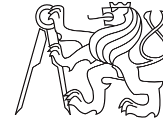
Insert here your thesis' task.

Czech Technical University in Prague
Faculty of Information Technology
Department of Software Engineering



Bachelor's thesis

# API for Code Generation and Refactoring

***Jakub Nesveda***

Supervisor: Ing. Jan Vraný, Ph.D.

5th February 2015

# Acknowledgements

Many thanks to my supervisor Ing. Jan Vraný, Ph.D. for his leadership and support.

# Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on 5th February 2015 . . . . . . . . . . . . . . . . . . . . .

Czech Technical University in Prague
Faculty of Information Technology
© 2015 Jakub Nesveda. All rights reserved.
*This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).*

### Citation of this thesis

Nesveda, Jakub. *API for Code Generation and Refactoring.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2015.

# Abstrakt

Při vývoji softwaru se může programátor setkat s dílčími úlohami, které se velmi často opakují, například přejmenování proměnné, implementace metod pro existující rozhraní nebo tvorba testovacích případů. To může být únavné a časově náročné, a proto jsme navrhli a implementovali API pro generování kódu a refaktoring v prostředí Smalltalk/X. Přestože STX již obsahuje takové nástroje, jejich API není příliš uživatelsky přívětivé. V této práci prezentujeme, jak je API navrženo a jak programátor může použít toto API k vytvoření generátorů kódu nebo refaktorování.

**Klíčová slova** API pro generování kódu, API pro refaktoring, Smalltalk/X, Podpora programování

# Abstract

During coding a programmer can face many repetitive tasks such as renaming variables, creating methods to implement an interface or creating test cases. This can be tedious and time consuming so we designed and implemented API for code generation and refactoring in Smalltalk/X. Although STX contains tools for such a task, they do not provide much user-friendly API. In this work

we present how is API designed and how a programmer can use this API to create code generators or refactorings.

**Keywords** Code generation API, Refactoring API, Smalltalk/X, Code Assistants

# Contents

**Introduction** **1**
Motivation . . . . . . . . . . 1
API . . . . . . . . . . 1
Code generation and refactoring . . . . . . . . . . 2

**1 Original state** **3**
1.1 Package structure . . . . . . . . . . 3
1.2 Code generators . . . . . . . . . . 5
1.3 Refactoring . . . . . . . . . . 7

**2 Analysis** **9**
2.1 Integration with IDE . . . . . . . . . . 9
2.2 Class CodeGeneratorTool . . . . . . . . . . 10
2.3 Package Refactory Browser . . . . . . . . . . 13
2.4 Method rewriter . . . . . . . . . . 15

**3 API in a nutshell** **17**
3.1 Usage in the IDE . . . . . . . . . . 17
3.2 Usage inside a code . . . . . . . . . . 18
3.3 Custom code generator . . . . . . . . . . 19
3.4 Custom refactoring . . . . . . . . . . 23
3.5 Overall schematic overview . . . . . . . . . . 24

**4 API design** **27**
4.1 Template class . . . . . . . . . . 27
4.2 Class model . . . . . . . . . . 29
4.3 Context . . . . . . . . . . 30
4.4 Perspective . . . . . . . . . . 32
4.5 Refactoring helper . . . . . . . . . . 32
4.6 Formatter . . . . . . . . . . 33

**5 Implementation** **35**
5.1 Generator or refactoring composition . . . . . . . . . . . . . . . 35
5.2 Integration with IDE . . . . . . . . . . . . . . . . . . . . . . . . 36
5.3 Class model extensions . . . . . . . . . . . . . . . . . . . . . . . 37
5.4 Refactoring helper . . . . . . . . . . . . . . . . . . . . . . . . . 39

**6 Testing** **41**
6.1 Mocks and stubs . . . . . . . . . . . . . . . . . . . . . . . . . . 42
6.2 Generators or refactorings tests . . . . . . . . . . . . . . . . . . 43
6.3 Testing summary . . . . . . . . . . . . . . . . . . . . . . . . . . 44

**Conclusion** **45**
Future improvements . . . . . . . . . . . . . . . . . . . . . . . . . . 45

**Bibliography** **49**

**A Acronyms** **51**

**B Contents of enclosed CD** **53**

# List of Figures

1.1 Old code generator class design . . . . . . . . . . . . . . . . . . . . . 5

3.1 Usage of TestCase setUp method code generator . . . . . . . . . . . 18
3.2 Context menu building flowchart with marked extension points . . 25
3.3 Code generator or refactoring execution flowchart with marked extension points . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26

4.1 Code generator or refactoring template class . . . . . . . . . . . . . 28
4.2 Class model . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
4.3 Context and perspective class diagram . . . . . . . . . . . . . . . . 31
4.4 CustomRefactoryBuilder class diagram . . . . . . . . . . . . . . . . 33

5.1 Shared objects diagram at the point of generator or refactoring execution . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36

6.1 Code coverage . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42

# Introduction

Automating repetitive programming tasks is key goal to improve productivity. Within this thesis we are focusing on code generation and refactoring in Smalltalk/X environment. Primary goal is to design and implement API which will programmers use to create their own code generators or refactorings with possibility to integrate them in STX IDE. Secondary goal is to create bunch of code generators and refactorings in order that we verify the usage of the API. This chapter briefly describes why the new API was introduced and explains essential terms related to this work.

## Motivation

STX allows creating code which performs a refactoring or generates new source code. If we would like to create a script which would add a method to existing class then it is easy as calling one method. This would be fine unless we will need something more like reuse script via click-able item in STX IDE. Old fashion way to do it in STX requires direct modification of IDE menu which hard to maintain and programmer needs to know where the menu definition is located. The new API solves seamless integration with IDE and other aspects related with changing source. Creating code to generate e.g. method should be easy as implement well designed API.

## API

Application programming interface [1] is a specification how to use or implement a software component. We defined interface which should be implemented by a code generator or refactoring. The interface itself consists of template class (4.1) which should be inherited and methods to be overridden. Therefore each code generator or refactoring should be represented by Smalltalk class.

Another part of API focuses on code generation or doing refactoring itself and is designed more like a library. STX already offers comprehensive code-base for this task. We extended existing API to archive better usability and possibility to provide custom implementation.

We figured out that testing single code generators or refactorings is a bit complicated when doing from scratch so we created testing API (6.2). With a few lines of code and without side effects is possible to test modifications in the source code.

## Code generation and refactoring

Both code generation and refactoring are meant to change the source code. However they have lot in common within our API we distinguish between them semantically. Code generator should create new code to deliver new functionality, but refactoring [2] should modify the code without changing its outer behaviour. In practice this difference is not so strict. We can do, for example, a refactoring where the original API is kept and we add a little more functionality. As a result, the behaviour is slightly modified, but API is still backward compatible. For example, if we replace a constant string with the translation call then we consider it more like a refactoring in scope of our API.

Chapter 1

# Original state

STX has variety of options how to automatically create or modify the source code. Some of them are even usable via click-able menu item in the IDE. We will describe the API beyond this thesis, because the new API is essentially based on top of this.

## 1.1 Package structure

Two main branches coexist together in STX to support codebase modifications. Both offer similar functionalities which leads to duplication. They will be probably merged together in the future to make the API more consistent.

The first one lives in packages `stx/libbasic`, `stx/libbasic3`, `stx/libcomp` where `stx/libbasic` contains fundamental classes like `ClassDescription` with methods to change codebase. Every Smalltalk class inherits from this class thus we can directly call methods to add a method and class 1.1.

```
"To add a method"
MyClass compile:code classified:category.

"To add a subclass"
MyClass subclass:nameSymbol
        instanceVariableNames:instVarNameString
        classVariableNames:classVarString
        poolDictionaries:pool
        category:cat.
```

Code 1.1: Creation of a method and class

In fact method "compile:classified:" is delegated to the compiler class which is for Smalltalk language located in `stx/libcomp` along with parser and AST classes. In order to be complete, STX contains also compilers for other lan-

guages like Java or JavaScript. Package `stx/libbasic3` has among other change classes which serve the same purpose as described in subsection 1.1.2.

The second one lives in the package `stx/goodies/refactoryBrowser` and provides more extensive API than the branch noticed above. Little inconvenience could be focus on just Smalltalk language. Its API divides into sub-packages which we briefly describe.

### 1.1.1 Browser

Refactory Browser library was imported from VisualWorks [3] and this sub-package mostly contains GUI for performing refactorings. The GUI became obsolete as the operations were integrated with STX IDE except several GUI classes like `MethodNameDialog`.

Important dependency to packages below is class `BrowserEnvironment` and its subclasses. They represent the environment (classes and methods) on which we operate.

### 1.1.2 Changes

Classes in this sub-package represents codebase modifications. In other words, when we add/modify a method or class then it is stored in a class instance. Key API is to perform the change to the system or undo it.

### 1.1.3 Helpers

This sub-package is the meta-model for classes (`RBClass`, `RBMetaclass`) and methods (`RBMethod`). Class `RBNamespace` serves as collector for their instances. Main purpose is to provide `Class` and `Method` API which lives in the Smalltalk environment (i.e. real system classes) with respect to non performed changes. For example, when we add a method to the meta-model and then ask the class if the method exists then the meta-model class should return true, but the Smalltalk environment is still without the method until we actually perform the change.

### 1.1.4 Lint

Sub-package contains rules which detect bad programming style or probable coding mistakes. However this sub-package is not much relevant code to this work, we will notice it, because it is part of Refactory Browser.

### 1.1.5 Parser

Offers similar Smalltalk language parsing and AST nodes like the parser in package `stx/libcomp`, but is enhanced with pattern searching and rewriting.

The search pattern looks similar to regular expression pattern, but specializes in searching source code fragments [4].

### 1.1.6 Refactoring

Majority of classes represents individual refactoring operations like rename variable or extract method. These refactorings depend on precondition classes which validates if the refactoring can be performed.

Search pattern rule classes are also part of this sub-package. They serve source code search and replace purpose noticed above 1.1.2 .

## 1.2 Code generators

Code generators are written as single methods in classes `CodeGeneratorTool`, `SmalltalkCodeGeneratorTool` and `JavaScriptCodeGeneratorTool`. In Figure 1.1 is simplified class design and important dependencies within STX (complete specification includes much more methods).

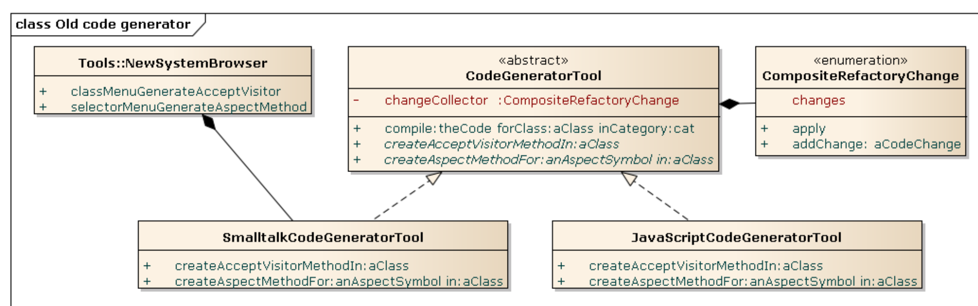



Figure 1.1: Old code generator class design

As you can see, class `CodeGeneratorTool` is meant to be abstract class which implementers should be specific for different programming languages (`SmalltalkCodeGeneratorTool` and `JavaScriptCodeGeneratorTool`). Each metaclass [5, page 40] responds to method `codeGeneratorClass` which should return proper generator class for programming language of the class.

