Transformation-based Refactorings: a First Analysis

N. Anquetil, M. Campero, **S. Ducasse,** J.P. Sandoval and P. Tesone

International Workshop on Smalltalk Technologies

Outline

- Vocabulary
- About the need of transformations
- Research questions
- Why RB engine?
- Analyses and results

Vocabulary

- **Refactoring**: behavior preserving modification of the source code.
- **Transformation**: behavior agnostic modification of the source code.
- **Pre-condition:** a condition to hold before applying a refactoring
- **Atomic refactoring:** A refactoring that is not implemented using some other refactorings
- **Composite refactoring**: Not Atomic.
- **Elementary operation:** A local modification of a program

Outline

- Vocabulary
- **About the need of transformations**
- Research questions
- Why RB engine?
- Analyses and results

Replace Call x by y

foo

self x

x

doing something

y

doing something else

bar

t x

foo

self y

x

doing something

y

doing something else

bar

t y

Replace Call Analysis

This is not a refactoring because there is no warrantee that after the application behavior stays the same

Replace Call Analysis

- Cannot easily be expressed with Rename Method Refactoring
- Now the implementation is close to Rename Method
  - Look for all the senders
  - Replace all the senders with the target method

Need for transformation too

- Developers are often left alone and do it manually
- There is a need for **transformations as well as refactorings**

Goal of the study

- Could both refactorings and transformations **share their code modification** logic?
- Could we decouple code transformations from refactorings to be able to reuse them when behavior preservation is not a concern?
- Could we **compose refactorings** from such **code transformations**?

Correlated questions

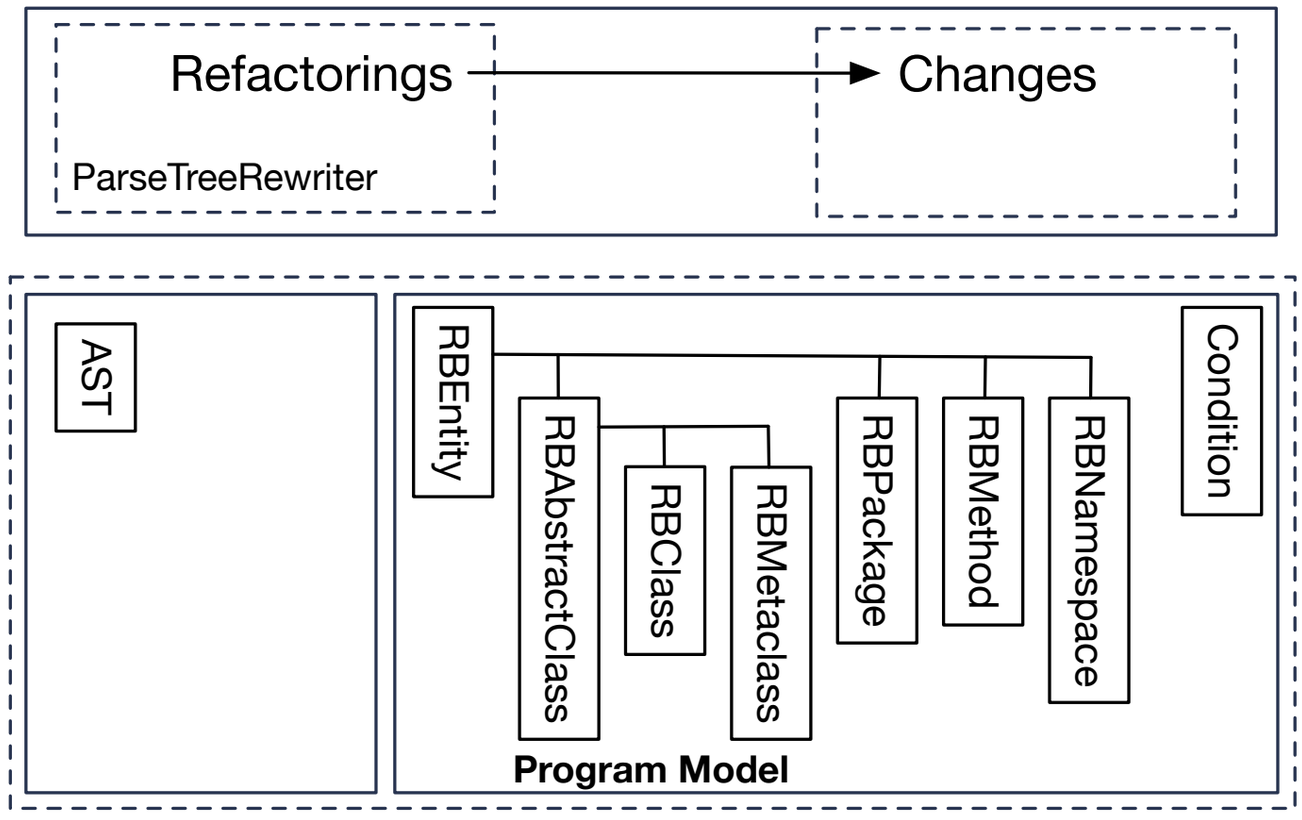
- Should we turn elementary operations into transformations?
- At which cost complex refactorings be expressed out of simpler ones?

