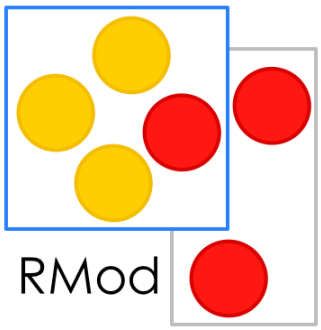
Final Project Report

Département Génie Informatique et Statistiques
Polytech Lille, Villeneuve d'Ascq

Language Transformation



Supported by : Vincent Blondeau	**Company name :** Equipe Rmod - INRIA
Tutor : Anne Etien	**Year :** 2013/2014

Contents

1 Introduction **3**

2 Why and how transforming code? **4**
2.1 Why transforming code? 4
2.2 Code Modeling 4
2.3 Transformation Existing tools 6
2.3.1 Transformation Framework 6
2.3.2 Transformation Language 7
2.4 The project 7

3 What is a transformation tool? **8**
3.1 Model navigation 8
3.2 Model filtering 9
3.3 Transformation rules 10
3.4 An other tool: The Model Displayer 13

4 Tasks and time scheduling **15**
4.1 Estimated schedule 15
4.2 Agile method 15
4.3 Realized tasks 17

5 Conclusion **18**

1 Introduction

In Polytech Lille, the final study project enables the students to work during almost three months on an innovative project. This project is composed of two parts: one of analysis and specification definition and the other based on conception, implementation and testing of the solution.

I had the opportunity to make this project in the Rmod research team. Set up at the Inria, it is specialized in code analysis and maintenance. The main used tool is a code analysis engine, called Moose. It allows to easily build a model representing the code from a source code.

The goal of this project is to add to Moose an API allowing to transform easily any Moose model into another one.

The second section explains why and how we can tranform code. In section 3, I will describe what is a code transformation tool and in section 4, I will present the different tasks I made in this project and the schedule.

If you want to try an to improve the tool, you can gofer it with:

Gofer it smalltalkhubUser: 'VincentBlondeau' project: 'FAST-Refactoring'; configuration; loadDevelopment.

on a fresh FAST-On-Moose image that you can download here :
https://ci.inria.fr/moose/job/FAST-On-Moose/lastSuccessfulBuild/artifact/FAST-On-Moose.zip.

2 Why and how transforming code?

In order to understand the context of this project, we are going to explain what is the goal of code transformation, how we can operate a such transformation and what are the main existing tools to do this.

2.1 Why transforming code?

We would transform code for many reasons. The first of these is to simplify the code. For example, your code uses a method that you want to rename, because the name doesn't match the behaviour, or you want to delete all the calls to an old method. Theses changes are code transformations.

Secondly, you can support the framework evolution. If your code is based on a library or external code that is evolving, you would like to follow the changes and consequently adapt you code.

Finally, you would translate your code from a language to another. For instance, you want to convert a Java program into a C++ or a Smalltalk one, it can be done by code transformation.

In the first two cases (simplify the code or support framework evolution), the transformations are refactorings. The code behaviour stays the same but the implementation differs. Moreover before and after the transformation, the model contains the same elements.

The langage translation is not a refactor but a heterogeneous transformation. It means that the source and target meta-models of the transformation (a meta-model is a model that describes an another model) are not the same. If we transform Java in Smalltalk, the Java language has not the same concepts that the Smalltalk one. So we have translated all the concepts included in one to the other.

For this report, we will take a short and simple transformation, the inline of the *ifTrue: message send*. It is the example I use to test the transformation tool I made.

Let take the following method named `return1`:

```
return1
 true ifTrue: [ ↑ 1 ]
```

It is a Smalltalk method containing an *ifTrue: message send* (`ifTrue: [...]`). The receiver of this statement is true so the block in the first argument will always be executed. Thus the method will always return the integer 1. So we want to transform it into:

```
return1
    ↑ 1
```

The *ifTrue: message send* is removed and the content of the block in first argument is moved in the body of the method because it is useless in the code. The transformation application is simplified if the code is not considered pure text but is abstracted by a model.

2.2 Code Modeling

The code under its initial form is only a string. It is difficult to easily and rapidly perform transformations on it because:

1. The string data structure is ambiguous: if a character is added or modified, you can't suppose easily that the changed code is always working.

2. The adding of new expressions is not easy. For this, the grammar has to be extended, and must stay coherent. Adding tokens and expressions in a grammar is difficult.