Detailed view of code 1.2 shows basic structure of a generator within method. This concrete example creates method `acceptVisitor:` for given class. Source code is built as string with expanded place-holders and then passed to the method `compile:forClass:inCategory:`. Here is transformed into the change object 1.1.2 and then applied to the system. Additional option `canUseRefactoringSupport` skips Refactory Browser and uses firstly noted branch (1.1).

```
createAcceptVisitorMethod:selector in:aClass
    withParameter: withParameter
```

```
    self assert:( aClass isMeta not ).

    (aClass includesSelector:#'acceptVisitor:') ifFalse
        :[
         self
             compile:
(('acceptVisitor:visitor %2
    "Double dispatch back to the visitor, passing my
        type encoded in
     the selector (visitor pattern)"

    "stub code automatically generated - please change
        if required"

    ^ visitor %1self %2
') bindWith:selector with:(withParameter ifTrue:[' with:
    parameter'] ifFalse:['']))
             forClass:aClass
             inCategory:#visiting.
    ]
```

Code 1.2: generates acceptVisitor: method

### 1.2.1 IDE Integration

Integration with IDE is done directly in the class `Tools::NewSystemBrowser`. Pop-up menu item (code example 1.3) holds code generator name, methods to tell click-ability and execution method – `classMenuGenerateAcceptVisitor` which calls the method from class `SmalltalkCodeGeneratorTool` for each selected class.

```
MenuItem
          enabled: hasClassSelectedHolder
          label: 'Visitor Method'
          itemValue: classMenuGenerateAcceptVisitor
          translateLabel: true
          isVisible: hasNonMetaSelectedHolder
```

Code 1.3: Menu item for acceptVisitor: method code generator

## 1.3 Refactoring

Refactoring operation (1.1.6) represents a subclass of the class `Refactoring` which requires methods `preconditions` and `transform` to be implemented. From required protocol we could sense that `preconditions` is called first to check if refactoring can be performed and then `transform` produces the change objects (1.1.2). All necessary actions are wrapped within the `execute` method which also applies the changes to the system.

Binding with STX IDE resembles to what we described in the section above (1.2). Interesting difference is that `Tools::NewSystemBrowser` initializes a refactoring object with some data like selected class or source and then calls its method `performRefactoring:` which applies the changes to the system (code example 1.4).

```
codeMenuExtractMethod
    self withCurrentMethodsClassAndSelectorDo:[:mClass :
       mSelector |
        | refactoring |

        refactoring := (ExtractMethodRefactoring
                          extract: (self
                             selectedInterval)
                          from: mSelector
                          in: mClass).

        refactoring source:self codeView
           contentsAsString.
        self performRefactoring:refactoring.
    ]
```

Code 1.4: Menu action for extract method refactoring

Chapter 2

# Analysis

Previous chapter gave us an idea how is designed the older API. We will analyse the disadvantages and inconveniences from our point of view in this chapter. We identified roles which can treat code generators or refactorings in different ways.

- The creator (programmer) of generators/refactorings
- The tester of generators/refactorings
- The user who run generators/refactorings from STX IDE

## 2.1 Integration with IDE

We can see (1.2.1) that we need to add a menu item directly in the particular method within `Tools::NewSystemBrowser`. Since the menu is globally shared between developers it is hard to achieve customization. Furthermore extending the menus requires additional knowledge of `MenuItem` and `Tools::NewSystemBrowser` which is kind of a big bunch of methods (2198 [1]).

When we already succeed to create a menu item then we need to pass some data to our code which is, for example, selected classes in the IDE. Most of them holds `Tools::NavigationState` which is instance variable of `Tools::NewSystemBrowser`. The data can be retrieved from methods of the browser or directly from its navigation state. This can lead to tight coupling [6, chapter 2] if we do not have any interlayer. We also need to find appropriate method between many of them and understand how it works. Code example 1.4 shows how could be the data from the browser retrieved and used. In conclusion, we can say that it is a bit hard work to extend browser with new code generator.

[1] STX version 6.2.5, including extension methods from other packages

## 2.2 Class CodeGeneratorTool

At first glance this class and its implementation `SmalltalkCodeGeneratorTool` looks like the God object [2] anti-pattern which knows all about code generation. If we take a closer look then we can see that it contains a few utility methods for code generation and bunch of methods which actually generates the source code.

Having each code generator written in a method within one class tends to include all the logic in here. This can lead to overly complicated code which is hard to test and understand. On the other hand, splitting the code into more methods would make the class bigger and less understandable. In code example 2.1 from `SmalltalkCodeGeneratorTool` we can see that putting all logic in one method produces complicated code.

Another inconvenience could be with adding new generators. Following the path outlined by `CodeGeneratorTool` we would need to define abstract method and then its implementation in each programming language. Smalltalk allows adding a method to a class from another package and these kind methods are called extension methods. This makes the task easier, but still we can sense that bloating the class does not follow good programming practices. If a programmer decides to bypass it and create its own implementations somewhere else then we are likely facing incompatibilities and hard to reuse the code in case of another programmer will do the same thing.

Idea that one abstract code generator (`CodeGeneratorTool`) specifies all what can be generated by subclass responsibility [5, page 88] is misleading. We would expect that every programming language specific implementation will have required methods, but this cannot be easily achieved. Due to many methods (61 [3]) with required implementation, creating a new subclass for another language is difficult task. Probably part of the required methods will have no usage for specific language. For example, class `JavaScriptCodeGeneratorTool` does not implement all required methods and lots of code is copy from class `SmalltalkCodeGeneratorTool` which throws parse exception. Nevertheless, having same method signature for code generators which actually does the same thing like generating getter/setter is correct design.

In code example 2.1 we can notice that for source code building is used string concatenation and expansion. If also many conditionals take the place then understand what is going to be generated is very difficult. Exposed code is extreme case. With its five boolean parameters promises $2^5$ versions of produced source code.

Another inconvenience which could be seen in code example 2.1 is need for wrapping the generator code with methods `startCollectChanges` and

[2] `http://lostechies.com/chrismissal/2009/05/28/anti-patterns-and-worst-practices-monster-objects/`

[3] STX version 6.2.5

`executeCollectedChangesNamed:`. One may easily forget it and also additional noise is present in the code which can be omitted.

```
createAccessMethodsFor:aCollectionOfVarNames in:aClass
   withChange:withChange asValueHolder:asValueHolder
   readersOnly:readersOnly writersOnly:writersOnly
   lazyInitialization:lazyInitialization
  |classesClassVars|

  self startCollectChanges.

  classesClassVars := aClass theNonMetaclass
     allClassVarNames.

  aCollectionOfVarNames do:[:name |
    |source varType methodName defaultMethodName argName
       |

    varType := (classesClassVars includes:name)
      ifTrue:['static']
      ifFalse:[
        (aClass isMeta ifTrue:['classInstVar'] ifFalse
           :['instance'])].

    methodName := name.
    argName := 'something'.

    "/ the GETTER
    writersOnly ifFalse:[
      lazyInitialization ifTrue:[
        defaultMethodName := 'default' , name
           asUppercaseFirst.
      ].

      (aClass includesSelector:(methodName asSymbol))
         ifFalse:[
        asValueHolder ifTrue:[
          source := methodName , '\'.
          generateComments ifTrue:[
            source := source , ' "return/create the
               ''%2'' value holder (automatically
               generated)"\\'.
          ].
          source := source , ' %2 isNil ifTrue:[\'.
```

```
        lazyInitialization ifTrue:[
          source := source
            , '    %2 := self class %3 asValue.\'.
        ] ifFalse:[
    "Similar approx. 100 lines of code continues here"
  ].

  self executeCollectedChangesNamed:('Add Accessors').
```

Code 2.1: Method for creating many variations of getter/setter

No direct support for creating classes and methods within one undo-able change (1.1.2) could be also inconvenient. To achieve this, we would need to instantiate `RBNamespace` then define the class here with definition string and retrieve the class back from the namespace in its object form. Next step would be adding a method to the class and finally adding the changes (1.1.2) from the namespace to the local change collector. Other option would be to play around with the change objects itself (1.1.2). Both options are unnecessarily complicated and can be simplified.

STX contains support for automatic source code formatting with customizable settings. It would be nice to have this feature built-in for each generator. User could decide how the code will be formatted so he would not need to reformat the code after a generator execution.

In some cases, an error can occur while applying code changes (1.1.2) to the global system. Especially errors during parsing a source code are very common. Method `compile:forClass:inCategory:skipIfSame:` implementation [4] allows to add an invalid source code to the changes (1.1.2). This can lead to partially applied code changes without undo operation so that IDE user has to manually solve the problem with likely unwanted code leftovers.

Each method which generates some code has different signature. Although each follows similar naming convention, it is easy to make some mistake. For example, `createWidgetCodeFor:` and `createUpdateMethodIn:` take a class as argument, but does not follow exactly same naming convention.

Hard-coded references to global class names could be also limitation. For example, implementation of method `executeCollectedChangesNamed:` contains reference to class `RefactoryChangeManager` which affects global collector with undo operations. Modification of it is unwanted behaviour for unit test cases, because we would have IDE full of mess when running tests. Similar problem suffers the call of `information:` method. It is nice to see the text in a `dialog` window as user, but this does not allow running tests automatically.

[4]STX version 6.2.5

## 2.3 Package Refactory Browser

Here we describe disadvantages and glitches of package Refactory Browser and its sub-packages noticed in section 1.1.

This package was ported from `VisualWorks` to STX [5]. These two systems slightly differ from each other thus some incompatibilities comes as result. For example, STX supports private classes [6], but ported package lacks API for them.

