Extending Renraku: the Quality Model of Pharo

Yuriy Timchuk, Stéphane Ducasse

October 25, 2023

Layout and typography based on the `sbabook` LaTeX class by Damien Pollet.

Contents

Illustrations

CHAPTER 1

Adding a new quality rule

To add a rule, subclass `ReAbstractRule` or one of the special rule classes that will be discussed. Most of tools rely on the classes hierarchy to select the rules for checking.

Each rule should provide a short name string returned from the `#name` method. You also have to override the `#rationale` method to return a detailed description about the rule. You may also put the rationale in the class comment, as by default #rationale method returns the comment of the rule's class.

1.1 Specifying an interest

The class-side methods #checksMethod, #checksClass, #checksPackage and #checksNode return true is the rule checks methods, classes or traits, packages and AST nodes respectively. Tools will pass entities of the specified type to the rule for checking. A rule may check multiple types of entities but please avoid checks for types inside of rules. E.g. if a rule checks whether an entity is named with a swearing word and does this by obtaining the name of the entity and matching substrings. It is completely fine to specify that the rule checks classes and packages as you don't have to distinguish which entity was passed to you.

1.2 Checking

The easiest way to check an entity is to override the `#basicCheck:` method. The method accepts an entity as a parameter and returns true if there is a violation and false otherwise.

1.3 Advanced checking

While there is a default implementation that relies on `#basicCheck:` and creates an instance of `ReTrivialCritique`, it is advised to override the `#check:forCritiquesDo:` method. This method accepts an entity and a block which could be evaluated for each detected critique. This means that one rule can detect multiple critiques about one entity. For example, if a rule checks for unused variables it can report all of them with a dedicated critique for each.

The block that should be evaluated for each critique may accept one argument: the critique object, this is why you have to evaluate it with #cull:. You may use the `#critiqueFor:` method if you don't feel comfortable with critiques yes. For example:

```
    self critiqueFor: anEntity
```

will return `ReTrivialCritique` about the entity. Later you can update your code to create other kinds of critiques more suitable for your case.

1.4 Testing

It is fairly easy to run your rule and obtain the results. Just create an instance of it and send it the `#check:` message with the entity you want to check. The result is a collection of critiques. For example inspecting

```
RBExcessiveMethodsRule new check: Object
```

should give you a collection with one critique (because the Object class always has many methods ;)). Go on click on the critique item and inspect it. You will see that there is a special "description" tab. This is the power of critique objects, they can present themselves in a different way. Guess what: you can even visualize the critique if needed.

1.5 Group and Severity

It's a good idea to assign your rule to a specific group. For this override the #group method and return string with the name of the group. While you can use any name you want, maybe you would like to put your rule into one of the existing groups: API Change, API Hints, Architectural, Bugs, Coding Idiom Violation, Design Flaws, Optimization, Potential Bugs, Rubric, SUnit, Style, Unclassified rules.

You can also specify the severity of your rue by returning one of: #information, #warning, or #error symbols from the #severity method.

1.6 Reset the cache

To have quality assistant pick up your changes you have to reset a cache. Do this looking for Renraku in the Setting Browser. Or simply executing:

```
ReRuleManager reset
```

When you load complete rules into the system, the cache will be reset automatically. But as you are creating a new rule and it is in the incomplete state you have to reset the cache once you are ready.

1.7 Running rules

Usually you don't have to run rules yourself, this is done by tools. But it is a good idea to be familiar with the API available in rules for developing your own tools and dubugging unexpected behaviour.

First of all tools traverse the hierarchy of rule classes and selects the required ones buy testing them with `checksMethod`, `checksClass`, etc...

The main method that the rules implement to check entities is #check:forCritiquesDo:. It accepts the entity to be checked as the first argument, and the block to be evaluated for each critique. The block may accept one parameter which is a critique object. For example:

```
rule
  check: anEntity
  forCritiquesDo: [ :critique |
    "do something for each critique" ]
```

There are also 3 convenience methods:

- check: Accepts an entity to check and returns the collection of critiques.

Part I

check:ifNoCritiques:

Similarly to previous one checks an entity and returns the collection of critiques. Additionally accepts a block that will be evaluated if no critiques are found.

Part II

check:forCritiquesDo:ifNone:

Similarly to #check:forCritiquesDo: checks an entity and evaluates the block for each detected critique. Additionally accepts a block that will be evaluated if no critiques are found.