So in front of all these problems, we decided to transform the text into a model.

1. It is an efficient and powerful data structure: there are no ambiguity. Each element has properties and is linked to the others by links. While the data structure respects the meta-model of the language, which is very easy to check, the model will stay a code model.

2. Adding new expressions is easy. You just have to add a new type of element in the meta-model and the links to make it works.

3. The model can be created from the text, using a parser, and code results from printing the model.

So the model has great advantages on the text.

This model is called Abstract Syntax Tree (AST). It is a tree because in a programming language, all the instructions are included recursively in another until the root (which is not included).

This model is implemented in Moose by FAST (FAMIX AST). FAMIX is the Moose meta-model. FAST contains all the tools and the model elements to create AST from Java or Smalltalk and to print them.

Once created, we can use it to apply transformations.
The figure 1 shows the different steps between original and transformed source code.

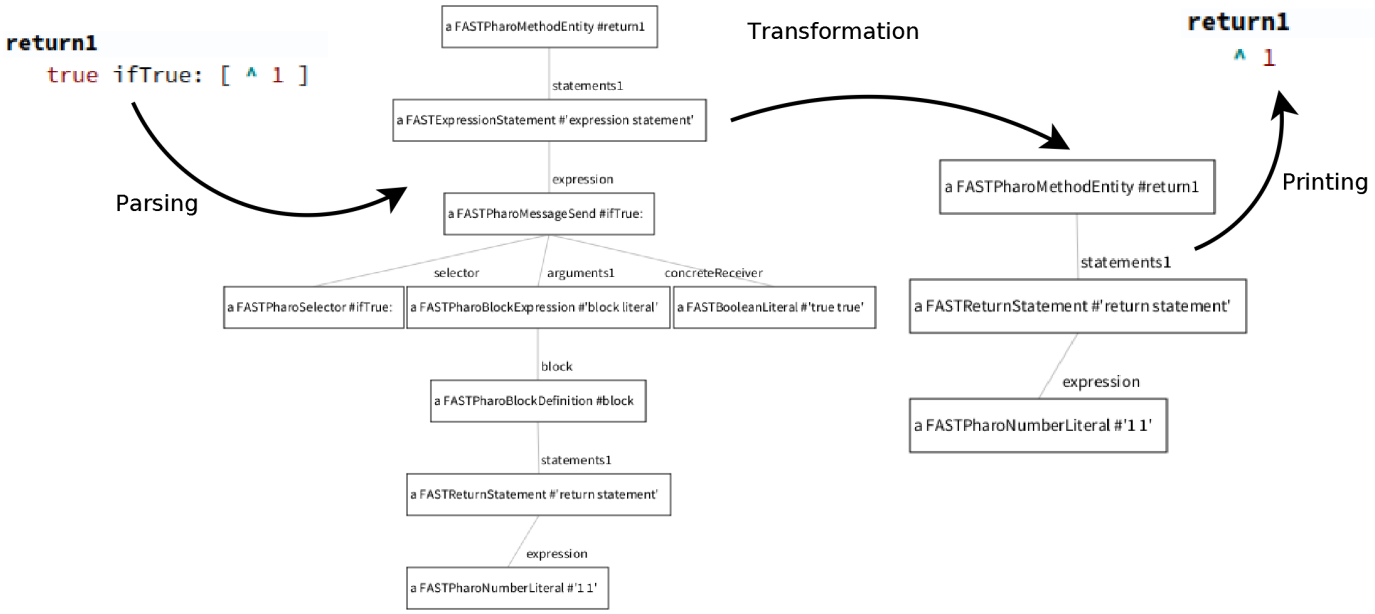


Figure 1: Code transformation steps

The first step is to parse the code to create a code model. The step is already implemented in Moose. The current parser supports both Java and Smalltalk languages.

Coming back to the example function, which is in Smalltalk, each element of the original function has a FAST model equivalent. `return1` is a method element.

true `ifTrue`: [...] is a *message send* element. Its selector is an *ifTrue: selector* receiver is is a boolean node with a value equals to *true*. It has an *argument* containing a block (Composed of two kind of nodes : *block expression* and *block definition*. This block contains a *return statmement* which is the model equivalent of the ↑1;

The *return statement* is linked to an expression which is *integer 1* equivalent to 1. In this way we have a tree describing the Smalltalk example code.

