# Rapport de stage

# Layouts pour Roassal

Par

Dehouck Mathieu

Tuteur entreprise : Stéphane Ducasse, Usman Bhatti
Tuteur universitaire : Céline Kuttler
Juin 2013



# Remerciements

J'ai beaucoup appris au cours de mon stage parmi les membres de l'équipe RMoD que je tiens à remercier dans leur ensemble. Ils ont su faire preuve de patience et m'ont transmis une partie de leur connaissance.

Je tiens surtout à remercier Stéphane Ducasse et Usman Bhatti qui m'ont aiguillé tout au long du stage, Nicolas Anquetil et Martin Dias qui m'ont accueilli dans leur bureau et qui ont répondu à toutes mes questions, ainsi qu'Alexandre Bergel et toute la communauté Moose qui n'a eu de cesse de m'encourager et de me donner un précieux retour sur mon travail.

Je remercie aussi les enseignants de Licence Informatique pour cette année de cours.

# Résumé

Lors de mon stage de fin de licence, j'ai implémenté de nouveaux algorithmes de placement ( layout ) pour Roassal, un outil de visualisation écrit en Pharo ( un dialecte Smalltalk ) utilisé par l'équipe RMoD d'INRIA Lille. L'équipe RMoD travaille en outre dans la rétro-ingénierie logiciel, et il est souvent nécessaire de représenter visuellement les informations extraites des vieux logiciels que l'on souhaite analyser, c'est l'objectif de Roassal. Les layouts servent à organiser ces informations pour les rendre plus lisibles.

Roassal n'est pas utilisé que par RMoD, mais aussi par toute la communauté Pharo, et par des entreprises privées, ainsi mon travail n'est pas destiné à l'équipe RMoD uniquement, mais aussi à d'autres personnes aux Chili et en Suisse notamment.



# Abstract

During my internship I implemented new layout algorithms for Roassal, a visualisation tool written in the Smalltalk dialect Pharo, used by INRIA Lille team RMoD. RMoD team works in software reverse engineering, and it is often necessary to visually represent information extracted from old software we want to analyse, this is the aim of Roassal. Layouts goal is to organise these information in order to make them more easily understandable.

Roassal is not used by RMoD team only, but also by the whole Pharo community, and by companies, thus my work is not limited to RMoD team, but will be used by other people in Chile and Switzerland inter alia.

# Table des matières

# Introduction

Actuellement étudiant en dernière année de Licence Informatique. J'ai effectué mon sage de fin de licence à l'INIRIA ( Institut National de Recherche en Informatique et Automatique ) au sein de l'équipe RMoD, du 2 avril au 28 juin 2013.

Ce stage est ma première confrontation avec le monde de la recherche, et a confirmé mon envie de rejoindre le monde de la recherche académique.

Le but de mon stage est d'améliorer certains des algorithmes de placement ( layout ) présents dans Roassal et d'en implémenter de nouveaux. Ces nouveaux layouts ont pour la plupart été demandés part la communauté et doivent être suffisamment robustes et malléables pour pouvoir être utilisés pour représenter différentes informations.

Dans ce rapport je vous présenterai comment j'ai réalisé mon objectif. Je commencerai par introduire l'environnement de travail. Je présenterai ensuite les outils avec lesquels j'ai travaillé, et je montrerai les algorithmes que j'ai implémenté avec des exemples.

# 1 Contexte

Je vais maintenant vous présenter l'environnement dans lequel j'ai effectué mon stage, l'INRIA (Institut National de Recherche en Informatique et en Automatique), RMod, l'équipe que j'ai rejointe et Smalltalk, le langage avec lequel j'ai travaillé.

## 1.1 INRIA

### 1.1.1 Présentation

INRIA est un établissement français sous la double tutelle du Ministère de l'Enseignement Supérieur et de la Recherche et du Ministère de l'Industrie ( actuel Ministère du Redressement Productif ) dont le but est de soutenir la recherche dans les domaines de l'informatique, des technologies de l'information et de la communication (TIC). L'institut fournit également un transfert de technologie fort et porte une attention particulière à la formation par la recherche, la diffusion du développement scientifique et technique, l'expertise et la participation à des programmes internationaux.

### 1.1.2 Composition

INRIA accueille 3800 personnes dans ses huit centres de recherche situés à Rocquencourt, Rennes, Sophia Antipolis, Grenoble, Nancy, Bordeaux, Lille et Saclay. 2800 membres du personnel sont des scientifiques de l'INRIA et l'autre partie sont des partenaires d'organisations (CNRS, universités). Au total, ils travaillent dans plus de 160 équipes de recherche. Beaucoup de chercheurs de l'INRIA sont également enseignants et leurs étudiants (environ 1000) effectuent leur thèse au sein des équipes de projet de recherche de l'INRIA.

### 1.1.3 INRIA-Nord Lille

L'INRIA Lille - Nord Europe, dirigé par David Simplot-Ryl, rassemble 13 équipes de recherches. INRIA Lille accueille plus de 220 personnes, près de la moitié sont payés par l'institut. Le centre INRIA est un atout pour la compétitivité du Nord - Pas-de-Calais dans la recherche et l'innovation.

