FAMOOSr 2009

3rd Workshop on FAMIX and MOOSE in Software Reengineering

Organizers:

Simon Denier	INRIA Lille-Nord Europe, France
Tudor Gîrba	University of Bern, Switzerland

Program Committee:

Gabriela Arevalo	LIFIA La Plata, Argentina
Johan Brichau	UniversitÃĺ catholique de Louvain, Belgium
Serge Demeyer	University of Antwerp, Belgium
Yann-Gaël Guéhéneuc	École Polytechnique de Montréal, Canada
Andy Kellens	Vrije Universiteit Brussel, Belgium
Michele Lanza	University of Lugano, Switzerland
Radu Marinescu	Politehnica University of Timisoara, Romania
Oscar Nierstrasz	University of Bern, Switzerland
Pascal Vollmer	ICT Software Engineering, Germany
Thomas Zimmermann	Microsoft Research

http://moose.unibe.ch/events/famoosr2009

About FAMOOSr

The goal of the FAMOOSr workshop is to strengthen the community of researchers and practitioners who are working in re- and reverse engineering, by providing a forum for building future research using Moose and FAMIX as shared infrastructure.

Research should be collaborative and supported by tools. The increasing amount of data available about software systems poses new challenges for reengineering research, as the proposed approaches need to scale. In this context, concerns about meta-modeling and analysis techniques need to be augmented by technical concerns about how to reuse and how to build upon the efforts of previous research.

That is why Moose is an open-source software for researchers to build and share their analysis, meta-models, and data. Both FAMIX and Moose started in the context of FAMOOS, a European research project on object-oriented frameworks. Back in 1997 Moose was as a simple implementation of the FAMIX meta-model, which was a language independent meta-model for object-oriented systems. However over the past decade, Moose has been used in a growing number of research projects and has evolved to be a generic environment for various reverse and reengineering activities. In the same time, FAMIX was extended to support emerging research interest such as dynamic analysis, evolution analysis, identifier analysis, bug tracking analysis, or visualization. Recent work includes analysis of software architecture and semantic annotations.

Currently, several research groups are using Moose as a platform, or FAMIX as a meta-model, and other groups announced interest in using them in the future.

Simon & Tudor
October 11, 2009

Table of Contents

Session 1: Tools

1. Identifying Cycle Causes With CycleTable
   Jannik Laval, Simon Denier, Stéphane Ducasse
2. Lumiere: An Infrastructure for Producing 3D Applications in Smalltalk
   Fernando Olivero, Michele Lanza, Romain Robbes
3. Miler – A Tool Infrastructure to Analyze Mailing Lists
   Alberto Bacchelli, Michele Lanza, Marco D’Ambros

Session 2: Models

1. Challenges for Domain Knowledge Driven Program Analysis
   Daniel Ratiu
2. Driving the refactoring of Java Enterprise Applications by evaluating the distance between application elements
   Fabrizio Perin
3. Why and How to Substantiate the Good of our Reverse Engineering Tools?
   David Röthlisberger

Identifying cycle causes with CycleTable

Jannik Laval, Simon Denier, Stéphane Ducasse
RMoD Team, INRIA, Lille, France
firstname.lastname@inria.fr

Abstract

Identifying and understanding cycles in large applications is a challenging task. In this paper we present CycleTable which displays both direct and indirect references. It shows how minimal cycles are intertwined through shared edges and allows the reengineer to devise simple strategies to remove them.

1 Introduction

Understanding the structure of large applications is a challenging but important task. Several approaches provide information on packages and their relationships, by visualizing software artifacts, metrics, their structure and their evolution. Software metrics can be somehow difficult to understand since they are dependent on projects. Distribution Map [1] alleviates this problem by showing how properties are spread over an application. Lanza et al. [2] propose to recover high level views by visualizing relationships. Package Surface Blueprint [3] reveals the package internal structure and relationships among other packages.

In previous work, we enhanced Dependency Structure Matrix (eDSM) [4] to visualize dependencies between packages, showing in each matrix cell information about the dependencies at the fine grain of classes (inheritance, class references, invocations, referencing and referenced classes/methods). eDSM proved useful to provide an overview of dependencies, detect direct cycles (a cycle between two packages) and provide information to remove them. However, removing all direct cycles does not necessarily remove all cyclic dependencies because there could be indirect cycles (between more than two packages). Although indirect cycles are also displayed by eDSM, they are hard to read in the eDSM layout, making the task inefficient.

In this paper, we present a new visualization, called CycleTable, entirely dedicated to cyclic dependencies assessment. CycleTable layout displays both direct and indirect cycles and shows how minimal cycles are intertwined through shared edges. CycleTable combines this layout with the enriched cells of eDSM to provide the details of dependencies at class level.

Next section introduces the challenges of cycle analysis with graph layout and eDSM. Section 3 explains CycleTable layout and enriched cells. Section 4 presents a sample of cyclic dependencies displayed with CycleTable and discusses the criteria to break cycles as highlighted by the visualization.

2 Cycle Visualization

Figure 2 shows a sample graph with five nodes and three minimal cycles. Each edge is weighted. Notice that cycle A-B-C and A-B-E share a common edge (in yellow) from A to B. This shared edge is interesting to spot since it joins two cycles and by removing it we would break those cycles.

Graph layouts offer an intuitive representation of graphs, and some handle cyclic graph better than others. On large graphs, complex layouts may reduce the clutter but this is often not simple to achieve. In addition, in the context of DSM we enhanced the DSM cells to convey much more information about the strength, the nature and consequences of a cycle. Now it is difficult to combine a graph layout with enriched cells (see Figure 4 for a cell sample), as an enriched cell represents an edge in a graph. Enriched cell is an important asset of our work with eDSM as it provides fine-grained details of a dependency between two packages, and enables *small multiples* as well as *micro-macro reading* effects [5].

In eDSM we used a matrix, the traditional support of DSM. It provides a regular structure which is the same at any scale: it handles the repetitive arrangement of enriched cells, enabling the above effects. It is a fundamental element of eDSM design. In Figure 1, four packages belonging to Pharo kernel (an open-source Smalltalk) are presented in a eDSM. It shows edges involved in direct cycles in red and pink cells, indirect cycles in yellow. More explanations about the design and usage of eDSM are available in [4].

While eDSM allows us to analyze direct cycles comfortably, we could not address the problem of indirect cycles left over after removal of direct cycles. The main reason

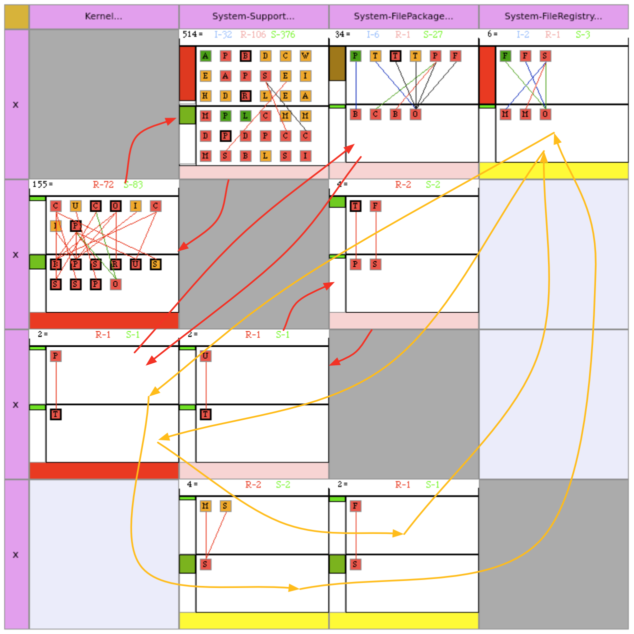


Figure 1. Some Pharo core packages in a eDSM (with cycles overlayed)

for this problem is that it is difficult to read an indirect cycle in the matrix, *i.e.,* to follow the sequence of cells representing the sequence of edges in the cycle. The order can appear quite arbitrary as one has to jump between different columns and rows (this problem does not exist with direct cycles as there is only two cells, mirroring each other along the diagonal). Cycles have been overlayed in Figure 1 to show the complexity of reading indirect cycles, intertwined with direct cycles.

3 CycleTable

We have built the CycleTable design with the single purpose of visualizing cycles. As a consequence, CycleTable does not show a complete overview of dependencies between packages as eDSM does. It complements eDSM. CycleTable is a rectangular matrix where nodes are placed in rows and cycles in columns. CycleTable (i) shows each minimal cycle clearly and independently, (ii) highlights shared edges between minimal cycles, and (iii) uses enriched cells [4] to provide information about edge internals, enabling *small multiples and micro-macro reading* [5] *i.e.,* variations of the same structure to reveal information. We detail each of these points now.

3.1 Minimal Cycle

A minimal cycle is a cycle with no repeated nodes other than the starting and ending nodes. For example, in Figure 2, A-B-E and A-B-C are two different minimal cycles, but A-B-C-D-C is not because C is present twice. With minimal cycles, the visualization provides all edges contains in cycles. Therefore it is not necessary to show complex (intertwined) cycles. In the CycleTable layout, each minimal cycle is represented directly and independently in its own column.

3.2 CycleTable Layout

The CycleTable layout is presented in Figure 3. This figure shows a sample CycleTable layout for the graph in Figure 2. The first column (header) contains the name of packages involved in cycles, then all minimal cycles are represented column by column. A box at the intersection of a row and a column indicates that the package is involved in the cycle.

One package per row. Each row contains dependencies (as boxes) from the package in the header. Number in each box represents the weight of the edge. In Figure 3, first row represents package A, which depends on package B with a weight of 10. Second row represents package B, which depends on E (weight 9) and C (weight 4).

One cycle per column. In the right part of the table, each column represents a cycle. In Figure 3, the first column involves packages A, B and E in a cycle. Each minimal cycle is represented independently.

Shared edges. Cells with the same background color represent the same edge between two packages, shared by multiple cycles. In Figure 3, first row contains two boxes with a yellow background color. It represents the same edge from A to B, involved in the two distinct cycles A-B-E and A-B-C. It is a valuable information for reengineering cycles. Indeed, removing or reversing A-B would solve two cycles.

3.3 CycleTable Cell

Box content is customized to display enriched cells as in eDSM [4]. An enriched cell displays all dependencies at class level from one package to the other. A CycleTable cell is structured in three parts: (i) on top, position in the cycle, (ii) in center, an enriched cell as in DSM, and (iii) on right, a colored frame if the edge is shared by multiple cycles.

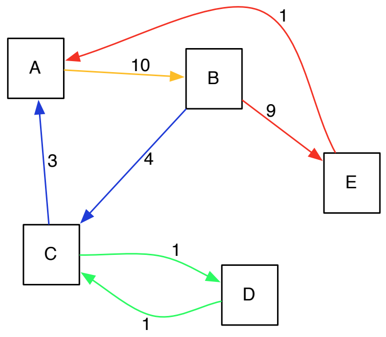


Figure 2. Sample graph with three cycles.

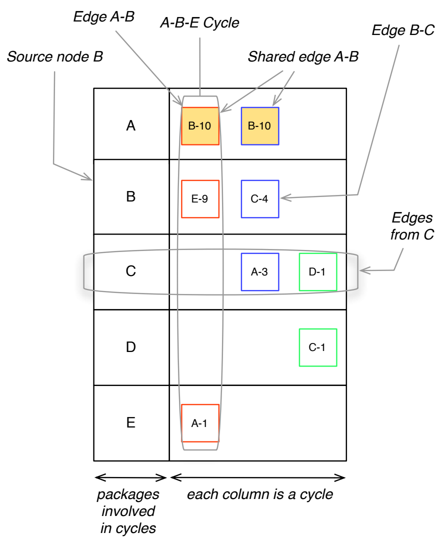


Figure 3. CycleTable for sample graph.

Position in the cycle. The position in the cycle represents a relative order between edges. This number is sometimes necessary to retrieve the exact order of edges in indirect cycles. In Figure 5, numbers allow one to read indirect cycles in third and fifth columns.

Enriched cell. Cell contents gives a detailed overview of dependencies from the package in the header of the row (*source package*) to the next package in the cycle (*target package*). Each cell represents a small context, which enforces comparison with others. The objective is to create *small multiples* [5].

An enriched cell is composed of three parts. The top row gives an overview of the strength and nature of the dependencies. The two large boxes detail dependencies going from the top box to the bottom box *i.e.,* from the *source package* to the *target package*. Each box contains squares that represent involved classes: referencing classes in the source package and referenced classes in the target package. Edges between squares links each source class (in top box) to its target classes (in bottom box). As this structure is the same as in eDSM [4], we give no more explanation in this paper.

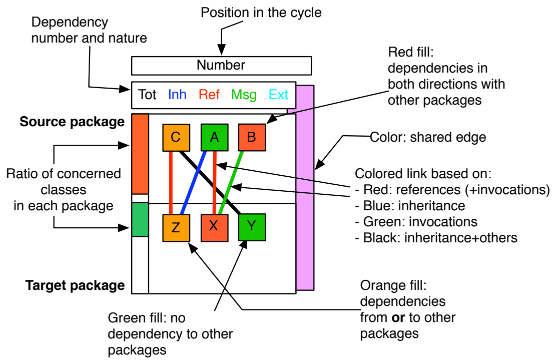

Dependency number and nature
Position in the cycle
Number
Tot Inh Ref Msg Ext
Red fill: dependencies in both directions with other packages
Source package
C
A
B
Color: shared edge
Ratio of concerned classes in each package
Colored link based on:
- Red: references (+invocations)
- Blue: inheritance
- Green: invocations
- Black: inheritance+others
Z
X
Y
Target package
Orange fill: dependencies from or to other packages
Green fill: no dependency to other packages

Figure 4. Information in CycleTable cell.

4 Breaking Cycles with CycleTable

Figure 5 shows a CycleTable with the sample four packages of Pharo core presented in Figure 1. Five minimal cycles are immediately made visible. It also appears that three edges are each involved in multiple cycles (with the red, blue, and orange frames).

