Towards a Versatile Reverse Engineering Tool Suite

Mahugnon Honore Houekpetodji[1,3], Benoît Verhaeghe[2,3], Clotilde Toullec[3], Mustapha Derras[2], Fatiha Djareddir[1], Jérôme Sudich[1], Stéphane Ducasse[4], and Anne Etien[3]

[1] CIM, Lille, France – {firstname}.{lastname}@sa-cim.fr
[2] Berger-Levrault, Montpellier, France – {firstname}.{lastname}@berger-levrault.com
[3] Université de Lille, CNRS, Inria, Centrale Lille, UMR 9189 – CRIStAL, France
[4] Inria, Université de Lille, CNRS, Centrale Lille, UMR 9189 – CRIStAL, France
{firstname}.{lastname}@inria.fr

Abstract. Reverse engineering is central activity to sustain the effort of multi-decade large software systems. Reverse engineering often consists of the same activities: querying, navigating, abstracting. However, from previous experiments and the literature, we report that tools used for software reverse engineering projects are language-specific or case-oriented and so need to be redeveloped or extended for each case. This raises the questions of the versatility and extensibility of such tools. Our answer to such challenge is an open architecture accepting a set of generic but essential reverse engineering tools (powerful query engine, abstracting engine based on virtual entity creation, a browser builder and a scripting visualisation engine). In addition such generic tools are specialized and extensible via several families of meta-models. This article presents our first architecture and key elements. It reflects on our current work with partner companies on Postgresql, PowerBuilder, Java, GWT and VisualBasic/Access.

Keywords: Tools · Software Reverse-Engineering · Model-Driven Engineering · Architecture

1 Introduction

"Reverse engineering is the process of analyzing a subject system to identify the system's components and their relationships, and to create representations of the system in another form or at a higher level of abstraction."[5] Since projects have multiple decade lifetime with developers changing projects, reverse engineering is underestimated but a central activity [7]. Reverse engineering is not limited to some mainstream languages [3] but is universal with projects in binary reverse engineering up to database [6] or on GUI [17].

Reverse engineering objectives is mainly understanding the system at various levels of abstraction: it is based on activities that such code navigation, queries, abstracting from code (architecture, domain model), creating reports using visualisations or metrics engine. At the same time, literature shows that reverse engineering projects require specific meta-models and dedicated tools [3, 15].

This raises the challenge of the adaptation of existing tools. Indeed many tools have a specific meta-model or a too generic one. This forces developers to find solution to adapt such specific metamodels and the tools built on top [4]. In addition, there is often a need for a specific tool working on a given metamodel (*e.g.,* querying specific embedded procedure on Postgresql[6], representing specific GWT idioms to support their migration [17]). Therefore there is a need for a versatile and extensible language-independent tool suite: this extension can happen at multiple levels - meta model, but also tools themselves (*i.e.,* acting on the model).

Bellay and Gall [2] proposed a large list of criteria to compare reverse engineering tools. Still they do not cover metamodel, metamodel extension and tools extensibitily as a main focus. Govin et al [11] identified five requirements for reengineering. Since we currently focus on reverse-engineering, we take into account the two requirements related to the analysis of the existing system to create an independent reengineering tool suite. In collaboration with different IT companies, we add three new requirements. Each requirement corresponds to actions developers perform in reverse engineering context.

We designed and implemented a versatile and extensible language-independent tool suite that fulfils the preceding requirements. To do so, we developed a language-independent meta-model that can be extended. Then, we used and developed core tools on top of the meta-model to help for the conception of new reverse engineering tools. Finally, we designed an architecture that eases communication between elements of the tool suite and developed first language-independent tools based on the core tools.

The rest of this paper is structured as follow: Section Section 2 presents the requirements inherent to reverse engineering activities. In Section 3, we detail the architecture of our tool suite. In Section 4, we present our tool suite. In Section 5, we present related work. Finally, Section 6, we conclude and present future work.

2 Reverse engineering requirements

Bellay and Gall [2] proposes a detailed list of criteria to evaluate reverse engineering tools: Analysis (supported source types, parser functionality, parsing funtionality), Representation (reports texutal, graphical, general report properties), Editing and Browsing (search, hypertext, General capabilities (supporting platform, multiuser, storing, tool extensibility, output...). While they provide a large list they do not cover the modelling aspects of the reverse engineering process: what entities and relations are manipulated beside mere source code, and whether new entities (such as domain-specific ones) can enrich the code metamodel.

Kienle and Müller [13] explore the issue of building reverse engineering tools. They propose three lenses (1) requirements, (2) construction and (3) evaluation.

[14]

In collaboration with different IT companies, we work on the reengineering of several software systems. For example, we work on the cleaning of a 3 MLOC PowerBuilder system where bad smells as code duplication, God classes, or code bloat can be found. For another industrial partner, we migrate the GUI of 8 GWT applications to Angular [17], each representing more than 500 web pages. Indeed, the framework written in Java and developed by Google is not maintained anymore and no new released came out since 2015. Developers and users have to use old browser versions. For the database architect of our university lab, we provide tools to refactor the database and the embedded programs written in PostgreSQL [6]. These projects involve large industrial software systems. Each of these projects implies to analyze the existing software system and to query it to highlight different issues depending on the purpose of the project.

