Widgets**, we want a text area so we will choose a `TextModel`.
2. In order to render the component we need to: create it and tell the layout that we want to render it
 - 2.1. Create the widget: define an instance variable and instantiate it during the **initializeWidget** using the Spec way to do it:

```
// initializeWidgets
```



```
self instantiateModels: #(
  text TextModel
)
```

We need to define text as an instance variable because *instantiateModels* assume that and try the elements of the array as an association: *variable – kind* where **variable** is a property in the object and makes it point to a new instance of the **kind**, in order to avoid the troubles we will define the variable.

Be careful with this (as Sean mention) if you define the variable while you are debugging you can get some nasty errors, so the advice is to define the variable first and then open the window.

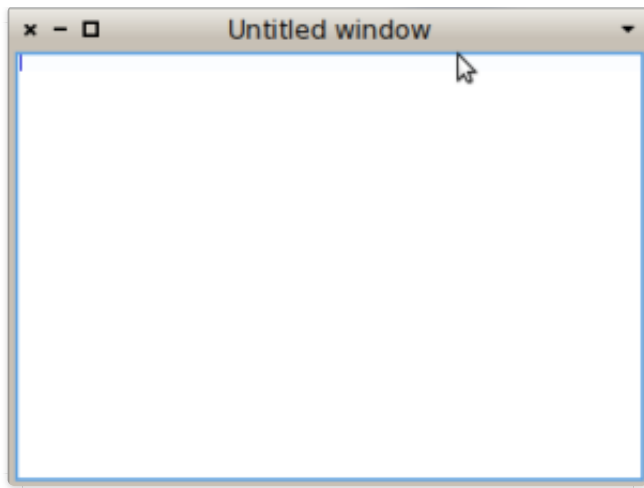
At this point if we get anxious to see our cool text component and we open the window we will see that is still empty, this is because we never added it to the layout! So it's not rendered.

2.2. Add the component in the layout, for this we will modify the *defaultSpec* class method and add a getter for the text property:

```
/// defaultSpec
  ^SpecColumnLayout new
  add: #text;
  yourself
```

We need the getter because the layout tries to access the variable sending the symbol as a message

3. Open it



Communicating Components

Now we can open a window and have our components, but now we want to have several components that depends between them.

We will build a window with a text editor, when the content in the text editor changes we will see the message “The text changed” we can accept that pressing a button that says “Ok” and then the message will change to “You are up to date” (until someone change the content in the text ...).

For doing this we will use:

- a TextModel called: example
- a LabelModel called: changes, starts with the text: “You are up to date”
- a ButtonModel called: acceptChanges, starts disabled and with the label: “Ok”

```

// initializeWidgets
self instantiateModels: #(
    example TextModel
    changes LabelModel
    acceptChanges ButtonModel ).
changes text: 'You are up to date'.
acceptChanges
label: 'Ok';
disable.

```

This is like we had before, only that we use other components that needs a little more of configuration.

The real challenge here is to link some effects: when the contents of *example* changes we want to change the message in *changes* and enable *acceptChanges*.

For defining this kind of interactions we should define the method **initializePresenters** defining the interactions, lucky our models understand several useful message to define an action to perform for an event, an example for a TextModel `#whenTextIsAccepted: , #whenTextChanged:, ...`

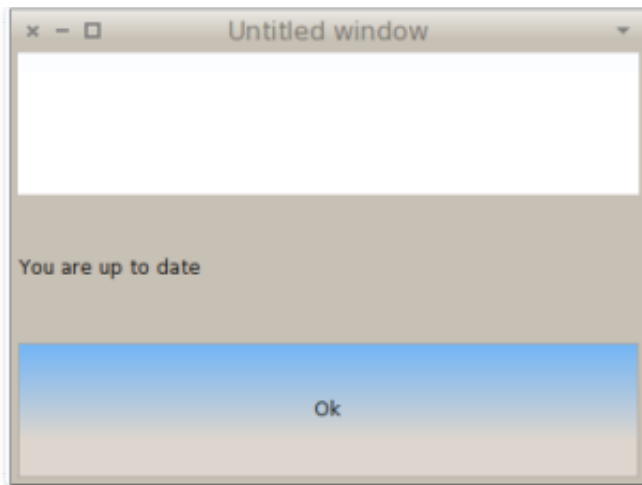
Let's define the actions:

```
// initializePresenter
example whenTextIsAccepted: [ changes text: 'The
text Changed'. acceptChanges enable ].
acceptChanges action: [ changes text: 'You are up to
date'. acceptChanges disable ].
```

In this part we are done, now remember that in order to render the components we need to add it to the layout:

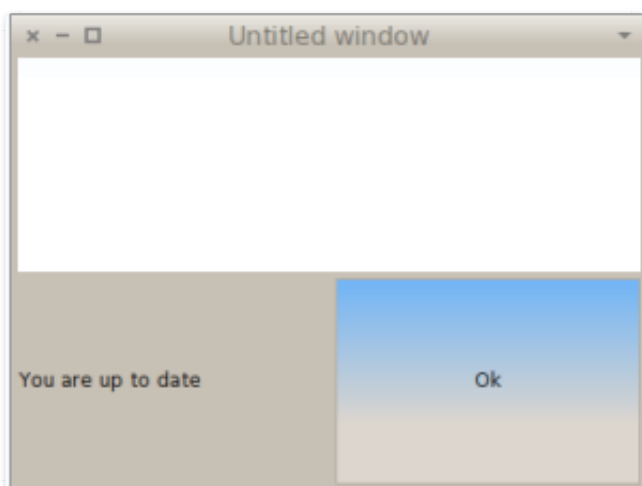
```
// defaultSpec
<spec>
^ SpecColumnLayout new
add: #example;
add: #changes;
add: #acceptChanges;
yourself.
```

And we see:



Quite easy and good, but lets improve the layout and play a little:

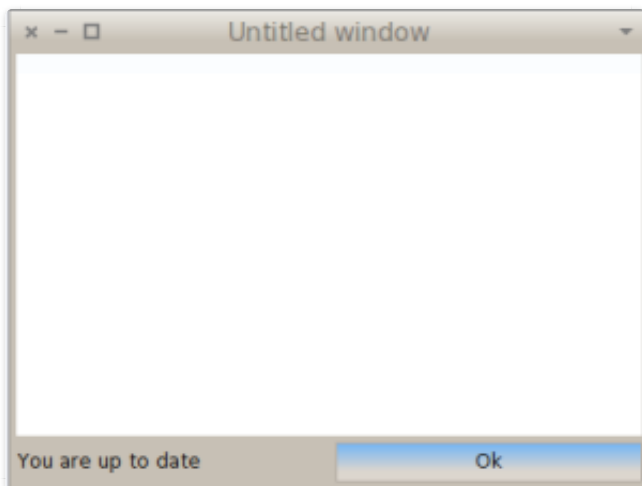
```
// defaultSpec
<spec>
  ^SpecLayout composed
  newColumn: [ :mainColumn | mainColumn
  add: #example;
  newRow: [ :feedbackRow | feedbackRow
  add: #changes;
  add: #acceptChanges]];
  yourself.
```



Better but still a little detail:

```
// defaultSpec
<spec>
```

```
^SpecLayout composed  
newColumn: [ :mainColumn | mainColumn  
add: #example;  
newRow: [ :feedbackRow | feedbackRow  
add: #changes;  
add: #acceptChanges  
] height: 26.];  
yourself.
```



A lot better! And It was quite easy, we can see that is well separated the layout from the components, and that we have a nice proposal for organizing our UI-Code:

- We define the layout in `#defaultSpec`
- We instantiate the components we are going to use in `#initializeWidget`
- We define the interactions in: `#initializePresenter`.

This is very nice, but now I have a UI that separates the logic but still has a lot of responsibilities, is the UI who knows if I've accepted a change or if I'm up to date, also knows exactly what to tell me... I don't think that this is the UI responsibility I want to model a class and that the UI shows it and not all in one like we have now.

And how do we do it?

Using a Model

We will have a model behind our window in 3 easy steps:

1. Create the model when initializing (you can create it or receive it, as you want, I will show the simplest way)

```

// initialize
model := AcceptableText new. "replace for your
domain class"
super initialize .

```

2. define the widgets, is almost the same that before

```

// initializeWidgets
self instantiateModels: #(
example TextModel
changes LabelModel
acceptChanges ButtonModel ).
acceptChanges
label: 'Ok';
disable.

```

3. define the interaction

```

// initializePresenter
example whenTextChanged: [:changed|
model text: changed.
self updateContents].
acceptChanges action: [
model acceptChanges.
self updateContents ]
updateContents
changes text: model status .
acceptChanges enabled: model hasUnreadChanges .