## 1.2 Equipe RMod

### 1.2.1 Présentation

L'équipe RMod travaille à la remodularisation des applications orientées objet. Cet objectif se traduit par de axe de recherches : la rétro-ingénierie logicielle pour l'analyse et la maintenance de vieux programmes, et la recherche sur les langages de demain.

Pour aider à la ré-ingénierie, de nouvelles méthodes d'analyses sont proposées afin de comprendre les grosses applications ( métriques, visualisations, etc ). Dans le contexte des langages de programmation des modules de vérifications sont étudiés.

L'équipe travaille également sur un nouveau noyau sécurisé pour Pharo, un environnement de développement intégré pour Smalltalk, utilisé et maintenu par l'équipe.

### 1.2.2 Remodularisation

L'évolution d'une application est limitée par des dépendances fortes entre ses versions antérieures. C'est pourquoi il est crucial de répondre aux questions suivantes :

- Comment pouvons-nous remplacer une partie par une autre avec un impact minimal ?
- Comment faire pour identifier les éléments réutilisables ?
- Comment modulariser une application quand il y a des liens incorrects ?

Répondre à ces questions est le but de Moose, l'environnement d'analyse de l'équipe, qui fournit une panoplie d'outils d'analyses. Ce travail est divisé en 3 parties :

- des outils pour comprendre le fonctionnement des grosses applications (packages/modules)

- des analyses pour une remodularisation
- des outils de qualité logiciel

### 1.2.3 Sémantique des éléments pour la modularité

Ce dernier paragraphe ce concentre sur la définition des éléments nouveaux pour la sémantique des langues dans le but de construire une application flexible et re-configurable. L'équipe travaille actuellement sur :

- la définition d'un langage uniquement basé sur Traits
- le rapprochement entre les langages réflexifs et la sécurité

## 1.3 Smalltalk

Le langage Smalltalk, autrement appelé Smalltalk-80, est un langage de programmation entièrement orienté objet, réflexif et dynamiquement typé, qui a vu le jour dans les années 80. Il influença de grands langages de programmation comme Python, Ruby ou encore Java.

Smalltalk est le langage de programmation utilisé par l'équipe, et est donc celui que j'ai moi même utilisé pendant mon stage.

### 1.3.1 Présentation de Smalltalk

Smalltalk est un langage de programmation orienté objet, réflexif et dynamiquement typé. Expliquons certains de ces mots :

- Orienté objet : en Smalltalk on manipule des objets par envoi de message, comme en Java ou C++.
- Réflexif : chaque objet peut inspecter et modifier sa propre structure pendant son exécution.
- Typage dynamique : les variables n'ont pas de type à l'initialisation, mais uniquement à l'exécution.
- Tout est objet : tout est objet, les classes, les méthodes, les messages ...

J'ai pour ma part pu apprendre ce langage avec Pharo By Example[**?**].

### 1.3.2 Quelques exemples

Déclaration de variable et initialisation :

```
| a |
a := 1.
```

Création et initialisation d'instance :

```
| a |
a:= Point new.
a x:1.
a y:2.
```

Tests d'égalités :

```
| a b |
a := 1@2.
b := 1@2.
a = b 'true'.
a == b 'false'.
```

Ouverture d'une fenêtre vide :

```
| aWindow |
aWindow := SystemWindow new.
aWindow openInWorld.
```

Ce que nous pouvons retenir de ces exemples ainsi que d'autres éléments de syntaxe :

- || déclare une variable
- := initialise une variable
- = tester l'égalité des valeurs de deux variables
- == tester si deux objets ont la même référence
- : façon de spécifier un paramètre aux méthodes
- self le receveur de la méthode, similaire à this en Java
- ^permet de retourner des valeurs, par défaut une méthode retourne self

# 2 Layouts pour Roassal

## 2.1 Roassal

Roassal est un outil de visualisation écrit en Pharo et intégré à l'environnement de programmation de Pharo. Il se présente sous la forme de deux fenêtres, une dans laquelle l'utilisateur écrit un script, cette fenêtre est appelée "easel", et une fenêtre de visualisation dans laquelle sont représentées les informations en suivant les spécifications de l'utilisateur.

Les données représentées par Roassal peuvent être très variées, ainsi il ne se charge que de les représenter et ne prend pas un "type" de données en entrée. Les données doivent être chargées dans Pharo par un outil tiers, Roassal créera ensuite son propre modèle ( des nœuds et des liens ) en se basant sur les méthodes du modèle avec lequel sont représentées les données.

Roassal a pour vocation de pouvoir représenter n'importe quel type d'informations, c'est une des raisons pour lesquelles il ne gère pas de modèle. Il est donc souhaitable de pouvoir diversifier les représentations de ces données autant que possible. Pour ce faire on a besoin notamment de placer les nœuds les uns par rapport aux autres de différentes manière, c'est de là que vient le besoin de layouts.

## 2.2 Objectif

Il existait déjà dans Roassal un layout d'arbre radial ( une façon de représenter les arbres avec la racine au centre et les différentes générations sur des cercles concentriques ) et un layout de force ( force-based en anglais, on considère les nœuds comme des particules chargées et les liens comme des ressorts, et on déplace les nœuds selon les forces qui s'y appliquent jusqu'à atteindre un certain critère : nombre d'itérations, minimum locale atteint ), mais leurs implémentations n'étaient pas suffisamment efficaces. Le première objectif ( défini avant le début du stage ) a donc été d'implémenter un nouveau layout d'arbre radial et un nouveau layout de force.