Additionally, the literature reports other specific MDE approaches. For example, Candel et al. [4] migrated PL/SQL triggers to Java. To do so, they had to make language-independant a preexisting Java meta-model.

According to Bruneliere Bruneliere et al. [3], the plurality of reengineering projects requires an *adaptable/portable* reverse engineering tools. If a tool is too specific to a technology, its advantages are lost when working with other technologies.

Our objective is to develop a versatile and extensible language-independent reverse engineering tool suite.

Indeed, such a tool suite would ease the conception of new reengineering tools and would enable each project to benefit from the improvement of others.

The tool suite needs to fulfil requirements that ensure its adaptability to any reengineering approach. In this paper, we focus only on the reverse-engineering step.

Govin [11] identified five requirements to support the rearchitecturing of embedding systems: *Modelization*, *Allocation*, *Selection*, *Abstraction*, and *Verification*. As a first step, we only consider requirements that focus on the reverse-engineering part. Consequently, we keep only *Selection* and *Abstraction*.

Modelization Creating new elements, as well as the relations between them using an abstract representation.

Selection Selecting elements based on their characteristics:
- Physical folder organization.
- Lexical pattern in source code.
- Contained elements: define a type *e.g.,* class or method, and identify all elements containing elements of this type.
- Associations: from a chosen element, selecting others following a given type of association.
- Multiplicity of allocations: selecting elements that have been allocated on several architecture elements.

Abstraction Gathering information in an upper containment level[5]. It concerns:

[5] Considering object-oriented languages, packages contain classes that contain methods and attributes... Packages are at an upper level than classes that at their turn are at an upper level than methods if we do not consider inner classes and so on

- Relations: representing the relationships between the containers whereas they are defined between contained elements.
- Metrics: aggregation of metrics computed for elements to the level of their container. For example, the number of lines of code of a package is the sum of the number of lines of code of its components.

In addition, based on our experience and the needs expressed by our industrial partners, we add the following requirements:

Duplication Emphasizing code duplication. A duplication is considered when it concerns at least a user-specified number of duplicate lines.
Source text Accessing the source code of a code element.
Navigation Navigating elements following relations between them.

3 Tool suite core

To answer the requirements detailed in Section 2, we developed a set of tools. Each tool is based on the core elements of the Moose platform [8], that we also contribute to develop. Indeed, the platform offers one generic meta-model and three main tools for software system analysis: Famix (Section 3.1), Moose Query (Section 3.2), Tags (Section 3.3) and Roassal (Section 3.4).

3.1 Famix

Famix is a meta-model that allows developers to represent programs. By default, it gathers elements of the language grammar such as function, module, class or method. However, it also enables users to define more abstract concepts, *e.g.,* widget or layout in the context of GUI representation [17]. Relations can be defined between model elements. These relations represent containment and associations (invocations, references, *etc.*) between a source and a target. Figure 1 gives an example of relations. A Famix abstraction of this program includes containment relations between each package and the class it contains, and between each class and its method. It also includes an invocation association (represented by the blue arrow) between *methodB* as source and *methodA* as target. Finally, a lot of concepts already exist but Famix extensibility enables the developer to define new languages if they are not already covered.

Moose tools are based on the Famix meta-model. Thus, any model created using Famix can be analyzed with the Moose tools.

3.2 Moose Query

Moose Query is an API to navigate Famix models through queries. Navigation is based on Famix relations of containment and association. It is thus possible to navigate from an element (for example *ClassA* in Figure 1), to its container (*PackageA*) or to the elements it contains (*methodA*). Moreover, from an element,

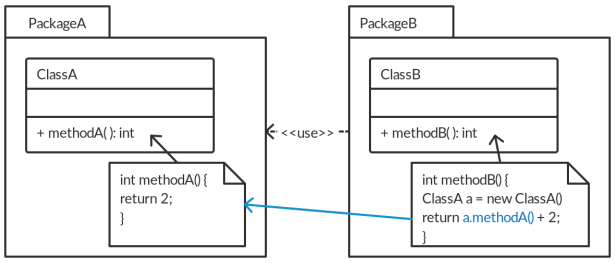


Fig. 1. Example of Famix relations.

it is possible to navigate through a specific or all types of associations to other elements. If the initial element is the source of an association, it is possible to reach the target of the association or the opposite. Finally, it is also possible to define the containment scope of the resulting element, for example, to abstract relations at a higher containment level.

Navigation queries can be composed, allowing users to express precise and more complex queries. For example, in Figure 1, the invocation between the two packages would be obtained with the composition of three parts: a containment query on the packages to get their methods, an association query on the methods to get the invocation and finally, a precision of package as the resulting scope.