An important asset of CycleTable is that it does not focus on a single solution to break cyclic dependencies. It rather highlights different options as there are many ways to resolve such cyclic dependencies. Only the reengineer can select what he thinks is the best solution based on a matter of criteria. We now discuss how CycleTable allows one to consider solutions for solving cycles in Figure 5.

First things to consider in CycleTable are the shared edges, the number of cycles they are involved in, and their weight. For example, the blue cell linking `System-FileRegistry` to `Kernel` is in the two indirect cycles. Yet it has a weight of six dependencies and involves six classes as well, which can require some work to remove. Finally, from a semantic point of view, `Kernel` is at the root of many functions in the system so it seems difficult to remove such dependencies from `System-FileRegistry`.

Instead, we turn our focus to the red cells, linking `Kernel` to `System-FilePackage`. It has a very low weight and involves only two classes. This is the minimal dependency we can get between two packages and seems as a prime candidate for removal. Removing this dependency (by moving a class, changing the method, or making a class

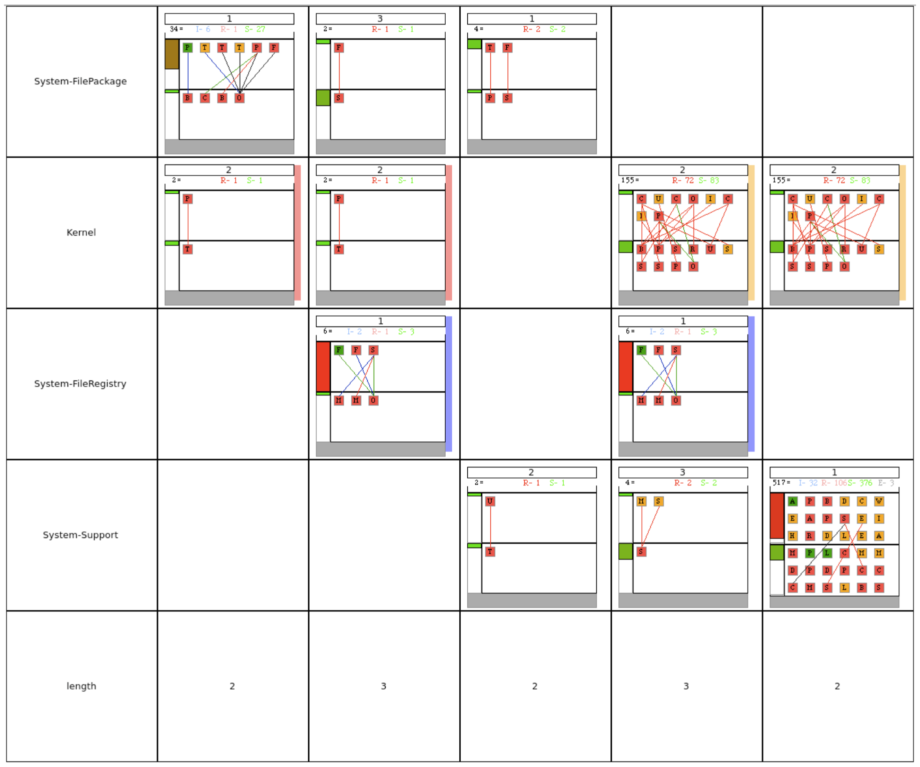


Figure 5. Pharo core from Figure 1 in CycleTable.

extension) would break the first two cycles from the left.

The third cycle links directly `System-FilePackage` and `System-Support` and is not intertwined with others. There is a weak link from the second package to the first which seems easy enough to remove.

Finally, the orange edge links the last two cycles. However, it is obvious that it is too complex to be removed. The fourth cycle seems solvable by removing its last edge (from `System-Support` to `System-FileRegistry`). The last cycle, linking directly `Kernel` and `System-Support` is too complex so that a simple removal strategy can be devised with CycleTable. Both packages are at the very core of Pharo system and are highly coupled together.

5 Conclusion

In this paper we presented CycleTable. This visualization shows cycles between packages in a system. Each cycle is presented in a separated column. A colored frame show which edge is shared by several cycles. To complete the visualization, enriched cells (proposed in [4]) have been adapted and integrated to represent each edge.

This visualization is for now a good complement of a DSM visualization. This visualization allows us to solve cycles separately or conjointly.

Future work will focus on computing heuristics for highlighting interesting edges in cycles (either because of their low weight or because they are shared by many cycles), prone to removal by the reengineer.

References

[1] S. Ducasse, T. Gîrba, and A. Kuhn. Distribution map. In *Proceedings of 22nd IEEE International Conference on Software Maintenance (ICSM '06)*, pages 203–212, Los Alamitos CA, 2006. IEEE Computer Society.

[2] S. Ducasse and M. Lanza. The class blueprint: Visually supporting the understanding of classes. *Transactions on Software Engineering (TSE)*, 31(1):75–90, Jan. 2005.

[3] S. Ducasse, D. Pollet, M. Suen, H. Abdeen, and I. Alloui. Package surface blueprints: Visually supporting the understanding of package relationships. In *ICSM '07: Proceedings of the IEEE International Conference on Software Maintenance*, pages 94–103, 2007.

[4] J. Laval, S. Denier, S. Ducasse, and A. Bergel. Identifying cycle causes with enriched dependency structural matrix. In *WCRE '09: Proceedings of the 2009 16th Working Conference on Reverse Engineering*, 2009.

[5] E. R. Tufte. *Visual Explanations*. Graphics Press, 1997.

Lumière: An Infrastructure for Producing 3D Applications in Smalltalk

Fernando Olivero, Michele Lanza, Romain Robbes
REVEAL @ Faculty of Informatics, University of Lugano
{fernando.olivero,michele.lanza,romain.robbes}@usi.ch

***Abstract*—With the goal of developing 3D applications using Smalltalk, we decided to build a lightweight 3D framework named *Lumière* from scratch, because after conducting a brief survey of the available frameworks, we found many of them to be either outdated, abandoned, undocumented or too heavyweight. In this paper we present the design and implementation of *Lumière*, an object-oriented framework based on a stage metaphor to display interactive 3D micro-worlds.**

I. Introduction

The Smalltalk language and its many dialects include several frameworks for developing 3D applications. Some of them are simple wrappers of low level rendering libraries (such as OpenGL or DirectX), while others are complete frameworks for producing 3D graphics. However, many of them are outdated, unmaintained, undocumented or heavyweight [2].

Therefore we built *Lumière*, the missing 3D framework in Smalltalk. We implemented it using Pharo and OpenGL, with the objective of producing 3D graphics with a simple, modern, lightweight and efficient framework. *Lumière* is an object-oriented framework that provides a layer of abstraction over graphical primitives and low-level rendering. The low-level rendering of the framework is done by OpenGL, an industry standard library for doing high performance graphics. We chose to use OpenGL for efficiency, maintainability, and portability reasons [2]. *Lumière* provides the infrastructure for building 3D applications using intuitive metaphors and high-level concepts such as shapes, lights, cameras and stages. Using *Lumière* a programmer can create complex 3D scenes that we call *micro-worlds*, using performant 3D graphics fully integrated with the underlying environment (Figure 1).

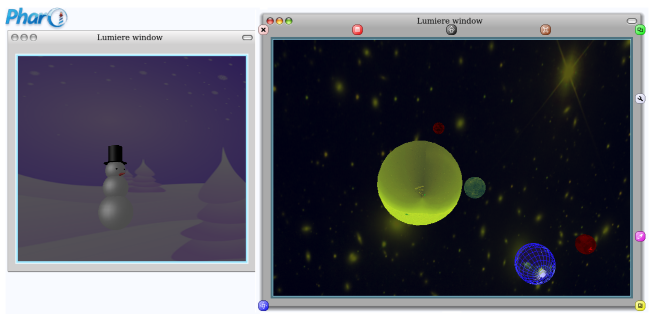


Fig. 1. Lumiere micro-worlds

In the following sections we describe the design and implementation of *Lumière*.

II. The Design of *Lumière*

When designing *Lumière* we chose to build the framework around an intuitive metaphor, promoting ease of usage and understanding, because of the immediate mental model that metaphors provide. We settled on a stage metaphor, where rendering takes place by cameras taking pictures of micro-worlds lit by the lights of the stage. Similar concepts were already present in other Smalltalk frameworks such as Ballon3D and Croquet [3], and also in foreign languages such as Open Scene Graph[1]. In the following we detail each aspect of the design and provide UML diagrams of the exposed APIs.

A. *Rendering a Scene*

A stage provides a context for taking pictures of a micro-world, therefore a stage contains a micro-world, cameras and lights, and other environmental properties such as ambient lights and fog (see Figure 2).

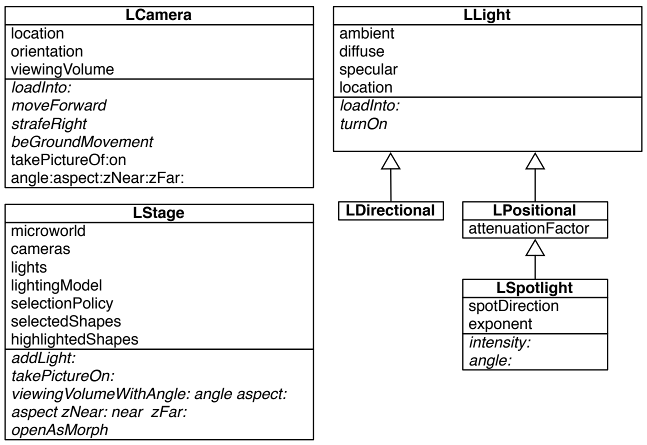


Fig. 2. UML class diagram of Lumiere stages, lights, and cameras

The lights are responsible for illuminating the scenes. There are several types of lights in *Lumière*, similar to the OpenGL lighting model. Each light can be positioned anywhere on stage and contributes to the final appearance of the shapes in a scene.

Cameras take pictures of the microworlds, rendering them on a canvas. They dictate several properties of the final drawing such as the distance, orientation and angle of sight from which the picture is taken. As such, they determine which shapes of the micro-world are visible in the rendered image.

[1]http://www.openscenegraph.org/projects/osg

B. *Modeling a micro-world*

A *Lumière* micro-world is a 3D world, programmatically modeled and populated by a diverse variety of visual objects called shapes. Shapes are the building blocks of *Lumière* scenes, they have visual properties such as scale, color, materials and textures. A shape can be a primitive or a composite shape that groups several lower-level shapes. *Lumière* provides special-purpose composite shapes with a predefined layout to simplify the positioning of the shapes composing it (see Figure 3).

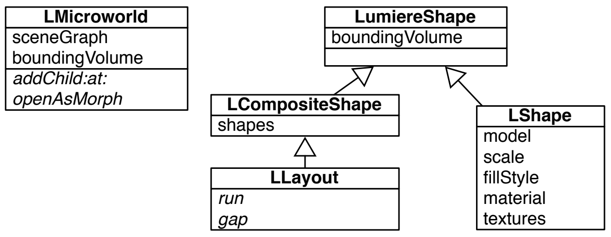


Fig. 3. Lumiere micro-world and shapes

The visual properties, orientation, and location of *Lumière* shapes are described programatically when designing a *Lumière* micro-world. The spatial relationships between shapes are modeled in a micro-world using a scene graph implementation, which is a hierarchical representation of the position of the shapes that populate a given micro-world. A scene graph is a directed graph composed by different types of nodes. There are nodes for loading translations, rotations, scales into a canvas. Other nodes when loaded into a canvas generate a particular figure to be drawn (see Figure 4).

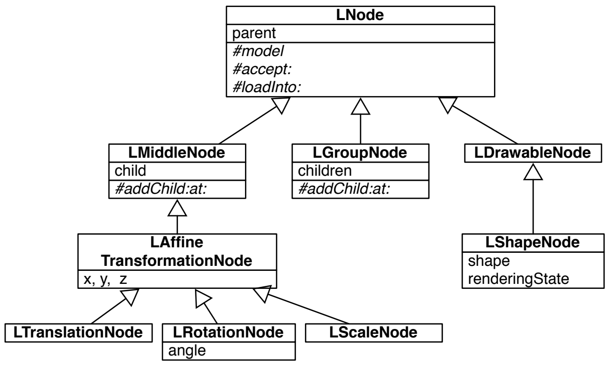


Fig. 4. Lumiere scene graph nodes

When a *Lumière* node is loaded into a canvas, it modifies the state of the latter according to the its type. For example when a scale node is loaded into a canvas it produces a modification in the size of every shape rendered afterwards.

III. The Implementation of *Lumière*

Lumière uses several underlying frameworks, from the base graphics renderer, OpenGL to the Pharo UI framework, Morphic. The architecture of *Lumière* is depicted in Figure 5.

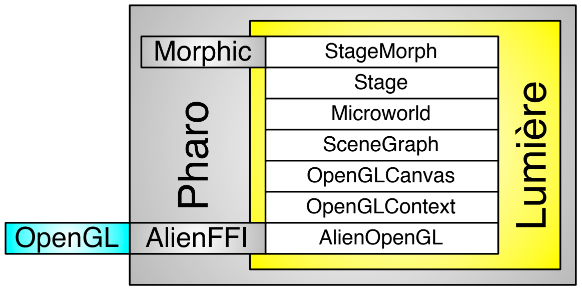


Fig. 5. The architecture of *Lumière*

Lumière uses OpenGL through AlienFFI, a foreign library wrapper framework. It uses a canvas framework to facilitate its interactions with the OpenGL renderer, and is integrated with the Pharo environment through the Morphic, Omnibrowser and Glamour frameworks.

A. *AlienOpenGL*

OpenGL is written in C to maximize performance and hence *Lumière* needs to interface with it. We implemented **AlienOpenGL**[2], a framework that uses AlienFFI to access the underlying functionality offered by OpenGL. AlienOpenGL allows one to invoke library functions by sending messages to a singleton instance of ALienOpenGLLibrary.