Par la suite, c'est en échangeant avec la communauté ( sur une mailing list ) que les besoins se sont révélés et que nous avons décidé des layouts à implémenter.

## 2.3 Méthodes de travail

Pour les différents layouts que j'ai implémenté, j'ai plus ou moins suivi le même processus. D'abord il convient de choisir un bon algorithme ( complexité de l'implémentation, complexité à l'exécution ), pour cela j'ai regardé des librairies, j'ai lu des articles de recherche.

Ensuite on fait une première implémentation, qu'il s'agit de débugger ( il y a toujours des bugs ) et d'améliorer jusqu'à avoir le résultat escompté.

Quand le code fonctionne, on fait un "commit", c'est à dire qu'on le met sur internet pour le rendre disponible pour les autres utilisateurs de Pharo. C'est une étape importante car ainsi les membres de la communauté peuvent tester notre code et nous donner du retour sur la mailing list. Avec ce retour on améliore encore le code jusqu'à avoir quelque chose qui répond au mieux aux besoins de la communauté.

Au début du stage, j'étais un novice en Smalltalk et je n'avais jamais codé dans un projet préexistant. A l'université on nous apprend à coder à partir de rien, parfois on nous fournit quelques parties de code, mais il s'agit finalement d'utiliser une API, alors que dans le cas de mon stage il s'agissait bel et bien d'écrire du code devant s'intégrer dans un ensemble plus complexe. Ainsi au début mon code n'était pas parfaitement intégré au reste de Roassal, mais en travaillant dans Roassal j'ai appris son architecture et j'ai retravaillé du code que j'avais déjà écrit pour qu'il se fonde mieux dans le code existant ( cette étape est appelée "refactoring" ).

## 2.4 Résultats

Nous avons écrit un article pour présenter notre travail à l'occasion de la $21^{eme}$ conférence internationale du Smalltalk organisée par L'ESUG ( European Smalltak User Group ) qui aura lieu à Annecy en septembre 2013.

Les coauteurs de cet article sont :

- Alexandre Bergel, le créateur de Roassal, ex-membre de l'équipe RMod, actuellement professeur assistant à l'Université du Chili à Santiago.
- Stéphane Ducasse, mon tuteur lors de ce stage, directeur de recherche, et chef de l'équipe RMod.
- Usman Bhatti, membre de l'équipe RMod avec lequel j'ai beaucoup travaillé lors de ce stage.

Comme l'article présente le travail effectué pendant le stage, nous l'avons inclus dans ce rapport. Nous ne l'avons pas traduit, et l'avons laisser tel qu'il sera présenté à l'ESUG.

# Pragmatic Visualizations for Roassal: a Florilegium

M. Dehouck U. Bhatti A. Bergel S. Ducasse

June 22, 2013

**Abstract**

Software analysis and in particular reverse engineering often involves large amount of structured data. This data should be presented in a meaningful form so that it can be used to improve software artefacts. The software analysis community have produced numerous visual tools to help understand different software elements. However, most of the visualization techniques, when applied to software elements, produce results that are difficult to interpret and comprehend.

This paper present five graph layouts that are both expressive for polymetric views and agnostic to the visualization engine. These layouts favor spatial space reduction while emphasizing on clarity. Our layouts have been implemented in the Roassal visualization engine and are available under the MIT License.

## 1 Introduction

Software analysis and reverse engineering large software system is a known to be difficult [DDN02]. Visualizing software eases analysis by using cognitive abilities to understand software and identify anomalies [Die07]. Visualizing software elements as a graph is a natural visual representation commonly employed:

- Graphs are relatively cheap and easy to visualize due to the amount of available dedicated libraries (*e.g.,* D3[1], Raphael[2]).
- Graphs are a structure effective to represent many different aspects of a software, including control flow and dependencies between structural elements.

Visualization techniques are known to be effective at analyzing package dependencies, correlating metric values, package connectivity and cycles, package evolution or the common usage of package classes (*e.g.,* [DLP05, LDDB09, vLKS+11]). A large body of existing work on software understanding is based on visualization approaches [HMM00, SvG05], in particular, on node-link visualizations [SM95, CIK03, KD03, HSSW06]. On one hand, some researchers explored matrix-based representation of graphs [HFM07, AvH04] or of software [MFM03] and its evolution [VTvW05].

[1] http://d3js.org
[2] http://raphaeljs.com

On another hand, important progress have been made to support navigation over large graphs and to propose scalable and sophisticated node-link visualizations for visualizing the connectivity graph of software entities [GFC05, HSSW06, Hol09].

The question here was not to invent new way of representing information, but to find relevant existing layouts and to implement them in Roassal, alongside basic Roassal basic layout such as grid, circle, tree. The novelty of our approach is that even when the nodes do not have same size they are drawn correctly.