1.8 Special rules

There are certain special rules with predefined functionality that allows to easily perform complex checks.

ReNodeMatchRule

The base rule for Pharo code pattern matching (relies on rewrite expressions). The rule operates on AST nodes.

Use the following methods in the initialization to setup your subclass:

- matches:
- addMatchingExpression:

add a string of rewrite expression to be matched by rule

- matchesAny:

same as previous but takes a collection of strings to match

- addMatchingMethod:

add a string of rewrite expression which should be parsed as a method

you may use #afterCheck:mappings: to do a post-matching validation of a matched node and mapping of wildcards.

```
ReNodeMatchRule >> addMatchingExpression:
  "add a string of rewrite expression to be matched by rule"

ReNodeMatchRule >> addMatchingMethod:
"add a string of rewrite expression which should be parsed as a
    method"

ReNodeMatchRule >> matches:
"add a string of rewrite expression to be matched by rule"

ReNodeMatchRule >> matchesAny:
"add a collection of rewrite expression strings to be matched by the
    rule"
```

ReNodeRewriteRule

The base rule for smalltalk code match & rewrite rules. The rule operates on AST nodes.

Use the following methods in the initialization to setup your subclass:

- replace:with:
- addMatchingExpression:rewriteTo:

add a ”from->to” pair of strings that represent a rewrite expression string to match and a rewrite expression to replace the matched node.

- addMatchingMethod:rewriteTo:

same as the previous, but the rewrite expression are parsed as method definitions

- replace:by:
- addMatchingExpression:rewriteWith:

add a ”from->to” pair, first element of which is a rewrite expression in a form of a string that is used to match nodes. The second parameter is a block that has to return a node which should replace the matched one. The block may accept 2 atguments: the matched node, and a dictionary of wildcard variables mapping.

1.9 Example

There are two main concepts in Renraku rule and critiques. These two concepts are represented by abstract classes: `ReAbstractRule` and `ReAbstractCritiques`.

Each `ReAbstractRule` represents a static validation rule that can be applied on a class, method, package or a node (a node refers to an AST node).

Une `ReAbstractRule` specifies the messages of the rule, its severity (’warning’ is the default) and the category in which the rule belongs (for example ’Optimization’).

Once executed, `ReAbstractRule` returns a list of `ReAbstractCritiques`. Each `ReAbstractCritique` is linked precisely to the place where the rule failed.

It is also possible to specify to automatically address the problem.

Elle permet aussi d’associer, si on le souhaite, un ”quickfix” pour automatiquement modifier le code.

Dans notre cas, le but de la règle est de mettre en évidence les variables temporaires non utilisées dans le code (uniquement non utilisées, pas non initialisée ou autre). Si des variables temporaires sont identifiées, alors, le quickfix proposé est de supprimer la variable en question.

Pour faire ceci, on commence par créer une nouvelle règle`NRTemporaryNeverUsedRule` dans un package `NewRule`, en héritant de `ReAbstractRule`:

```
ReAbstractRule subclass: #NRTemporaryNeverUsedRule
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'NewRules'
```

Comme notre règle travaille sur des éléments très fin d'un méthode analysée, on ne peut rester au niveau de la "méthode" pour l'analyse, il faut décendre au niveau de chaque `Node` de l'ast de la méthode. Il faut donc indiquer que cette règle s'applique uniquement sur des `Nodes`. Pour faire ceci, il faut surcharger la méthode de classe `ReAbstractRule class>>checksNode`:

```
NRTemporaryNeverUsedRule class >> checksNode
  ^ true
```

On précise ensuite le groupe auquel la règle va être associé en surchargeant `ReAbstractRule>>#group`:

```
NRTemporaryNeverUsedRule >> group
  ^ 'Optimization'
```

Puis, on indique le message qui sera affiché si la règle trouve un élément fautif en surchargeant `ReAbstractRule >> name`:

```
NRTemporaryNeverUsedRule>>name
  ^ 'Temporary variable is never used'
```