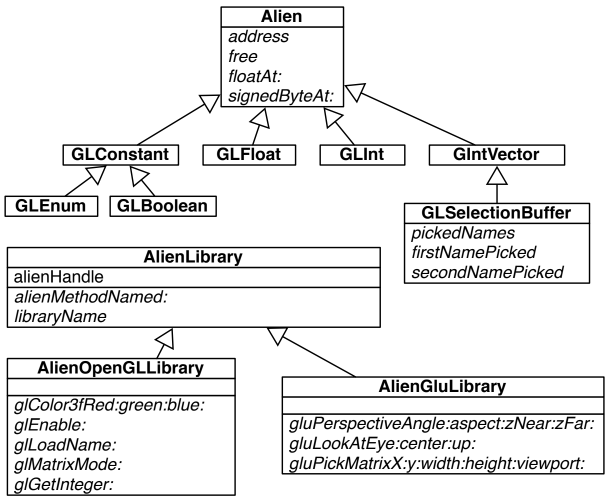


Fig. 6. AlienOpenGL API

See Figure 6 for a UML class diagram of the API of AlienOpenGL.

This framework also provides OpenGL data types reification, access to the Glu library (an extension to the base OpenGL library), and facilities for creating and manipulating an OpenGL drawable surface, managed both by the operating system and AlienOpenGL, where the rendering takes place.

[2]AlienOpenGL is available at http://www.squeaksource.com/AlienOpenGL

B. Canvas and context

When taking pictures of a micro-world with a camera, *Lumière* traverses the scene and loads the primitive shapes onto a canvas, which is an abstraction of a 3D surface, where primitive figures can be rendered. A canvas has a context, an object that reifies an OpenGL context (a particular state of the base rendering system), and acts as an adapter between *Lumière* code and the basic OpenGL protocol of the AlienOpenGL framework. The canvas context allows us to provide some optimizations by maintaining a cache of the state changes, and forwarding only real state changes to the low level AlienOpenGL library.

A canvas forwards the load primitives requests to the appropriate class of geometry object of *Lumière*, for example the message `#loadSphereScaled:` is forwarded to an instance of `LSphere`. This enables to implement different vertex loading mechanisms without modifying any of the canvas load methods, decoupling the request of loading a primitive from the actual low-level implementation. In Figure 7 we display the canvas, context and geometry class diagram.

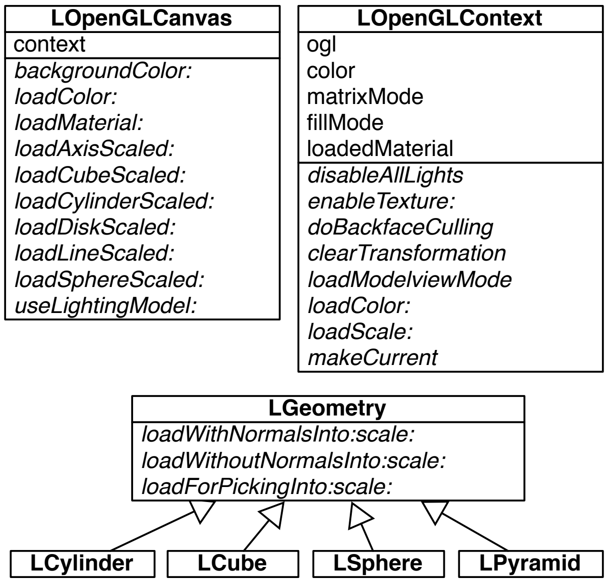


Fig. 7. Canvas, context and geometry

C. Scene graph

The scene graph dictates the final appearance, orientation, and location of the shapes that populate a micro-world. Each node in the path from the root node to a drawable node, contributes to modify the mentioned properties. For example, inserting a scale node after the root node of a micro-world that contains only one drawable node that renders a cube, affects the final size of the cube being rendered.

A scene graph is a convenient structure for accessing all the shapes in a micro-world when performing several tasks. *Lumière* uses the Visitor design pattern [1] to streamline this process. *Lumière* includes three different visitors (see Figure 8) for traversing scene graphs:

1) `LRenderingVisitor`: A rendering visitor traverses the scene and loads the visible nodes into a canvas to display images on the screen.
2) `LScenePickingVisitor`: Picking is the process trough which OpenGL determines which 3D object is under the mouse cursor. OpenGL renders the scene in a hidden buffer for this, so the picking visitor renders a lower detail version of the drawables for optimization purposes.
3) `LBVHCullingVisitor`: Culling is the process of determining which parts of the scene are visible or not. The culling visitor traverses the scene performing intersection tests, pruning the nodes of the shapes that are outside a particular viewing volume.

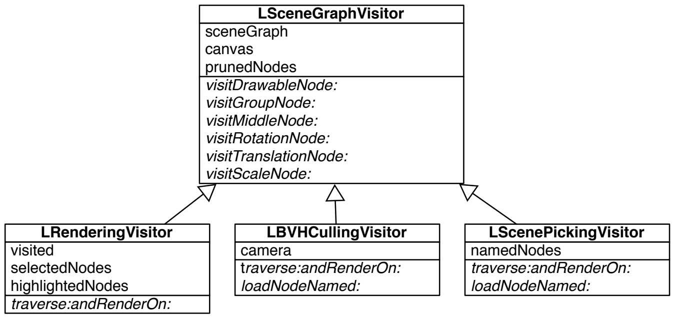


Fig. 8. Lumiere scene visitors

D. Underlying environment integration

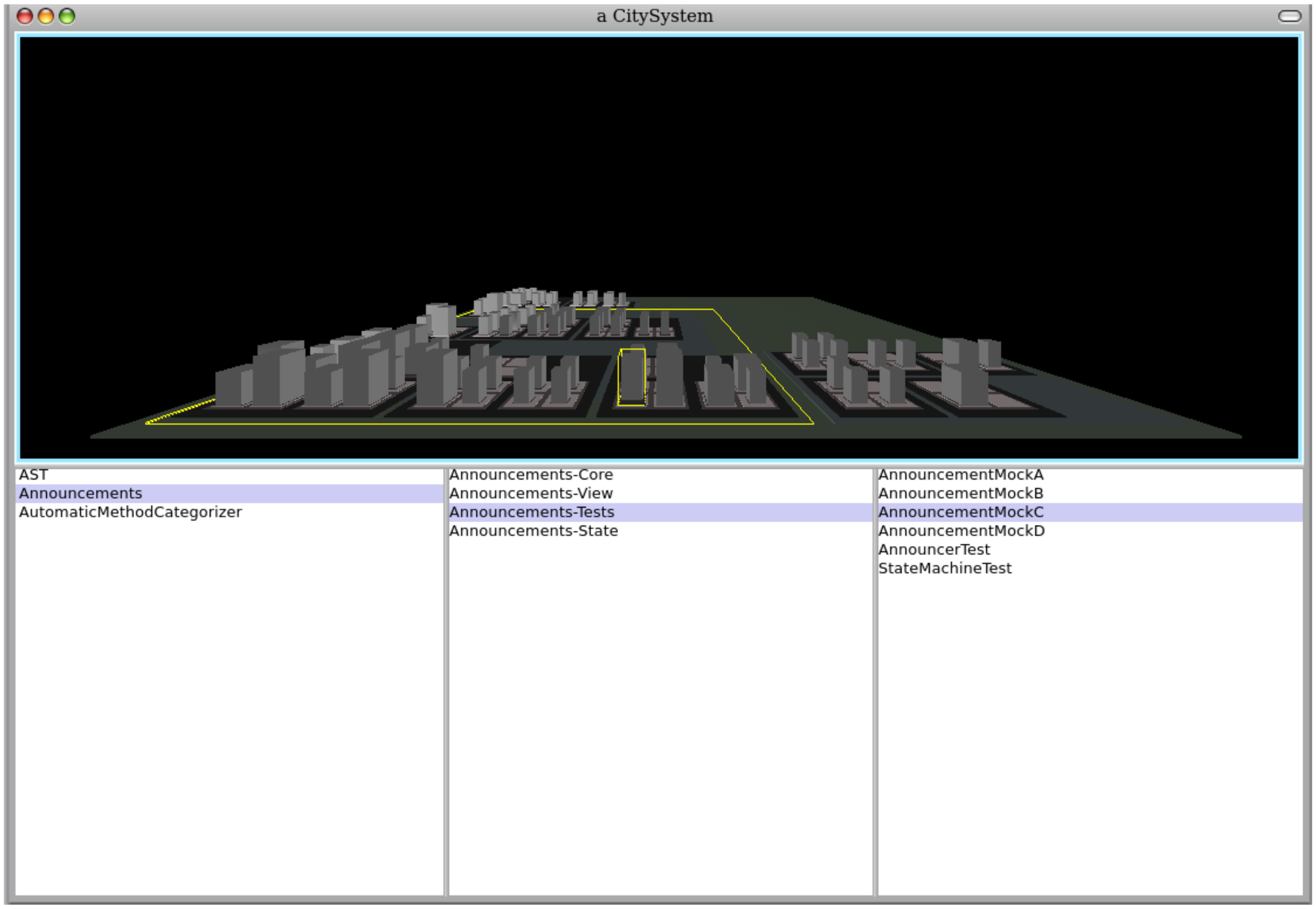
Lumière is fully integrated in the Pharo smalltalk environment. A stage can be opened in a window, integrated in browsers and respond to mouse and keyboard events.

Integration with Morphic. *Lumière* stages are integrated into Morphic, the standard UI framework of Pharo. Lumiere shapes, micro-worlds and stages answer the message `#openAsMorph`, opening an instance of a `StageMorph`. A Stage Morph holds an AlienOpenGL draw-able for performing the low level rendering, displaying pictures of the micro-worlds taken by the cameras of the stages.

Integration with Omnibrowser. Stage morphs can be inserted into Omnibrowser, the standard window environment of Pharo. A stage morph answers the messages `#addWindow` and `#removeWindow`, for adding or removing the window decorating it.

Integration with Glamour. *Lumière* stages can also be integrated into Glamour, a new scriptable browser framework for Pharo. Any stage can be rendered as a *Lumière* presentation inserted into a glamour browser (see Figure 9).

Responding to user events. The shapes of micro-worlds displayed by stage morphs can react to user input, from the keyboard and mouse. *Lumière* provides modifiable stage interaction policies to control the highlighting, selection, clicking and keyboard event handling. For example some stage interaction policies specify single or multiple selection, floating over shapes awareness and different camera keyboards movements.



AST
Announcements
AutomaticMethodCategorizer

Announcements-Core
Announcements-View
Announcements-Tests
Announcements-State

AnnouncementMockA
AnnouncementMockB
AnnouncementMockC
AnnouncementMockD
AnnouncerTest
StateMachineTest

Fig. 9. Glamour integration

In Figure 10 we present to the right a stage morph with a selected shape, and to the left a stage morph with a highlighted shape integrated into Omnibrowser.

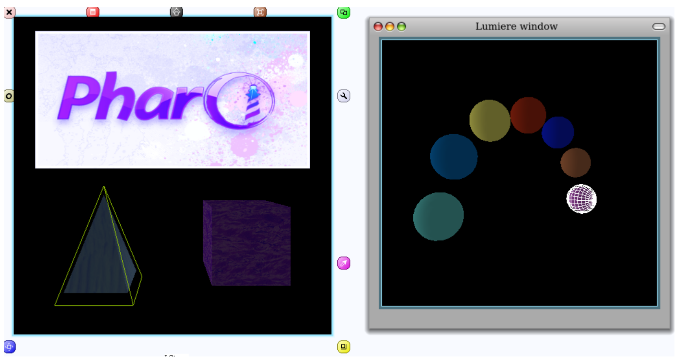


Fig. 10. Pharo integration

IV. Conclusion

In this paper we presented the design and implementation of *Lumière*, our novel 3D framework implemented in Smalltalk. *Lumière* uses a stage metaphor to render micro-worlds – graphs of 3D shapes generated programmatically– in OpenGL. *Lumière* is implemented using Pharo and is fully integrated with the underlying environment. The 3D scenes generated by *Lumière* can be integrated in Pharo's windows and browsers, and *Lumière* provides support for customizable keyboard and mouse interactions.

References

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, Mass., 1995.
[2] F. Olivero, M. Lanza, and R. Robbes. Lumiére: A novel framework for rendering 3d graphics in smalltalk. pages xxx – xxx. ACM Press, 2009.
[3] D. Smith, A. Kay, A. Raab, and D. Reed. Croquet - a collaboration system architecture, Jan. 2003.

Miler – A Tool Infrastructure to Analyze Mailing Lists

Alberto Bacchelli, Michele Lanza, Marco D'Ambros
REVEAL @ Faculty of Informatics - University of Lugano, Switzerland

***Abstract*—The information that can be used to analyze software systems is not limited to the raw source code, but also to any other artifact produced during its evolution. In our recent work we have focused on how archives of e-mails that concern a system can be exploited to enhance program comprehension.**

In this paper we present Miler, a tool we have built for that purpose. We describe its architecture and infrastructure, and the FAMIX meta-model extension we have devised to model mailing list archives.

I. Introduction

In software systems, not only the source code, but any other artifact revolving around them (requirements, design documents, user manuals, bug reports, *etc.*) concurs to define their shapes. Such artifacts add information either by describing a specific piece of the source code or generally introducing concepts or necessities.

E-mail archives are widely employed during the development of software systems and contain information at different levels of abstraction. E-mails are used to discuss issues ranging from low-level decisions (*e.g.,* implementation of specific software artifacts, bug fixing, discussing user interface problems) up to high-level considerations (*e.g.,* design rationales, future planning). They can often be written and read by both software system developers and end-users, and always come with additional meta-information (*e.g.,* timestamp, thread, author) that can be taken into account.

Since the FAMIX meta-model was designed to be extensible, adding new information to the source code entities it models is straightforward. However, to add information from e-mails it is first necessary to import messages from the archives in which they reside. Then, the resulting data must be stored in an easily accessible persistent format, in order to be used for subsequent analyses. Finally, the information contained in e-mails must be linked to the entities in the system model, described according to the FAMIX meta-model.