We have thus proposed five new graph layouts, each one focusing on a particular aspect: the *radial-tree* focuses on representing hierarchies, while with regular trees the root is repulsed to the top, radial-tree keeps the root at the center of the visualisation. *Force-based* layout allows one to represent cyclic graphs such as dependency graphs. The *compact tree* family is just another implementation of trees using the same algorithm as radial-tree so that it saves space for large hierarchies. The *reversed radial tree* layout is another way of representing hierarchies where the position of an element does not depend on its depth but on its distance from the bottom of the graph. The *rectangle-packing* layout is an implementation of a rectangle packing algorithm to allow representing a lot of elements of different sizes in a reasonably restricted space.

To avoid confusion, we define terms used in this paper. A *layout* is an algorithm that determine position of the graphical elements contained in a visualization following some particular constraints. A *node* or *vertex* is basic element of a graph, typically in software analysis a package, a class or a method. An *edge* or a *link* represents a relation between two nodes, typically inheritance, composition or call. A *tree* is an acyclic directed graph, for example a simple object hierarchy. A *root* node is an entry point from which all the nodes are reachable by transitivity, typically the superclass of a class hierarchy.

In Section 2 we will introduce the problem and then in Section 3 describe the different layouts we have implemented, explaining for each the intention we had, the problems we encountered, the way we solved them and the limits of our solution.

## 2 Problem Description

In reverse engineering we deal with old and complex systems that are not understood easily. Moose provides powerful tools to analyse these pieces of software and we end up with large amount of data and metrics. But it is hardly more understandable, thus we need a way of having a quick and smart overview of the relevant information.

The solution is to map the most important textual information to graphical features, and to organize them to be easily readable. The aim of the layouts is to organize these visual information. As we may have to represent various kinds of data and we may want to focus on different features, hence it is necessary to have several layouts which will organize data.

The main constraint is computing duration, hence the choice of a pragmatic answer. For example, we give to the rectangle packing layout a desired size for the resulting rectangle and do not really compute the optimal arrangement which would have minimized the surface because it is time consuming.

# 3 A Florilegium of Visualizations

In this section we present some algorithms we added to Roassal. In particular we present the general intention of the algorithm, the main challenges it poses and the solutions we chose.

For each algorithm proposed here, we show the resultant layout with the Collection class hierarchy of Pharo: there are 131 classes in this hierarchy.

## 3.1 Radial Tree

**Intention.** When dealing with inheritance it is natural to have large trees, and the problem with regular tree representation is that the root is repulsed to the top of the visualisation. Sometimes we want to avoid that, and to keep the root amidst the visualisation. This is the aim of the radial tree.

**Difficulties.** There are several difficulties when drawing a radial tree.

- **Parent position node.** Firstly we had to choose if the position of a parent node would influence its children nodes position, or if the parent node position would be influenced by its children nodes position.
- **Supporting interaction.** Another constraint was that in Roassal we do not just want to represent data, but we also want to interact with them, so it was important to have an airy representation. It was the problem encountered with the old implementation, the representation was so compact that it was not possible to interact properly with the nodes.
- **Algorithm selection.** The last problem and maybe the most important one, is what kind of algorithm should we use to compute nodes position. If we choose to compute directly radial position for each node, as a circle has a finite perimeter then we take to risk of having to displace each node several times, that give a complexity in $O(n^2)$, and we can do better for such a layout.

**Solutions.** We propose a solution inspired by the modified version of Reingold-Tilford algorithm [BJL02]. We compute node position beginning at the leaves and then we ascend the tree to the root, displacing subtrees when they overlap. We do this in a Cartesian coordinate system, with some minor modifications to nodes position so that the radial tree looks nice at the end: typically the space between nodes depends on the layer they belong to. And then we transform our regular tree into a radial one wrapping the layers around the root (Figure 1).

**Limits.** This layout is interesting for visualizing reasonable hierarchies, since we want to interact with the nodes, there must be enough space between them, and so when there are many nodes on a layer, then the tree has an enormous diameter and the root remains all alone in the middle of the visualization, far from its children. This is the main drawback of the radial-tree layout.