The second step is to transform the model. This part is not yet available on Moose and this is the purpose of the project.

In the third step, the transformed model is printed. This operation already exists in Moose. It consists to serialize the model into source code.

To summarize, the only thing missing is a tool transforming a model to another one. But before creating one from scratch, we will see what are the existing tools.

2.3 Transformation Existing tools

We studied two different tools. One coming from the Pharo platform, and the other from the Eclipse one.

2.3.1 Transformation Framework

In the Pharo platform, the Refactoring Browser (RB) is a framework enabling refactoring of applications. It allows to make some small code modifications that doesn't change the behaviour of the code. Renaming methods or variables, and adding some getters or setters are some examples. Theses modifications are very useful and common. Moreover the representation used by the Refactoring Browser is very similar to the FAST one. But this is not the same meta-model.

The central problem is the language dependency. The refactoring browser is adapted to the Pharo language model. The difficulty to adapt it to another language would be high.

Indeed, this framework uses specific patterns to express transformations. Theses patterns are written for Smalltalk and can't be translated to other languages.

For instance, with RB, we can use the pattern '`object size`. This is a pattern for searching all the elements in code whose the message `size` is sent. The search is an step in transformation that we will explain in refModel filtering.In the pattern, '`object` is a meta-variable representing anything in the code. This notion of meta-variable and message passing is specific to the Smalltalk language. It is not applicable to others langage models. Indeed in Smalltalk, all the operations are message sending. In Java, C++,... there are operations like the *if statement* or the *while statement* which are not message passings. Moreover as this tool is optimized for real time operations, the patterns are compressed. They use apostrophes to express the patterns like in the example.

Moreover RB relies on a string representation of the code and uses the position of the char to identify an element into the code model. To make a refactoring, RB build an AST on demand of the method, make the transformation based on text related information (char positionin the method), and print the code. RB even remembers the spaces and indents before the refactoring to restore them after.

To summarize the code is difficult to understand, to read and strongly linked to the text.

The Refactoring Browser can't be modified to express other language transformations. Because the Smalltalk is one of the more simple languages. Using another language will add other concepts that can't be understood by the Refactoring Browser because they are not presents in this tool. These concepts will compromise the optimized system on which it is built.

2.3.2 Transformation Language

The Object Management Group (OMG) standard for transformation is called QVTo (Query View Transformation operational). A tool using this standard has been implemented in the Eclipse platform.

It has the advantage to be a standard in the transformation community and it can express easily some complex transformations whatever the model to transform. Indeed, in this language, each transformation is divided in rules. Each rule describe on which kind of node and on which conditions the rule can be applied. A rule contains the operations to transform a part of the model and have the possibility to call other rules.

The main drawback of this language is the non interoperability between Moose and Eclipse platforms.

For transforming a model representing code, tools already exist. If we can't use them for our software, we can get from them some strategies and ideas.

The QVTo language brings some basics elements and concepts to make transformations. So we have to understand and translate them for a use in Moose.

2.4 The project

In the project, we have to implement and define a transformation tool with the following specifications:

- The tool can operate on every language separately (Java, Smalltalk,...)
- We should be able to express easily every transformation
- It is inspired by the QVTo standard.

The tool I propose allows to define specific transformations to each model element and to call other rules. Indeed, each transformation is defined by a set of rules, because one rule is not enough to describe a whole transformation.

Now that the project is defined, let's see how a transformation tool works and can be implemented.

3 What is a transformation tool?

Formally transformation is composed of 3 parts:

- navigation in the model
- model filtering
- model creation or modification by rules application

During this project, I made a Domain Specific Language (DSL) allowing to filter and to apply rules on a model. I also made an engine using the DSL to execute the transformation.

3.1 Model navigation

The model navigation enables to access model elements from others. As all the elements of a model are objects, we can access, thanks to the language reflection, all the properties of an object and its references to others objects.

Let's take our running example of the method `return1`. The figure 2 shows the method model. In the box with broken lines is the *ifTrue: message send* element.

Figure 2: Model Navigation

We assume this element is the current one. From this element, the parent element (dotted) can be accessed. It is the element containing all the method statements (in this case there is only one: the current element).

Similarly the children elements (in dots and traits) can be reached. They correspond to the *ifTrue: receiver* (true) and the *block statement* ([↑ 1]).

Recursively, by this two means, we can traverse a model to reach any element.