In this article we present *Miler*, a tool infrastructure that tackles these issues and allows one to analyze e-mail archives of software systems. Miler is implemented in VisualWorks[1] Smalltalk, uses the Moose Reengineering Environment for the modeling tasks, and uses GLORP [4] and the Metabase [2] for the object persistency. In previous work we used this infrastructure to create a benchmark for e-mail analysis and to test different lightweight methodologies for linking e-mails and source code [1].

Structure of the paper In Section II we present an overview of the Miler architecture introducing its components and showing how they interact. Then we provide the details of our technique to import archives of mailing lists, and we explain how we store e-mails in a database transparently thanks to a simple meta-model description. In Section III, we discuss how to use Miler to deal with the merging of the model of the system and the model of the e-mail archive. We conclude in Section IV.

II. Miler

Figure 1 depicts an overview of Miler's architecture. After a target system is chosen (point I), the source code is imported into the MOOSE Reengineering Environment through an importer module. The importer parses the source code, generate the corresponding model according to the FAMIX meta-model and exports it in a format that can be loaded by the MOOSE Environment. It is possible both to use a third-part importer or to implement a specific one. For example, when working with Java systems we used iPlasma[2] as importer, while we implemented specific importers to deal with languages (*e.g.,* PHP, Actionscript) not supported by any external tool. After the necessary models are available from Moose, they are included into the core of Miler as "System Models".

After the target system is selected, the e-mails in the archive of mailing lists are also imported into Miler. Different systems use different applications to manage and archive mailing lists, thus not offering a consistent way to access to their data. Moreover, such applications could change during the system lifetime. For this reason, in the worst case scenario, it might be necessary to write at least one importer per system to collect e-mails. We tackled this issue by using MarkMail[3], a free service for searching mailing list archives, which are constantly updated. More than 7,000 mailing lists, taken especially from open source software projects, are stored and displayed in a consistent manner. It is possible both to search e-mails through queries and to access all the e-mails of a specific mailing list. We implemented an importer (Figure 1, point II) that crawls the MarkMail website and extracts all the e-mails from the selected mailing lists and instantiates them

[1]http://www.cincomsmalltalk.com/

[2]http://loose.upt.ro/iplasma/
[3]http://markmail.org/

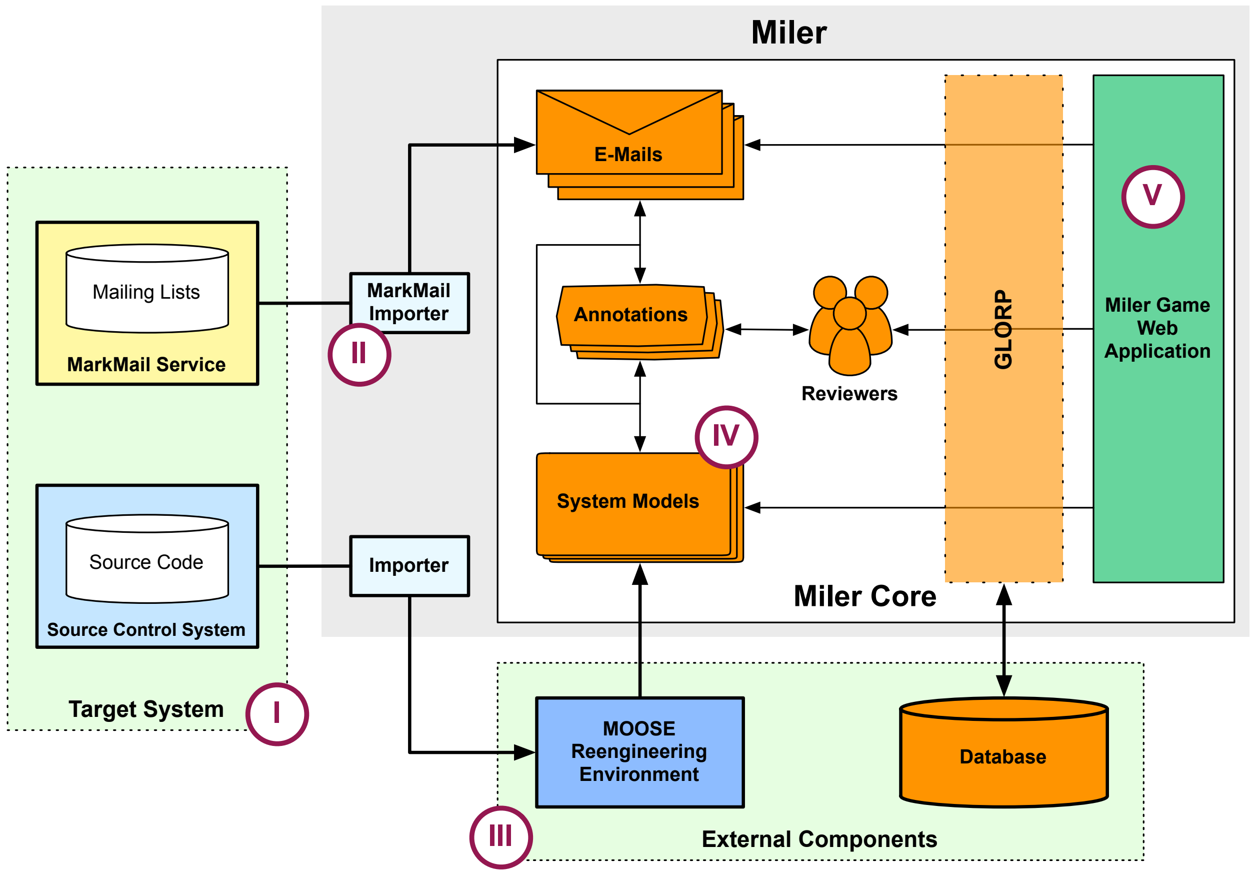


Figure 1. Miler architecture

as objects that are part of the Miler core. As it appears from the architecture diagram, it is easy to add various importers capable of extracting data from other sources than MarkMail service and instantiate them as objects.

To store information gathered from a mailing list, we use an approach based on object persistency rather than using text files (*e.g.,* as done with MOOSE, whose last, and currently used, file format is MSE[4]). Text files do not require a DBMS, however data cannot be accessed remotely, and they generate performance bottlenecks, since the entire text file must always be parsed (*i.e.,* it is not possible to import only parts of the model). When considering mailing lists, the performance aspect is relevant as they often contain thousands of e-mails.

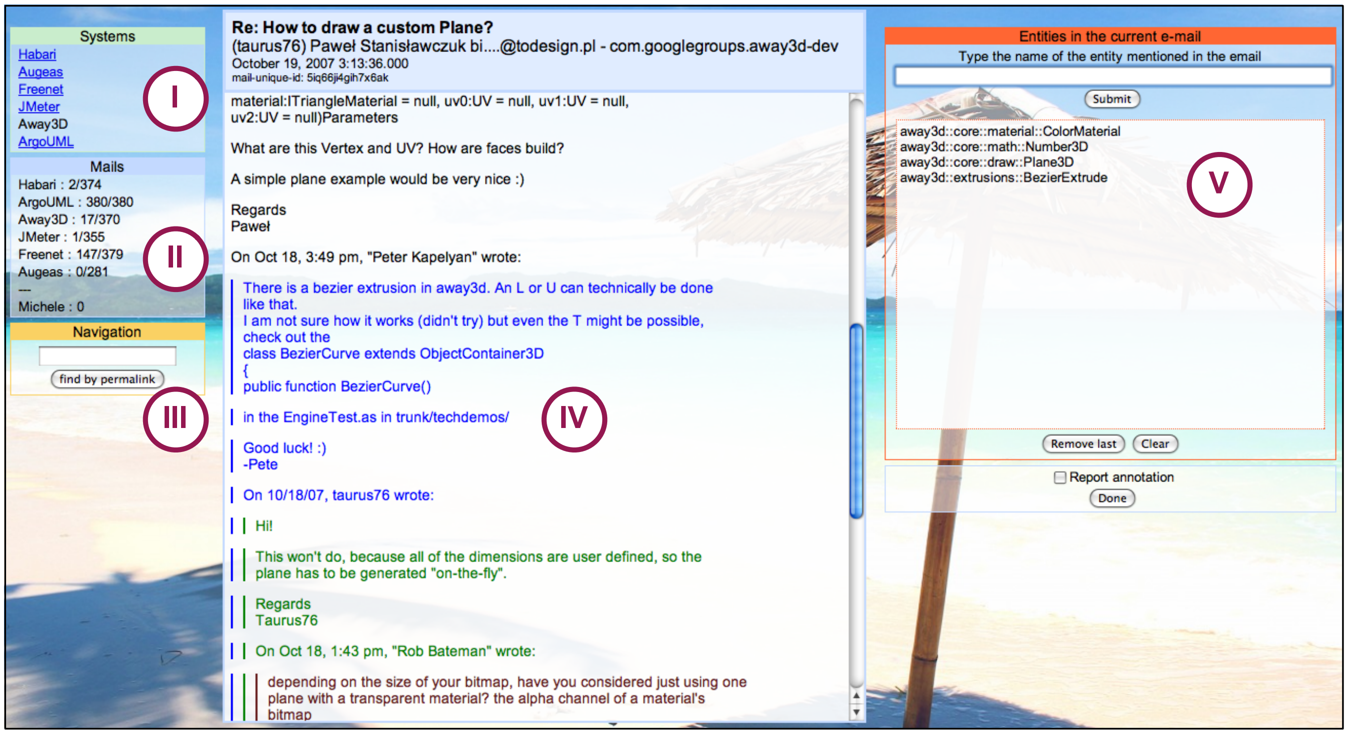
In Figure 1, the orange components that reside inside the core of Miler (Point IV) are modeled according to a meta-model written in Meta[5]. Thanks to these meta-descriptions, the Metabase component is capable of automatically generating the corresponding GLORP class descriptions, which define the mapping between the Smalltalk classes and the database tables [2]. In this way, objects are stored and retrieved from the chosen database transparently through the GLORP layer: It is sufficient to save the objects of the model the first time they are created and to create a connection with the database when loading Miler. In addition, since objects are stored in a common database, it is possible to access them, even remotely, from different languages and applications.

The last component of the Miler architecture is the “Miler Game”(Figure 1, Point V). This is a web application used to manually *annotate* the entities of the systems with the e-mails discussing them. This application is built on the top of the Miler Core using the Seaside web framework [3]. In Section III, we describe in detail the application from a user point of view.

III. Entities and E-Mails

After the core of Miler is filled with the necessary data, gathered from both source code and message archives, the step that follows is extending the FAMIX models with the relevant information that resides inside the model of e-mails, *i.e.,* linking software system entities (*e.g.,* classes) with the

[4]http://scg.unibe.ch/wiki/projects/fame/mse
[5]Meta is the previous version of Fame (http://scg.unibe.ch/wiki/projects/fame/)



Habari : 2/374
ArgoUML : 380/380
Away3D : 17/370
JMeter : 1/355
Freenet : 147/379
Augeas : 0/281

Michele : 0

Re: How to draw a custom Plane?
(taurus76) Paweł Stanisławczuk bi....@todesign.pl - com.googlegroups.away3d-dev
October 19, 2007 3:13:36.000
mail-unique-id: 5iq66ji4gih7x6ak

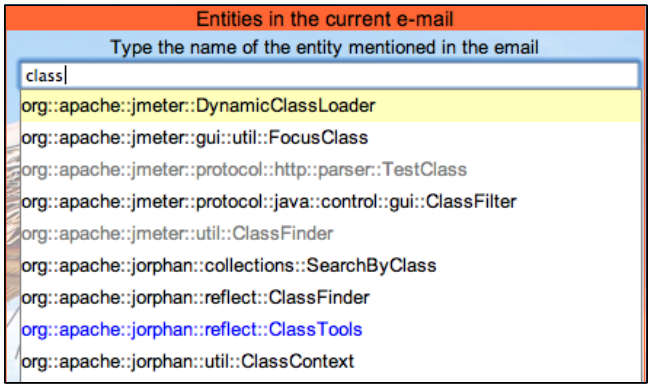
```
material:ITriangleMaterial = null, uv0:UV = null, uv1:UV = null,
uv2:UV = null)Parameters

What are this Vertex and UV? How are faces build?

A simple plane example would be very nice :)

Regards
Paweł

On Oct 18, 3:49 pm, "Peter Kapelyan" wrote:

| There is a bezier extrusion in away3d. An L or U can technically be done
| like that.
| I am not sure how it works (didn't try) but even the T might be possible,
| check out the
| class BezierCurve extends ObjectContainer3D
| {
| public function BezierCurve()

| in the EngineTest.as in trunk/techdemos/

| Good luck! :)
| -Pete

| On 10/18/07, taurus76 wrote:

| | Hi!

| | This won't do, because all of the dimensions are user defined, so the
| | plane has to be generated "on-the-fly".

| | Regards
| | Taurus76

| | On Oct 18, 1:43 pm, "Rob Bateman" wrote:

| | | depending on the size of your bitmap, have you considered just using one
| | | plane with a transparent material? the alpha channel of a material's
| | | bitmap
```

away3d::core::material::ColorMaterial
away3d::core::math::Number3D
away3d::core::draw::Plane3D
away3d::extrusions::BezierExtrude

Figure 2. The Miler Game Web Application

e-mails discussing them. Since e-mails are written in free-form text, automatically finding such missing links is not a trivial task [1]. To create a benchmark against which to compare approaches that establish such links, we devised the "Miler Game", a web application that permits to efficiently annotate the system entities with the corresponding e-mails.