Une règle doit ensuite fournir une implémentation de "#basicCheck:". Cette méthode retourne vrai si l'élément passé en argument viole la règle. Elle est implémentée ici simplement, si le noeud est une variable temporaire, alors, on vérifie qu'elle a bien été utilisée dans la séquence de node qui la contient.

```
NRTemporaryNeverUsedRule>>basicCheck: aNode
  aNode isTemp ifFalse: [ ^ false ].
  ^ (self checkTemp: aNode isUsedIn: (aNode whoDefines: aNode name))
    not
```

L'implémentation du sélecteur `checkTemp:isUsedIn:` est plutôt directe. On regarde dans toute les expressions de la séquence d'instruction s'il existe une variable qui est utilisée est qui porte le même nom que la variable temporaire dont on veut savoir si elle est effectivement utilisée.

```
NRTemporaryNeverUsedRule>>checkTemp: aTemp isUsedIn: statements
  statements
    do: [ :statement |
      statement
        nodesDo: [ :node |
          (node isVariable and: [ node name = aTemp name and: [ node
    isUsed ] ])
            ifTrue: [ ^ true ] ] ].
  ^ false
```

Maintenant que la règle est écrite, il est possible d'écrire une nouvelle critique qui va permettre de mettre en surbrillance dans le code la variable temporaire fautive. On commence donc par créer une sous-classe de`ReAbstractCritique`. Comme cette critique va devoir garder un lien vers le node temporaire, on lui ajoute une variable d'instance "temp" (le nom est pas génial) :

```
"Critique creation"
ReAbstractCritique subclass: #NRTemporaryNeverUsedCritique
instanceVariableNames: 'temp'
classVariableNames: ''
package: 'NewRules'
```

On ajoute aussi des accesseurs à la variable d'instance `temp`:

```
NRTemporaryNeverUsedCritique>>temporary
  ^ temp
```

```
NRTemporaryNeverUsedCritique>>temporary: aTemp
  temp := aTemp
```

On propose maintenant une méthode de classe pour la création de nouvelles instances qui va prendre en paramètre un "ReSourceAnchor". Un "ReSourceAnchor" est un wrapper vers l'obets d'intérêt :

```
NRTemporaryNeverUsedCritique class>>withAnchor: anAnchor by: aRule
    temporary: aTemp
  ^ (self withAnchor: anAnchor by: aRule)
    temporary: aTemp;
    yourself
```

Avec cette méthode, la "Critique" créée aura à la fois un lien vers l'objet d'intérêt via le "sourceAnchor", mais aussi un lien au node représentant la variable temporaire fautive.

Néanmoins, dans l'état actuel des choses, la "Critique" créée n'est pas encore liée à notre nouvelle règle. Par défaut, ce sont des instances de "ReTrivialCritique" qui sont proposée. Pour changer ceci, il faut surcharger "ReAbstractRule»#critiqueFor:". À la place de la création de la "ReTrivialCritique", on va créer une instance de la "Critique" que l'on vient juste d'écire :

```
NRTemporaryNeverUsedRule>>critiqueFor: aTemp
  ^ NRTemporaryNeverUsedCritique
    withAnchor: (self anchorFor: aTemp)
    by: self
    temporary: aTemp
```

Il faut noter ici l'utilisation du message "#anchorFor:" qui permet d'obtenir un "ReSourceAnchor" à partir d'un élément.

Dans l'état actuel, c'est déjà pas mal, mais pas complètement satisfaisant. On remarque que le "ReSourceAnchor" créé est un simple "ReSourceAnchor" et qu'il ne donne pas plus d'information. Dans notre cas, nous aimerions pouvoir avoir accès à l'interval qui représente la variable dans le source code (en gros, le début et la fin de la temporaire dans la variable).

Il existe plusieur sous-classes de ”ReSourceAnchor”, dont ”ReIntervalSourceAnchor” qui stocke par la même occasion l'interval de l'objet ”observé” dans le code source. On surcharge donc ”ReAbstractRule#anchorFor:” dans notre règle :

```
NRTemporaryNeverUsedRule>>anchorFor: aNode
  ^ ReIntervalSourceAnchor entity: aNode interval: aNode
    sourceInterval
```