3.3 Tags

Tags are labels used to enrich the source code with information that cannot be directly inferred from the source code. Govin et al. [12] presented tags systems as helping tools for software rearchitecting project. As with any other elements, it is possible to navigate tags using Moose Query.

Our tags system allows users to *(i)* map abstract concepts to the source code, *(ii)* navigate a logical view of the system, *(iii)* create a high-level view and analyze it to check code conformance, and *(iv)* compare competing views of the same system.

3.4 Roassal

Roassal is the visualization tool integrated into Moose. Visualizations represent elements of a program as nodes and the associations between them as edges. Users can configure visualizations to display the information they need. For example, a user may choose to represent a program as a class hierarchy with classes as nodes and inheritance as edges. The size of the nodes can be configured to represent the number of lines of code in each class. Roassal can also be used with Moose Query to represent more complex relations.

4 Browsers

Additionally to the core tools, we built browsers that offer a UI to visualize and analyze software systems.

4.1 Browsers architecture

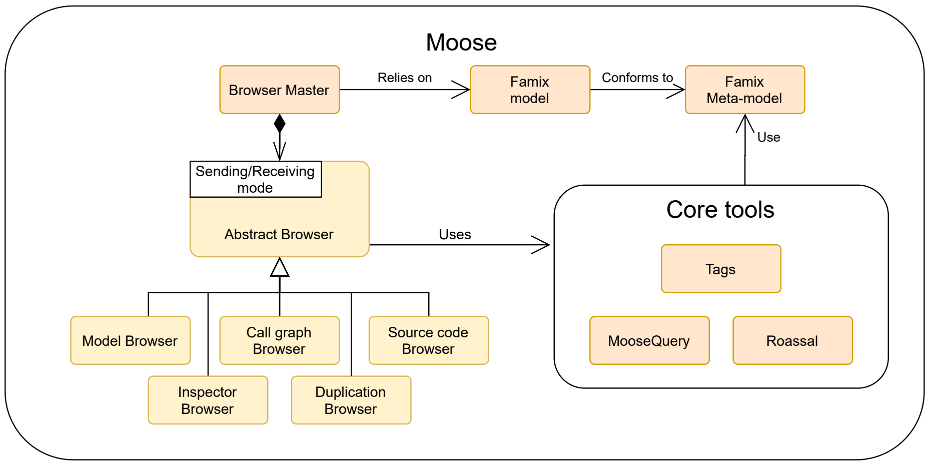


Fig. 2. Tool suite architecture

The overall architecture (Figure 2) of our software reverse-engineering tool suite is built in the Moose platform. The tool suite is mainly composed of a BrowserMaster and browsers. It runs on an instance of a Famix model.

The BrowserMaster is responsible for information exchange between browsers. It knows all opened browsers. Each browser handles **focus entities** that can be a single entity or a group of entities. When an event occurs in one browser, the BrowserMaster notifies all the other opened browsers about it according to propagation (outgoing) and reception (incoming) strategies. Browsers receiving the notification change their focus entities to the ones propagated by the BrowserMaster according their reception strategy.

The *do not propagate* mode does nothing. In the *propagate* mode, the browser notifies the BrowserMaster of the selection. The BrowserMaster publishes the selected entities among all opened browsers. Each browser then reacts according to its reception strategy. Except for the Inspector Browser, all browsers use the *propagate* mode by default.

The reception strategy of a browser defines how the latter behaves when it receives a selection: In the *follow* mode, the browser updates its focus entities to the received selection and refreshes. In the *highlight* mode, the browser searches for the received selection in its focus entities and highlights it if found. Finally, in the *ignore* mode, the browser does nothing.

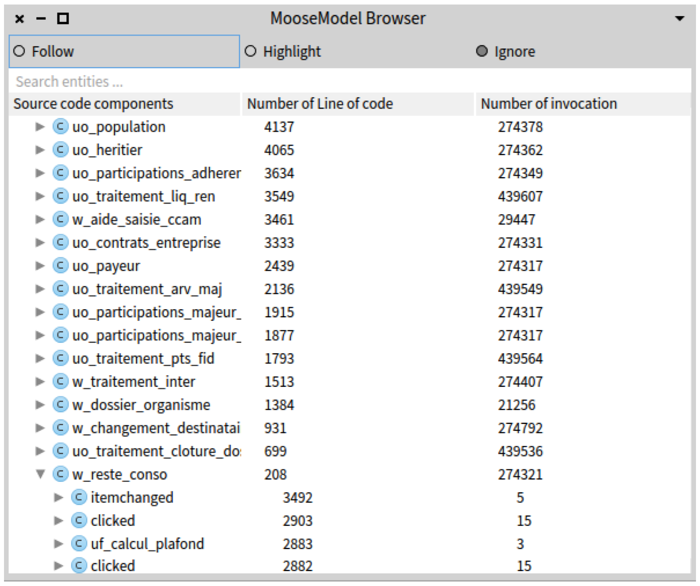
These different modes can be set up on each browser through radio buttons in a header and changed according the user's needs.