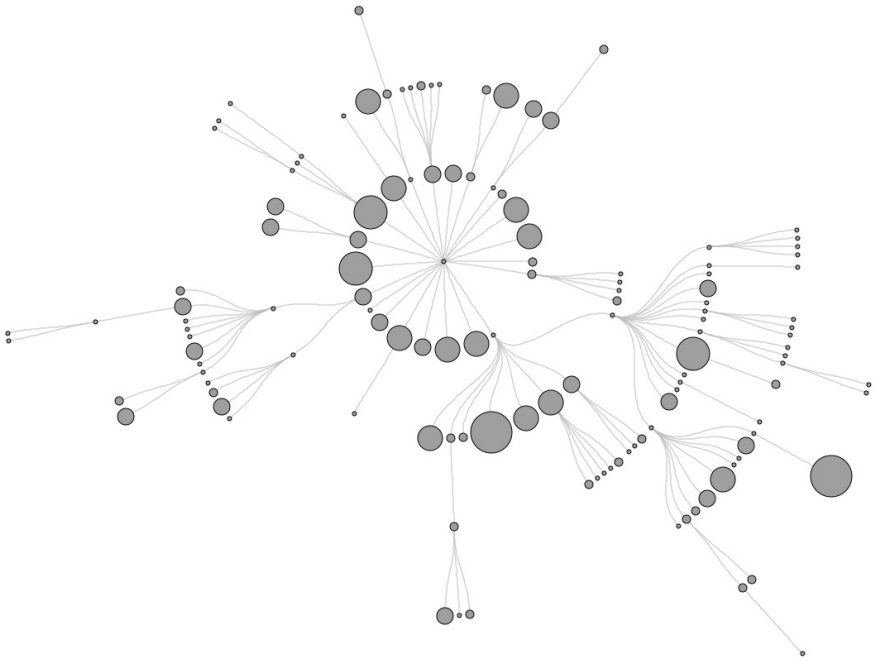

Figure 1: Radial-Tree Layout

## 3.2 Force Based

**Intention.** When dealing with methods invocations or module dependencies, trees are seldom encountered due to cyclic connections. And sometimes it does not make sense to give more importance to a node in particular, so the tree is not a good representation. Then we need a layout that treats all the nodes the same way and that does not break cycle sense. The force based layout considers nodes as repulsive charges and links as springs, then we have a representation which respect nodes connectivity.

**Difficulties.** The main problem of force-based layout algorithms is their temporal complexity which is considered to be $O(n^3)$ for the most trivial implementations, as each iteration has a quadratic complexity (we compute force action for each pair of nodes) and we must iterate enough times (which is thought to be of the same order as the number of nodes) to reach a local minimum. And since the goal is to represent big graphs, thus it is necessary to have a less time consuming algorithm.

**Solutions.** Our solution is inspired by D3 Javascript library implementation and the FADE algorithm [QE01]. Quadtrees reduce the number of calculi at each step and thus has a *O(n log(n))* complexity. It is also possible to specify charge for a particular node,

strength of a link, gravity center. Our force based layout is highly parametrisable, so that it is possible to focus on different aspects of a system (see Figure 2).

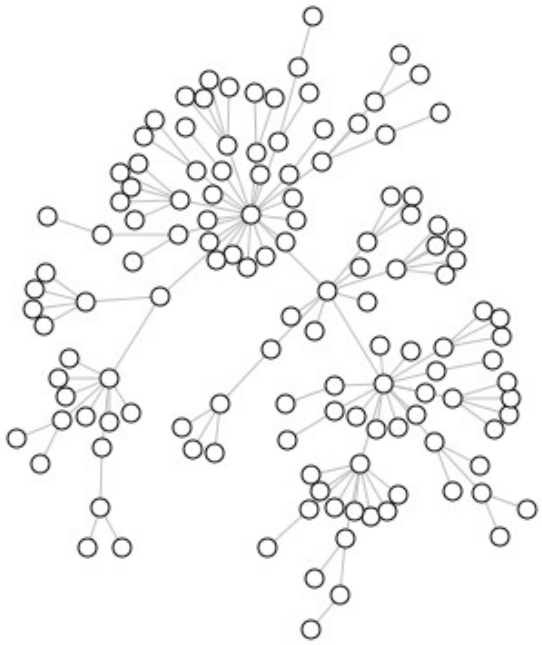

Figure 2: Force-Based Layout

**Limits.** Here the limit is the running time. Even with a complexity in *O(n log(n))*, large graphs takes much longer, and then it may be difficult to use it in live.

## 3.3 Compact Tree

**Intention.** There was already tree layouts in Roassal, but they make large graph since they only keep track of the biggest abscissa where a node has been set. Thus our goal here was to have a less space consuming algorithm, which permits to draw condensed tree when there is not much space.

**Difficulties.**

- **Vacant position.** A trap in this kind of algorithm is that we need to know for each layer the abscissa where we can set nodes, this can be done multiple ways, but the trivial way consists of checking all the previously set nodes, and then you

have a complexity in O($n^2$), which a loss of time in this case since tree can be drawn with a smaller complexity.

- **Node shifting.** Then as this algorithm is recursive, when setting a child node we do not know where the parent node will be set, and then when setting the parent node, sometimes it occurs that we have to move children nodes, and here the trivial solution has also a complexity in O($n^2$).

**Solutions.** Here we also use a Reingold-Tilford like algorithm with some improvements such as pointers for left-most and right-most children of a node. This is done so that we do not look at all the previously set nodes when we need to know where we can put a node. When placing a new node, we just skim the contour of the graph (the right-most and the left-most nodes of each layer) and it is less time consuming. In the same way, when we have to move children nodes to correspond to their parent node position, instead of moving them each time, the parent keeps a "modification" value, that spread to the children when they are drawn, once again it saves time (see Figure 3).