### 2.3.1 Changes

Each method or class in STX can be categorized to `packages`. The information what package we would like to have for a new/modified class or method is transferred via `PackageQuerySignal` (see code example 2.2). For some use cases this API is not very comfortable so `RefactoryChange` has instance variable `package`. Unfortunately, current changes implementation does not allow custom modification of this variable. With this behaviour are related two unpleasant bugs. Package information is lost when creating undo change (method `asUndoOperation`) and package value is not respected in class `AddClassChange`.

```
sampleUsageOfPackageQuerySignal
  Class packageQuerySignal
    answer: #package_name
    do: [
      SomeClass compile: 'method_code_here ^ self'
    ]
```

Code 2.2: Sample usage of Class packageQuerySignal

### 2.3.2 Class model

The only way how to define a new class in `RBNamespace` is with `defineClass:` method with class definition string [7]. The definition string can be constructed with help of `RBClass` or directly written inside a code. First option basically requires filling all class attributes so that we can retrieve the definition with `definitionString` method. If we define the class with definition string from precisely created `RBClass` instance then this instance is not stored in `RBNamespace`, but another internally created `RBClass` instance. Second option is just passing hand written definition string. Code inside a string will

[5] `http://live.exept.de/doc/online/english/programming/goody_refact.html`

[6] `http://live.exept.de/doc/online/english/programming/namespaces.html`

[7] `http://live.exept.de/doc/online/english/getstart/tut_3.html` - fifth and sixth image

not be highlighted in text editor and we need to escape single quote characters, which is tedious. Also unavailable code assist (due to code in string) will make the work even more inconvenient.

Methods in this package works with class names defined as symbol [8]. If someone passes a string instead of symbol to some method as class name then it does not behave as expected. Since string can be easily converted to the symbol, there is no barrier to do it internally so that the API can be more user-friendly.

### 2.3.3 Parser

Users of code rewriting class `ParseTreeRewriter` may suffer from reformatting whole method source code after performed replacement. This problem solves its subclass `ParseTreeSourceRewriter` [9], but it does work just for complete methods and not expressions.

### 2.3.4 Formatter

Although Refactory Browser package (1.1) includes class `RBFormatter` for formatting a source code, it has some disadvantages due to its global nature. Settings how to format the code are stored in class (static) variables therefore we cannot simply have many custom settings for different use cases. For instance, if we need to format a source code with some specific setting then it will be set globally so other formatting actions will be affected. Sample use case within this thesis is code generator or refactoring test cases. We need the generated code formatted with predefined settings so that code comparison can be easily made.

The formatting within another part of STX system is done with reference to the formatter class stored in class (static) variable. This design is easy to implement, but introduces difficulties with usage of many formatters within different scopes of usage.

Source code pattern search and replace functionality from package `parser` (1.1.5) performs the replacements by parsing the code to AST nodes and their modification. Only way to retrieve the new code is to compose it from the AST tree which is done with help of a formatter class. The formatting process completely reformats the whole code. This could be unwanted behaviour especially when our precisely hand-formatted method will be garbled due to some expression replacement.

[8] `http://live.exept.de/ClassDoc/classDocOf:,Symbol`

[9] with help of class RBReplaceStringInOriginalSource

## 2.4 Method rewriter

Class `Tools::MethodRewriter` is a tool which opens a new window where we can write source code search and replace pattern [4]. This code rewriting can be performed on selected classes which are also part of the window.

Described disadvantages are mostly from IDE user point of view. Search and rewrite works only over selected classes, but it is possible to have it for each type of code (e.g. classes, methods, packages, selected source code part, etc.). Saved rewrite expressions (called as "templates") are defined as single methods in the class so adding a new template is not very intuitive. Rewrite expression selection and execution can be simplified to single menu item placed in the IDE context menu like actual code generators (1.2.1).

Chapter 3

# API in a nutshell

Goal of this chapter is to show how to use the API to create own code generators and refactorings. After that should be more obvious how and why API is designed (4) the way it is. On the other hand, this should not supply complete user manual with all features documented.

## 3.1 Usage in the IDE

STX [10] IDE (`Tools::NewSystemBrowser`) can have various arrangements whereas we present default layout. In Figure 3.1 is system browser with package view enabled (top menu selection View → Package). Each inner window shows different part of codebase and is marked in Figure 3.1 by number 1 to 6:

1. List of packages [5, page 28]
2. Method [5, page 54] source code editor
3. List of classes [5, page 277] under selected packages
4. List of instance variable [5, page 81] for selected class
5. List of protocols [5, page 37] for selected class
6. List of methods for selected class

User can select some items from any of the list or part of the source code and then perform a code generator or refactoring. Figure 3.1 shows how to use a code generator on a class. The process itself consists of the following steps:

1. Left-click on the class to select it

[10] `https://swing.fit.cvut.cz/jenkins/job/stx_jv/`

2. Right-click to show pop-up context menu

3. Point mouse cursor over "Generate - Custom" to show available code generators

4. Left-click on some menu item to actually perform code changes

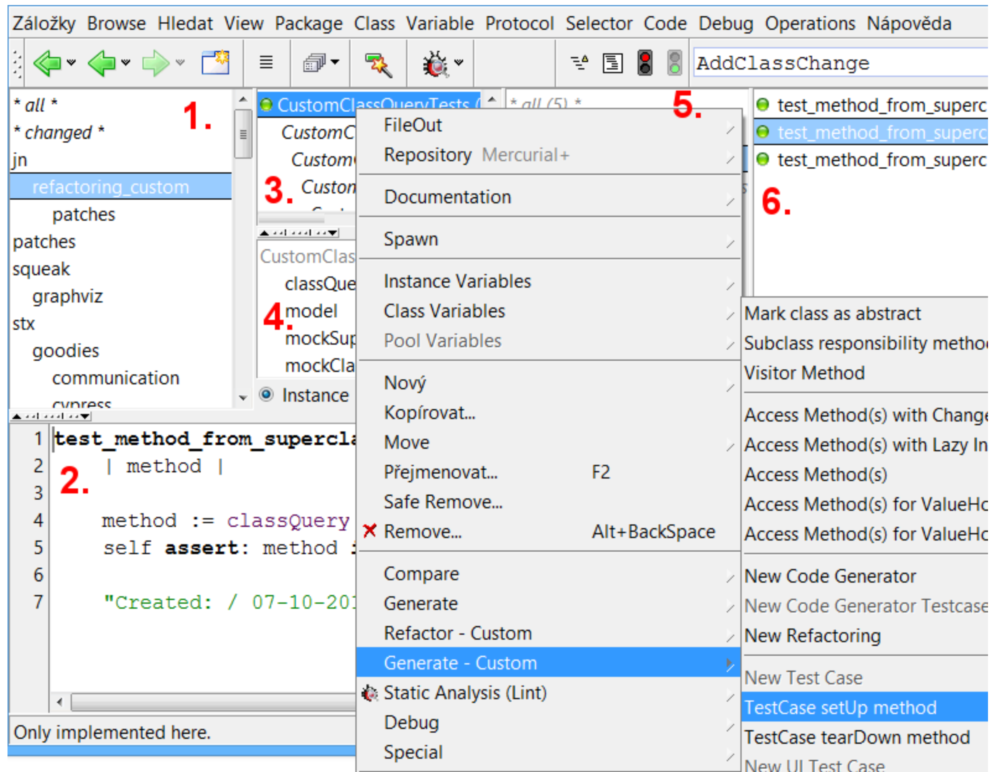



Figure 3.1: Usage of TestCase setUp method code generator

## 3.2 Usage inside a code

Code sample 3.1 creates test case class named `SomeClassTests` with generator `CustomTestCaseCodeGenerator`. Local variable context (4.3) holds a subset of the codebase (e.g. selected classes, methods, packages, etc.) which we described here 3.1. Additional generator settings are implemented as accessors under method protocol named `"accessing"`.

Another code generator or refactoring has the same concept, but can have different or none accessors for custom settings. Also each of them can make sense for different context. In other words, some generator uses selected classes. In contrast, another one uses selected methods.

```
sampleUsageOfGenerator
    | generator context |

    context := CustomSubContext new.
    context selectedClasses: (Array with: SomeClass).

    generator := CustomTestCaseCodeGenerator new.
    "These two lines are custom generator settings"
    generator testClassName: #SomeClassTests.
    generator testSuperName: #SomeClass.
    "Will create the test case class"
    generator executeInContext: context.
```

Code 3.1: Sample usage of a code generator or refactoring

## 3.3 Custom code generator

To create new code generator we need to add new class which inherits from `CustomCodeGenerator`. For this purpose we can use "New Code Generator" menu item visible in Figure 3.1. The helper will show pop-up window where we can enter a name. When we confirm the name then the new class is created along with required method stubs.

### 3.3.1 Instance methods

Only one method `buildInContext:` is required and also contains main functionality – build the source code. On a few examples we will show how to add new methods and classes.

#### 3.3.1.1 Method creation

Code example 3.2 creates new method `isClassName` for each selected class. Instance variable `model` (4.2) represents class model which acts as mirror to the global one in `Smalltalk` with respect to the changes done inside generator. If we add a method then it is immediately reflected the `model`, but in global space are the changes applied afterwards (when all is built successfully). For example, if we add line "class includesSelector: ('is', className) asSymbol" after the `compile` call then `true` should be returned although real class is unaffected at the moment of the method (`includesSelector:`) invocation.

Method `createMethod` returns initialized model method instance where we can set multiple attributes like owning class, protocol name (i.e. method category) and the source code. Special feature `replace:with:` replaces the place-holders in the source code with variable contents. Search patterns are

special for code matching (detailed documentation can be found here [4]) and replacement should be valid code fragment (expression or method call). Finally, method `compile` will actually add the new method and among other things formats the source code with custom formatter.

```
buildInContext: aCustomContext

  aCustomContext selectedClasses do: [ :class |
    | className |

    className := class theNonMetaclass name.

    model createMethod
      class: class;
      protocol: 'testing';
      source: ('`@isClassName

        ^ self theNonMetaclass == `@className
      ');
      replace: '`@isClassName' with: ('is', className);
      replace: '`@className' with: className asString;
      compile
  ]
```

Code 3.2: Required sample method which creates isClassName method

Code example 3.3 shows how to add a method with `compile:classified:` method call. With this simple call we cannot use the replace functionality and also custom code formatting is not supported – the reason of weird indentation in front of "`^ self`".

```
buildInContext: aCustomContext

  aCustomContext selectedClasses do: [ :class |
    class
      compile: 'myMethod
  ^ self'
      classified: 'a protocol'
  ]
```

Code 3.3: Required sample method which creates myMethod method

#### 3.3.1.2 Class creation

Method `createClass` in code sample 3.4 returns initialized model class where we can set many attributes like class name, its superclass, list of instance

variables and category. Method `compile` will define the class in the model and then is, for example, accessible via "model classNamed: #MyClass" or "model allClassesDo:" .

```
buildInContext: aCustomContext

  model createClass
    name: #MyClass;
    superclassName: #Object;
    instanceVariableNames: #('myInstVar');
    category: 'My-Category';
    compile
```

Code 3.4: Required sample method which creates new class

#### 3.3.1.3 Optional instance methods

Method `validateInContext:` can be implemented in order to validate the context. If we need to initialize some additional options then we can implement `configureInContext:`. This method is called only if the generator is run in the interactive mode i.e. a user invokes the generator from IDE pop-up menu. Therefore we can use it for showing a pop-up window where a user can enter some additional data like custom class name. Sample method for entering a class name shows code example 3.5. Variable `dialog` holds an instance of class `DialogBox` wrapper implementation. Class name entered in the `dialog` will be stored as instance variable named `myClassName`. Each configuration variable like `myClassName` requires getter and setter method for fluent work with class `AspectAdaptor`.

```
configureInContext: aCustomContext

  myClassName := self defaultMyClassName.
  dialog
    addClassNameEntryOn: ((AspectAdaptor forAspect:#
       myClassName)
        subject:self)
    labeled: 'Class name'
    validateBy: nil.

  dialog addButtons; open.
```

