

Insert here your thesis' task.



CZECH TECHNICAL UNIVERSITY IN PRAGUE  
FACULTY OF INFORMATION TECHNOLOGY  
DEPARTMENT OF SOFTWARE ENGINEERING



Bachelor's thesis

# Layouting of Diagrams in the DynaCASE Tool

*Peter Uhnák*

Supervisor: Ing. Robert Pergl, Ph.D.

January 12, 2016



---

# Acknowledgements

Foremost I must thank my supervisor Ing. Robert Pergl, Ph.D., for providing me with an ongoing amazing and supportive environment not just during my work on this thesis, but ever since we have begun the development of DynaCASE. Without his dedicated support and involvement I wouldn't have been able to achieve what I have achieved.

I would like to also thank the whole Pharo Community that supplied me with constant motivation and a real sense of belonging.

Finally I want to express my thanks to my family, Andy and Elisa in particular, for supporting me and helping me stay sane during my studies.



---

## Declaration

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

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the “Work”), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity.

In Prague on January 12, 2016

.....

Czech Technical University in Prague

Faculty of Information Technology

© 2016 Peter Uhnák. All rights reserved.

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

### **Citation of this thesis**

Uhnák, Peter. *Layouting of Diagrams in the DynaCASE Tool*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2016.

---

# Abstrakt

V této práci se věnujeme problematice automatického grafického rozvrhování diagramů běžných v softwarovém a business inženýrství, specificky UML diagramům tříd a BORM OR diagramům. Představujeme nejmodernější techniky rozvrhování UML diagramů tříd, které ačkoliv jsou často studovány, tak stále představují značnou výzvu. Pro BORM OR diagramy analyzujeme rozvrhové potřeby BORM notace a představujeme jednoduchý algoritmus pro rozvrhování.

V rámci této práce implementujeme obě diagramové notace spolu s vhodným automatickým rozvrhovačem jako součást DynaCASE modelovací platformy.

Protože implementace plně automatických rozvrhovacích algoritmů představuje složitou problematiku, zaměřujeme se a implementujeme rovněž sadu polo-automatických a interaktivních řešení, které zjednoduší ruční rozvrhování.

Nakonec zhodnocujeme dosažené výsledky a komentujeme budoucí plány jak DynaCASE platformy, tak i automatického rozvrhování.

**Klíčová slova** DynaCASE, layouting, automatic layouting, UML, BORM, Class Diagram, Pharo, Smalltalk, Roassal, modeling, CASE

# Abstract

In this thesis we explore the problematics of automatic diagram layouting of common software and business engineering notations, namely UML Class Diagrams and BORM Object Relation Diagrams. We evaluate the current state-of-the-art of UML Class Diagrams layouting, which although well studied still poses a special challenge. For BORM Object Relation Diagrams we analyze the layouting needs of the notation and propose a simple algorithmic solution.

As part of this thesis we also implement both notations, together with fitting automatic layouts in the DynaCASE modeling platform.

Because implementation of fully automated layouting algorithms posses a special challenge, we also explore and implement more cost-effective solutions in the area of interactive and semi-automatic layouting.

Finally we evaluate achieved results and discuss the future work for both the DynaCASE platform and it's automatic layouting.

**Keywords** DynaCASE, layouting, automatic layouting, UML, BORM, Class Diagram, Pharo, Smalltalk, Roassal, modeling, CASE

---

# Contents

<b>Introduction</b>	<b>1</b>
Motivation . . . . .	1
Problem Statements . . . . .	1
Structure of the Thesis . . . . .	2
<b>1 Formal Foundations</b>	<b>3</b>
1.1 Graphs . . . . .	3
1.2 Drawings . . . . .	4
<b>2 Visual Aesthetics</b>	<b>7</b>
2.1 Aesthetic Criteria . . . . .	7
2.2 Graph Layouts . . . . .	10
<b>3 Layouting</b>	<b>15</b>
3.1 Topology-Shape-Metrics . . . . .	16
3.2 Planarization . . . . .	17
3.3 Mixed Upward Planarization . . . . .	18
3.4 Edge Labeling Problem . . . . .	21
3.5 Interfacing with GraphViz . . . . .	24
<b>4 Polylines</b>	<b>27</b>
4.1 Managing waypoints . . . . .	28
<b>5 Interactive Layouting</b>	<b>31</b>
5.1 Grid . . . . .	31
5.2 Snap To Grid . . . . .	32
5.3 Alignment Snapping . . . . .	33
5.4 Snap And Go . . . . .	33
5.5 Constraint Snapping . . . . .	34
5.6 Edge Label Constraint . . . . .	35