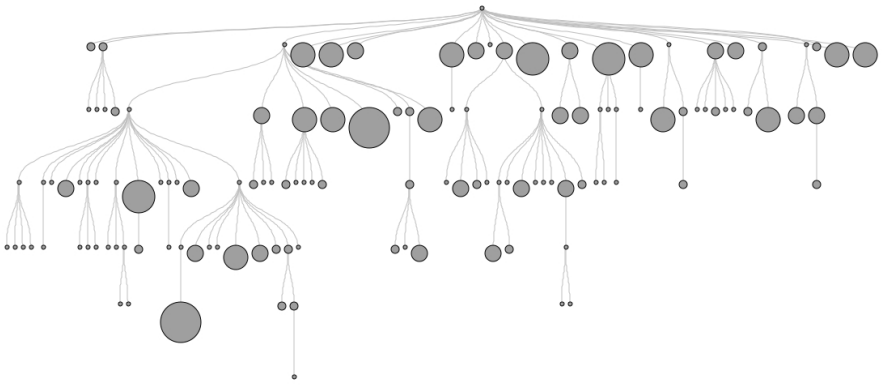

Figure 3: Vertical Compact Tree Layout

**Limits.** Our solution is pragmatic thereby computed tree is not the narrowest since even when children order is not important the tree is drawn as if the children were ordered. But then it would be necessary to go through the hierarchy several times to sort the nodes in order to have the narrowest tree, and it would have a high complexity.

## 3.4 Reversed Radial Tree

**Intention** The reversed radial tree layout is another tree layout for hierarchy representation, but when most of the trees focus on the distance between the nodes and the

root, the reversed radial tree layout focuses on the position of nodes compared to the whole tree, thus leaves are on the border regardless of their distance from the root.

**Difficulties** There is no real difficulties for the reversed radial tree layout. It is just important to avoid useless route in the graph.

**Solutions** We skim the tree from the leaves to the root, recording for each node the maximum distance to the leaves in the subtree induced by the node. And then as we have the list of leaves, we compute nodes position from the leaves to the root (see Figure 4).

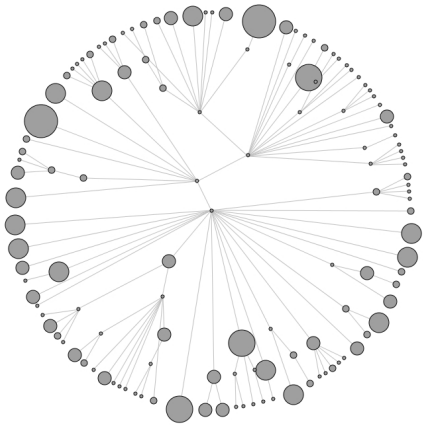

Figure 4: Reversed radial tree Layout

**Limits.** Here we have the same kind of problems as with the radial tree. As leaves are all on the border of the visualization, if there are lots of leaves (in a tree fine more large than deep), then the diameter of the visualization may be huge and the visualization may be almost empty, we will have lots of nodes on the border (the leaves) and very few nodes in the circle, with long edges between them.

## 3.5 Rectangle Packing

**Intention** Sometimes we want to represent a lot of elements of different sizes and a grid layout is not always a good choice as it does not use the visual space efficiently. The goal of the rectangle packing layout is to show many elements of various sizes in the available restricted visual space.

**Difficulties** The problem of rectangle packing is NP-hard, that means that we cannot find a solution in polynomial time but we cannot afford excessive computation time.

**Solutions** Here our solution is very pragmatic : instead of looking for the arrangement that will minimize the surface occupied by the elements, we provide the layout a ratio (2/3 by default), which corresponds to the width divided by the height of the rectangle we want to fill with our elements (Figure 5). Then the layout starts placing the elements and resize the containing rectangle until it has succeed in placing every element.

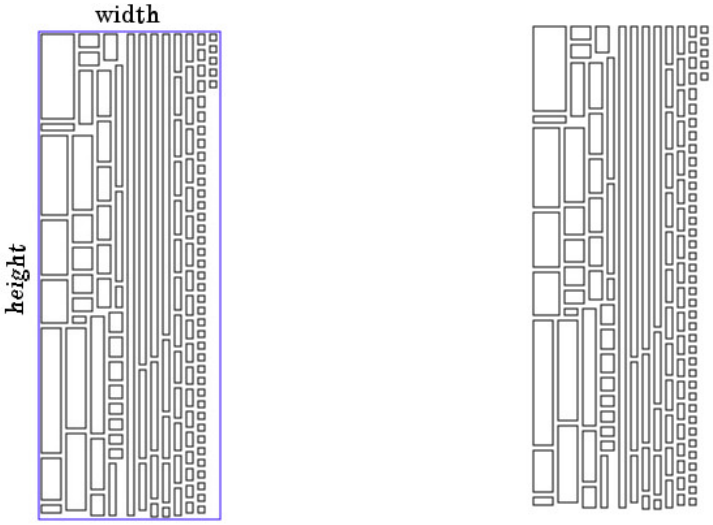


Figure 5: Rectangle Packing Layout

**Limits.** There are two limits :

- **Running time.** Even without looking for the best arrangement, it is time consuming. Thus it is difficult to apply it on a large number of elements.
- **Biggest elements.** We are dependent of the biggest elements (typically the longest and the widest). Sometimes we have little elements and a few big ones, then if we ask for a shape oriented in the other direction as the big ones, we will not have it. In our example, we provided the ratio 1/1 since we wanted to arrange elements in a square, but as there is a very long and thin one, we do not have a square at all, but a thin rectangle.

  Here we may raise the question of node resizing which is a touchy one, since we may break the sense originally provided by node size. And then, how do we resize nodes? Do we resize all the nodes, or only the biggest ones?