Concerning the navigation, I have unified the methods to access parents and children in the model. Thus based on polymorphism, any element of the model can be reached using the same methods whatever their type. This unity is needed to have a common behaviour on all the elements.

The DSL specified the function `parentNode` and `childrenNodes` to respectively access the parent and the children of a node.

3.2 Model filtering

Filtering enables to restrict a model to a collection according to a given criterion, i.e., talking in terms of tree, to select a subtree satisfying a given collection.

In the example, we want to collect only the *ifTrue: message send* whose receiver is true. i.e. the *ifTrue: message* linked by the `receiver` accessor to a boolean (whose value is *true*).
The figure 3 shows how the filtering works.

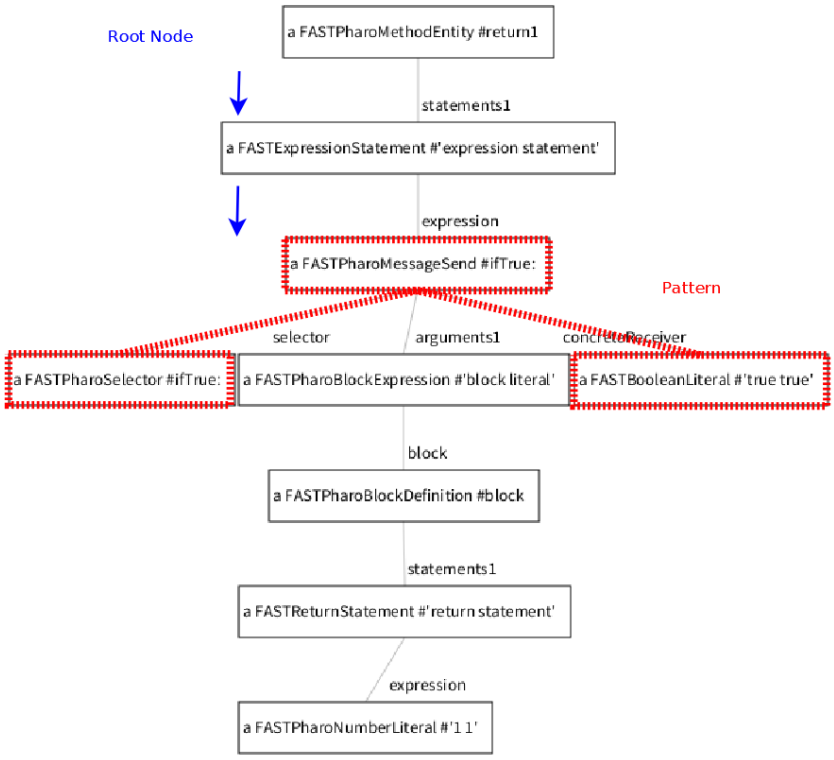


Figure 3: Model Filtering

The search starts from the root of the model (the method element) and cross all the elements by accessing their children until the pattern (dotted on the figure) is found. At this time, the *ifTrue: message send* is collected and the search continues in the sub-elements (the block elements i.e. the *block definition statements*).

Obviously, the model can be extended to a whole program containing a huge amount of methods. The root of the search will be not only a method but a collection of them. The result will contains all the matched elements.

I have implemented such a model filtering in my tool.
With the DSL, you can use:

```
find: <aTypeOfElement> in: <aNodeRoot> [where: <aBlock>]
```

This method enables you to filter any element of type `aTypeOfElement` in a model defined by his root (`aNodeRoot`), and where `aBlock` is verified. `aBlock` takes one argument which is one of the element of type `aTypeOfElement` in the model. The where argument is not mandatory.

For example, you can write:

```
self find: FASTPharoMessageSend in: return1
where: [:messageSend |
  messageSend selector name = #ifTrue: and: [
      a receiver value = true ] ]
```

This piece of code filters the `FASTPharoMessageSend` (FAST equivalent for a *message send*) in the method `return1` where the `messageSend` selector is `ifTrue:` and its receiver value is true. The search is only applied to the elements whose the kind is set in the `find:`.

3.3 Transformation rules

The third step consists in modifing the piece of model resulting from the filtering or to create new elements depending on the filtering.

On the running example, the figure 4 shows the two versions between before and after the transformation. The elements dotted are the *ifTrue: message send* previously searched on which we will apply the transformation rule. The transformation consists in moving the block statements to the method body and to delete the whole *ifTrue: message send*: the expression containing the message, the message, and the block given in the message argument. In broken lines, we can see that the argument block content is moved but not impacted by the move.