4.2 Model browser

The model browser provides navigation through a Famix model. It is composed of a header, a search field, and a content as shown in Figure 3.

The search field of the browser allows the user to write a search query on the focus entities. This search query can be lexical, as a simple string for a lexical pattern search, or structural.

For instance, if the user searches *include: FamixPWBAttribute*, the result will be all entities containing Powerbuilder attributes. The content of the model



Source code components	Number of Line of code	Number of invocation
▶ uo_population	4137	274378
▶ uo_heritier	4065	274362
▶ uo_participations_adherer	3634	274349
▶ uo_traitement_liq_ren	3549	439607
▶ w_aide_saisie_ccam	3461	29447
▶ uo_contrats_entreprise	3333	274331
▶ uo_payeur	2439	274317
▶ uo_traitement_arv_maj	2136	439549
▶ uo_participations_majeur_	1915	274317
▶ uo_participations_majeur_	1877	274317
▶ uo_traitement_pts_fid	1793	439564
▶ w_traitement_inter	1513	274407
▶ w_dossier_organisme	1384	21256
▶ w_changement_destinatai	931	274792
▶ uo_traitement_cloture_do	699	439536
▼ w_reste_conso	208	274321
▶ itemchanged	3492	5
▶ clicked	2903	15
▶ uf_calcul_plafond	2883	3
▶ clicked	2882	15

Fig. 3. Model browser with a PowerBuilder source code classes (PowerBuilder object)

browser mainly presents entities satisfying the search criterion (**Selection** requirement) in a tree (first column on Figure 3) where parent relationship corresponds to containment. A node of the tree is an entity and its node children correspond to its contained entities. If this entity contains others, it is possible to expend the corresponding node to see them. For example, when you expend a class node (*e.g.,* w_rest_conso), you see attributes and methods (*e.g.,* itemchanged, clicked).

The attributes nodes are leaves, and methods nodes are containers and so can be expended. When a method node is expended one can see variables nodes.

The content of the model browser can be adapted to specific needs. In Figure 3 we adapted the browser for one of our industrial partner by adding two extra columns. The second column shows the number of lines of code of the entities. Finally, the third column presents the number of invocations of each entity (**Abstration** requirement). For instance, the number of invocations of a class is the aggregation of the number of invocations of its methods.

fulfillment: The model browser satisfies **Selection** and **Abstraction** of metric requirements.

4.3 Call graph browser

In Figure 4, the call graph browser provides a visualization of the model as a call graph. This graph can eventually be coloured according to tags (see Section 3.3). It is mainly composed of a visualisation content.

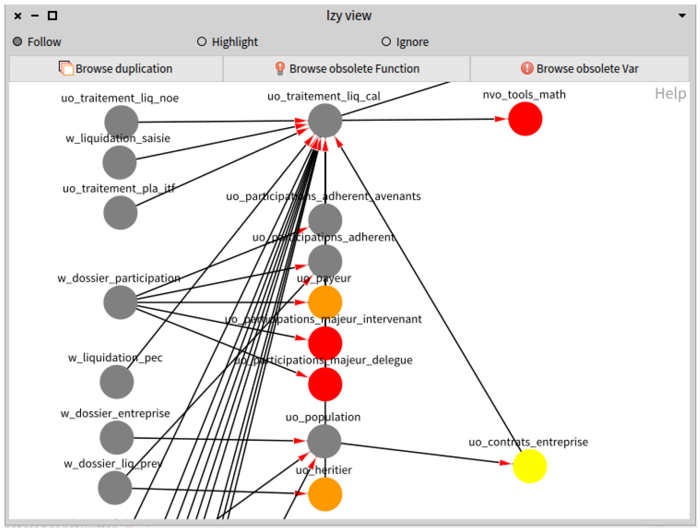


Fig. 4. Call graph browser

The content of the browser is an oriented graph. Each node of the graph is a circle, which, for example, represents a class. Considering two classes *C1* and *C2* if there exist invocations between methods of *C1* and methods of *C2*, there is an edge between the nodes representing these two classes. The browser shows only classes (i) whose methods invoke directly or not the focus entities of the browser or (ii) whose methods are invoked directly or not by the focus entities (**Linked**

elements selection requirement). The indirection level can be modified. For example, Figure 4 shows classes linked to *uo_traitement_liq_cal*.

It is possible to add information corresponding to tags previously put on elements through colours. By default, nodes do not have tags. Thus there are grey. For instance, the developers of our industrial partner company decided to enrich the visualization by tagging some classes as:

– isDeprecated: from developers knowledge of the system, these classes belong to useless modules (in red in Figure 4). Thus, for example, *uo_payeur*, *uo_participations_majeur_intervenant*, and *uo_participations_majeur_delegue* are tagged as isDeprecated.
– toRefactor: these classes appear to have invocations to other classes whereas according to the developers some of them are henceforth useless. These classes, in orange in the figure, have to be updated.
– notSure: For these classes represented in yellow, developers needed further investigation. In Figure 4 *uo_contrats_entre_prise* is tagged as not sure.

fulfillment: This browser fulfills the **Linked elements selection** requirement.