# 4 Conclusion

For large amounts of data, Roassal and similar visualization engines need to find a pertinent representation so that data are presented in a meaningful form and understood by the end users. In this paper, we have presented five graph layouts that are both expressive for polymetric views and agnostic to the visualization engine. The layouts favor spatial space reduction while emphasizing on clarity. Our solution is tractable and diverse, as the variety of layouts allows to analyze data in various forms. It should be noted that even with good layouts, if the amount of information is too big then it is hardly understandable, and the user has to himself select the most relevant information to be shown. We can make our layouts even more customisable, by for example proposing nodes staggering which can sometimes be a good way of saving even more space.


## Acknowledgements

This work was supported by Ministry of Higher Education and Research, Nord-Pas de Calais Regional Council, FEDER through the 'Contrat de Projets Etat Region (CPER) 2007-2013', the Cutter ANR project, ANR-10-BLAN-0219 and the MEALS Marie Curie Actions program FP7-PEOPLE-2011- IRSES MEALS.


# References


[AvH04] James Abello and Frank van Ham. Matrix zoom: A visual interface to semi-external graphs. In *10th IEEE Symposium on Information Visualization (InfoVis 2004), 10-12 October 2004, Austin, TX, USA*, pages 183–190. IEEE Computer Society, 2004.

[BJL02] Christoph Buchheim, Michael Jünger, and Sebastian Leipert. Improving walker's algorithm to run in linear time. In *Revised Papers from the 10th International Symposium on Graph Drawing*, GD '02, pages 344–353, London, UK, UK, 2002. Springer-Verlag.

[CIK03] Neville Churcher, Warwick Irwin, and Ron Kriz. Visualising class cohesion with virtual worlds. In *APVis '03: Proceedings of the Asia-Pacific symposium on Information visualisation*, pages 89–97, Darlinghurst, Australia, Australia, 2003. Australian Computer Society, Inc.

[DDN02] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.

[Die07] Stephan Diehl. *Software Visualization*. Springer-Verlag, Berlin Heidelberg, 2007.

[DLP05] Stéphane Ducasse, Michele Lanza, and Laura Ponisio. Butterflies: A visual approach to characterize packages. In *Proceedings of the 11th IEEE International Software Metrics Symposium (METRICS'05)*, pages 70–77. IEEE Computer Society, 2005.

[GFC05] Mohammad Ghoniem, Jean-Daniel Fekete, and Philippe Castagliola. On the readability of graphs using node-link and matrix-based representations: a controlled experiment and statistical analysis. *Information Visualization*, 4(2):114–135, 2005.

[HFM07] Nathalie Henry, Jean-Daniel Fekete, and Michael J. McGuffin. Nodetrix: a hybrid visualization of social networks. *IEEE Trans. Vis. Comput. Graph.*, 13(6):1302–1309, 2007.

[HMM00] Ivan Herman, Guy Melançon, and M. Scott Marshall. Graph visualization and navigation in information visualization: A survey. *IEEE Transactions on Visualization and Computer Graphics*, 6(1):24–43, 2000.

[Hol09] Danny Holten. *Visualization of Graphs and Trees for Software Analysis*. PhD thesis, Computer science department, 2009. ISBN 978-90-386-1882-1.

[HSSW06] Holt, Schürr, Sim, and Winter. Gxl: A graph-based standard exchange format for reengineering. *Science of Computer Programming*, 60(2):149–170, April 2006.

[KD03] Said Karouach and Bernard Dousset. Visualisation de relations par des graphes interactifs de grande taille. *Journal of ISDM (Information Sciences for Decision Making)*, 6(57):12, March 2003.

[LDDB09] Jannik Laval, Simon Denier, Stéphane Ducasse, and Alexandre Bergel. Identifying cycle causes with enriched dependency structural matrix. In *WCRE '09: Proceedings of the 2009 16th Working Conference on Reverse Engineering*, Lille, France, 2009.

[MFM03] Andrian Marcus, Louis Feng, and Jonathan I. Maletic. 3D representations for software visualization. In *Proceedings of the ACM Symposium on Software Visualization*, pages 27–ff. IEEE, 2003.

[QE01] Aaron Quigley and Peter Eades. Fade: Graph drawing, clustering, and visual abstraction. In *Proceedings of the 8th International Symposium on Graph Drawing*, GD '00, pages 197–210, London, UK, 2001. Springer-Verlag.

[SM95] Margaret-Anne D. Storey and Hausi A. Müller. Manipulating and documenting software structures using SHriMP Views. In *Proceedings of ICSM '95 (International Conference on Software Maintenance)*, pages 275–284. IEEE Computer Society Press, 1995.

[SvG05] Margaret-Anne D. Storey, Davor Čubranić, and Daniel M. German. On the use of visualization to support awareness of human activities in software development: a survey and a framework. In *SoftVis'05: Proceedings of the 2005 ACM symposium on software visualization*, pages 193–202. ACM Press, 2005.