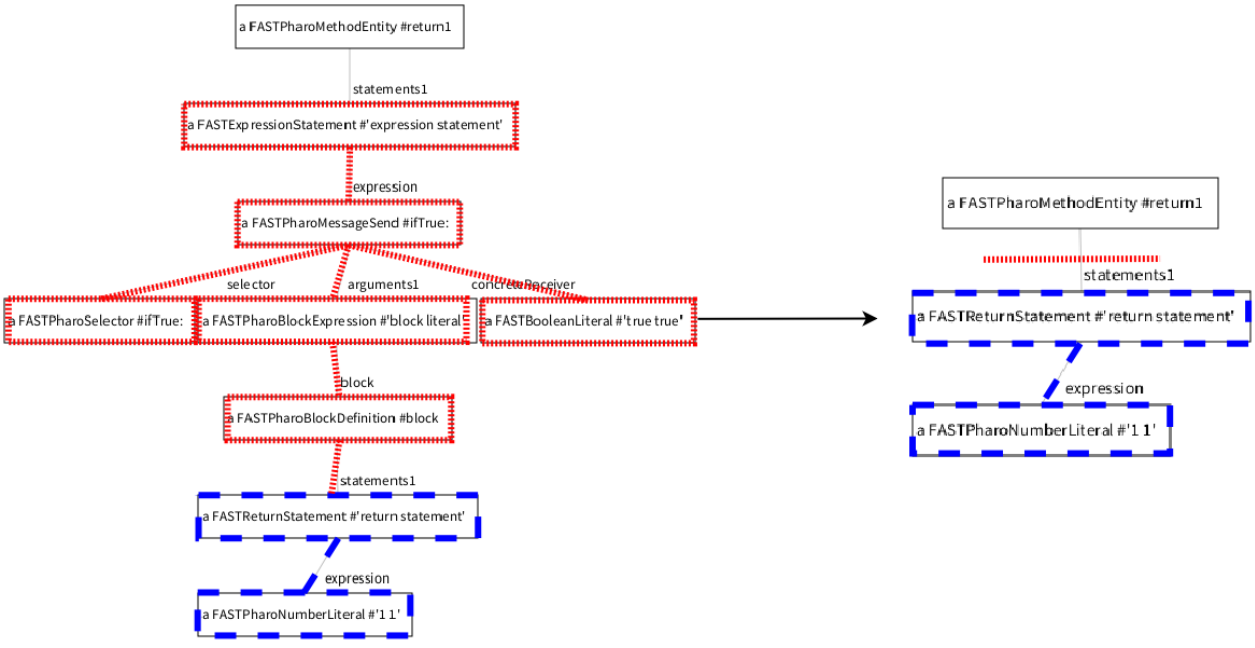


Figure 4: Transformation

The tool I propose defines a DSL divided in two parts: the rule application elements and the model elements modifiers.

On the side of the rule application, there are:

1. A meta-description of the transformation rules:

   The transformation rules are meta-described. The meta-description allows to specify on which kind of element the rule must be applied. It must be present to define a rule.

   This meta-description is translated in the Smalltalk language under Pragmas. This pragma:

```
<transformOn: <aTypeOfElement>
  [where: <aBooleanReturningSelector>] >
```

   applied to a method, says that the rule will be executed only on elements of type `aTypeOfElement` and for which the method `aBooleanReturningSelector`: will return true. The where argument is not mandatory and the associated method should take as argument the model element. It also allows to differentiate the transformation rules behaviour and we can define several rules for the same kind of element.

   On our example, we want to apply the transformation only if the receiver value is true. So the transformation can be meta-described by :

```
<transformOn: #FASTPharoMessageSend where: #isAnIfTrueStatement>
```

   The transform engine will execute the rule only if the kind of element is an FASTPharoMessageSend and if the FASTPharoMessageSend checks the condition defined in the method #`isConditionTrue`. This method checks only if the element is a boolean.

2. A method to call the others rules:

   Once the rules are defined, the user has to choose the application order and on which element it must be applied. So at any moment, only one rule is applicable. The default usage of this method is :

```
self apply: <aTransformationRule> on: <anElementOfTheModel>
```

   This method tries to call the rule `aTransformationRule`: taking in parameter a subtree model whose root is `anElementOfTheModel` and matching the rule pragmas described above. Indeed, the engine tries to recursively apply this rule to each children of the model given until there is no more child left.