Contributions

- An analysis of the original implementation of Refactorings.
- The identification of different **kinds of pre-conditions**
- Understanding program modification API
- Identify atomic refactorings
- The identification of **missed reuse opportunities** in current refactorings implementation
- Path for the real work

Why RB engine?

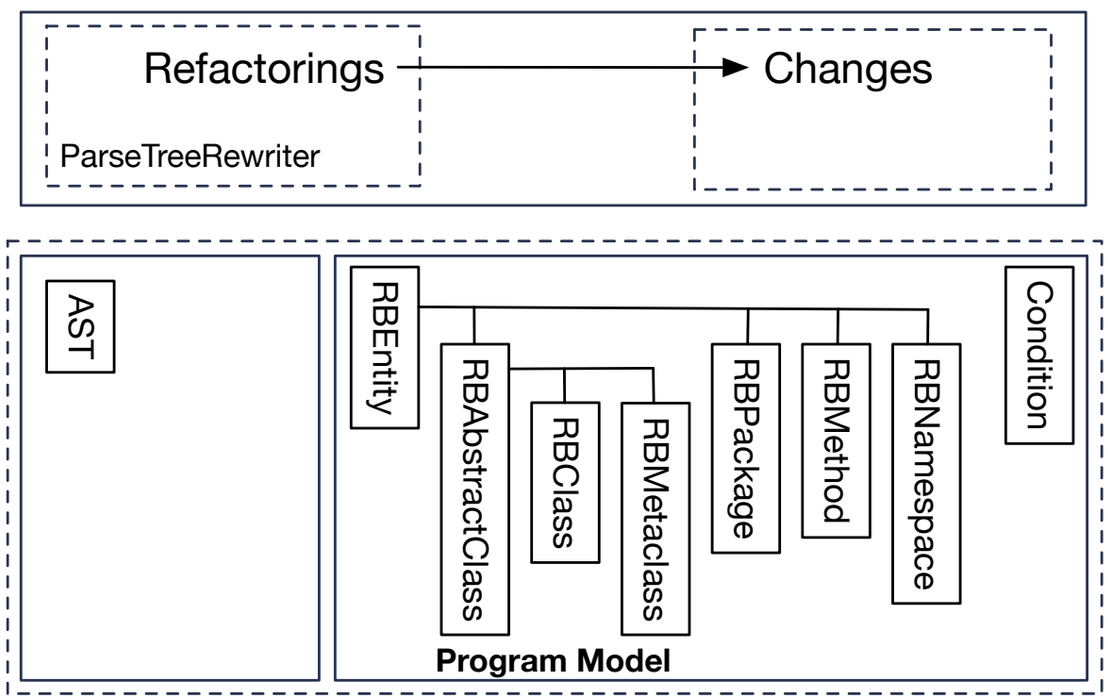
- Seminal work
- Documented in PhD of D. Roberts
- Available and fully working
- Nice architecture (compared to mainstream engines)
  - preview
  - support off line code



Applying a refactoring

The architectural elements

- A refactoring reason on a **program model**
- **Check preconditions** on such a program model
- Produces **changes** that can be previewed
- Then actual modifications are done



Outline

- Vocabulary
- About the need of transformations
- Research questions
- Why RB engine?
- **Analyses and results**