[vLKS+11] Tatiana von Landesberger, Arjan Kuijper, Tobias Schreck, Jörn Kohlhammer, Jarke J. van Wijk, Jean-Daniel Fekete, and Dieter W. Fellner. Visual analysis of large graphs: State-of-the-art and future research challenges. *Comput. Graph. Forum*, 30(6):1719–1749, 2011.

[VTvW05] Lucian Voinea, Alex Telea, and Jarke J. van Wijk. Cvsscan: visualization of code evolution. In *SoftVis '05: Proceedings of the 2005 ACM symposium on Software visualization*, pages 47–56, New York, NY, USA, 2005. ACM.

## 2.5 Au delà des layouts

Lors de ce stage j'ai essentiellement travaillé sur l'implémentation de nouveaux layouts pour Roassal, mais j'ai aussi travaillé sur d'autres aspects de Roassal.

Un des objectifs de la communauté est de faire utiliser Athens ( une API pour dessin vectoriel sous Pharo ) par Roassal, ainsi les visualisations seront rendues en vectoriel. Mais pour l'instant, Roassal utilise encore Morphic ( l'API de rendu matriciel de Pharo ), il est donc nécessaire de coder dans Roassal une partie du traitement qui sera natif dans Athens et qui ne l'est pas dans Morphic.

J'ai donc implémenté un traceur de courbes de Bézier qui calcule une suite de segments, approximation de la courbe quadratique dont on a passé l'origine et la destination en paramètres. Les points de contrôle sont calculés différemment en fonction du layout.

J'ai ensuite également implémenté un traceur d'arcs de cercle. Pour pouvoir diversifier la représentation des liens entre les nœuds. Au début de mon stage il n'y avait en effet que les lignes droites, et la classe représentant les "escaliers" pour joindre les nœuds en utilisant uniquement des lignes verticales et horizontales, était boguée.

# Conclusion

## Bilan technique

### Langages de Programmation

Pendant ce stage j'ai codé en Pharo, un dialecte Smalltalk, un langage très différent de ce que l'on a vu en Licence ( C, Caml, Java ) où tout est objet. C'est un langage très intéressant et qui demande une approche différente du C ou du Java, du point de vu des structures de contrôle notamment.

Une partie des layouts que j'ai implémenté sont inspirés ( l'un d'eux est traduit ) de la librairie D3, j'ai donc du acquérir des bases de Javascript pour mener à bien ma mission.

### Langues Naturelles

L'équipe RMoD est connue au niveau international et comporte de nombreux membres étrangers, l'anglais est donc de rigueur dans l'équipe. De plus il y a plusieurs argentins au sein de l'équipe, j'ai donc pu parlé régulièrement espagnol. Ce fut donc un stage très enrichissant d'un point de vue des langues.

# Bilan humain

## L'équipe RMod

Les membres de l'équipe m'ont appris beaucoup, tant au niveau de l'ingénierie logicielle qu'au niveau de la recherche académique et de l'informatique en générale. Ce stage m'a conforté dans le choix que j'ai fait de devenir chercheur.

## La communauté Pharo

La communauté Pharo est très active. J'ai surtout communiqué avec la communauté Moose ( un sous ensemble de la communauté Pharo orienté rétro-ingénierie ) tout au long de mon stage. La communication un élément important de la recherche, en effet je n'étais pas le seul à travailler sur Roassal qui a plusieurs contributeurs réguliers, dont un en Suisse ( Tudor Girba ) et un au Chili ( Alexandre Bergel ).

# Glossaire

**Pharo** est une implémentation libre créée en 2009 sous licence MIT avec quelques parties sous licence Apache du langage de programmation Smalltalk.

**Roassal** est un outil de visualisation écrite en Pharo. Il se compose d'une fenêtre de script dans laquelle l'utilisateur spécifie la façon de représenter les données, et d'une visualisation où sont représentées ces données. Roassal n'importe pas de données directement, il crée son modèle en se basant sur le modèles disponibles dans Pharo.

**Licence MIT** ou licence X11 est une licence de logiciel libre et open source, non copyleft, permettant donc d'inclure des modifications sous d'autres licences, y compris non libres.

**Machines virtuelle** : utilisée pour exploiter les logiciels d'une machine qui n'existe plus dans le commerce (ordinateur, console de jeu, assistant personnel, ...), pour cacher la machine simulatrice et simuler une machine fictive, telle que la machine virtuelle Java.

**Layout** : façon d'agencer différents éléments les uns par rapport aux autres et respectant un certain nombre de contraintes. Par extension on appelle aussi **layout** les algorithmes qui permettent d'obtenir ces agencements et leurs implémentations.

**Layout de force**, ou **force based layout** : façon de placer des nœuds en considérant qu'ils sont tous porteurs d'une charge répulsive et que leurs liens sont des ressorts.

**Arbre** : graphe acyclique qui possède un élément à partir duquel on peut atteindre tous les autres ( en tant que graphe orienté ), on appelle **racine** le nœud à partir duquel tous les autres peuvent être atteints. Par extension, on appelle **arbre** la représentation de ces graphes.

**Arbre radial** : représentation d'arbre dans laquelle la racine est au centre, et les générations sont représentées sur des cercles concentriques.

Un problème de **packing** est un problème dont l'objectif est d'arranger un certain nombre d'éléments pour minimiser la surface occupée par ces éléments.