   So coming back to the example, we can do :

```
self apply: removeIfTrueStatement on: return1
```

   to remove the if statement and to move its body of the if. Here we apply the transform rule named `removeIfTrueStatement` on the code model whose the root is `return1`.

In any transformation, you can use the filtering described in the previous subsection by two means:

- As the entry point of the transformation: you can use it in the `entryPoint` method. This method is the first rule executed by the transformation engine. In this method, you must call the first rule to be applied by using the `apply:on:` method. The `on:` argument can be the result of a filtering made in this method.
- As a query in a rule: in a rule, you can do a filtering on the node in argument of the rule.

The elements modifiers are some methods to modify easily the elements of the model. There are:

- `add: <anElement> after: <anOtherElement> in: <aSelectorReturningACollection>`: allowing to add `anElement` after `anOtherElement` in the collection returned by the selector `aSelectorReturningACollection`.
- `add: <anElement> insteadOf: <anOtherElement> in: <aSelectorReturningACollection>`: has the same behaviour than the previous one but replace the element instead of adding it.
- `addLast: <anElement> in: <aSelectorReturningACollection>`: Similar to the others but add at the end of the returned collection.
- `remove: <anElement> from: <aSelectorReturningACollection>`: Remove the element from the collection returned by the selector.
- `is: <aTypeOfElement>`: return true if and only if the element is a kind of `aTypeOfElement`.

To come back to the running example, we will show how the transformation rules can be written. If you want to try it, it is the `FASTERTransformTrueIfTrueBlock` class of the tool. This class contains all the rules to do the transformation.

The *ifTrue:* inlinement can be defined in two rules.

The first is a rule that is triggered when a *message send* (a `FASTPharoMessageSend` in the FAST model), is an *ifTrue:* statement. Indeed, this rule whose code is given below is executed only if the method `isAnIfTrueStatement` applied on a `FASTPharoMessageSend` returns true, as the meta description of the rule indicates.

The rule returns the block corresponding to the message send argument and apply the main rule (explained later) on the block.

```
getTrueIfTrueFrom: aNode
    <transformOn: #FASTPharoMessageSend where: #isAnIfTrueStatement>
    | block |
    block ← aNode arguments first block.
    self apply: #transformTrueIfTrueFrom on: block.
    ↑ block
```

This method `isAnIfTrueStatement` is used by the rule before to check if the node is an *ifTrue: message send*.

```
isAnIfTrueStatement: aNode
    ↑ aNode selector name = #ifTrue:
        and: [ (aNode receiver is: FASTBooleanLiteral) and: [ aNode ↩
           receiver value = true ] ]
```

The second rule (`transformTrueIfTrueFrom`) is the main rule. It is applied on the block level or method (in FAST, it is the *FASTBehaviouralEntity* element). In the `behaviouralEntities` variable, we get a collection containing the element returned by the first rule i.e. a block. In our example, there is only one *ifTrue: message send*. However in some cases, we can get several elements because of the the children crossing. That is why the `apply:on:` method always returns a collection.

On each block in `behaviouralEntities`, we get the *localVariables* and the *statements* (the contents of the block) and we add them to the current *FASTBehaviouralEntity*. By this way, we raise the block contents by one level.

```
transformTrueIfTrueFrom: aNode
    <transformOn: #FASTBehaviouralEntity>
    aNode statements copy
        do: [ :statementNode |
            | behaviouralEntities statements localVariables |
            "Get the data from a deaper entity"
            behaviouralEntities ← self
                apply: #getTrueIfTrueFrom on: statementNode.
            behaviouralEntities do: [ :aBehaviouralEntity |
                    localVariables ← aBehaviouralEntity localVariables.
                    statements ← aBehaviouralEntity statements.

                    "Affect the statements of the block to the upper ↩
                       BehaviouralEntity"
                    aNode
                        add: statements
                        insteadOf: statementNode
                        in: #statements.
                    "Move the variables of the block to the upper ↩
                       BehaviouralEntity"
                    aNode addLast: localVariables in: #localVariables ].
             ].
    ↑ aNode
```

Recursively, by applying this both rules, we manage to inline all the *true ifTrue: message send*.

Once your rules are implemented, you have to run the engine to apply them on a model. On an instance of the transformation class, you must use the `scope:` selector to set the model on which you want to apply the rule. Once the model is set, you can send the message `run` to your instance to run the transformation rules on your model.