4.4 Duplication browser

The duplication browser shows duplication between the browser focus entities methods (**Duplication** requirement). This browser represents clone fragments in a way that the user can easily inspect them. For this purpose, it uses a string matching algorithm [9] to detect duplication of level 1. This algorithm can be changed to a more sophisticated algorithm to detect duplication [16].

There are two different ways to visualize the results of duplication detection: clone fragments by entities and entities by clones fragments.

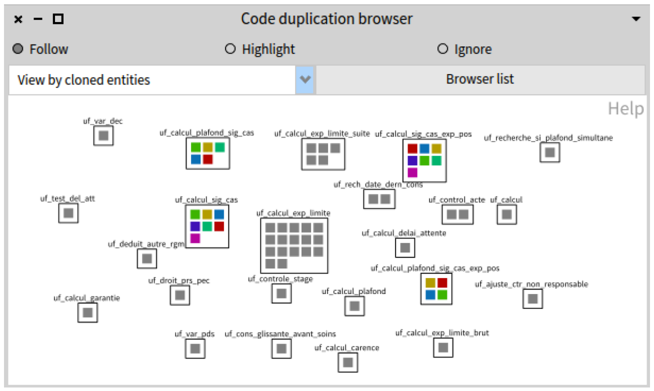


Fig. 5. Duplication browser

Visualize clone fragments by entities

Illustrated in Figure 5, this perspective represents entities containing clone fragments (**Abstraction** requirement). Methods are represented with a rectangle. The clone fragments in the methods are represented by small squares inside the rectangle. By default, inside a method, all squares representing clone fragments are grey. The user can select a method to see which entities share clones with it. Selecting a method colours its clone fragments, with a different colour for each different clone fragment. Coloured clone fragments present in other methods are highlighted with the same colour. Thus all squares of a given colour correspond to different clones of the same fragment of code.

Visualize entities by clone fragments

This visualization groups entities by clone fragment. A source code fragment is represented by a rectangle. All entities in which the fragment exist are represented by a small square inside this rectangle. A similar colour mechanism indicates where a given fragment is spread. This view gives a quick focus on methods having at least one clone fragment in common.

Figure 5 presents a duplication browser where clones are grouped by methods. *uf_calcul_plafond_sig_cas_exp_pos* shares clone fragments with *uf_calcul_plafond_sig_cas*, *uf_calcul_sig_cas* and *uf_calcul_sig_cas_exp_pos*. Nevertheless, these three methods have more fragments that they share together or not.

fulfillment: The duplication browser satisfies the **Duplication** and the **Abstraction** of metric requirements.

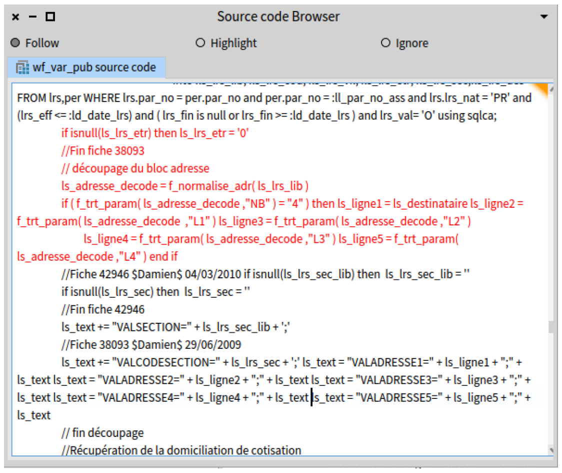
4.5 Source code browser

The source code browser displays source code as text. Usually, a text is written in black. But the source code of clone fragments is displayed in red. Figure 6 depicts the source code browser in the case of a method with clones.

fulfillment: The source code browser fulfils the **Source code visualization** and the **Duplication** requirements.