Precondition Analysis

4 Families

- None
- Applicability check
- Break check
- Complex ones

Precondition: None

Looks like there were added by developers not following the design

Should really be revisited

Precondition: None

- Abstract Class Variable
- Abstract Variables
- Category Regex
- Class Regex
- Extract Method To Component
- Extract Set Up Method And Occurrences
- Expand Referenced Pools
- Protocol Regex
- Source Regex

Precondition: Applicability

Verifies that the program elements are available

Precondition: Applicability

```
RBCreateAccessorsForVariableRefactoring >> preconditions

^ classVariable
    ifTrue: [ RBCondition definesClassVariable: variableName
asSymbol in: class ]

    ifFalse: [ RBCondition definesInstanceVariable: variableName in:
class ]
```

Precondition: Applicability

- Abstract Instance Variable
- Accessor Class
- Add Class
- Add Class Variable
- Add Instance Variable
- Add Method
- Add Parameter
- Children To Siblings
- Copy Class
- Copy Package
- Create Accessors For Variable
- Create Cascade
- Create Lazy Initialization
- Move Inst Variable To Class
- Move Method To Class
- Move Method To Class Side
- Move Variable Definition
- Protect Instance Variable
- Pull Up Class Variable
- Pull Up Instance Variable
- Push Down Method
- Realize Class
- Remove All Senders
- Remove Hierarchy Method
- Deprecate Class
- Deprecate Method
- Extract Method And Occurrences
- Extract Set Up Method
- Extract To Temporary
- Find And Replace
- Find And Replace Set Up
- Generate Equal Hash
- Generate Print String
- Inline All Senders
- Inline Method
- Inline Parameter
- Merge Instance Variable Into Another
- Rename Argument Or Temporary
- Rename Class
- Rename Class Variable
- Rename Instance Variable
- Rename Method
- Rename Package
- Replace Method
- Split Cascade
- Split Class
- Swap Method

Precondition: Break check

- None
- Applicability check
- Break check
- Complex ones

Precondition: Break check

- None
- Applicability check
- Break check
- Complex ones

Precondition: Break check

- Verifies if the operation is done whether the system would be broken

Precondition: Break check

```
RBRemoveClass >> preconditions
    ^ classNames
        inject: self emptyCondition
        into: [ :sum :each | | aClassOrTrait |
        aClassOrTrait := self model classNamed: each asSymbol.
        aClassOrTrait ifNil: [
        self refactoringFailure: 'No such class or trait' ].
        sum & ((self preconditionIsNotMetaclass: aClassOrTrait)
            & (self preconditionHasNoReferences: each)
            & (self preconditionEmptyOrHasNoSubclasses: aClassOrTrait)
            & (self preconditionHasNoUsers: aClassOrTrait)) ]
```

Precondition: Break check

- Remove Parameter
- Remove Sender
- Inline Method From Component
- Remove Class Variable
- Push Down Instance Variable
- Remove Instance Variable
- Temporary To Instance Variable
- Remove Class
- Remove Class Keeping Subclasses
- Push Down Class Variable

Precondition: Complex

- Facing real code some things are not like in the books!
- To be cleaned up!

Precondition: Complex

RBPullUpMethod >> **preconditions**
self requestSuperClass.
^(selectors inject: (RBCondition hasSuperclass: class) into: [:cond :each | cond & (RBCondition definesSelector: each in: class)]) & (RBCondition withBlock: [self **checkInstVars**.
self checkClassVars.
self checkSuperclass.
self checkSuperMessages. true])

RBPullUpMethod >> **checkInstVarsFor**: aSelector

class instanceVariableNames do: [:each |
((class whichSelectorsReferToInstanceVariable: each) includes: aSelector) ifTrue: [(self confirm: ('<1p> refers to #<2s> which is defined in <3p>. Do you want push up variable #<2s> also ?' expandMacrosWith: aSelector with: each with: class))

ifTrue: [self **pushUpVariable**: each]
ifFalse: [self refactoringError: 'You are about to push your method without the instance variable it uses. It will bring the system is an inconsistent state. But this may be what you want. So do you want to push up anyway?']]]

RBPullUpMethod >> **pushUpVariable**: aVariable
| refactoring |
refactoring := RBPullUpInstanceVariableRefactoring model: self model variable: aVariable class: targetSuperclass.

self performCompositeRefactoring: refactoring.