```

Look that when we are defining the actions we do 2 things:

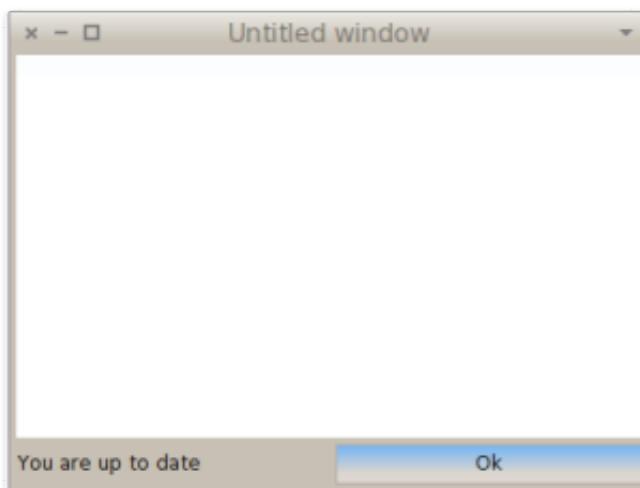
- 1) modify the model
- 2) update the content of the widgets to show the new state of the model

We can see this actions that are using blocks as **controllers** (in a MVC pattern) since they modify the model from an input in the view and then update the view to show the new state in the model, in other frameworks we can use Bindings that do exactly the same.

■ ■ *In fact Ben pointed that Spec is far more closer from a MVP pattern than an MVC*

In order to write a good UI we have to be very careful in how much code goes there, we probably will like to have small controllers delegating in the model instead of big controllers that does all the work for the model if we are not careful there we may end with anemic objects and spaghetti uis.

But to much talk and see how it ends:



Basically the same that before but now we have the linked ui using our domain object! Quite nice, and really really really simple to use.

We have a name

Posted on June 21, 2013

As you can see now in the tittle... **Flamel** is the name of the rw rule tool

Posted in [Flamel](#) | [Leave a comment](#)

The class definition message

Posted on June 21, 2013

Q: What is a class definition in Smalltalk?

A: Just another message send!

I love that simplicity... We don't need control structures because we send messages and as we have an abstraction that represents some piece of code everything is happiness (yeah... blocks are cool stuff).

But still knowing it I go to the System Browser... I see the class definition and I don't think that is another common message send, in fact when I realize that I can write that "magic code" in a workspace and have the same effect (creating the class) and see it in the system browser I said wow! After a while I thought "so lame... of course is just a message send".

But however I go to the class definition again select an instance variable and I want to apply the rename refactoring... so... I ask for suggestions... and I got... nothing... just selecting a string and that confused me and then I realize, ok it's just a message send and is just a string as argument so... that's logic. The thing is that is not true that is just a message send... because for some reason we expect more in some points, by example when we are in the class definition...

So I did some prototypes for changing the representation and meanwhile I change my mind in the process.

1. The class definition is not represented as a message node

I was radical... I said: A Class Definition is always a class definition, so I changed the parser and when parsing I tried to parse a class definition

The prototype for this can be found in:

■ ■ *MCHttpRepository*

location:

'<http://smalltalkhub.com/mc/gisela/ParsingClassDefinition/main>’

user: "

password: "

I started with some issues here... The main one is:

How do I know I'm in a class definition, in the prototype you will see a very "innocent" implementation for realizing this... I just assume that if the message identifier I parsed is #subclass: it's because a full message definition. There is no way to know if you are in a class definition until you finish the parsing.

Lot of the behavior is similar to a message send, and to implement the program node API we have to see how to share this code, it seems an effort without a real motive, because all

the problems I was having relies in the fact that I assumed that a class definition is not a message send...

2. The class definition is created from a message send

Something in between, I still changed the parser but...

transforming the message node parsed only if it was a class definition.

The prototype is in:

```
/// MCHttpRepository
location:
'http://smalltalkhub.com/mc/gisela/MessageClassDefinition/main&#8217;
user: "
password: "
```

I had less problems 😊 but still some strange behaviors, it was because I changed the AST and all the users of the ast should change, specially the visitors because now you have classDefinitionNode, categoryNode, variableDefinitionNode (...).

And then I have lots of failing test and for olving in the major part I endedUp delegating in the message node reference I had hold.

But this was expected, since I changed the AST structure in some point all the users of that structures need to be modified, but ended up having like the same that before... lot of behavior shared between a message node and a class definition, and sometimes the "new representation wasn't even desiarable.

By example asking for suggestions... if I'm writting a method and write something like:

```
/// Object
subclass: #JunkClass
```

```
instanceVariableNames: 'zzz'
classVariableNames: "
poolDictionaries: "
category: 'DeleteMe-1'.
```

If I ask for suggestions in zzz I expect something like...

“Extract local” and no “Rename” but in the definition space... I want that behavior, so... since that I decided that in some occasions you look the class definition message as a special stuff, and that depends on the context where you are, so changed the parser wasn't so great because at that point you don't how they would like to treat that particular message, that it's mostly user-decision.