Figure 2 illustrates the main page of the web application, which is displayed after the user login. Different interactive panels form this page: the "Systems" panel (Point I) shows the list of the software systems for which both the FAMIX model and the e-mails are available in Miler. The user here chooses the system to be considered. The "Mails" panel (Point II) informs the user about how many e-mails have been read for each system. Since it is possible to setup a predetermined number of mails to read per system (*e.g.*, to create benchmarks [1]), this number is also displayed. The "Navigation" panel (Point III) allows the user to retrieve an e-mail knowing its unique permalink (as we were using the MarkMail importer, we decided to use the one present in the MarkMail service). The main panel of the application is the e-mail panel (Point IV), in which the headers and content of an e-mail are displayed. Headers are displayed on top of the message, including the subject, the author, the date and the list to which the e-mail was sent, and the unique permalink of the message inside Miler. The message content is colored like in common e-mail readers: "Threaded" messages that are part of a larger discussion often quote sentences from previous e-mails, thus, in order to increase readability, the Miler Game colors quoted text differently. Finally, there is the "Annotation" panel (Point V) with two components: the list of the entities that are already annotated (*i.e.,* are discussed in the e-mail) and an autocompletion field (Figure 3).



Entities in the current e-mail
Type the name of the entity mentioned in the email

class

org::apache::jmeter::DynamicClassLoader
org::apache::jmeter::gui::util::FocusClass
org::apache::jmeter::protocol::http::parser::TestClass
org::apache::jmeter::protocol::java::control::gui::ClassFilter
org::apache::jmeter::util::ClassFinder
org::apache::jorphan::collections::SearchByClass
org::apache::jorphan::reflect::ClassFinder
org::apache::jorphan::reflect::ClassTools
org::apache::jorphan::util::ClassContext

Figure 3. Autocompletion in the Miler Game

This field helps the user in annotating the e-mail at different levels: first, it allows the user to see all the entities whose name include the letters she inserted; second, it avoids typos and forces the user to enter only entities that are really present in the FAMIX models; third, the entity names are colored according to the time proximity of the e-mail date and the entity release. In Miler, it is possible to have more than one release of the same system, and when the entity names are displayed their date is taken into account. If the entity is present in the last release before the e-mail date, then its name will be colored in black in the autocompletion menu, if the entity is older, then it will be colored in light gray. On the contrary, if the entity appears in the version released after the e-mail date, its color will be blue, otherwise light blue, if present in a version which was released later. This helps the user discerning the appropriate entity. For example, if we consider the class named *ClassFinder*, that is also present in Figure 3, the autocompletion menu shows two different entities with this name: "org::apache::jmeter::util::ClassFinder", in light gray, and "org::apache::jorphan::reflect::ClassFinder", in black. If the class is mentioned only by its name in the e-mail, without the package, and there is no other information, the user can decide to take the latter, as it is more probable that the e-mail is referring to it.

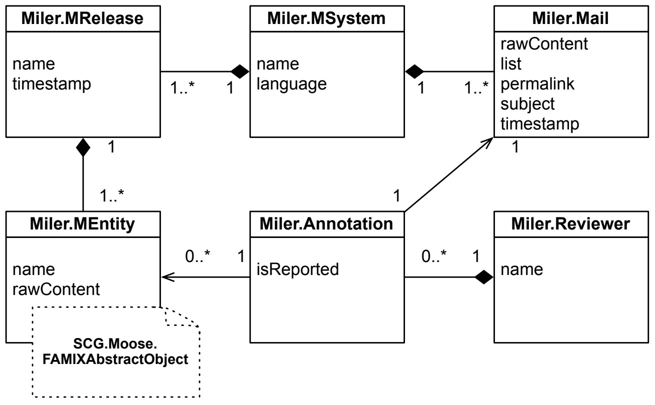


Figure 4. UML Schema of the Miler core

Figure 4 shows the core meta-model behind Miler. *MSystem* is the class representing a system that is imported in Miler. Since GLORP adds an hidden "id" instance variable to every object it stores, it is possible for the name to be not unique. Each *MSystem* owns a collection of *MReleases*, which represent the various version of the source code. Each *MRelease* is characterized by a "timestamp" and has a collection of unique *MEntities*. In our case, we decided to create a new class to represent entities of the system, instead of extending Moose definitions, however it is possible to substitute this abstraction with a *FAMIXAbstractObject* class. To do so, the developer must describe that class meta-model using Meta to specify the information, *i.e.,* instance variables, to be stored and retrieved from the database.

In Figure 4, the class *Reviewer* represents the abstraction of the users of Miler. When a reviewer reads a new e-mail, this generates a new *Annotation* that expresses the reviewer's opinion on the connection between a *Mail* and zero or more *MEntities*. This class shows the "missing link" between source code entities and e-mails. Once the links are validated (*e.g.,* by a review or other expert users), the *Annotation* can be put aside and the *MEntity* can be directly extended with the new information. If *FAMIXAbstractObject* takes the place of *MEntity*, it is possible to extend it either by using a new instance variable or by using the *property* attribute already existing in the class.

Annotations can be generated not only by the manual work of users, but also implementing an automated method. For example, it was shown that searching for entity names into the content of mails using regular expressions is often sufficient to establish a correct link between an e-mail and source code artifacts [1].

IV. Summary

In this paper we have presented Miler, a novel tool infrastructure to establish links between e-mails and source code artifacts. We described the architecture of Miler, discussing the different modules composing it, and presented our implementation and how it can be extended. We then presented Miler Game, a web application we devised for manually linking the information in the e-mails with the entities of the system.

As future work, we plan to extend the "Miler Game" web application to allow selected users to easily perform administrative tasks, such as adding new systems, releases or mailing lists, by simply providing a link to the version control repository or to the mailing list archive.

Acknowledgments We gratefully acknowledge the financial support of the Swiss National Science foundation for the project "DiCoSA" (SNF Project No. 118063).

References

[1] A. Bacchelli, M. D'Ambros, M. Lanza, and R. Robbes. Benchmarking lightweight techniques to link e-mails and source code. In *Proceedings of WCRE 2009 (16th Working Conference on Reverse Engineering)*, pages xxx–xxx. IEEE CS Press, 2009.

[2] M. D'Ambros, M. Lanza, and M. Pinzger. The metabase: Generating object persistency using meta descriptions. In *Proceedings of FAMOOSR 2007 (1st Workshop on FAMIX and Moose in Reengineering)*, 2007.

[3] S. Ducasse, A. Lienhard, and L. Renggli. Seaside: A flexible environment for building dynamic web applications. *IEEE Software*, 24(5):56–63, 2007.

[4] A. Knight. Glorp: generic lightweight object-relational persistence. In *OOPSLA '00: Addendum to the 2000 proceedings of the conference on Object-oriented programming, systems, languages, and applications (Addendum)*, pages 173–174. ACM Press, 2000.

Challenges for Domain Knowledge Driven Program Analysis

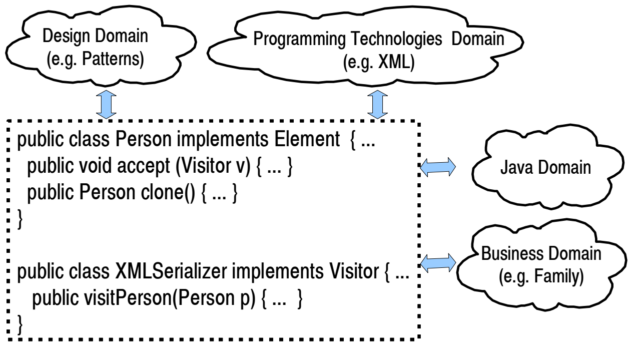
Daniel Ratiu
Technische Universität München
ratiu@in.tum.de

Abstract

Programming is a knowledge intensive activity. In order to write and evolve programs, programmers make use of knowledge of different kinds like about the business domain, programming technologies or design. Most of the current reverse engineering approaches do not explicitly take into consideration the domain knowledge implemented in programs when the analyses are performed and thereby they lose important information. This situation is due to a series of difficulties that need to be overcome when performing analyses that interpret the code from the point of view of domain knowledge. In this paper we advocate the need for the explicit use of domain knowledge in program analyses and we discuss the most important challenges that have to be addressed to realize such analyses in the practice: making the knowledge explicit, obtaining the knowledge, defining appropriate interpretations, and recovering the interpretations automatically. We present islands of solutions to approach these challenges and we argue that to systematically overcome them is required a community effort.

1 Introduction

Programs model aspects of the real world. Design heuristics and object-oriented design guidelines suggest that there needs to be a correspondence between program modules and the domain knowledge that they implement [3]. The situation in the practice is however different: units of knowledge are scattered between many program modules and a single program module contains more knowledge units (phenomena known as delocalization and interleaving) – e. g. Figure 1 illustrates how are multiple kinds of knowledge used in different program parts: different classes reference knowledge about the business domain (e. g. family), the programming technologies (e. g. XML), architecture (e. g. Visitor pattern), and Java core library. Thereby the structure of the program is only weakly related to the partition of the domain knowledge that it implements. By regarding programs purely from a structural point of view


Design Domain (e.g. Patterns)
Programming Technologies Domain (e.g. XML)
Java Domain
Business Domain (e.g. Family)

```
public class Person implements Element { ...
  public void accept (Visitor v) { ... }
  public Person clone() { ... }
}

public class XMLSerializer implements Visitor { ...
  public visitPerson(Person p) { ... }
}
```

Figure 1. Weaving of different kinds (dimensions) of domain knowledge in programs

we lose the connection with the domain knowledge. Instead of analyzing programs from the point of view of their structure, one could change the focus and regard the code primary from the point of view of the domain knowledge that it uses. Our aim is to obtain and use a new interpretation of the code given in terms of the domain knowledge (Figure 2). This high level interpretation is highly desirable (and more natural) since the end-user changes requests, technology changes, the need to reuse already written code are all expressed in terms of domain knowledge (about technical or business domain) and how it is reflected in the code.

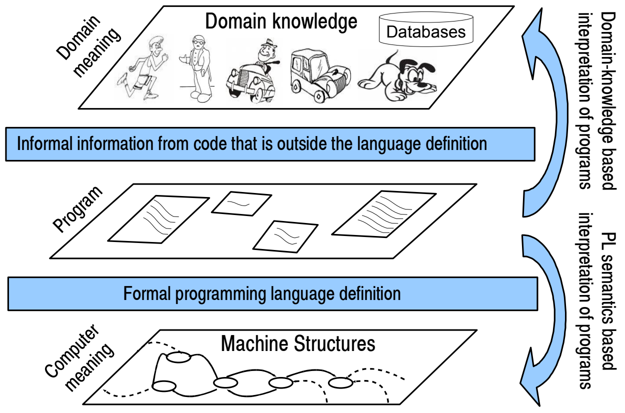

Domain meaning
Domain knowledge
Databases
Informal information from code that is outside the language definition
Program
Formal programming language definition
Computer meaning
Machine Structures
Domain-knowledge based interpretation of programs
PL semantics based interpretation of programs

Figure 2. Program meanings

Currently, the use of domain knowledge is only sporadically considered into automatic programs analysis practice. Instead of using the domain knowledge up-front in the analyses, today the reverse engineers implicitly interpret the domain knowledge agnostic analysis results from the point of view of his background knowledge. In automatic program analysis approaches (Figure 3-top), the domain knowledge is regarded as a helper in the analyses (at best), and beside several use-cases (concepts location being the most representative one), the domain knowledge is mostly ignored. In our vision (Figure 3-bottom) rather than being only a helper in analyzing programs, the domain knowledge should play a fundamental role in any kind of program analysis. Based on the domain meaning we can define completely new analyses and enhance the existent ones – e. g. the domain meaning can be used to lift the current analyses at the logical level such as: instead of assessing the structural modularity of a program we could assess its logical modularity, instead of detecting code clones we could to address logical redundancy, etc. We advocate that the systematic use of domain knowledge when analyzing programs can leverage the current analyses at a higher level of abstraction, this being the central aim of reverse engineering [4].

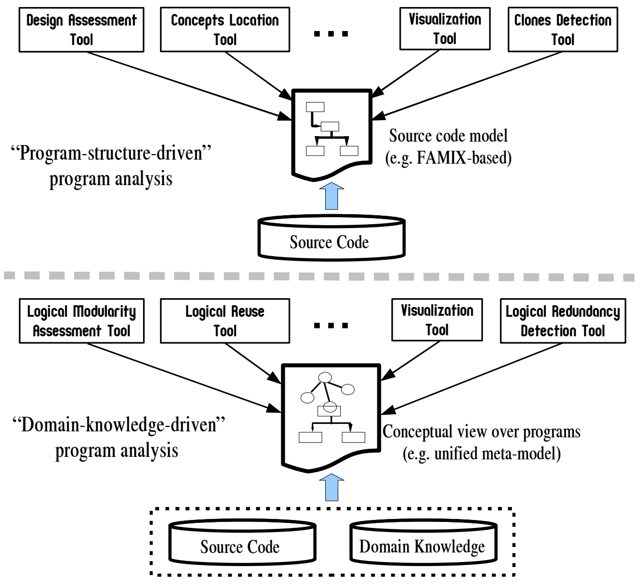

Design Assessment Tool
Concepts Location Tool
...
Visualization Tool
Clones Detection Tool
"Program-structure-driven" program analysis
Source code model (e.g. FAMIX-based)
Source Code
Logical Modularity Assessment Tool
Logical Reuse Tool
...
Visualization Tool
Logical Redundancy Detection Tool
"Domain-knowledge-driven" program analysis
Conceptual view over programs (e.g. unified meta-model)
Source Code
Domain Knowledge

Figure 3. Classical reverse engineering vs. domain knowledge driven program analysis

Outline. In the next sections we present a generic framework for assigning domain meaning to programs (Section 2), we discuss some challenges that need to be overcome in order to perform domain knowledge driven program analyses on a large scale (Section 3), we present islands of solutions for tackling these challenges (Section 4) and our call for a community effort (Section 5).

2 Assigning domain meaning to programs

In Figure 4 we present a generic framework to assign domain meaning to programs. The key parts in the framework are *the semantic domain*, *program abstraction*, and *interpretation* as we detail below.

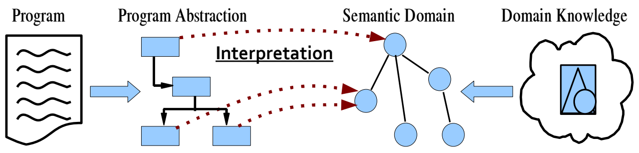


Figure 4. Framework for assigning (domain) meaning to programs

Program abstraction. The program abstraction is a model of the program that is rich enough to allow the desired analyses. The abstraction should not be too detailed to hinder the analyses due to too much information; at the same time, the abstraction should not be too coarse-grained to make the results of the analyses inaccurate.