Code 3.5: Optional sample method which configures dialog for entering class name

#### 3.3.1.4 Composition with another generator

Code example 3.1 works rather well inside another generator, but the result might be slightly different from what we would expect. For example, there will be at least two undo operations after the generator with its inner generator will be executed. Code sample 3.6 reveals how can be two or more generators composed together. These methods help to create new sub-generator with the same set-up (e.g. same formatter, class model, etc.) as in the actual generator.

```
buildInContext: aCustomContext
  | subGenerator |

  "Creates initialized generator instance with same
     settings"
  subGenerator := SomeCodeGenerator
     subGeneratorOrRefactoringOf: self.
  subGenerator executeInContext: aCustomContext.

  "Shorthand to code above, executes initilized
     generators"
  self executeSubGeneratorOrRefactoringClasses: {
      SomeCodeGenerator.
      AnotherCodeGenerator
    }
    inContext: aCustomContext
```

Code 3.6: Sample usage of a nested code generator or refactoring withing another one

### 3.3.2 Class (static) methods

First three methods (code example 3.7) provides additional information about the generator. Method `description` should return detailed description about what the generator does. Method `label` should return label which will be visible in the menu. Optional method `group` should return a collection of category names in which the generator belongs. Generators with same group will grouped together in the menu.

Figure 3.1 shows IDE layout with open context sub-menu filled with generators. Each inner IDE window (marked with red numbers) contains different type of code (e.g. classes, methods, packages, etc.) and for each is displayed different context sub-menu. Method `availableInPerspective:` determines in which of these context sub-menus will be the generator present. Code example 3.7 shows method `isClassPerspective` which allows to include the generator in the context sub-menu inside the IDE class list (number 3. in Figure 3.1).

User selections within IDE (e.g. classes, methods, packages, etc.) are collected into the `context` (4.3) and method `availableInContext:` tells if the generator can be executed in it. Generators with improper user selection are visible in the context sub-menu, but they are greyed out and not executable. Telling the availability for a `context` (4.3) can be more complex than what is shown in the code example 3.7. For example, we can check if the class inherits from certain superclass or whether the class implements some methods.

```
description
    ^ 'Detailed description of the generator'

group
    ^ #('Group name' 'Subgroup name')

label
    ^ 'Label of the generator which will be visible in
      the menu'

availableInPerspective: aCustomPerspective
    ^ aCustomPerspective isClassPerspective

availableInContext: aCustomContext
    ^ aCustomContext selectedClasses notEmptyOrNil
```

Code 3.7: Required class methods of code generator or refactoring

## 3.4 Custom refactoring

Refactoring classes share the same API as described above for generators (3.3), but should inherit from class `CustomRefactoring`. Code example 3.8 shows how to use helper `refactoryBuilder` (4.5) for doing source code replacements. This particular example is meant to wrap selected source code fragment in the IDE editor (number 2. in Figure 3.1) with translation call "resources string: code_selection". More detailed description of code patterns search and replacements can be found here [4].

```
buildInContext: aCustomContext

  refactoryBuilder
    replace: '`@expression'
    with: '(resources string: (`@expression))'
    inContext: aCustomContext
```

Code 3.8: Required method to replace selected expression with another one

## 3.5 Overall schematic overview

Template class `CustomCodeGeneratorOrRefactoring` defines required and optional methods. The following subsections describe how they are accessed to achieve desired behaviour with flowchart. Methods which are intended to be overridden wrap angled brackets – $\langle methodName \rangle$.

### 3.5.1 Context menu building

Figure 3.2 illustrates single steps that are taken to build context menu with generators or refactorings for IDE (see also Figure 3.1). Method `group` is optional; other methods wrapped with angle brackets are required. Menu building itself is located in the class `CustomMenuBuilder` while iteration over generator classes depends on class `CustomManager`.

### 3.5.2 Generator or refactoring execution

Figure 3.3 shows how the public generator/refactoring execution point (method `executeInContext:`) works from inside. Methods `configureInContext:` and `validateInContext:` are optional; other methods wrapped with angle brackets are required.

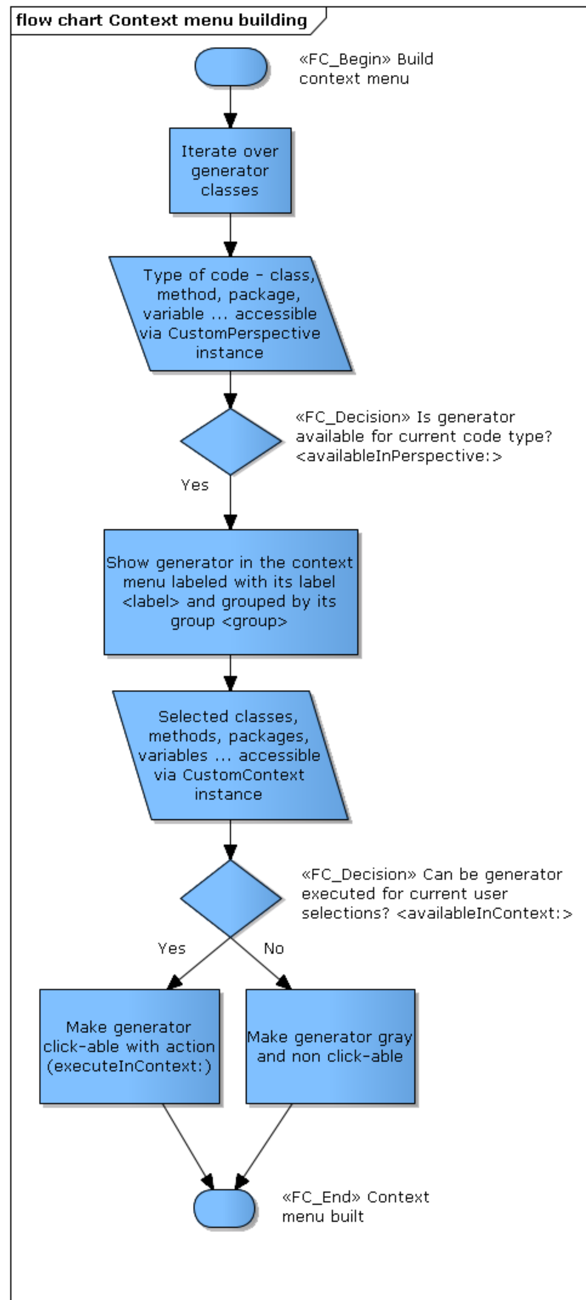



Figure 3.2: Context menu building flowchart with marked extension points

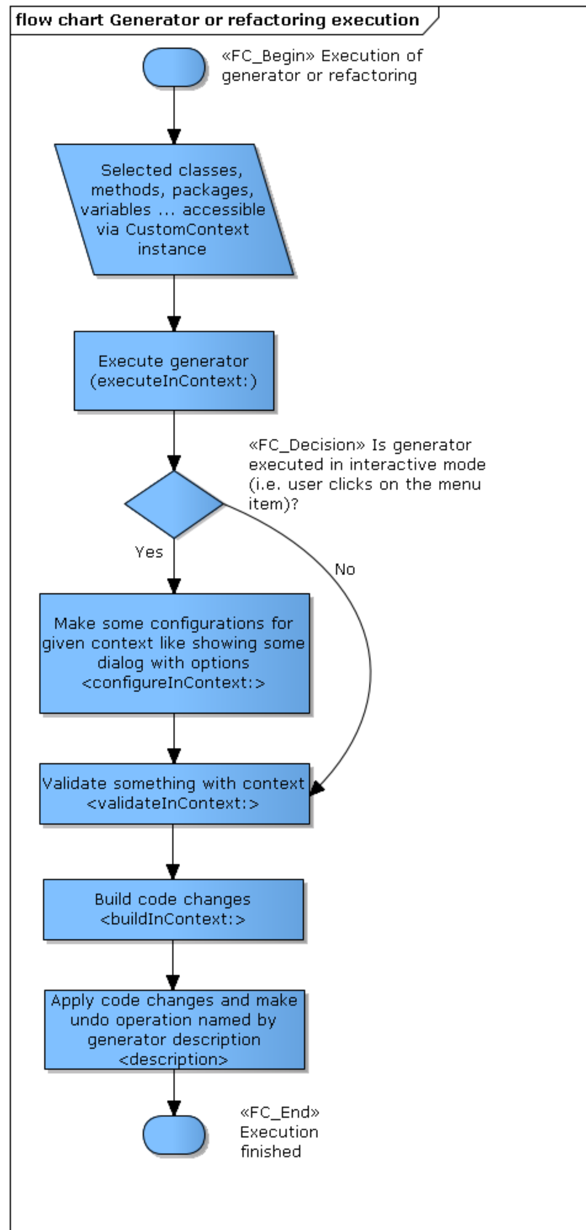



Figure 3.3: Code generator or refactoring execution flowchart with marked extension points

Chapter 4

# API design

In this chapter we will introduce how is API structured from class perspective and it solves issues described in the analysis chapter. Simplified class design is shown in Figure 4.1 (full method API is hidden along with less important classes). Key idea is that each code generator or refactoring will be realized as single class with implemented required methods. This allows us to have benefits like generator composition (5.1), seamless IDE integration (5.2) and unit test API (6.2).

## 4.1 Template class

Abstract class `CustomCodeGeneratorOrRefactoring` acts as template method design pattern [7] for code generators or refactorings. You can see that complete API is shared for both of them. We decided that separating could be more disadvantage then advantage, because in its nature both modify codebase and the difference is more like semantic. As a result, its subclasses `CustomCodeGenerator` and `CustomRefactoring` only categorize their subclasses by methods `isCustomCodeGenerator` and `isCustomRefactoring`.

Following sections describe each part of the API, because from detailed view is their purpose slightly different.