5.7	Rail Constraint . . . . .	37
<b>6</b>	<b>UML Class Diagrams Layouting</b>	<b>39</b>
6.1	Diagram, Subject, Model, Meta-model . . . . .	39
6.2	UML Class Diagram Notation . . . . .	41
6.3	UML Class Diagram Layouting . . . . .	43
<b>7</b>	<b>BORM Object Relation Diagrams Layouting</b>	<b>47</b>
7.1	BORM ORD Model . . . . .	47
7.2	Diagram overview . . . . .	49
7.3	Participant's finite state automaton . . . . .	50
7.4	Communications and data flows . . . . .	53
7.5	Layouting Algorithm . . . . .	54
7.6	Conclusion . . . . .	57
<b>8</b>	<b>Testing</b>	<b>59</b>
8.1	SUnit Tests . . . . .	59
8.2	Continuous Integration . . . . .	60
8.3	Visual Debugging . . . . .	61
	<b>Conclusion</b>	<b>63</b>
	Faced Challenges . . . . .	63
	Achieved Results . . . . .	63
	Future Work . . . . .	64
	<b>Bibliography</b>	<b>65</b>
<b>A</b>	<b>Acronyms</b>	<b>69</b>
<b>B</b>	<b>Contents of enclosed CD</b>	<b>71</b>

---

## List of Figures

1.1	A graph with simple (c) edge, directed edge (a), loop (b) and multiedge (d,e) . . . . .	4
1.2	Planar graph with non-planar (left), and planar (right) drawing . .	5
1.3	Kuratowski graphs $K_{3,3}$ and $K_5$ do not have any planar drawings .	5
1.4	Same graph with two different embeddings (edge ordering is counted clockwise) . . . . .	6
1.5	Dual graph of 1.4b . . . . .	6
2.1	(a) creates an impression that $b$ is connected to $a$ and $c$ , while in fact it is not (b) . . . . .	8
2.2	An orthogonal edge crossing with a bridge . . . . .	8
2.3	Haphazardly organized graph . . . . .	11
2.4	An expression tree . . . . .	11
2.5	Fig: Layouting collection of elements in a single line . . . . .	11
2.6	A multiline layout . . . . .	12
2.7	Examples of an alignment . . . . .	12
2.8	Horizontal, vertical, and radial tree layout . . . . .	12
2.9	Force-directed layout applied on a hierarchy of classes . . . . .	13
2.10	An example of improvements of a drawing . . . . .	14
3.1	Edge insertion and the routing graph[20] . . . . .	20
3.3	All considered label positions (a), and after removing overlaps (b)	24
3.4	A diagram layouted through GraphViz . . . . .	25
4.1	Polyline with activated handles . . . . .	27
4.2	Diagram of classes which provide polyline connection . . . . .	28
4.3	. . . . .	28
4.4	Dragging the midpoint down (a) will create a new waypoint and two new midpoints (b) . . . . .	29
4.5	Dragging handle (a) to position (b), after which the handle is removed (c) . . . . .	29

5.1	RTGridView . . . . .	32
5.2	A simple graph created without (a) and with (b) snap-to-grid . . .	33
5.3	yEd editor utilizing alignment snapping (top line) . . . . .	34
5.4	Examples of automatic distance constraints between objects in the yEd graphical editor . . . . .	34
5.5	UML Association with five different labels . . . . .	35
5.6	Properties of the constraint . . . . .	35
5.7	(a) initial placement of the label, (b) nudging from line orthogo- nally to it, (c) nudging from source element in the direction of the line . . . . .	36
5.8	UML Class Diagram utilizing the edge label constraint . . . . .	36
5.9	BORM <i>DataFlow</i> “pizza description” placed on a <i>Communication</i> line . . . . .	37
5.10	Rail constraint with visualized rail anchor and rod . . . . .	37
6.1	Roles of the subjects . . . . .	40
6.2	Model of a FAMIX metamodel including DynaCASE extensions . .	41
6.3	UMLEdge with various diagram settings . . . . .	42
6.4	Portion of DD/DI implemented in this thesis . . . . .	42
6.5	Notation and diagram elements used to visualize an Association element[29, B.2] . . . . .	42
6.6	Relations between model and its visualization . . . . .	43
6.7	Inheritance drawn without (a), and with (b) arc joining . . . . .	44
7.1	ORD diagram depicting various elements . . . . .	48
7.2	Metamodel of BORM Object Relation Diagram . . . . .	48
7.3	Comparison of participants organization . . . . .	49
7.4	Participant placed to the left of the primary one . . . . .	50
7.6	A participant with a back edge . . . . .	52
7.7	Possible visualizations of a service-oriented participant . . . . .	53
7.8	Communication with two data flows . . . . .	54
7.9	Addition of a line bend point to maintain line symmetry . . . . .	55
7.10	Back edge with a crossing . . . . .	56
7.11	Multiple participants with communications and data flows . . . . .	57
8.1	Hapao test coverage visualization applied on our FAMIX extensions	60
8.2	Considered positions during edge labeling . . . . .	62

---

# Introduction

## Motivation

Dissatisfied with the current level of modeling tools for software and business engineering alike, we have recently begun the development of a new open-source modeling platform named DynaCASE. The aim of the platform is to provide a moldable modeling environment for students, researches, and industry experts.