By this way you can apply any set of transformation rules on any model.

3.4 An other tool: The Model Displayer

In this project, I needed to visualize the model on which we are doing the transformations. Indeed with only an object inspector, it is hard to materialize a model.

Thanks to the `childrenNodes` and `parentNodes` accessors, I created a model displayer using the Mondrian Framework of Roassal, a Moose tool to create visualisations. All the figures representing code model in this report are created by this displayer.

To display a model or a collection of models, the only thing to do is to send the method `display` on it. It will open a window in which we can see and navigate the model. You can

inspect an element or *browse* the class of the element by right clicking on this element.

During this project I designed and implemented a model transformation tool containing the three elements described before. In the next section, we will see how the work has been divided and scheduled.

4 Tasks and time scheduling

This project was divided into two parts. The first part during which I wrote the specifications, the second theoretically reserved for implementation, tests and documentation.

After I described the estimated schedule, I will expose the Agile methodology we used to perform tasks.

4.1 Estimated schedule

Initially, I planned to do the following tasks:

- Define the transformation description language.
- Realize the transform engine.

Theses two tasks are linked together. Because the language needs the engine to be executed and the engine can't do nothing if the language is not defined. So firstly, we have planned to specify the transformation language and as soon it is done to write the transformation engine.

4.2 Agile method

With the RMod team, we decided to use the Agile software development method. It is based on iterative and incremental development. The development is divided in cycles in which I developed a working application and I presented it to the team.

The figure 5 shows how the cycle is decomposed.

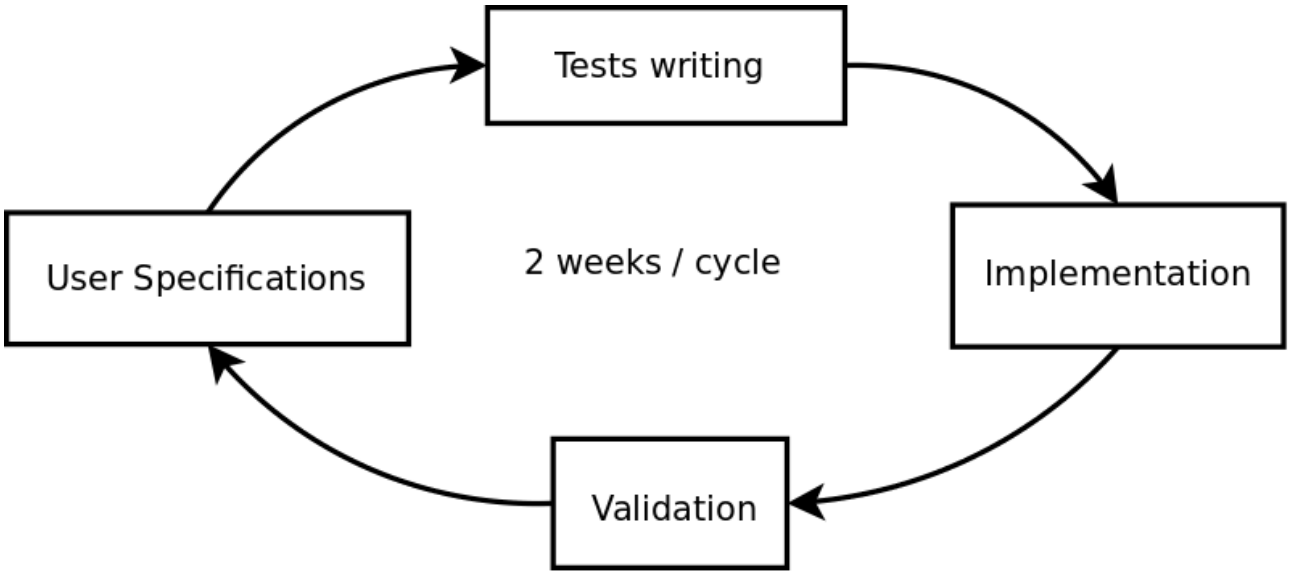


Figure 5: Agile Cycle

A cycle is composed of 4 steps:

1. User specifications:

   This step aims is to talk to the final users of the tool and convert their wishes to reality. We have to understand the wanted features and tell them if it's realizable or not. If not, we have to suggest them something else that can be easily understood.