### 4.1.1 Code modification

Public template method `executeInContext:` applies code modifications assembled inside `buildInContext:` which has to be implemented by a subclass. In other words, the first one stands for public execution point and the second one is meant to be overridden to build custom code changes (1.1.2). With these two methods we can separate the responsibility to create the code change from actually applying them to the global system space. Therefore we can profit from it these advantages:

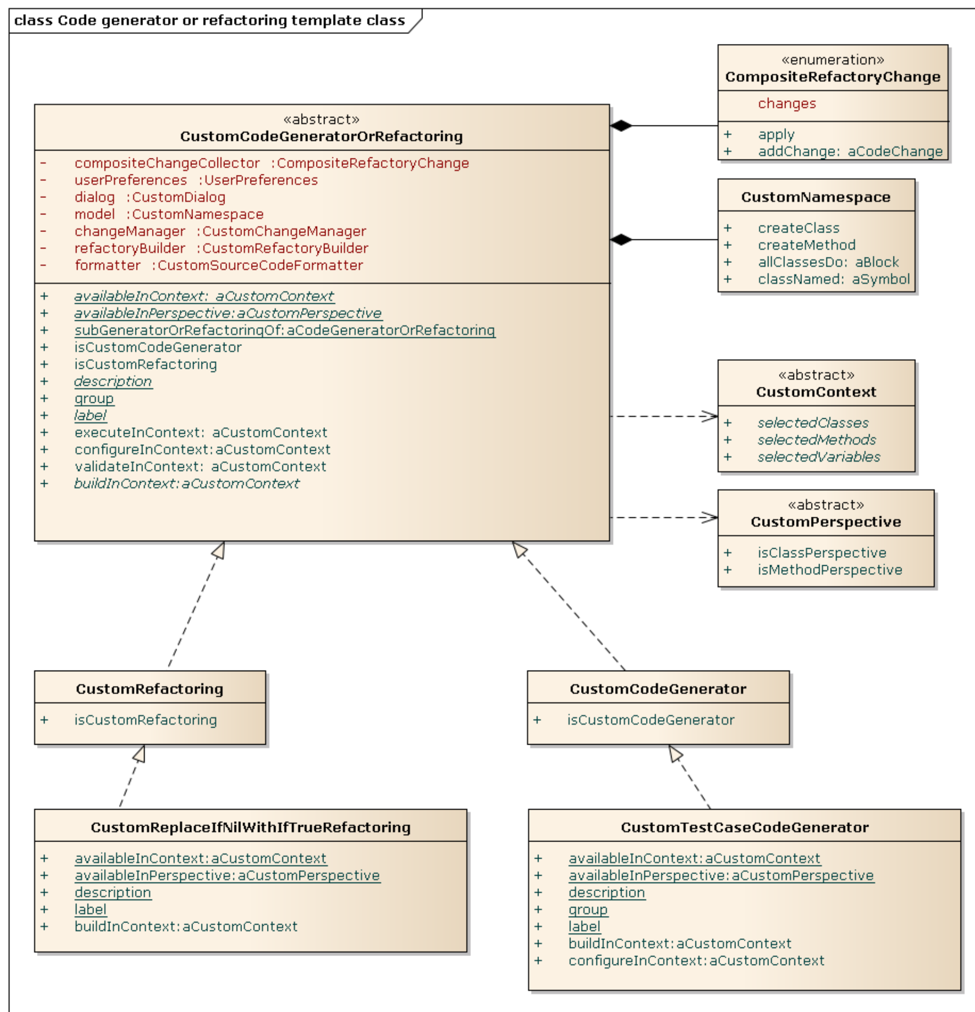



Figure 4.1: Code generator or refactoring template class

- Failed building does not affect global space (i.e. changes are not applied)
- No need to apply changes inside each subclass

### 4.1.2 Receiving input

Smalltalk codebase consists of elements like classes, methods and packages. A subset of these collects `CustomContext` instance (4.3) and determines on which codebase parts should generator or refactoring operate. Majority of them works with some subset of codebase and the `context` fits to be standardized input container for this purpose. Thus it solves the problem with different method signatures (2.2).

Data or options which do not fit universal container described above can be stored as individual instance variables with getters and setters. Variable `dialog` holds helper for constructing pop-up window to enter additional data. More general configuration options stores variable `userPreferences` (instance of `UserPreferences`). Instance of `CustomPerspective` subclass identifies type of code (4.4).

### 4.1.3 Scope limitation

Method `availableInPerspective:` tells with which type of code (e.g. classes, methods, packages, etc.) can be a generator or refactoring associated. In fact this allows to locate a generator or refactoring in the IDE context sub-menus (3.1). Method `availableInContext:` determines whether a generator or refactoring can be executed for concrete codebase parts (e.g. classes, methods, packages, etc.). Both methods are defined as static, because there is no need of any additional local class data. Also selecting generator or refactoring classes for `context` and `perspective` is easier, because creating an instance is not needed.

### 4.1.4 Class description

Description methods present the implementer to the end user. Method `label` should return short string label, `description` should return more detailed description of the class and finally optional `group` can provide list of group names in which the class belongs. All these methods are static, because they relates only to the class itself.

## 4.2 Class model

Class `Smalltalk` holds all classes and thus serves as global system namespace. For similar purpose is in class `CustomCodeGeneratorOrRefactoring` used `CustomNamespace` (child of `RBNamespace`) stored as instance variable `model`. Key goal is to keep code changes local while building them unless they are applied globally. All this respects the class and namespace API (basically the reflection API [8]) so the changes are immediately accessible in the local `model`. For example, if we create a new class then it will be directly stored and accessible in the `model`, but in global namespace `Smalltalk` will be the new class when the changes are applied.

Figure 4.2 shows simplified class design of the `model`. `RBNamespace` keeps track of single code changes (1.1.2) along with new, removed and modified classes. These are represented by `RBClass` and `RBMetaclass` with collection of `RBMethod` in order to emulate real class reflection API. Instance variable `environment` is an instance of `BrowserEnvironment` or any of its subclass. The class/subclasses itself have many features, but here are used to access

real classes within global namespace and eventually convert them to model classes. `BrowserEnvironment` allows all real classes to be accessed whereas most of its subclasses limits the classes with some condition like belonging to a package, category etc.

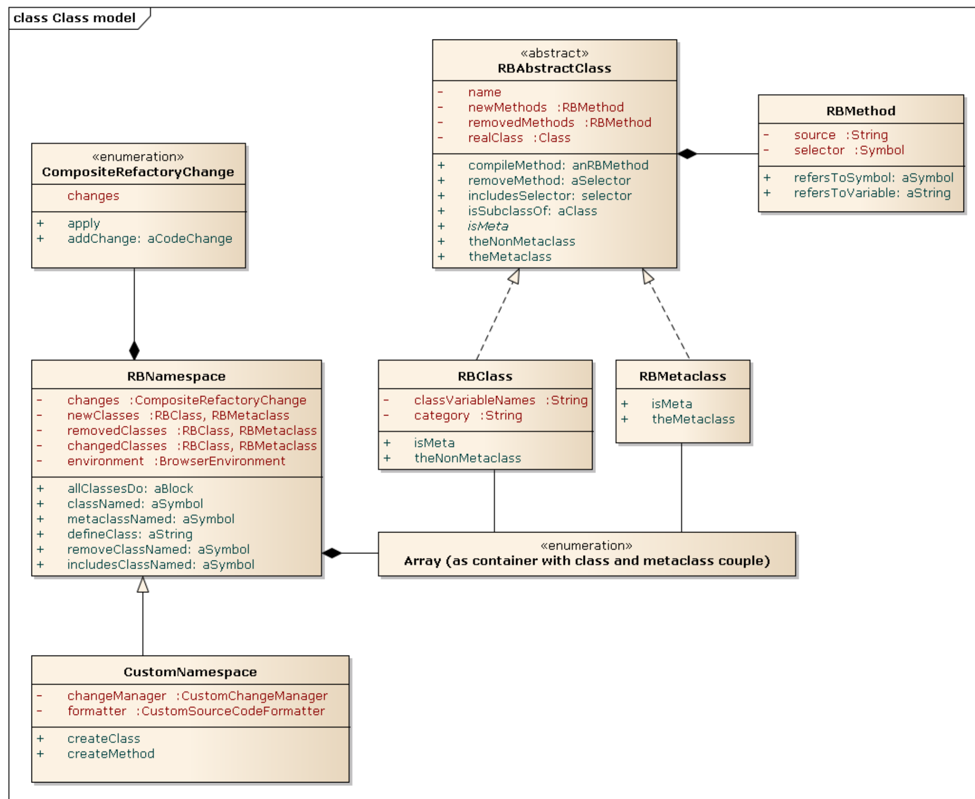



Figure 4.2: Class model

Majority of classes and methods comes from Refactory Browser package (1.1.3). We extended the functionality from two aspects:

- Missing class and method reflection API (5.3.1)
- Helpers to ease code building itself (5.3.2)

## 4.3 Context

The `context` design is inspired by `Tools::NavigationState` (most of the method names) which holds codebase selections within the STX IDE (classes, methods, packages etc.). As already mentioned, the purpose is to provide a codebase subset on which should operate code generators or refactorings. Fig-

ure 4.3 shows abstract class named `CustomContext` with two implementations along with `CustomPerspective` (4.4).

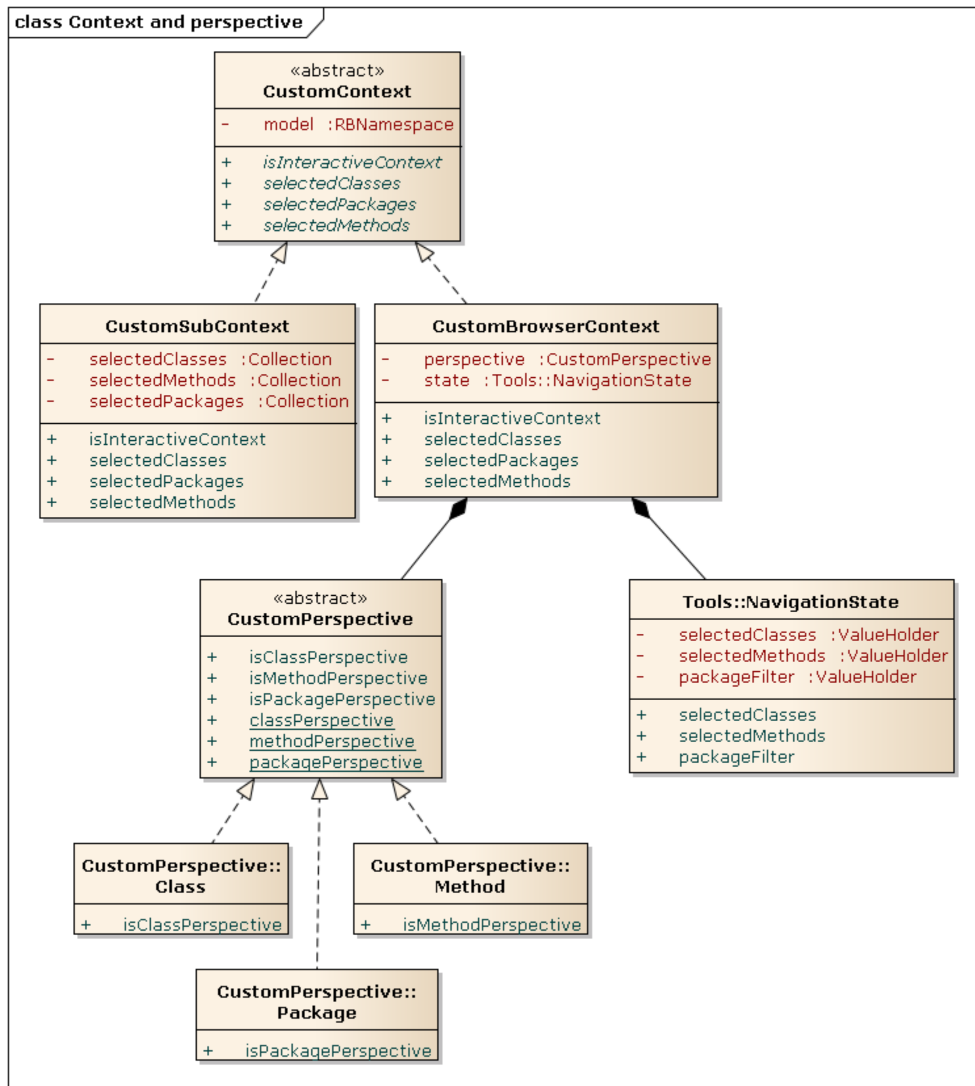



Figure 4.3: Context and perspective class diagram

Since code modifications are intended work with `model` (4.2), the reference to `RBNamespace` helps with converting real classes and methods to their mirror counterparts from the `model` inside each `context` implementation. Without this utility could be easily done an unwanted change to the global space. Furthermore, the creator of a generator would need to think that both real and model classes/methods could be passed as input. For example, "class compile: aCode" will create/modify the method immediately in the global space with real class whilst `model` class will register a code change

(1.1.2) to be applied afterwards.