3. Request the class definition explicitly

I don't change the parser, I add a message to the nodes
“asClassDefinition”

```
■ ■ MCHttpRepository
location:
'http://smalltalkhub.com/mc/gisela/ClassDefEnhanceTree/main&#8217;
user: "
password: "
```

And then let the user of that tree decide in which case he is and if it's who apply to do the transformation or not.

With this approach the direct user of the ast must know if he wants this representation or not, the original tree doesn't change, and we can think that is that we let that two representation exists, is a similar approach to the semantic analysis by default you don't have the information and if you want that you explicitly need to say... ok do the analysis.

And in the end

I finish in the beginning, the AST represents the syntax structure and that is how we understand it and how we manipulate it, sometimes we want a transformation, we want more and we enhance that structure for better understanding.

At least for the suggestions it was only in the class definition view. And after dealing with the consequences of change I get convinced that we send message and that's it, we don't have and neither need an special syntax for defining a class, is just sending the right message to the right object, it's simple and it's very good.

And when we want another approach now we can have it, very easily.

Posted in [Pharo](#), [Programming](#) | Tagged [AST](#), [Class Definition](#), [Pharo](#), [Programming](#), [Software](#) | 1 Comment

Defining the UI

Posted on June 13, 2013

The main feature we want for the rewriting tool UI it's to be example driven. Two very similar examples:

Option 1 – All in the same view

RW Tool

Scope

☐ Class
 ☒ Category
 ☐ Package
 ☐ Method

Match

<expression defined>

Search

Transform

<expression defined>

Run

Example method showing matching

Example method showing transformation

Option 2 – A tabed view

RW Tool

Scope

☐ Class
 ☒ Category
 ☐ Package
 ☐ Method

Match

Transform

<expression defined>

Example method showing matching

Search

RW Tool

Scope

☐ Class
 ☒ Category
 ☐ Package
 ☐ Method

Match

Transform

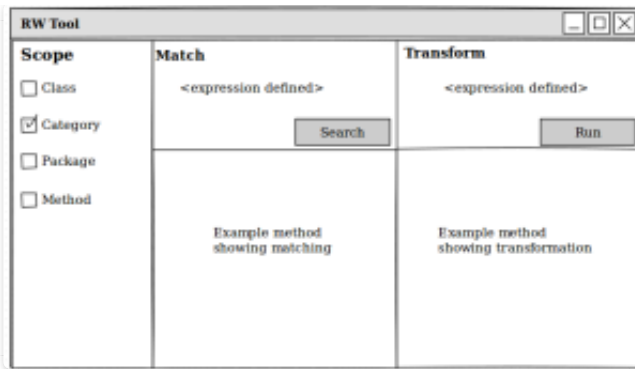
<expression defined>

Example method

Example method showing transformation

Transform

Option 3 – A 3-Column layout



We also want to be possible:

- select a method and drop into the example text editor.
- manipulate the ASTs nodes in the text editor to make simple define transformations and matching.
- highlight the matched parts from the example.
- select scopes to apply the rule.
- preview the changes.
- show the transformation in the example expression.
- modify an expression throw the example or writing the pattern.
- output the smalltalk valid code that produces the effect we see in the example.

Posted in [Flamel](#), [Pharo](#), [Programming](#) | Tagged [Flamel](#), [Gsoc](#), [Pharo](#), [UI](#) | 2 Comments

Next steps...

Posted on June 12, 2013

Now we have a very **powerful** tool **but** quite **complex** to use... The pattern's definitions ends up been confusing and lot of people dismiss the tool for this.

The fact that the **patterns are strings** is one of the causes in misunderstanding and encourage to produce lot of mistakes, when we want to match an expression thinking in a method definition we think in terms of: a method contains this statement, or declares this variable and not in complex regular expressions, it's also true that sometimes we end up with a cryptic expression, hard to understand by others (even for ourselves).

It would be great to have a very expressive api for what we want to do, for example make possible to say something similar to:

// *“some scope” pattern*
includesStatement: *anStatement;*
hasTemporaryVariable: *aVariable.*

We should write some nice **examples of usage** and also take a look into the **rewriting engine**, we could do lot of improvements.

All of this sounds very ambitious, so I will start for defining an UI for make the patterns auto generated in an easy way with lot of cool features, specially example driven when building expressions.

Once we have the UI and a nice video showing how to use it... we can improve the API and make it more expressive and if we have time we will take a look in the rewrite engine... but first let's do these easy to use!

Posted in [Flamel](#), [Programming](#) | Tagged [Flamel](#), [Gsoc](#), [Pharo](#) | 1 Comment