4.6 Inspector browser

The inspector browser enables an endless navigation between model entities. By default, it is composed of a single panel representing a list of entities. This list correspond either to a selection resulting from another browser or to the whole model. A new panel opens on the right when the user selects an element of the list. If this element is a collection, it is presented as a list. Then the user can select another element, opening a new panel on the right and so on. If the selected element corresponds to a single entity, the panel is composed of four subbrowsers:



```
FROM lrs,per WHERE lrs.par_no = per.par_no and per.par_no = :ll_par_no_ass and lrs.lrs_nat = 'PR' and
(lrs_eff <= :ld_date_lrs) and ( lrs_fin is null or lrs_fin >= :ld_date_lrs ) and lrs_val= 'O' using sqlca;
        if isnull(ls_lrs_etr) then ls_lrs_etr = '0'
        //Fin fiche 38093
        // découpage du bloc adresse
        ls_adresse_decode = f_normalise_adr( ls_lrs_lib )
        if ( f_trt_param( ls_adresse_decode ,"NB" ) = "4" ) then ls_ligne1 = ls_destinataire ls_ligne2 =
f_trt_param( ls_adresse_decode  ,"L1" ) ls_ligne3 = f_trt_param( ls_adresse_decode ,"L2" )
            ls_ligne4 = f_trt_param( ls_adresse_decode ,"L3" ) ls_ligne5 = f_trt_param(
ls_adresse_decode ,"L4" ) end if
        //Fiche 42946 $Damien$ 04/03/2010 if isnull(ls_lrs_sec_lib) then  ls_lrs_sec_lib = ''
        if isnull(ls_lrs_sec) then  ls_lrs_sec = ''
        //Fin fiche 42946
        ls_text += "VALSECTION=" + ls_lrs_sec_lib + ';'
        //Fiche 38093 $Damien$ 29/06/2009
        ls_text += "VALCODESECTION=" + ls_lrs_sec + ';' ls_text = "VALADRESSE1=" + ls_ligne1 + ";" +
ls_text ls_text = "VALADRESSE2=" + ls_ligne2 + ";" + ls_text ls_text = "VALADRESSE3=" + ls_ligne3 + ";" +
ls_text ls_text = "VALADRESSE4=" + ls_ligne4 + ";" + ls_text ls_text = "VALADRESSE5=" + ls_ligne5 + ";" +
ls_text
        // fin découpage
        //Récupération de la domiciliation de cotisation
```

Fig. 6. Source code browser

the navigation browser, the tree browser, the fame browser and the property browser.

The *navigation browser* enables users to navigate to all entities linked to the selected one whatever the type of links. For a class, it is possible to access its contained entities (*i.e.*, methods or attributes) or its subclasses or the methods referring it…

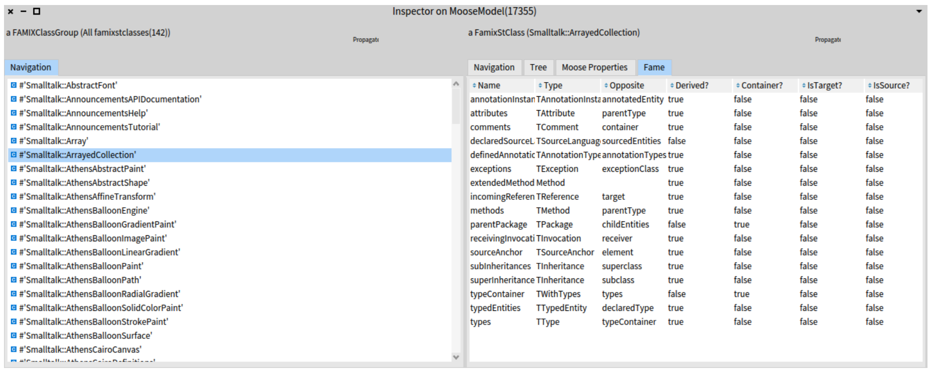
The *tree browser* presents recursively the contained entities of the selected entity as a tree, similarly to the model browser.

The *fame browser* displays the meta-information about the selected entity, such as its properties name and their types (Boolean, Integer, *etc.*). It also displays more information relative to Famix such as the containment and the relations.

The *property browser* displays the value of the entity properties for example for a class: the number of lines of code or the number of methods. Any element in these subbrowsers can be selected, opening a new panel on the right. Only two panels are simultaneously displayed but it is possible to move from one to the other. Additionally to those features, the inspector browser comes with a *propagate* button. Clicking on it manually activates the *propagate* mode and so notifies the BrowserMaster of the selection.

Figure 7 shows the inspector browser focused on the ArrayCollection Pharo class. The left-hand panel displays all the classes present in the inspected model. The right-hand panel displays information about the ArrayedCollection entity. In particular, the inspector is opened on the fame browser.

fulfillment: The inspector browser fulfils the **Linked elements selection** and the **Abstraction** requirements.



a FAMIXClassGroup (All famixstclasses(142))

```
#'Smalltalk::AbstractFont'
#'Smalltalk::AnnouncementsAPIDocumentation'
#'Smalltalk::AnnouncementsHelp'
#'Smalltalk::AnnouncementsTutorial'
#'Smalltalk::Array'
#'Smalltalk::ArrayedCollection'
#'Smalltalk::AthensAbstractPaint'
#'Smalltalk::AthensAbstractShape'
#'Smalltalk::AthensAffineTransform'
#'Smalltalk::AthensBalloonEngine'
#'Smalltalk::AthensBalloonGradientPaint'
#'Smalltalk::AthensBalloonImagePaint'
#'Smalltalk::AthensBalloonLinearGradient'
#'Smalltalk::AthensBalloonPaint'
#'Smalltalk::AthensBalloonPath'
#'Smalltalk::AthensBalloonRadialGradient'
#'Smalltalk::AthensBalloonSolidColorPaint'
#'Smalltalk::AthensBalloonStrokePaint'
#'Smalltalk::AthensBalloonSurface'
#'Smalltalk::AthensCairoCanvas'
```

a FamixStClass (Smalltalk::ArrayedCollection)