`CustomBrowserContext` wraps `Tools::NavigationState` as partial facade pattern [7] implementation – only code selections within IDE are covered (see Figure 3.1). The information from what IDE part was `context` populated is held in `perspective` (4.4).

`CustomSubContext` stores code selections directly in its instance variables with getter/setter methods. This becomes useful if we need to populate custom data and pass them into a code generator or refactoring.

## 4.4 Perspective

`Perspective` provides simple marker for each type of code (e.g. classes, methods, packages, etc.). In usage inside IDE it represents the type of code the user works with and what is his "focus" list (i.e. classes, methods, packages, etc.) window. Class `CustomPerspective` defines class-type test method and factory method for each of its subclass which represents a single `perspective`. As seen in Figure 4.3, for example, `::Class` perspective overrides only method `isClassPerspective` which returns `true` while its parent class `CustomPerspective` defines `isClassPerspective` returning `false` as default value and class (static) method `classPerspective` returning an instance of `::Class`.

## 4.5 Refactoring helper

Class `CustomRefactoryBuilder` is inspired by `RefactoryBuilder` which implements partial facade for single refactoring classes in Refactory Browser package (1.1.6). Original class does not fit to the described generator/refactoring API design thus we do not reuse as superclass of `CustomRefactoryBuilder`. Code changes (1.1.2) are not stored in the same collector and they are not reflected in the `model` (4.2). Without shared change collector cannot be easily achieved correct order of changes. Outdated `model` would return wrong data which can lead to secondary misbehaviour.

In Figure 4.4 is simplified class diagram of `CustomRefactoryBuilder` with only methods for performing refactorings. Apart from simple operations like class category change, the helper offers source code search and replace functionality based on `ParseTreeSourceRewriter`. How to write code search patterns is well documented in [4]. Main advantage is that we can operate directly on codebase subset stored in the `context` (4.3). This includes ability to search on selected part of a method source code (e.g. just some expression within the source code).

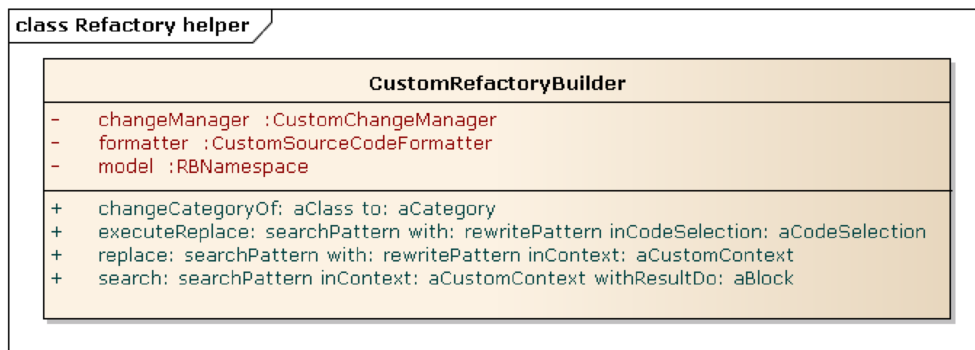



Figure 4.4: CustomRefactoryBuilder class diagram

## 4.6 Formatter

To solve the problems listed in section 4.6 we defined abstract formatter `CustomSourceCodeFormatter` with required method `"formatParseTree:"` and two different implementations. Class `CustomRBLocalSourceCodeFormatter` uses internally `RBFormatter`, but stores formatting settings locally in instance variable. Class `CustomNoneSourceCodeFormatter` keeps source code original formatting.

Chapter 5

# Implementation

Previous chapter gave us detailed overview of the architecture and most probably revealed part of obvious implementation. Here we will dig into implementation details which we considered worth to mention.

## 5.1 Generator or refactoring composition

Methods for composing generators or refactorings are noticed along with code example in subsection 3.3.1.4. Class `CustomCodeGeneratorOrRefactoring` defines several instance variables which holds initialized components useful for code generation or refactoring. Most of these need to be shared across all components during the execution in order to achieve proper behaviour.

Change object (1.1.2) collector (`CompositeRefactoryChange`) stored in instance variable `compositeChangeCollector` should be common for each part which creates change objects. With this premise can be easily achieved correct order of changes. First implementation simply put together changes from various change collectors, but later came out that incorrect order of changes may produce wrong results. For example, add class change needs to be before add method change if the method belongs to the class. Current implementation makes sure that only one change collector will be shared across all dependencies.

Similar case as noticed above is with `model` (4.2). If there were at least two different instances of `model` within generator or refactoring execution then one may become obsolete and thus return wrong values. Even `context` (4.3) which is given as method argument to the generator or refactoring requires the same `model`, because the `context` itself converts real classes and methods to their `model` counterparts (`RBClass` and `RBMethod`).

Figure 5.1 illustrates how objects are referenced by each other for instance of generator or refactoring being executed (`executeInContext:` method call). If we strip away `aCustomContext` instance (which is method argument) from the diagram then it will show common set-up.

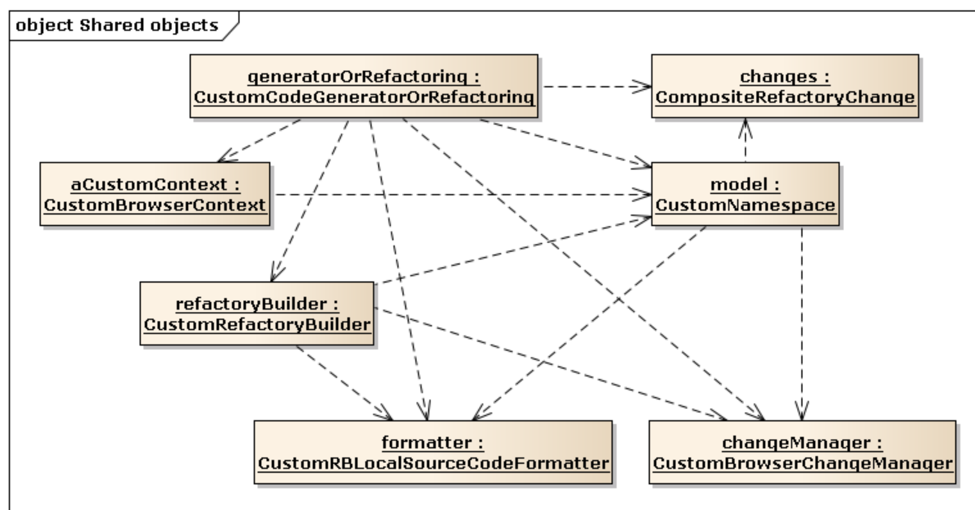



Figure 5.1: Shared objects diagram at the point of during generator or refactoring execution. Note: Arrows represent "contains reference to object" relation

For generator or refactoring execution is intended to use only method `executeInContext:` or its wrappers. Question could be how is then determined when can be changes applied to global system space if the method is used also for inner generators or refactorings? Instance variable named `compositeChangeNesting` holds nesting count. At the end of execution method is called `executeCollectedChangesNamed:` which decrements the counter and on zero (indicates top generator or refactoring) applies the changes.

## 5.2 Integration with IDE

Functionality described in section 3.1 implements class `CustomMenuBuilder` (main part of implementation). Context menu for each IDE inner window (classes list, source code editor, etc.) can be dynamically modified with extension methods. Code example 5.1 shows extension method for class `Tools::NewSystemBrowser` (IDE implementation) which adds menu item labelled "Generate - Custom". Annotation `menuextension:` specifies type of menu which will be passed as method argument on its invocation. These menu extension methods are called every time when is context menu shown (e.g. every time user does right-click). In practice this means that changes made to the context menu are reflected immediately.

Method `customMenuBuilder` returns instance of menu builder with preset attributes common for each such extension method. Menu building starts in method `buildMenu` which creates menu item with sub-menu containing menu items assembled from generators or refactorings. Class `CustomManager`

provides access for generator or refactoring classes and is used to select appropriate generators or refactorings for the sub-menu. Simplified algorithm is mentioned in the subsection 3.5.1.

```
classMenuExtensionCustomGenerators: aMenu
  <menuextension: #classMenu>

   self customMenuBuilder
     perspective: CustomPerspective classPerspective;
     menu: aMenu;
     submenuLabel: 'Generate - Custom';
     afterMenuItemLabelled: 'Generate';
     generatorOrRefactoringFilter: [
       :generatorOrRefactoring |
       generatorOrRefactoring isCustomCodeGenerator ];
     buildMenu.
```

Code 5.1: Context menu extension for classes list in the IDE

## 5.3 Class model extensions

### 5.3.1 Missing reflection API

STX total method (both object and class/static) count for class, method and namespace model classes is 1489 [11]. Although all of them do not belong to the reflection API, this number illustrates that the API is overall huge. Moreover, this simple sum does take in account other programming language included in STX (Java [9, chapter 2], JavaScript and Groovy). Rather than implementing complete API one by one method, we selected only those which were useful for generators or refactorings we created. One could ask why we have not simply reused the original. The reason is that the original represents existing elements (classes, methods and namespaces), but `model` described in section 4.2 represents existing, new and modified elements thus requires different implementation.