Maintenant, la règle et la critique créé sont liée. Néanmoins, on ne peux toujours pas effectuer de modification automatiquement du code. Il faut fournir quelques méthode en plus à notre nouvelle critique pour ça. La méthode ”#providesChange” va permettre d'indiquer que notre critique fournit un moyen de ”fix” le code, et la méthode ”#change” va retourner une instance de ”RBRefactoryChange”. Cette instance représente la modification effecivement effectuée sur le code source. Il en existe beaucoup et certaines sont probablement très pratiques, mais celle que j'ai utilisée est la ”RBAddMethodChange”.

```
NRTemporaryNeverUsedCritique>>providesChange
  ^ true
```

```
NRTemporaryNeverUsedCritique>>change
  | parseTree |
  parseTree := sourceAnchor entity parseTree.
  parseTree
    nodesDo: [ :node |
      node isSequence
        ifTrue: [ node removeTemporaryNamed: temp name ] ].
  "alternative possible :
  rewriter := RBParseTreeRewriter removeTemporaryNamed: temp name.
  parseTree := rewriter executeTree: sourceAnchor entity parseTree;
    tree."
  ^ RBAddMethodChange compile: parseTree newSource in: sourceAnchor
    entity methodClass
```

La méthode de modification du code source est elle aussi plutôt directe, on traverse l'ast et on enlève les ”temporary” fautives de chaques séquences. Il faut noter ici que le ”sourceAnchor entity” retourne la méthode dans laquelle le soucis a été détecté et non pas le node. Même si l'on a explicitement créé la ”critique” avec le ”TemporaryNode” comme ”sourceAnchor”, il se trouve que ”ReCriticEngine»#nodeCritiquesOf:” réassigne la ”CompiledMethode” comme ”sourceAnchor” après (on peut le voir l.27 à 31) :

```
rule
  check: node
  forCritiquesDo: [ :critique |
    critique sourceAnchor initializeEnitity: aMethod.
    # stuffs that overrides the node, also typo: '.*Enitity' ->
     '.*Entity'
    critiques add: critique ]
```

1.10 Extending renraku

SmartTest selects automatically some tests to be run automatically each time a new method is compiled. It is clearly a new kind of rule.

Pour SmartTest, comme nous sommes sur un tout nouveau type de règle, j'ai étendu la classe 'ReAbstractRule' pour creer les règles. Il y a deux règles, l'analyse

- Pour les méthodes
- Pour les classes

Pour les rules, il y a deux méthodes importante à chaque fois

- #check:forCritiquesDo:
- #basicCheck:

Part III

basicCheck: return true si le critique block doit être executé (#basicCheck: est appellé dans #check:forCritiquesDo:

Part IV

check:forCritiquesDo: execute un critique block avec en paramètre une critique.

Dans un premier temps on vérifie que s'il y a des tests relatifs à la classe ou à la méthode. Si oui on execute le critique block avec en parametre une critique expliquant que l'on a besoin de tests Si non avec une critique donnant les tests trouvés.

Il y a quatre critiques que j'ai créées

- SmTMethodNeedTestsCritique
- SmTClassNeedTestsCritique qui étendent de SmTNeedTestsCritique
- SmTMethodRelativeTestsCritique
- SmTClassRelativeTestsCritique qui étendent de SmTRelativeTestsCritique

Pour ces dernières: Il n'y a que le #title et la #description qui changent SmTRelativeTestsCritique définit aussi une #actions qui ouvre la fenetre avec les tests

Pour les `SmTNeedTestsCritique` Pour `SmTClassNeedTestsCritique`, il n'y a que le `title` et la `description`

Pour contre pour SmTMethodNeedTestsCritique, il y a la petit clef à molette. Donc j'ai étendu `providesChange` pour que cela retourne true

Ensuite j'ai juste eu à écrire la méthode`execute` qui crée un test (et la fenetre du fix qui apparait etc. est déjà dans Reneraku.)

CHAPTER 2

Renraku Model

2.1 Entities

Renraku is a framework for defining and processing quality rules. The framework operates with three main concepts: entities, rules and critiques.

- Entities. Entities are not a part of Renraku, but Renraku is validating entities. Theoretically entity can be any object, but in practice we mostly focus on code entities such as methods, classes, packages, AST nodes.
- Rules. Rules are the objects that describe constraints about entities. A rule can check an entity and produce critiques that describe the violations of the entity according to the rule. Rules are constraint descriptions about entities. Think of a rule as a function that consumes an entity and produces a collection of critiques about it (which can be empty if there are no violations)
- Critiques. Critique is an object that binds an entity with a rule that is violated by that entity. The critique describes a specific violation, and may provide a solutions to fix it.