Defining the program abstraction and extracting it with different parsing technologies from the code is one of the mostly studied issues in the reverse engineering. An example of an established program abstraction is the FAMIX meta-model [5].

Semantic domain. The semantic domain is an adequate representation of domain knowledge, that is used to interpret the program. The semantic domain should be flexible enough to represent accurately (i. e. with minimal encoding bias) concepts from a wide variety of domains (e. g. from business domains to programming technologies), at different levels of detail, from different perspectives (e. g. structure, dynamics), and it should be rich enough to enable the desired analyses.

Since in domain knowledge driven program analysis programs are interpreted from the point of view of the knowledge that they implement (rather than based on the semantics of the programming language) and since the domain knowledge is captured in the semantic domain, the limits of the semantic domain determines the limits of the logical analyses.

Interpretation. The interpretation is a mapping between the program abstraction and the chosen semantic domain. The mapping assigns meaning to the program and is the key ingredient for the definition of advanced analyses. In order to enable the analysis of big programs, we need to recover (or approximate) the interpretation automatically.

Recovery the interpretation is a problem similar to concepts location [9] and concepts assignment [2], whereby the concepts are a priori well defined by the semantic domain, belong to different kinds of knowledge ranging from business domain to programming (as shown in Figure 1) and the meaning of the mappings between the concepts and program parts is well defined. In domain knowledge driven program analyses, the interpretation is not seen as external analysis (like concepts location is depicted in upper part of Figure 3) but rather as fundamental (inherent) part of the programs representation (Figure 3-bottom).

Most of the reverse engineering approaches that use external knowledge to interpret programs (e. g. [1, 6, 7, 8, 12]) can be classified according to the framework from Figure 4. These approaches differ in the formality of the semantic domain (e. g. natural language documentation, weak ontologies, logics-based knowledge bases), in the definedness of the interpretation (e. g. traceability links, reference of individual concepts), or in the chosen program abstraction.

3 Challenges in domain knowledge driven program analysis

In the ideal way, domain knowledge driven analyses should be similar to the investigation of a project guru that knows exactly the structure of the program, the knowledge implemented in the program and how does it map to the program entities. The guru would further use these ingredients as basis for more complex analyses. The pure structural analyses approaches are a particular case of domain knowledge driven ones, in the sense that the domain knowledge is not used at all (called the "zero knowledge assumption").

From the above ingredients for implementing the domain knowledge driven program analyses, obtaining the program abstraction is by far the most understood. Both the making the semantic domain explicit, populating it with the knowledge, defining an adequate interpretation and recovering it automatically are open issues.

1) Expressing and obtaining the domain knowledge. A precondition of performing complex analyses is to have domain knowledge in a machine processable form. Obtaining the domain knowledge and sharing it are problems addressed by knowledge engineering (knowledge acquisition, representation and sharing).

Ontologies are envisioned to be fundamental means for sharing knowledge. However, there is much knowledge that can not be captured in ontologies – e. g. knowledge about the dynamics of a system, physical laws, complex relations, constraints, knowledge that is not lexicalized (knowledge units that do not have (yet) associated a name). Furthermore, obtaining adequate ontologies for program analyses is an open issue. Off-the-shelf ontologies that exist today[1] are highly inadequate for analyzing the programs (have other focus, are at another abstraction level, and are not detailed enough).

[1]`http://swoogle.umbc.edu`

2) Defining and recovering the interpretation. Defining and automatically recovering the exact mappings between the domain knowledge and the code is a difficult issue – for example, a domain concept can be defined, represented, encoded or only referenced in a part of a program. Depending on the exact interpretation we can apply different analyses over the program – e. g. whether a certain concept is redundantly defined, or only referenced in several places.

Once adequate interpretations are defined, in order to be applicable on large programs, we need to recover them in a (semi-)automatic manner. For most of the domain concepts (especially those of non-technical nature) the clues for recovering the interpretation stay outside of the programming language definition and mostly in informal sources of information – for example the names of program identifiers represent the most important clues. The recovery of interpretation based on this information is sensitive to the quality of identifiers and the modularization of the program. Therefore, in the most general case the interpretation can be only approximated. These facts lead to difficulties in recovering exactly the interpretation and subsequently performing automatic program analyses.

4 Approaching the challenges

Implementing domain knowledge driven program analyses in the widest sense is a difficult endeavor that would require human intelligence. However, depending on certain (classes of) use-cases even small and simple knowledge bases and rough approximations in the recovery of the interpretation function can bring substantial advantages.

4.1 Expressing and obtaining the domain knowledge

Expressing the semantic domain as light-weighted ontologies. Light-weighted ontologies (concepts and relations among them) can be used as a relatively simple and convenient way to express lexicalized parts of the domain knowledge.

Extracting domain knowledge by analyzing the similarities of domain specific APIs. Knowledge about programming technologies (e. g. XML, GUI, databases, communication) is used in virtually every program and therefore due to its pervasive occurrence in programs, many analysis use-cases involve it. Therefore, once knowledge bases that contain programming knowledge are built, they can be shared and re-used by a community of users for analyzing different programs.

Much of the knowledge of technical nature (e. g. GUI, XML, databases) is captured by the APIs that are well established in the programming community – e. g. the standard APIs of each programming language. We started to build a repository of programming technologies knowledge[2] that contain light-weighted ontologies extracted by analyzing the commonalities of multiple APIs that address the same domain. This represents only a first step in sharing the technical knowledge relevant for interpreting a wide variety of programs.

Building knowledge bases through reverse knowledge engineering. In the case when no knowledge base is available that fulfills our needs, we can perform reverse knowledge engineering in order to build the semantic domain by manually investigating the code and building fragments of the semantic domain (e. g. fragments of ontologies that contain the domain knowledge) [6, 11]. However, in order to do this we need adequate tool support and a well defined methodology.

4.2 Defining and recovering adequate interpretations

Defining the interpretations based on use-cases. Depending on the use-case, we need more or less powerful interpretations. For example, assessing the logical modularity of programs [11] requires a weaker interpretation as opposed to the assessment of domain appropriateness of APIs [10]. Depending on the powerfulness of the interpretation it can be easier (or not) to recover it from programs in an automatic manner.

Recovering the interpretation. One way to recover the interpretation is to use the information contained in the names of program identifiers or in the code documentation. The advantage is that in this manner we can recover information belonging to a wide variety of domains. However, this approach is very sensitive with respect to the quality of the identifiers. Other approaches that recover knowledge closer to programming (e. g. design patterns) use pattern matching for interpreting the code [8]. Finally, there are analysis approaches based on the manual definition of the interpretation [6, 12].

A minimal unified meta-model. In Figure 5 we present a minimal meta-model that contains the program abstraction and the domain knowledge as two graphs. The interpretation is given by a relation between concepts and program elements. Both the program graph and the knowledge graphs can be refined with more structure (FAMIX is an extension of the program graph), or enhanced with constraints.

[2]`http://www4.in.tum.de/~ratiu/knowledge_repository.html`

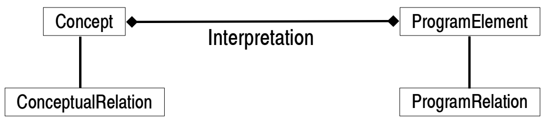


Figure 5. A minimal unified meta-model that considers programs abstraction and domain knowledge

5 The need for a community effort

Finally, we advocate that in order to systematically address these challenges and to use domain knowledge in automatic program analyses, we need a community effort. Systematic building of knowledge bases about programming technologies that are frequently used, annotations of standard APIs with the conceptual information, definition of interpretations adequate for performing certain analyses require efforts that can be addressed only in a community.

References

[1] G. Antoniol, G. Canfora, G. Casazza, A. D. Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Trans. on Sof. Eng.*, 28(10):970–983, 2002.

[2] T. J. Biggerstaff, B. G. Mitbander, and D. Webster. The concept assignment problem in program understanding. In *ICSE '93*, pages 482–498. IEEE CS Press, 1993.

[3] G. Booch. *Object-Oriented Analysis and Design with Applications (3rd Edition)*. Addison Wesley, 2004.

[4] E. J. Chikofsky and J. H. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, 1990.

[5] S. Demeyer, S. Tichelaar, and S. Ducasse. Famix 2.1 — the famoos information exchange model. Technical report, University of Bern, 2001.

[6] P. Devanbu, R. Brachman, and P. G. Selfridge. Lassie: a knowledge-based software information system. *Commun. ACM*, 34(5):34–49, 1991.

[7] M. T. Harandi and J. Q. Ning. Knowledge-based program analysis. *IEEE Software*, 7(1):74–81, 1990.

[8] W. Kozaczynski, S. Letovsky, and J. Q. Ning. A knowledge-based approach to software system understanding. In *KBSE*, pages 162–170, 1991.

[9] V. Rajlich and N. Wilde. The role of concepts in program comprehension. In *IWPC'02*. IEEE CS Press, 2002.

[10] D. Ratiu and J. Juerjens. Evaluating the reference and representation of domain concepts in APIs. In *ICPC'08*. IEEE CS, 2008.

[11] D. Ratiu, J. Jürjens, and R. Marinescu. The logical modularity of programs. In *Proceedings of the Working Conference on Reverse Engineering (WCRE'09)*, 2009.

[12] M. P. Robillard and G. C. Murphy. Representing concerns in source code. *ACM Trans. on Sof. Eng. Meth.*, 16(1):3, 2007.

Driving the refactoring of Java Enterprise Applications by evaluating the distance between application elements

Fabrizio Perin

Software Composition Group
University of Bern, Switzerland
`http://scg.unibe.ch`

***Abstract*—Java Enterprise Applications (JEAs) are complex systems composed using various technologies that in turn rely on languages other than Java, such as XML or SQL. Given the complexity of these applications, the need to reverse engineer them in order to support further development becomes critical. In this paper we show how it is possible to split a system into layers and how is possible to interpret the distance between application elements in order to support the refactoring of JEAs. The purpose of this paper is to explore ways to provide suggestions about the refactoring operations to perform on the code by evaluating the distance between layers and elements belonging those layers. We split JEAs into layers by considering the kinds and the purposes of the elements composing the application. We measure distance between elements by using the notion of the shortest path in a graph. Also we present how to enrich the interpretation of the distance value with enterprise pattern detection in order to refine the suggestion about modifications to perform on the code.**

***Keywords*-Reverse engineering; Java Enterprise; Architecture.**

I. Introduction

Since Java 2 Platform Enterprise Edition (J2EE) was introduced in 1999 it has become one of the standard technologies for enterprise application development. J2EE applications are complex systems composed using various technologies that in turn rely on languages other than Java, such as XML or SQL. In order to perform different kinds of analyses on JEAs it is important to collect information from all sources and put it together in a model that can include everything relevant. In this paper we focus our attention on analyzing the structure of JEAs. As known from the literature [1], [2], [3], [4] JEAs, by their very nature, can be split into layers. Each layer can differ from the others in the type of the elements that it contains or in the task that those elements have been created to perform. Evaluating the distance between different layers and between elements belonging those layers we can reveal violations of the application's architecture that should be modified in order to improve readability and maintainability.

In this paper we propose a technique to drive this refactoring. We describe the layering scheme that we adopt, the index that we use to identify architectural violations and how we can modify the interpretation of that index using enterprise pattern detection. We plan to implement our proposal in Moose [5], a software analysis platform, in order to exploit our existing infrastructure for static and dynamic analysis. Moose is a reengineering environment that provide several services including a language independent meta-model. On top of Moose have been build several tools that provide different services like: static analysis, dynamic analysis, software visualization, evolution analysis.

II. Layering

Java Enterprise Applications are complex software systems composed of different elements. Because those elements have different purposes and behaviors JEAs can be split into layers [1]. Comparing different types of layering schemes [1], [2], [3], [4] we split the applications into 4 layers: Presentation, Service, Business and Data layer. Every element belonging to a layer has a specific purpose and they work together to solve the user's problems. The Presentation layer contains all elements concerning the front-end of the application such as a rich UI or an HTML browser interface. The Service layer is part of the Business layer and contains all those elements that define the set of available operations and manage the communications between the Presentation layer and the domain logic classes. The Business layer includes all classes that implement and model the domain logic. The Data layer contains all classes that access the database and map the data into objects that can be used by the application. In Figure 1 illustrates the layering system that we adopt. The large external rectangles represent the layers.

When implementing a service it is always important to create a complete structure that involves all layers. There are two main reasons for this: the first, and most important one, is related to code understanding and the second to maintainability. If, for some reason, a service does not need to process some data, an element belonging to the Presentation layer can invoke directly an element in the Data layer. However in this way whoever will read the code will miss the domain model related to that particular service. So it becomes much more difficult to understand the code.

It is also important go through all layers for reasons of maintainability. If at the beginning of development it is not necessary to process some data, it could became necessary afterward and it is usually tricky to modify the structure of the code preserving understandability.

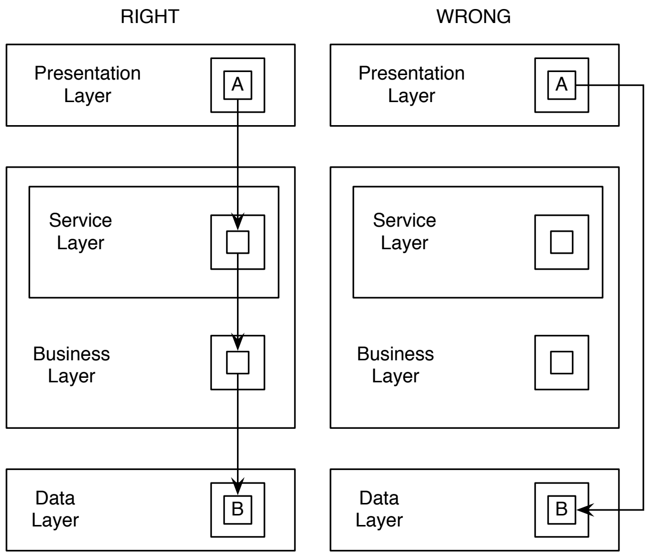


Figure 1. Layering scheme and basic invocation chian.