Contrary to common beliefs

- **Transformations can have preconditions too!**

Study the API of Program Model

- What is basically the API that is used to perform changes?
- Could such API turned into first class transformation?

Study the API of Program Model

RBAbstractClass

addInstanceVariable:
addMethod:
addSubclass:
compile:
compile:classified:
compile:withAttributesFrom:
compileTree:
convertMethod:using:
removeInstanceVariable:
removeInstanceVariable:ifAbsent:
removeMethod:
removeSubclass:
renameInstanceVariable:to:around:

RBClass

addClassVariable:
addPoolDictionary:
addProtocolNamed:
comment:
removeClassVariable:
removeClassVariable:ifAbsent:
removePoolDictionary:
removeProtocolNamed:
renameClassVariable:to:around:

RBMethod

compileTree:

RBNamespace

addClassVariable:to:
addInstanceVariable:to:
addPackageNamed:
addPool:to:
addProtocolNamed:in:
category:for:
changeClass:
comment:in:
compile:in:classified:
convertClasses:select:using:
createNewClassFor:
createNewPackageFor:
defineClass:
description:
performChange:around:
removeClass:
removeClassKeepingSubclassesNamed:
removeClassNamed:
removeClassVariable:from:
removeInstanceVariable:from:
removeMethod:from:
removePackageNamed:
removeProtocolNamed:in:
renameClass:to:around:
renameClassVariable:to:in:around:
renameInstanceVariable:to:in:around:
renamePackage:to:
reparentClasses:to:
replaceClassNameIn:to:

Atomic refactorings and operations

- Analysed refactorings to identify atomic ones
- And also which ones are modifying the program model

Atomic refactorings and operations

- identify atomic

Class	Used By
AddClass*	ChildrenToSiblings, CopyClass, SplitClass
AddClassVariable*	CopyClass
AddInstanceVariable*	CopyClass, SplitClass
AddMethod*	CopyClass
AddParameter	
CategoryRegex	
CreateCascade	
DeprecateMethod	
ExpandReferencedPools	AbstractVariables, PullUpMethod, PushDownMethod
ExtractMethod	ExtractMethodAndOccurrences, ExtractMethodToComponent, FindAndReplace
InlineMethod	InlineAllSenders
InlineParameter	
InlineTemporary	
ProtocolRegex	
RealizeClass	
RemoveClass*	
RemoveClassVariable*	
RemoveInstanceVariable*	SplitClass
RemoveMethod	
RemoveParameter	
RemoveSender	RemoveAllAccessors
RenameArgumentOrTemporary	
RenameClass	RenamePackage
RenameClassVariable	
RenameMethod	
ReplaceMethod	
SourceRegex	
SplitCascade	

Atomic refactorings and operations

- Studied the refactorings that directly use the program API elements
- They could be turned into composite refactorings
- Could the program API elements be turned into first class transformations?

Potential reuse

Potential reuse in implicit composite refactorings

Refactoring	could use…	…instead of
ChangeMethodName	RemoveMethod	removeMethod:
CreateAccessorsForVariable	AddMethod	compile:
CreateAccessorsWithLazy-InitializationForVariable	AddMethod	compile:
DeprecateClass	AddMethod	compile:
GenerateEqualHash	AddMethod	compile:
GeneratePrintString	AddMethod	compile:
MoveInstVarToClass	AddMethod RemoveInstanceVariable	addMethod: and compile: removeInstanceVariable
MoveMethodToClass	AddMethod RemoveMethod	addMethod: and compile: removeMethod:
PullUpClassVariable	RemoveClassVariable	
PullUpInstanceVariable	RemoveInstanceVariable	
PushDownClassVariable	RemoveClassVariable	
PushDownInstanceVariable	RemoveInstanceVariable	
RemoveMethod	RemoveMethod	
SwapMethod	AddMethod and RemoveMethod	compile: and removeMethod:
TemporaryToInstanceVariable	RemoveInstanceVariable	

Conclusion

- **Transformations can have preconditions too!**
- **Interaction should be removed from refactorings**
- **Identify Atomic refactorings**
- **Identify potential reuse in implicit refactoring**
- **Next step is to try to reify transformations and use them**