Most of the extension methods implementation is trivial so we explain only interesting ones. Among not so straightforward things we can include method dictionary. Regular class stores its methods in special collection – instance of `MethodDictionary`, but `model` class keeps track only of new and removed methods. Method dictionary is built on demand (i.e. call of `methodDictionary`) from existing (from real class) and new methods minus removed ones. This solution does not excel in speed, because every time is

[11]STX version 6.2.5, classes: Behavior, ClassDescription, Class, Metaclass, PrivateMetaclass, NameSpace, Smalltalk, CompiledCode, Method

MethodDictionary instantiated and filled from scratch, but is easy to implement and thus error prone. If we would create the dictionary on model class initialization (i.e. constructor) then we would need to keep it updated every time when methods are created and removed along with real class changes. Similar problem would exist with some caching strategy, because its invalidation (known as cache invalidation) would be complicated.

Due to package origin (2.3), private classes [12] reflection API was completely missing. Significant difference between private and public class is that private class belongs to some class (private or public) – the owning class. For newly created private class is the owning class retrieved from definition string [13]. Existing private class already has its owner and thus is returned. Unlike regular private class which has implementation in special class PrivateMetaclass, here we use just RBMetaclass which stands for both private and public class. This simplifies both usage and implementation, because we can change the "privacy" just by setting class owner.

### 5.3.2 Code building helpers

Section 2.3.2 criticizes original possibilities from API user aspect. Factory method createClass returns instance of RBClass with filled default class attributes. As seen in subsection 3.3.1.2 user can fill custom attributes (e.g. class name, instance variables, etc.) and then reflect it in model (4.2) with compile method. Advantage of using factory method instead of direct class instantiation is more flexible code, because we can easily change the class without rewriting the code. Default values saves the time when writing code. For example, majority of classes has none poolDictionaries, but class definition requires them so we can assume that they will be empty unless user fills a custom value. On the contrary compile method is a compromise. Although it eases the usage in comparison with the original implementation, it could be completely omitted with different implementation. One of possible implementation could be change tracking (for instance via update:with:from:) in each setter method and different logic for creating change objects (1.1.2). The reason why we have not implemented this is due to lack of enough time to do it properly – biggest challenge would be legacy issues.

In a similar way works method createMethod, but returns initialized instance of RBMethod. Key feature is ability to define code template with placeholders and replacements (3.3.1.1). Although it is not a full blown template engine [10], it fulfils some aspects like metalanguage, AST usage and syntax check. In contrast with code building mentioned in section 2.2 this improves generator readability thus we can better understand what is going to

[12] http://live.exept.de/doc/online/english/programming/namespaces.html#PRIVATECLASSES

[13] http://live.exept.de/doc/online/english/getstart/tut_3.html - fifth and sixth image

be generated when looking on generator code. The metalanguage is taken from Refactory Browser package (1.1.5). Instance of class `CodeGenerator` holds replacements, template source code and also provides API for transforming the template to final source code. The template code is first parsed with class `RBParser` and then by traversing AST tree are replaced the placeholders. Class `CustomSourceCodeGenerator` follows this implementation and makes possible method call replacements (instead of only expression) along with local source code formatter (4.6). Finally `RBMethod` loosely integrates the functionality just by exposed `replace:with:` method where replacements are performed in `compile` method.

## 5.4 Refactoring helper

Here we follow section 4.5 with more detailed implementation description. Code example 5.2 shows that search and replace are separated into two methods. Argument named `aCustomContext` (4.3) can contain various parts of codebase (e.g. packages, classes, methods, etc.). Pattern searching works on method source code therefore we need, for example, an algorithm which is able to search on methods belonging to some packages. Similar algorithm is useful also for classes or class categories. For each such algorithm is defined private class [14] of `CustomRefactoryBuilder`. Search simply iterates over these classes and calls their method `search:inContext:withResultDo:`.

The user who browses through codebase with IDE has usually selected multiple types of code (e.g. packages, classes, methods, etc.) at once. All these selections are present in the `context` (4.3) as well. With described behaviour above would be all these selections subject of search and replacement, but from user point of view is this inappropriate feature. To solve this glitch, each private class implements class (static) method `availableInPerspective:` which tells on what type of code the class operates (4.4). This helps to search only on focused user selection. For example, if user selects a class and executes search then only methods in this class are searched (instead of all selections in the IDE).

Instance of class `CustomSourceCodeSelection` is passed as result of matching search pattern to method which performs the replacement. This class represents container for selected source code fragment in method and can contain either whole code or, for example, just some expression. The replace method `executeReplace:with:inCodeSelection:` firstly performs a replacement on the selected source code and if whole method is not selected then the new code is injected into the method's source. Finally whole new method source code is compiled to the class using `model` (4.2).

[14] `http://live.exept.de/doc/online/english/programming/namespaces.html#PRIVATECLASSES`

```
replace: searchPattern with: rewritePattern inContext:
   aCustomContext
  "Searches for given pattern in methods source code or
     selected code fragments and if source code matches
     then executes replacement"

  self search: searchPattern
    inContext: aCustomContext
    withResultDo: [ :sourceSelection |
      self executeReplace: searchPattern
        with: rewritePattern
        inCodeSelection: sourceSelection
    ]
```

Code 5.2: Method for souce code search and replace

Chapter 6

# Testing

The development process consisted of writing new generators or refactorings along with underlying API. To verify that the developed code behaves as expected is used sunit test framework [15] which represents classic xUnit framework [16]. Each class has a test case class which name ends with `Tests` and its category ends with `-Tests`. Tests for extension methods (i.e. methods for existing classes outside this project) are located in test case classes named `Custom<ClassName>Tests`.

Main levels of testing [6, chapter 4] are unit, integration and system tests. If we consider unit as method, class or even whole package then the written tests are mixed unit and integration tests (depends on concrete test method). For instance, testing that code generator creates a method involves nearly all dependent components (Refactory Browser, Compiler, etc.) to create real existing method. Many tests relate to codebase modification therefore they are integration tests. According to [11] (if we omit tabloid/offensive phrases) our integration tests suffer from slower execution time and also the fact that we are not testing only our package isolated from other packages. When something wrong comes out from outer packages then it is hard to find out where the error/misbehaviour comes from. On the other hand, the tests are written per method as the unit and follow multiple execution paths which keep the code coverage [6, chapter 4, subsection 4.2.1.] high (Figure 6.1). However, what stands for best unit size and isolation level is questionable [12] and depends on project nature. In most of the tests the code goes through parser and compiler which we consider valuable.

Created generators or refactorings were manually tested through UI on Linux and Windows environment which can be considered as system tests. Nevertheless, none of created code is intended to be platform dependent. Since we have Jenkins job [17] it would be nice to have repeatable (regression) system

[15] `http://live.exept.de/doc/online/english/tools/misc/testfram.htm`
[16] `http://www.martinfowler.com/bliki/Xunit.html` (JUnit, NUnit, PHPUnit, etc.)
[17] `https://swing.fit.cvut.cz/jenkins/job/custom_refactorings_reports/`

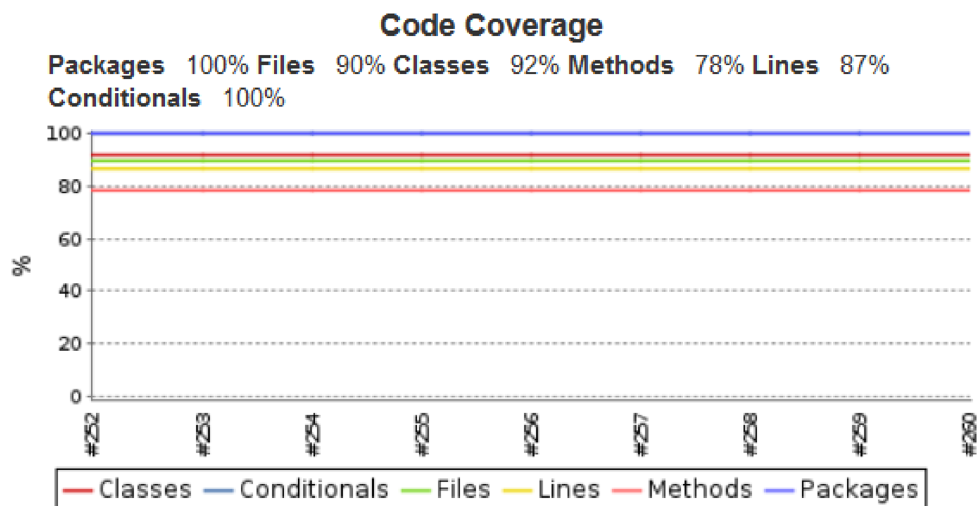



Figure 6.1: Code coverage

tests, but after searching the web it does not seem that exists easy to use prepared solution.

UAT tests [6, chapter 4, subsection 2.2.1.] should check if end user requirements are satisfied. The only known end users were the thesis supervisor and me thus we evaluated how user-friendly the API is. If the API will be presented to the Smalltalk programmers community then probably someone will be willing to provide valuable feedback, but this was not realized so far.

## 6.1 Mocks and stubs

At the time of writing this thesis there was no mock/stub [18] framework in STX and none of existing solutions in other Smalltalk dialects does not seem to be suitable [19] [20] [21]. Mocks/stubs for test cases are created with class `CustomMock` which is "poor man's" mocking helper. Code example 6.1 shows basic usage. Implementation simply creates new subclass with generated name and allows to compile methods without modification of change file (i.e. text file where are stored code changes). To be complete, Jan Vraný created class (static) methods with mocking possibilities into the class `MessageTracer` under protocol `"method mocking"`. These methods are designed to be thread safe (considering test method isolation) and could possibly replace `CustomMock` in many

[18] http://www.martinfowler.com/articles/mocksArentStubs.html
[19] http://www.squeaksource.com/Mocketry.html
[20] http://codedaemon.com/sMock/
[21] https://joachimtuchel.wordpress.com/2011/12/31/yet-another-mock-object-framework-for-va-smalltalk/

places, but the solution is less powerful, because it attaches method per class therefore it cannot be simply used for different mocks/stubs of a single class.

All mocks/stubs used within this thesis could be done also with real classes and methods (i.e. existing classes/methods which exist during whole project lifetime), but it would require more effort writing tests and all mock classes and methods would make codebase bigger thus less understandable. On the other hand, this mock/stub implementation would fasten the tests execution speed.

```
sampleUsageOfCustomMock
  | mockHelper mock |

  mockHelper := CustomMock new.
  "Creates instance of Object mock"
  mock := mockHelper mockOf: Object.
  "Creates/overrides method"
  mock compileMockMethod: 'selector ^ true'.
  "Check if method works as expected"
  self assert: mock selector = true.
  "Removes all mocks"
  mockHelper unmockAll.
```

Code 6.1: Sample usage of CustomMock

## 6.2 Generators or refactorings tests

They have the same interface and basically do the same thing – codebase modification. This means that also the tests will follow very similar patterns. Specialized test case class `CustomCodeGeneratorOrRefactoringTestCase` along with generator for creating such test case classes aims to simplify the testing. Code example 6.2 shows how can be tested that particular method is correctly generated.