Name	Type	Opposite	Derived?	Container?	IsTarget?	IsSource?
annotationInstan	TAnnotationInsta	annotatedEntity	true	false	false	false
attributes	TAttribute	parentType	true	false	false	false
comments	TComment	container	true	false	false	false
declaredSourceL	TSourceLanguag	sourcedEntities	false	false	false	false
definedAnnotatic	TAnnotationType	annotationTypes	true	false	false	false
exceptions	TException	exceptionClass	true	false	false	false
extendedMethod	Method		true	false	false	false
incomingReferen	TReference	target	true	false	false	false
methods	TMethod	parentType	true	false	false	false
parentPackage	TPackage	childEntities	false	true	false	false
receivingInvocati	TInvocation	receiver	true	false	false	false
sourceAnchor	TSourceAnchor	element	true	false	false	false
subInheritances	TInheritance	superclass	true	false	false	false
superInheritance	TInheritance	subclass	true	false	false	false
typeContainer	TWithTypes	types	false	true	false	false
typedEntities	TTypedEntity	declaredType	true	false	false	false
types	TType	typeContainer	true	false	false	false

Fig. 7. Inspector browser

4.7 Using Simultaneously Several Browsers

Using several browsers linked through the BrowserMaster is very useful in different contexts, for example to visualize and compare clone fragments. First, the developers open a duplication browser. Then they open a source code browser and set its reception browser strategy to *Follow*. In the duplication browser, they can select a clone. The source code of the selected clone is directly displayed in the opened source code browser. If this entity is relevant, they switch the browser reception strategy of the opened source code browser to *Ignore* to block the browser on this entity, whatever occurs in the other browsers. To compare this code fragment with another, the developers open a second source code browser and keep the browser reception strategy to *Follow*. Then they select the second duplication in the duplication browser. The code of the entity appears in the second source code browser. It is now possible to see the two code fragments. It is not necessary to open a source code browser for each occurrence of the clone. The strategy enables to easily go through the code of all the methods containing a given clone fragment.

4.8 Browsers and Requirements

Each browser is designed to respect software reengineering requirements defined in the Section 2. Table 1 summarizes these requirements.

Only the physical folders organization selection requirement is not currently satisfied by one of the developed browsers. Since the information can be stored in the Famix metamodel, it is part of our future work. Moreover, all the browsers rely (i) on the associations or the containment relationships between elements, (ii) on name of elements, (iii) on properties or (iv) on source code and eventually duplication. Those abstractions exist in all languages. Our browsers are thus completely generic.

Browsers	Selection	Abstraction	Duplication	Source code visualization	Navigation
Model	Lexical Structural	Metric	No	No	Yes
Call graph	Linked elements	Relationship	No	No	No
Duplication	Duplication	No	Yes	No	No
Source code	Source code	No	Yes	Yes	No
Inspector	Linked elements	Metric Relationship	No	No	Yes

Table 1. Browsers with satisfied requirements

5 Related work

In the following, we present already published works dealing with software reverse engineering tool suites.

Bruneliere et al. [3] proposed and implemented the MoDisco framework. MoDisco is divided into 4 layers, eclipse modelling project, infrastructure, technology, and use cases. The infrastructure layer contains generic tools such as the one to navigate and query models. The technology layer contains the specific metamodel, *i.e.,* Java, XML, JSP. And the use cases layer contains specific action for a model, *i.e.,* Java code refactoring.

Their layers are similar to ours, the main differences concern the available tools. To ease the conception of new tools, we added Roassal as an integrated visualisation framework.

Ebert et al. [10] proposed an integrated language-independent workbench to support program understanding named GUPRO. It is based on a graph representation of languages with a graph query engine. Additionally, they proposed different generic tools to understand software systems. In particular, they offer a code view and a table view. The table view enables the users to select an element and then to start a query in the code view from it. The code view enables the users to create a graph query and to see the result in a table view.

Their tool suite does not integrate a visualisation tool. Moreover, users have to learn their query language. This might be an obstacle to framework adoption.

Arcelli et al. [1] developed an eclipse plugin with a generic meta-model. They created multiple views on top of the meta-model tool such as system complexity, class diagrams, and class blueprint. All those views can be used for any languages accepted by the meta-model. Finally, this meta-model can be extended to compute specific metrics for different languages.

Unlike our tool, their generic meta-model includes only one way to describe the relationship between two elements. This constraint leads to less precise queries because users cannot select directly only some kinds of relations, *e.g.,* they can select all relations and not only the invocations of a method by another.