III. Distance between elements

We use value of the distance between JEA's elements to identify violations in the architecture. We compute the distance between elements using the notion of the shortest path in a graph. We can therefore use graph theory to solve our problem [6]. In the following we summarize the idea of distance used to solve the shortest path problem.

We have a weighted, directed graph $G = \langle V, E \rangle$, with a weight function $\omega : E \to \mathrm{R}$ mapping edges to real-valued weights. The weight of a path $p = \langle \upsilon_0, \upsilon_1, \upsilon_2, \ldots, \upsilon_n \rangle$ is the sum of all weights of its edges:

$$\omega(p) = \sum_{i=1}^{k} \omega(\upsilon_{i-1}, \upsilon_i) \quad (1)$$

The shortest path weight from u to υ is defined as:

$$\delta(u, \upsilon) = \begin{cases} \min\{\omega(p) : u \rightsquigarrow \upsilon\} & \text{if there is a path} \\ & \text{from } u \text{ and } \upsilon, \\ \infty & \text{otherwise} \end{cases} \quad (2)$$

Every edge in the graph has a weight that is necessary to find which is the shortest path between two elements. In our case we want to assign to each of them the weight 1. It is possible to calculate the minimum number of invocations from one method to another using Dijkstra's algorithm. We define the *distance* between two elements as the shortest path between those two elements.

We will define threshold values to determine which distance is correct and which is not. To simplify the concept in this work, even if the Presentation layer could contain HTML or JS Pages as well as GUI elements in any language, in the following we will consider an element contained in this layer as a class with methods.

The basic distance to calculate is the distance between different methods. The distances between classes and layers are derived from the distance between methods, therefore it is not necessary to apply the algorithm used to calculate distances on those methods, it is just necessary to regroup the methods of a path as part of a class or a layer.

In Figure 1 on the left we show the basic invocation chain that implements a normal user request. This chain contains classes and the smallest squares are methods or a generic JSP or HTML page. In Figure 1 on the right we also show what we consider to be a wrong invocation chain. In fact in this case the element A belonging to the Presentation layer invokes directly the method B belonging to the Data layer.

The distance between classes and layers is important to calculate because we cannot be sure that everything is fine just looking at the distance between methods. If the distance between methods has an acceptable value this doesn't means that every layer is touched in the implementation.

We will present in the following some examples that cover some normal cases that can be found in a normal implementation of an enterprise application. It is important to underline that the real threshold values to adopt to evaluate the code are still to be decided and they will be defined by analyzing some huge industrial case studies with a number of classes up to 1800. Below we exemplify our idea considering as right a distance value between two elements belonging the Presentation and the Data layer equals to 3.

Example 1: In Figure 1 on the left: The distance between method A and method B is 3 as well as the distance between the class that contains A and the class that contains B and the distance between the Presentation layer and the Data layer. We consider this situation a basic right implementation where all layers are touched. If instead we consider two classes, the first belonging to the Presentation layer and the second belongs to the Data layer, then if the distance between those classes is 1 it means that there is a direct invocation from the Presentation layer to the Data layer. This is the most basic and recurrent case of wrong implementation.

Example 2: In Figure 2 are shown a couple of correct paths. On the left side the distance between method A and method B is 4, instead the distance between the class that contains A and the class that contains B as well as the distance between the Presentation layer and the Data layer is 3. From those

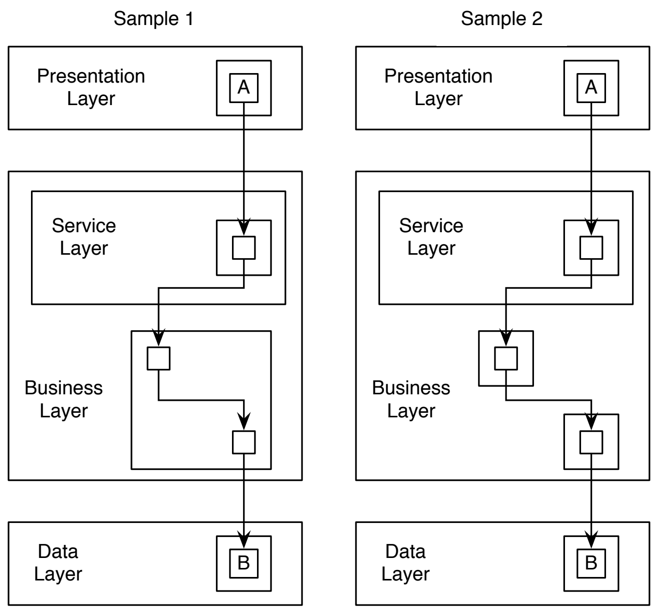


Figure 2. Layering scheme and basic invocation chian.

values we can see that all layers have been touched and that in a class there is an internal invocation because of the difference between the distance between methods and the distance between classes. On the right in Figure 2 all distance values are the same and equal to 4. Also in this case we can see that everything seems fine. What is important in a situation like this is that the number of invocations is not too high. A too high value is a symptom of complexity, so maybe that service implementation could be simplified.

Example 3: In Figure 3 on the left the distance between method A and method B is 3 so we consider it right. On the other hand the distance between their classes is just 2 as well as the distance between the Presentation layer and the Data layer. What is missing is a representation of the domain logic. On the right in Figure 3 the distance between A and B and the distance between the classes that contain them is 4 so it is right but the distance between layers is 2. What is missing is an entry point for that specific service because method A accesses directly a method belonging to the Business layer.

IV. Distance and Enterprise Patterns

There is a large body of development patterns gathered by the engineering community. There are patterns for enterprise applications [1] in general and patterns for J2EE [4] in particular. The description of design patterns provides information about the structure, the participant's roles, the interaction between participants and, above all, the intent for which they should be used. Our intent is to mix the value of the distance between elements and data source architectural patterns [1] tuning the results obtained by just looking for the distance index.

By being able to identify data source architectural patterns in the applications it is possible to provide more specific suggestions on the operation to accomplish during the refactoring. It is also possible to identify potential errors in a correct invocation sequence or vice versa.

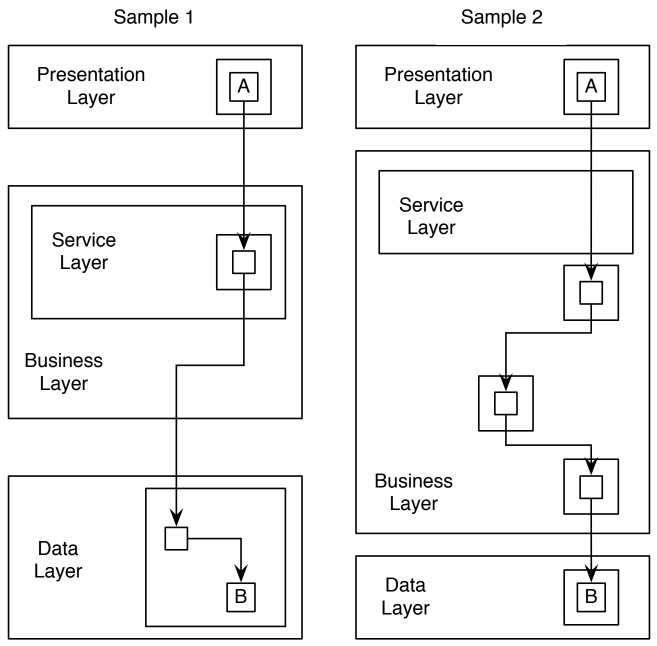


Figure 3. Layering scheme and basic invocation chian.

For example in Figure 1 on the right is shown a wrong invocation where the distance between A and B is 1. In this case the business logic is missing as well as the invocation to the Service layer that define the services available in the system. A standard modification to the code will be to implement one class belonging to the Business layer and another belonging to the Service layer. Supposing now that the class that contains B implements the pattern Active Record [1]. By it own definition, this pattern should contain some domain logic. So the modification to enact is not just to implement the missing classes in the Business and the Data layer but also to move the domain logic in the new class down to the Business layer.

V. Related Works

Some effort has been already spent in the context of architectural conformance checking [7], [8], [9]. In particular [9] extracting information from source code and byte code in Java and C++ and storing this information in a database that models all information that can be extracted from the code. From this information it is possible to perform different kind of analyses like checking illegal relationships within layers.

There are some differences between our work and [9]. We plan to apply our idea to an enterprise system that contains not only Java but also other languages such as XML, JS or SQL. The information harvested from the system will be modeled with FAMIX [10]. Our model will include all aspects concerning an enterprise application: the FAMIX meta model will not only contain structural information but also higher level information such as methods involved in a transaction. Using Mondrian [11] on the information contained in the meta model we can generate many different software visualizations on the code. Our intent is not only to identify which part of code could contain errors or inconsistencies but also to suggest possible modifications for the refactoring.

VI. Conclusion

In this paper we summarized our proposal to drive the refactoring of JEAs by comparing the distance between the application's elements to that of the data source architectural patterns.

In particular we presented the layering scheme that we adopt to regroup different elements of a JEAs. We also explore how to apply the concept of method distance to create an index that can be used to detect the presence of a wrong application structure. Finally we relate the concept of distance to the data source architectural patterns in order to modify the interpretation of the distances. In this case we propose to compile a catalogue of patterns and distances together with heuristics to drive the refactoring.

The basic idea is that every service has to be implemented touching every layer starting from the Presentation one. The catalog will be able to indicate what is wrong in the implementation and how the code should be modified in order to have a right structure. The approach could also be used to expose code that is too complex, *i.e.*, if the distance is too high.

We plan to implement our proposal in FAMIX [10] which already includes a generic meta-model for Object-Oriented application that can be extended to analyze Enterprise applications in Java. We want to refine FAMIX by adding all parts and relations that are necessary to model a JEA. Based on a consistent meta-model, it is possible to define a quality model based on metrics and pattern detection. We will evaluate the performance impact of calculating every time the distance or cacheing it. Another solution could be pre-compute all distances between all elements. In this case the Floyd-Warshall algorithm will fit better.

In order to validate this work we plan to perform experiments using an industrial partner we have been collaborating with.

Acknowledgments We gratefully acknowledge the financial support of the Hasler Foundation for the project "Enabling the evolution of J2EE applications through reverse engineering and quality assurance" (Project no. 2234, Oct. 2007 – Sept. 2010). We would also want to thank Tudor Gîrba, Oscar Nierstrasz for their comments on this paper and their support on this project.

References

[1] M. Fowler, *Patterns of Enterprise Application Architecture*. Addison Wesley, 2005.

[2] F. Marinescu, *Ejb Design Patterns: Advanced Patterns, Processes, and Idioms with Poster*. New York, NY, USA: John Wiley & Sons, Inc., 2002.

[3] K. Brown and G. C. et al., *Enterprise Java Programming with IBM Websphere*. Addison Wesley, 2001.

[4] D. Alur, J. Crupi, and D. Malks, *Core J2EE Patterns: Best Practices and Design Strategies*. Pearson Education, 2001.

[5] O. Nierstrasz, S. Ducasse, and T. Gîrba, "The story of Moose: an agile reengineering environment," in *Proceedings of the European Software Engineering Conference (ESEC/FSE'05)*. New York NY: ACM Press, 2005, pp. 1–10, invited paper. [Online]. Available: http://scg.unibe.ch/archive/papers/Nier05cStoryOfMoose.pdf

[6] T. H. Corman, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. MIT Press, 1990.

[7] G. Murphy, D. Notkin, and K. Sullivan, "Software reflexion models: Bridging the gap between source and high-level models," in *Proceedings of SIGSOFT '95, Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*. ACM Press, 1995, pp. 18–28.

[8] B. Laguë, C. Leduc, A. L. Bon, E. Merlo, and M. Dagenais, "An analysis framework for understanding layered software architectures," in *Proceedings IWPC '98*, 1998.

[9] W. Bischofberger, J. Kühl, and S. Löffler, "Sotograph – a pragmatic approach to source code architecture conformance checking," in *Software Architecture*, ser. LNCS. Springer-Verlag, 2004, vol. 3047, pp. 1–9.

[10] S. Tichelaar, S. Ducasse, S. Demeyer, and O. Nierstrasz, "A meta-model for language-independent refactoring," in *Proceedings of International Symposium on Principles of Software Evolution (ISPSE '00)*. IEEE Computer Society Press, 2000, pp. 157–167. [Online]. Available: http://scg.unibe.ch/archive/papers/Tich00bRefactoringMetamodel.pdf

[11] M. Meyer, T. Gîrba, and M. Lungu, "Mondrian: An agile visualization framework," in *ACM Symposium on Software Visualization (SoftVis'06)*. New York, NY, USA: ACM Press, 2006, pp. 135–144. [Online]. Available: http://scg.unibe.ch/archive/papers/Meye06aMondrian.pdf

Why and How to Substantiate the Good of our Reverse Engineering Tools?

David Röthlisberger
Software Composition Group, University of Bern, Switzerland
roethlis@iam.unibe.ch

Abstract

Researchers and practitioners are usually eager to develop, test and experiment with new ideas and techniques to analyze software systems and/or to present results of such analyzes, for instance new kind of visualizations or analysis tools. However, often these novel and certainly promising ideas are never properly and seriously empirically evaluated. Instead their inventors just resort to anecdotal evidence to substantiate their beliefs and claims that their ideas and the realizations thereof are actually useful in theory and practice. The chief reason why proper validation is often neglected is that serious evaluation of any newly realized technique, tool, or concept in reverse engineering is time-consuming, laborious, and often tedious. Furthermore, we assume that there is also a lack of knowledge or experience concerning empirical evaluation in our community. This paper hence sketches some ideas and discusses best practices of how we can still, with moderate expenses, come up with at least some empirical validation of our next project in the field of reverse engineering.