Template test case class for generators or refactorings provides common configuration inside `setUp` method where are initialized generator or refactoring instance variables (4) along with input `context` (4.3). To make testing independent of global user settings which can vary and thus make the testing difficult, local instance of user settings is created with predefined values. Same case is with source code formatter (4.6). Local change (1.1.2) manager allows making code changes without affecting undo/redo menu in IDE along with cancelling code changes in `tearDown` method. Also change file modification is prevented with help of `"Class withoutUpdatingChangesDo:"` [22]. This helps with real classes and methods testing in that IDE is without code leftovers.

[22] These are different changes from the changes in the undo/redo menu

During testing generators or refactorings became apparent that some assertions and `context` are constantly repeating. Specialized assert methods are located under protocol `"asserting"` and methods creating `context` templates are under protocol `"context templates"`. Code example 6.2 contains call of `executeGeneratorInContext:` method with `context` template name (method name) as argument. At the end of outlined test is called method `assertMethodSource:atSelector:` which checks source code of method under given selector for class created inside method preparing `context` (i.e. the `context` template method).

```
test_simple_setter_method_generated_without_comments
  | expectedSource |

  userPreferences generateCommentsForSetters: false.

  expectedSource := 'instanceVariable:something
  instanceVariable := something'.

  self executeGeneratorInContext:
    #classWithInstanceVariable.
  self assertMethodSource: expectedSource
    atSelector: #instanceVariable:
```

Code 6.2: Test case method for generator which creates setter method

## 6.3 Testing summary

The tests were very valuable, because some parts of the project were reworked multiple times. Without tests, refactoring would be hard and very vulnerable to introducing new bugs. However, the tests do not make sure that the project is completely free of bugs.

# Conclusion

We have presented API for creating code generators or refactorings along with testing API. By meeting goals stated at the thesis beginning, our work offers following major benefits:

- Hidden implementation details of IDE integration (5.2) from programmer
- Standardized interface (4.1) and inputs (4.3) of created generators or refactoring
- Class and method creation factories along with template-like definition of source code (5.3.2)
- Testing API with test case generator (6.2)

Code generators and refactorings were created during the whole development process in order to design, improve and verify the API. Part of them consists of rewritten existing generators in STX. Others are created for development purposes within this work. Test case class/method generators were the most helpful and made the testing process easier.

## Future improvements

There are many potential improvements/features which could make code generating or refactoring better.

### Switch to generated code in IDE

When a new class or method is generated then IDE user can have automatically opened new IDE tab window with selected newly generated class or method.

### Comments inside new code

Some generators have the option to generate methods with or without method. Code example 6.3 shows how are comments included or excluded from source code. However, comments on/off switching can be also done with `rewriteOn` and `rewriteSavingCommentsOn` parser and scanner setting which could completely remove the if-branch and thus simplify the code.

```
sourceForClass: aClass variableName: aName
  "Returns simple getter method source code for given
     class and variable name"

  | methodName comment |

  methodName := self methodNameFor: aName.
  comment := ''.

  userPreferences generateCommentsForGetters ifTrue:[
    | varType |

    varType := self varTypeOf: aName class: aClass.
    comment := '"return the %1 variable ''%2'' (
       automatically generated)"'.
    comment := comment bindWith: varType with: aName.
  ].

  ^ self sourceCodeGenerator
    source: '`@methodName
    `"comment

    ^ `variableName';
    replace: '`@methodName' with: methodName asSymbol;
    replace: '`variableName' with: aName asString;
    replace: '`"comment' with: comment;
    newSource.
```

Code 6.3: Current implementation how are comments enabled/disabled in source code

### General purpose codebase iterator

Refactoring helper (4.5) iterates through set of methods based on `context` (4.3) to perform search operation. This iteration is implemented directly within the helper. However, similar iterator would be useful also for other code types like classes, protocols, packages, etc. which could be used outside

the helper. Class `BrowserEnvironment` with its subclasses implements such iterator and can be possibly used to some extent, but its usage inside generators or refactorings is rather complicated and some features may be missing. Desired API could be that `context` (4.3) will implement method returning initialized iterator which will offer methods like `classesDo:`, `methodsDo:`, `protocolsDo:` and filtering options.

### Keyboard shortcut bindings

Currently are generators or refactorings executable via menu items (3.1) or by method call (3.2). Frequently used ones could be also executable with press of a keyboard shortcut (e.g. `F6`, `Ctrl+l`, etc.) which could be assigned in user preferences. This would allow better usage for more than one programmer using a generator or refactoring.

Keyboard shortcut can be assigned to a menu item with `shortcutKey:` method. This seems like possible implementation for generators or refactorings even if the menu for them is built dynamically, because it is rebuilt on each keyboard shortcut key press.

### Integration with SmallSense

SmallSense [23] is programmer productivity plugin for STX. Features include code-completion [24] which shows list of possible code elements as programmer type. Extending this list with custom items each with custom action could help a programmer to be more productive. For example, if a programmer often uses expression `ifTrue:[]` then extending the list in a way that after writing just letter "i" it will offer the expression at first position, would be helpful. The completion action could be more complex than just insert static piece of code.

### API for interactive code changes confirmation

Some refactorings may create a lot of changes in existing code. Part of them may not meet the programmer's requirements. In such a case it would be better to display a confirmation window with old and new code for each change.

### Progress-bar for long time refactorings

With previous subsection relates possible long time execution of refactoring. In such case would be good to show progress-bar were will be visible how much work has been completed and how much work remains to finish the operation.

[23] https://bitbucket.org/janvrany/stx-goodies-smallsense

[24] http://code-recommenders.blogspot.cz/2010/05/its-all-about-intelligent-code.html

### Class model setters with changes creation

Section 5.3.2 describes the reason of `compile` method and suggestion how could be implementation changed. In other words, all `model` (4.2) setter methods which shadow real class and method interface could create changes (1.1.2) on the fly. For instance, `RBClass` method `category:` can create class category change object if the class is defined. Unfortunately, the class `model` is used also outside this thesis scope thus this improvement should be aware of legacy issues.

### Visibility in user preferences

Many generators and refactorings in the context menu can be confusing and hard to use, especially when the menu is bigger than the display screen. Possibility to limit list of generators and refactorings in user preferences would be more user-friendly than direct code modification of particular generator/refactoring – especially when it will be shared across many programmers.

Class `CustomManager` has instance variable which is the source for generators or refactorings. Here can be injected an object implementing method `generatorsAndRefactoringsDo:`.

### Multiple programming language support

STX primarily focuses on Smalltalk programming language, but to some extent supports Java, Groovy and JavaScript [25]. This thesis is built with heavy usage of Refactory Browser package (1.1) which focuses also on Smalltalk therefore creating code generators or refactorings for other programming languages is very limited.

We were interested in how this could be implemented. First task is ability to assign code generator or refactoring to concrete programming languages in order to show only relevant ones in context menus. Template class for generators or refactorings (4.1) contains class (static) method returning list of supported programming languages [26] and method which tells availability for programming languages [27] within `context` (4.3). We also created two prototype code generators for JavaScript and Java so we can provide early observations. JavaScript in STX is different in comparison to well known web-browser implementation and resembles to the Smalltalk class structure (e.g. separated class definition and methods). Each JavaScript method is compiled separately thus works better with changes package (1.1.2), but fails with parsing the source code. In contrast, Java code is compiled as whole class with methods.

[25] Currently included in `https://swing.fit.cvut.cz/jenkins/job/stx_jv/`, STX version 6.2.5

[26] availableForProgrammingLanguages

[27] availableForProgrammingLanguagesInContext:

# Bibliography


[1] Application programming interface [online]. Wikipedia, The Free Encyclopedia. October 2014, page last edited 2014-09-02 [cit. 2014-10-11]. Available from: `http://en.wikipedia.org/w/index.php?title=Application_programming_interface&oldid=623906555`

[2] Fowler, M. *Refactoring. Improving the Design of Existing Code.* Addison-Wesley, 1999, ISBN 0-201-48567-2.

[3] Refactory Browser [online]. ST/X Smalltalk Online Documentation. October 2014, [cit. 2014-10-22]. Available from: `http://live.exept.de/doc/online/english/programming/goody_refact.html`

[4] Code Search Patterns [online]. ST/X Smalltalk Online Documentation. October 2014, [cit. 2014-10-23]. Available from: `http://live.exept.de/doc/online/english/help/Browser/RBSearchPatterns.html`

[5] Andrew P. Black, O. N., Stéphane Ducasse; Pollet, D. *Pharo by Example.* Square Bracket Associates, Switzerland, 2009, ISBN 978-3-9523341-4-0. Available from: `http://pharobyexample.org/`

[6] Bourque, P.; R.E. Fairley, e. *Guide to the Software Engineering Body of Knowledge, Version 3.0.* IEEE Computer Society, 2014, ISBN 978-0-7695-5166-1. Available from: `www.swebok.org`

[7] Gamma, E.; Helm, R.; Johnson, R.; et al. *Design Patterns: Elements of Reusable Object-oriented Software.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995, ISBN 0-201-63361-2.

[8] Smalltalk reflective model [online]. Mariano Martinez Peck. April 2011, [cit. 2015-01-10]. Available from: `https://marianopeck.wordpress.com/2011/04/30/smalltalk-reflective-model/`

[9] Hlopko, M. *Java implementation for Smalltalk/X VM*. Master's thesis, Czech Technical University in Prague, 2011. Available from: https://dip.felk.cvut.cz/browse/pdfcache/hlopkmar_2011dipl.pdf

[10] Jeroen Arnoldus, A. S. J. B., Mark van den Brand. *Code Generation with Templates*. Atlantis Press, Paris, France, 2012, ISBN 978-94-91216-56-5. Available from: http://books.google.cz/books?id=UvCOMJHSqjkC&source=gbs_navlinks_s

[11] Rainsberger, J. B. Integrated Tests are a Scam: Part 1 [online]. March 2010, [cit. 2015-01-18]. Available from: http://www.jbrains.ca/permalink/integrated-tests-are-a-scam-part-1

[12] Martin Fowler, D. H. H., Kent Beck. Is TDD Dead? [online]. May 2014, [cit. 2015-01-18]. Available from: http://martinfowler.com/articles/is-tdd-dead/

Appendix A

# Acronyms

**API** Application programming interface

**STX** Smalltalk/X

**IDE** Integrated development environment

**AST** Abstract syntax tree

**UI** User interface

**UAT** User acceptance tests

Appendix B

# Contents of enclosed CD

```
readme.txt ....................... the file with CD contents description
src ..................................... the directory of source codes
  jn .................................... implementation source codes
  thesis .............. the directory of LaTeX source codes of the thesis
stx ................... the directory with STX including implementation
  bin ................................. the directory with executables
  lib .......................... the directory of complete source codes
text ......................................... the thesis text directory
  thesis.pdf ........................... the thesis text in PDF format
  thesis.ps ............................. the thesis text in PS format
```