6 Conclusion and future work

Reverse-engineering is the analysis of a software system and is an essential step of reengineering. The tools needed to perform reverse-engineering are similar whatever the language regardless of the perspective used for software analysis. We identified from literature and the experience of our industrial partners five requirements that such tools should fulfill: *Selection*, *Abstraction*, *Duplication*, *Source code visualization* and *Navigation*. In this paper, we presented a generic tool suite for reverse-engineering. This suite is language-independent and can thus be used for the analysis of different software systems. Our tools interact together, providing the user with an environment that fulfills the identified reverse-engineering requirements.

In the future, we will complete our tool suite to respond to other aspects of reengineering. We will develop generic tools to help modification of software systems. Those tools will have to meet requirements related to this step of reengineering.

Bibliography

[1] Francesca Arcelli, Christian Tosi, Marco Zanoni, and Stefano Maggioni. The marple project: A tool for design pattern detection and software architecture reconstruction. In 1st International Workshop on Academic Software Development Tools and Techniques (WASDeTT-1), pages 325–334, 2008.

[2] Berndt Bellay and Harald Gall. An evaluation of reverse engineering tools. Journal of Software Maintenance: Research and Practice, 1998.

[3] Hugo Bruneliere, Jordi Cabot, Grégoire Dupé, and Frédéric Madiot. Modisco: A model driven reverse engineering framework. Information and Software Technology, 56(8):1012–1032, 2014.

[4] Carlos Javier Fernández Candel, Jesús García Molina, Francisco Javier Bermúdez Ruiz, Jose Ramón Hoyos Barceló, Diego Sevilla Ruiz, and Benito José Cuesta Viera. Developing a model-driven reengineering approach for migrating pl/sql triggers to java: A practical experience. Journal of Systems and Software, 151: 38–64, 2019.

[5] Elliot Chikofsky and James Cross II. Reverse engineering and design recovery: A taxonomy. IEEE Software, 7(1):13–17, January 1990. https://doi.org/10.1109/52.43044. URL http://dx.doi.org/10.1109/52.43044.

[6] Julien Delplanque, Anne Etien, Nicolas Anquetil, and Stéphane Ducasse. Recommendations for evolving relational databases. In 32nd International Conference on Advanced Information Systems Engineering, 2020.

[7] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. Object-Oriented Reengineering Patterns. Morgan Kaufmann, 2002. ISBN 1-55860-639-4.

[8] Stéphane Ducasse. Reengineering object-oriented applications. Technical report, Université Pierre et Marie Curie (Paris 6), September 2001. TR University of Bern, Institute of Computer Science and Applied Mathematics — iam-03-008.

[9] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. In Hongji Yang and Lee White, editors, Proceedings of 15th IEEE International Conference on Software Maintenance (ICSM'99), pages 109–118. IEEE Computer Society, September 1999. https://doi.org/10.1109/ICSM.1999.792593.

[10] Jürgen Ebert, Bernt Kullbach, Volker Riediger, and Andreas Winter. GUPRO — generic understanding of programs, an overview. Fachberichte Informatik 7–2002, Universität Koblenz-Landau, 2002. URL http://www.uni-koblenz.de/fb4/publikationen/gelbereihe/RR-7-2002.pdf.

[11] Brice Govin. Support à la rénovation d'une architecture logicielle patrimoniale : Un cas réel chez Thales Air Systems. PhD thesis, Université de Lille, June 2018.

[12] Brice Govin, Nicolas Anquetil, Anne Etien, Stéphane Ducasse, and Arnaud Monegier Du Sorbier. Managing an Industrial Software Rearchitecting Project With Source Code Labelling. In Complex Systems Design & Management conference (CSD&M), Paris, France, December 2017. URL https://hal.inria.fr/hal-02095200.

[13] Holger M. Kienle and Hausi A. Müller. The tools perspective on software reverse engineering: Requirements, construction, and evaluation. In Advanced in Computers, volume 79, pages 189–290. Elsevier, 2010.

[14] Holger M. Kienle, Adrian Kuhn, Kim Mens, Mark van den Brand, and Roel Wuyts. Tool building on the shoulders of others. IEEE Software, 26(1):22–23, 2009. ISSN

0740-7459. https://doi.org/10.1109/MS.2009.25. URL http://dx.doi.org/10.1109/MS.2009.25.

[15] Oscar Nierstrasz, Stéphane Ducasse, and Tudor Gîrba. The story of Moose: an agile reengineering environment. In Michel Wermelinger and Harald Gall, editors, Proceedings of the European Software Engineering Conference, ESEC/FSE'05, pages 1–10, New York NY, 2005. ACM Press. ISBN 1-59593-014-0. https://doi.org/10.1145/1095430.1081707. Invited paper.

[16] Chanchal Kumar Roy and James R Cordy. A survey on software clone detection research. Queen's School of Computing TR, 541(115):64–68, 2007.

[17] Benoît Verhaeghe, Anne Etien, Nicolas Anquetil, Abderrahmane Seriai, Laurent Deruelle, Stéphane Ducasse, and Mustapha Derras. Gui migration using mde from gwt to angular 6: An industrial case. In 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), Hangzhou, China, 2019. URL https://hal.inria.fr/hal-02019015.