Keywords: reverse engineering, software analysis, empirical software engineering, evaluation, validation

1 Introduction

In recent years, the reverse engineering community invented and realized many interesting, promising and potentially practically useful techniques, concepts, processes, or tools such as Moose [4], Program Explorer [8], Graphtrace [7], or Mondrian [12]. Many useful visualizations of software systems, presenting on a high or low level both software structure and behavior have been invented (for instance, Polymetric Views [10], Class Blueprints [3, 9], Real-time visualization [15], or even 3D visualizations [5]).

While all these innovations are promising to actually support software developers in various software engineering tasks such as gaining an understanding for an unfamiliar application, analyzing collaborations between source artifacts, correcting software defects, or extending and enhancing key software features, nobody really knows whether any of those invented techniques contributes any value in practice, and if so, to which degree these techniques add value. Thus many questions are left open, for example how big is the impact of a given reengineering tool or technique on productivity or whether practitioners actually employ the concepts our community developed. Another interesting question is which techniques are best suited for which kind of software tasks, that is, which techniques or tools should a software developer use when faced with a task in areas such as defect correction, feature implementation, refactoring, or porting an application from one programming language to another. Certainly we are also interested in learning how a specific reverse engineering tool or technique is best employed in practice to be most helpful. As long as these questions remain unanswered and largely neglected, we cannot be sure whether our work has an impact on software developers working in industry on practical, real-world software development and maintenance problems and challenges.

For most of the fore-mentioned tools, techniques, or visualizations we completely lack quantitative knowledge about their practical usefulness. Thus we discuss in this paper several means how we can concretely acquire such crucial knowledge. The ideas presented here should serve as a thought-provoking impulse for how to evaluate and validate existing reverse engineering tools or at least newly invented and implemented tools. Moreover, we want to provoke a discussion in the community to sensibilize researchers and practitioners in our field for the importance of empirical research and to motivate them to actually perform such validation, that is, to conduct empirical studies and experiments. The ideas for empirical evaluations in our field presented in the following should give the basic knowledge about how to get started with empirical research.

This paper is structured as follows: In Section 2 we stress the importance of empirical validation with some concrete examples of the benefits of such validation. Section 3 presents and discusses different kinds of empirical validation, and how we can apply them to the context of reverse engineering research. Finally, Section 4 wraps up the paper with some concluding remarks.

2 Why Empirical Validation is Important

In this section we briefly list the three most important reasons why our community should really invest in empirical research:

- Validating our work is crucial, particularly in research. Empirical evidence is hereby a bold argument for the quality of our research projects.
- Quantitative empirical evidence allows us to compare different reengineering techniques, also with respect to types of software engineering tasks for which they are useful, and to thus develop guidelines in which kind of software tasks we should use which technique or combinations thereof.
- Our primary goal as reverse engineering researchers should eventually be to have the largest possible impact on software development and maintenance in practice. Only validating our tools empirically in practice can give us information about how to actually achieve this goal and to learn what to improve to maximize our impact on practice.

3 How to Conduct Empirical Evaluation

In this section we discuss different means to empirically validate tools and techniques developed to reverse engineer software systems. We start with means focusing on acquiring qualitative feedback assessing the usability of evaluated techniques while later approaches are also able to quantitatively report on the impact of these techniques on, for instance, productivity of software developers. Quantitative measures are considered as a more powerful validation as their informative value is higher [11], in particular as concrete numbers allow researches to compare the performance of different techniques. This is not directly possible with qualitative evaluations, yet they still provide important insights in a more explorative way than quantitative evaluations can do; that is why we consider both, qualitative and quantitative evaluation procedures, as valuable.

3.1 Surveys

A survey gathers data by asking a group of people their thoughts, reactions or opinions to fixed questions. This data is then collected and analyzed by the experimenters to obtain knowledge about how the interrogated persons consider, for instance, the usefulness of a visualization to understand run-time collaborations between different modules of a software system. Often surveys ask subjects to rate specific aspects or impacts of a technique or a tool in a Likert scale (typically from 1, "not useful", to 5, "very useful"). Analyzing surveys with such questions yields quantitative data, however, this data is based on personal and subjective judgement of the subjects. Thus the outcome of surveys cannot be considered as highly reliable as it is too much dependent on the specific subjects interrogated.

Surveys have several benefits as well as disadvantages: They are cost effective and efficient as a large group of people can be surveyed in a short period of time. The disadvantages of using surveys to conduct research include the validity based upon honest answers. Answer choices could not reflect true opinions and one particular response may be understood differently by the subjects of the study, thus providing less than accurate results.

3.2 Case Studies

Unlike a survey, a case study closely studies an individual person or an individual application to be analyzed. Surveys always ask several people and might consider several systems being analyzed or covered by the reverse engineering tools under study. Case studies were developed from the idea of a single case (*e.g.*, a single application to be analyzed by a new technique) being tried in a court of law. Comparing one case study to another is often difficult, thus it is usually not possible to draw a significant conclusion from one or from a low number of similar case studies. However, a case study can often be seen as a starting point for further analysis, for instance by testing how people use and interact with a new visualization tool or a new analysis procedure. A case study can thus serve as a pre-study to gather preliminary, qualitative feedback and insights to learn how to design another, possibly controlled study which eventually quantitatively evaluates the impact of the reverse engineering technique on the variable of interest (*e.g.*, programmer productivity).

The benefits of performing a case study thus include getting an in-depth view into subjects behavior and can also help to determine research questions to study for a larger group. A disadvantage of case studies is that the researcher may start off only looking for certain data such as specific usage patterns of the studied tool. Such a narrowed focus might lead to overlooking other interesting and important aspects, such as shortcomings or hidden benefits of the tool or technique under study. A way to mitigate this threat is to not conduct a case study with a specific goal, *e.g.*, testing a particular hypothesis. Rather researchers should employ case studies purely as an explorative means to unprejudicedly observe how a reverse engineering tool is perceived and used by study subjects to actually benefit from the advantages of case studies, for instance their potential to reveal unknown and unanticipated behavior of people concerning how they use a tool.

Table 1. Comparison of different kinds of empirical studies

Study	*Costs*	*Quantitative*	*Qualitative*	*Generalizability*	*Reproducibility*
Survey	+	Partially	Yes	+	-
Case Study	++	No	Yes	- -	- -
Observational Study	+	No	Yes	-	-
Contest	-	Partially	Partially	+	+
Controlled Experiment	- - -	Yes	No	++	++

3.3 Observational Studies

An observational study is similar to a case study. Typically, a observational study includes several single and possibly independent case studies with, for instance, several subjects, software systems to be analyzed, or analysis tools and techniques being used. Thus observational studies help experimenters to generalize the findings and results of one case study to a broader context.

Conducting observational studies has similar advantages and disadvantages as case studies. Of course they require more time and effort to conduct than just a single case study, the amount of insights and reliability they yield is larger and broader though.

3.4 Contests

Contests are another, rather specialized means to evaluate reverse engineering tools. The basic idea is to let different subjects or different analysis tools compete against each other, for instance by imposing some tasks or questions on the subjects that have to solve these problems under time pressure and while competing against other subjects. The eventual goal is to win the contest, for instance by solving the problem in the shortest amount of time or in the highest quality. The performance of each subject is analyzed, the experimenters observe the contest and try to locate the cause why and how much a specific reverse engineering tool helped subjects to solve the problem in a particular time or accuracy.

The fundamental difference to an observational study is the fact that a contest puts significant time pressure on the subjects; thus they are more motivated to solicit the best out of the available tools they obtain to solve the given problem. Thus the results might be more reliable and comparable between different employed tools as the motivation of the subjects was high to do their best to learn, understand and employ a tool in the most efficient manner.

3.5 Controlled Experiments

(Controlled) experiments are crucial in finding the answers to questions such as "does tool X provide value in software maintenance activities". The cause and effect of a particular problem can be studied through an experiment, providing it has "a set of rules and guidelines that minimize the possibility of error, bias and chance occurrences" [6]. Surveys and case studies require observation and asking questions, whereas experiments require controls and the creation of constrained situations to be able to actually record beneficial data.

The benefits of controlled experiments include a more scientific and thus more accepted approach of collecting data, as well as limiting potential bias that could occur in a survey, case study or an observational study. One disadvantage to many experiments is the cost factor involved; often laboratory procedures can be expensive. Another potential disadvantage is that a controlled experiment virtually always has a narrow focus, that is, it tries to study one or several very particular variables and thus neglecting any other, potentially interesting analysis of, for instance, developer reactions to specific characteristics of a reverse engineering technique.

Examples of well conducted controlled experiments in the area of software engineering and specifically in the field of program comprehension, software analysis, and reverse engineering are for instance: Cornelissen *et al.* [2] that evaluated EXTRAVIS, a trace visualizing tool, with 24 student subjects. Quante *et al.* [14] evaluated by means of a controlled experiment with 25 students the benefits of Dynamic Object Process Graphs (DOPGs) for program comprehension. Arisholm *et al.* [1] quantitatively analyzed in a large, long-lasting controlled experiment with 295 professional software engineers on three different levels (junior, intermediate, expert) whether pair programming has a positive impact on time required, correctness and effort for solving programming tasks with respect to system complexity and programmer expertise.

3.6 Summary

Table 1 summarizes the different kinds of empirical studies we presented in this section. The table shows for instance which studies to use to gather quantitative or qualitative feedback (note that each study can certainly be extended to gather qualitative feedback as well even though its primary goal is to quantitatively analyze cause-effect relationships, for instance by observing the study subjects or by additionally handing out a questionnaire). With "generalizability" we refer to whether the study concept itself easily allows us to obtain results that can be generalized to practical, real-world situations. Generalizability is certainly very much dependent on how a concrete study is designed. With "reproducibility" we suggest how likely a type of study is to yield the same results when conducted several times.

Researchers sometimes only require the use of one of the discussed methods to successfully find the answer to a question, however it is often worthwhile to combine different methods when evaluating a specific research project as each evaluation method has its specific strengths. Thus a combination of different methods often yields more insights and results that are more reliable and generalizable to other contexts, software systems, or analysis techniques.

Kitchenham *et al.* [6] thoroughly report on preliminary guidelines for empirical research in software engineering; these guidelines are also applicable to the field of reverse engineering and reengineering. Di Penta *et al.* [13] encourage researchers and practitioners to design and carry out empirical studies related to program comprehension and to develop standards how to conduct such studies. They give some early hints how to establish a community motivated to perform empirical engineering in our field.

4 Conclusions

In this paper we first motivated the need for conducting empirical research in the reverse engineering and reengineering field by raising some important questions that remain unanswered as long as only very little empirical validation on our techniques and tools is performed. In a second step, we introduced several types of empirical studies such as surveys, case studies, or controlled experiments that we can employ to actually conduct empirical studies. Moreover, we gave some hints how to concretely conduct which kind of study in which context and referred to important work on empirical software engineering done by other researchers.

Acknowledgments. We gratefully acknowledge the financial support of the Swiss National Science Foundation for the projects "Bringing Models Closer to Code" (SNF Project No. 200020-121594, Oct. 2008 – Sept. 2010).

References

[1] E. Arisholm, H. Gallis, T. Dyba, and D. I. Sjoberg. Evaluating pair programming with respect to system complexity and programmer expertise. *IEEE Transactions on Software Engineering*, 33(2):65–86, 2007.

[2] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering*, 2009. To appear.

[3] S. Ducasse and M. Lanza. The class blueprint: Visually supporting the understanding of classes. *Transactions on Software Engineering (TSE)*, 31(1):75–90, Jan. 2005.

[4] S. Ducasse, M. Lanza, and S. Tichelaar. Moose: an Extensible Language-Independent Environment for Reengineering Object-Oriented Systems. In *Proceedings of CoSET '00 (2nd International Symposium on Constructing Software Engineering Tools)*, June 2000.

[5] D. Holten, R. Vliegen, and J. J. van Wijk. Visual realism for the visualization of software metrics. In *VISSOFT*, pages 27–32, 2005.

[6] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. E. Emam, and J. Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Trans. Softw. Eng.*, 22(8):721–734, 2002.

[7] M. F. Kleyn and P. C. Gingrich. GraphTrace — understanding object-oriented systems using concurrently animated views. In *Proceedings of International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'88)*, volume 23, pages 191–205. ACM Press, Nov. 1988.

[8] D. Lange and Y. Nakamura. Interactive visualization of design patterns can help in framework understanding. In *Proceedings ACM International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'95)*, pages 342–357, New York NY, 1995. ACM Press.

[9] M. Lanza and S. Ducasse. A Categorization of Classes based on the Visualization of their Internal Structure: the Class Blueprint. In *Proceedings of 16th International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '01)*, pages 300–311. ACM Press, 2001.

[10] M. Lanza and S. Ducasse. Polymetric views—a lightweight visual approach to reverse engineering. *Transactions on Software Engineering (TSE)*, 29(9):782–795, Sept. 2003.

[11] T. C. Lethbridge, S. E. Sim, and J. Singer. Studying software engineers: Data collection techniques for software field studies. *Empirical Software Engineering, Springer Science and Business Media, Inc., The Netherlands*, 10(3):311–341, July 2005.

[12] B. A. Myers, D. A. Weitzman, A. J. Ko, and D. H. Chau. Answering why and why not questions in user interfaces. In *CHI '06: Proceedings of the SIGCHI conference on Human Factors in computing systems*, pages 397–406, New York, NY, USA, 2006. ACM Press.

[13] M. D. Penta, R. E. K. Stirewalt, and E. Kraemer. Designing your next empirical study on program comprehension. In *Proceedings of the 15th International Conference on Program Comprehension*, pages 281–285, Washington, DC, USA, 2007. IEEE Computer Society.

[14] J. Quante. Do dynamic object process graphs support program understanding? - a controlled experiment. In *Proceedings of the 16th International Conference on Program Comprehension (ICPC'08)*, pages 73–82, Washington, DC, USA, 2008. IEEE Computer Society.

[15] S. P. Reiss. Visualizing Java in action. In *Proceedings of SoftVis 2003 (ACM Symposium on Software Visualization)*, pages 57–66, 2003.