2.2 Critiques

Critique is an object that binds an entity with a rule that is violated by that entity. The critique describes a specific violation, and may provide a solution to fix it.

`ReAbstractCritique` is the root of the critiques hierarchy.

A critique has a reference to the rule that reported the violation. The rule's `name` is used as the critique's `title` and the rule's `rationale` is used as the `description` of the critique.

A critique has a reference to the criticized entity. This link is established through `ReSourceAnchor`. A source anchor has a reference to the actual class, method, or other entity that is criticized. An anchor also has a `providesInterval` method that returns a boolean indicating if the anchor provides a selection interval to the actual source of the critique. The interval can be accessed through the `interval` method.

There are two subclasses of `ReSourceAnchor`.`ReIntervalSourceAnchor` stores the actual interval object which is set during initialization.`ReSearchStringSourceAnc` stores a searchString which will be searched for in the entities source code on demand to find an interval of substring.

A critique has the `providesChange` method which returns a boolean value specifying whether the critique can provide a change which will resolve the issue. The `change` method can be used to obtain an object of `RBRefactoryChange` kind.

One can override `actions` method to return a list of `RePropertyAction` objects. Tools can use these objects to provide a user with custom actions defined by critiques themselves.

I am an action that appears in the Nautiluas qa plugin next the the item's title.

icon - a Form that will appear on the button (green square by default)

description - the description that will be present on popup on hower

action - a two (ortional) parameter block that is evaluated with the critic and the current code entity (class, method...) when the button is pressed No newline at end of file

2.3 Migrating rules to renraku

Lately Pharo tools moved to Renraku framework which requires a slightly different implementation from rules as defined in the refactoring browser.

While you can achieve much more features by reading the whole documentation and using the complete set of Renraku possibilities, this book contains a few simple steps to help you converting existing rules to work with Renraku model.

Generic rule

To convert a generic rule (one that simply checks method, class or package) to work with Renraku you have to follow 3 simple steps:

- Inheritance: change superclass to ReAbstractRule.
- Interest: Specify which entities your rules is interested in by overriding a method on the class side to return true. If the rule was checking methods, then override =checksMethod`, for classes override` checksClass`, and for packages` checksPackage‘.
- Checking: Implement `basicCheck:` in a way that it will return true if the argument violated the rule and false otherwise. The quality tools will pass you only the arguments of the type you've expressed interest in.

To convert parse tree rules Parse tree rules are subclasses of RBParseTreeLintRule, change their superclass to ReNodeMatchRule.

Then change the initialization method. Instead of sending match-specifying methods to `matcher`, send them to `self`. The rest of API is similar:

- `matches:do:` -> `matches:`
- `matchesMethod:do:` -> `addMatchingMethod:`
- `matchesAnyOf:do:` -> `matchesAny:`

So the old initialization:

```
self matcher
  matches: '`var := `var'
  do: [ :node :answer | node ]
```

will become:

```
self matches: '`var := `var'
```

You have noticed that new API is missing the ”do:” part. First of all, almost no rules use this functionality and you can check node in the matching expression with `` `{:node | "check node" \} `` syntax.

2.4 Rewrite rules

To convert parse tree rules, subclasses of `RBTransformationRule`, change their superclass to `ReNodeRewriteRule`.

Then change the initialization method. Instead of sending transformation-specifying methods to `rewriteRule`, send them to `self`. The rest of API is similar:

- `replace:with:` -> `replace:with:` (noChange)
- `replaceMethod:with:` -> `addMatchingMethod:rewriteTo:`
- ? -> `replace:by:` (second argument is a block which accepts matched node and returns a node that should be used for replacement).

So the old initialization:

```
self rewriteRule
  replace: '`var := `var' with:
```

will become:

```
self replace: '`var := `var' with:
```

2.5 Better post-checks

The new rules also give you a move powerful way of post-checking matched nodes. You can override `afterCheck:mappings:` method and return true if node really violates the rule or false otherwise. The first argument passed to the method is the matched node object, while the second argument is a dictionary of bindings for the wildcards in the rule. For example if the pattern `'`var := `var'` will match expression `'a := a'` the matches dictionary will contain one entry where key is `RBPatternVariableNode\(`var\)` and value is ‘RBVariableNode(a)==.

Bibliography