   For the project, I have done few transformation examples, so we can discuss on it and improve it until find a good solution.

2. Tests writing:

   As soon as the specifications made, I started to write the tests defining the external behaviour of the engine. I invented some small examples like the Java example described in section 2 and 3 to have a more realistic tool.

3. Implementation:

   Once the tests are written, the transformation engine can be implemented. The only thing to do is to write the methods invloved in the tests in order to pass them all. During this process, it is possible we see that some features can not be implemented or hard to implement.

4. Validation and checking:

   In this step, we should assert that the piece of software made works fine. Thanks to tests written before, we can easily check the good behaviour of the program and represent the result expected by the user. If the tests are well done and pass, the application has some chances to work perfectly. Moreover the users have to find all the features they wanted. If all is good, the project is finished.

   But it is never like that. There are always a new feature to implement, one not correctly implemented, or impossible to implement. In theses cases, we must discuss with the final users to see what are the possibilities, and try to find a way to a good solution. In the project, some wished features were not implementable "as is" so I proposed others solutions to the users. We realize also that the transformation language was not easily understandable. So in theses cases, we do another iteration.

Each cycle had a duration of almost two weeks. That seems short but thanks of that, we are sure that the project is on the rails and follow a good direction. Inside the cycle I used to ask the RMod users their feelings of the use of the tool.

For this project, I did several iterations:

1. Create the tool:

   The first cycle of the project was dedicated to the language creation from the specifications and analysis made in the first part of the project. I did a tool transforming any model corresponding to the specifications. I also modified the original code model. Indeed, it was not ready to support the transformation. So I added some methods to modify easily elements and links between them.

2. Add cohesion between filtering and transformation parts:

   In the first version of the tool, filtering and creation parts were divided and almost separated. This step purpose was to agglomerate these essentials parts of the transformation. Indeed the transformation must be based on the result of the search.

3. Simplify the expression of the transformations for the user:

   Once all the parts of the transformation was melted, the transformation expression was hard to understand for the user.

   During the implementation part, some concepts has been modified or deleted and the name of the methods composing the language wasn't obvious when reading. For instance, to apply a transformation we used to use the $-->$ selector followed by the container name (a class) of all the transformations rules (which are methods). But the meaning of the arrow (interesting at the first cycle) was not very readable and the understanding was not

ensured. So we change this method to `apply:on`: described in section 3 which was more obvious.

4. Improve the meta-desciption of the rules :

   In the first rules, the meta-description was limited to the kind of element the rule should apply. But it was blocking for some transformations whose the transformation of the If Statement. So we added, like in QVTo, a `where`: argument to the meta-description to reduce the scope of the rule. Therefore we can write more specified rules.

However, it was a research project in Agile method. The method preconizes to do the step little by little, and not to develop the whole software in one block. In this case either we have all, or we have nothing.

4.3 Realized tasks

The goal of this project was to made a proof of concept on the implementation of a transformation tool in the Moose platform.

The tool I realized contains the three essential parts to any transformation language:

- Navigation between the model elements
- Filtering of the model. Actually the possibility to select any part of the model.
- Definition of the transformations rules and their applications.

It works on some basic examples like the deletion of the *ifTrue Statement* or the translation of a Java *if(true)... Statement* to a Smalltalk *ifTrue:[..] Statement*. But the possibilities of adding new rules are infinite. The filtering of patterns in the model works and should be enhanced to support multiple requests. With this kind of filtering, we will be able to have the elements contained in the If Statement as result of the search. The transformations will be easier to write.

However, as some unpredicted and unanticiped event are appeared, because I thought that the code model was a more developped project, I didn't write all the rules to transform Java source code to Smalltalk one. And there are no complex example of search or transformation. But it should be done in the next project.

5 Conclusion

Thanks of this project, I had the opportunity to work on a real research project. This project is the continuation of FAST which is the code model I use and with which we can parse, print and navigate the code model.

I worked with autonomy and improved my practice in Agile software development. I am now familiar with the practices of Moose, Pharo, Smalltalk and test driven development.

I also deepened my knowledge in Abstract Syntax Tree concept, a standard data structure to modeling code, and in code transformation which is an interesting project.

This project is the first stone of the model transformations with Moose. For now, Moose is only used for calculating metrics around the code model. But with this project, in a close future, Moose will be able to generate transformed and optimized code.

For that, we may add more complex transformation rules of model modification (for improvement) or creation (for translation).