One of the problems modeling experts have to regularly deal with is the visual organization of model diagrams. As the visual organization has a major impact on both fast and accurate understanding, it is vital for the modeler to produce a good one. Such task is however very tedious and the resulting organization — layout is brittle. Even small change to the model may require complete redo, which a modeler may skip at the expense of letting the diagram's visual quality deteriorate. Likewise an automatic generation of diagrams from source code is more and more common, for such diagrams we do not have any layout at all.

To solve this problem, the art of automatic layouting has been of high interested and study. An automatic layout should be able to quickly turn diagram of arbitrary quality into a an aesthetically pleasing one, all while emphasizing aspects particular to the visualized domain.

As we consider it valuable, we would like to offer such automatic layouting in our DynaCASE platform.

We have chosen UML Class Diagrams and BORM Object Relation Diagrams as a representative diagram notations of software and business engineering.

## Problem Statements

The objective of this thesis is to analyze, design, and implement a set of automatic layouting algorithms for representative conceptual models for both

software and business engineering in the DynaCASE modeling platform.

As the platform is still in its early stages, the necessary models and notations are also to be implemented. Additional modifications and additions to DynaCASE may be required to accommodate the implementation. Finally tests should be performed of the achieved results.

DynaCASE[1] is implemented in the Pharo Smalltalk live environment[2] and uses the Roassal visualization library[3]; we therefore expect the implementation to be in this environment.

To summarize, in this thesis we:

- Review state-of-the-art of automatic diagram layouting with focus on UML Class Diagrams and BORM ORD
- Implement UML Class Diagram and BORM Object Relation Diagram model and graphical notations
- Extend the DynaCASE platform and its graphical framework to make the implementation of notations and layouting possible
- Design and implement a set of layouting algorithms for the implemented notations
- Test the software, and evaluate the results

## Structure of the Thesis

The structure of this thesis is as follows:

In chapter 1 we introduce basic vocabulary and definitions of graph theory and graph drawings necessary for understanding of layouting.

In chapter 2 we explore aesthetic requirements of graph and diagram layouting and the benefits of several basic layouts.

In chapter 3 we describe in detail two major layouting techniques — the topology-shape-metrics and edge label placement. We also briefly explore interfacing with external layouting libraries.

In chapter 4 and 5 we focus on layouting and other auxiliary utilities that help user achieve better layouts.

In chapter 6 we discuss the architecture behind the implementation of model and notation of UML Class Diagrams. Based on existing literature we present an overview of layouting requirements and algorithms of UML Class Diagrams layouting.

In chapter 7 we study the layouting needs of BORM OR Diagrams, based on our analysis we propose a simple layouting algorithm.

In chapter 8 we briefly explore some ways of testing used during implementation.

Finally in Conclusion we summarize achieved results and discuss future roadmap.

# Formal Foundations

As most of layouting rests on graph theory it is necessary to introduce some of its concepts that will be used throughout this work. Many introductions to graph theory should also cover this topic, such as [4].

## 1.1 Graphs

**Definition 1.1.1 (Graph).** A graph is an ordered triple  $G = (V, E, \psi)$ , where  $V$  is a set of vertices,  $E$  set of edges, and  $\psi$  incidence function between  $V$  and  $E$  —  $\psi(E) \subseteq V \times V$ .

Examples of incidences:

- $\psi(e_1) = (v_1, v_2)$  — edge  $e_1$  is between vertex  $v_1$  and  $v_2$
- $\psi(e_1) = (v_1, v_1)$  — edge  $e_1$  is a loop (self-edge) from vertex  $v_1$  to itself
- $\psi(e_1) = \psi(e_2) \wedge e_1 \neq e_2$  — two different edges have same incidences (a multigraph)

**Definition 1.1.2 (Graph (alternative)).** A graph is an ordered pair  $G = (V, E)$ , where  $V$  is a set of vertices, and  $E$  is a set of pairs of incident vertices  $E \subseteq V \times V$ .

**Digraph (Directed graph)** is a directed graph if the incidence pair is ordered. In such case the first vertex of the pair is the edge's source and the second its target.

**Undirected graph** is a graph whose incidence pairs are unordered.

**Multigraph** can contain multiple edges with same incidences.

**Simple graph** has no loops and no two edges connect the same (unordered) pair of vertices.

**Loop** (self-loop) is an edge that connects a vertex with itself.

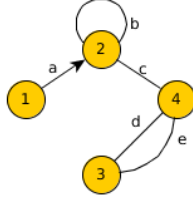


Figure 1.1: A graph with simple (c) edge, directed edge (a), loop (b) and multiedge (d,e)

**Degree** is the number of edges incident with a vertex. For digraphs we are also interested in **indegree** (number of incoming edges), and **outdegree** (number of outgoing edges).

**Walk** is a finite sequence of vertices and edges  $W = v_0 e_1 v_1 \dots e_k v_k$ .

**Trail** is a walk, where all edges  $e_i$  are distinct. (No edge is visited twice.)

**Path** is a trail, where also all vertices  $v_i$  are distinct. (Thus every vertex and edge in the sequence is only once.)

**Cycle** is closed trail (closed if  $v_0 = v_k$ ) with at least three edges (two for directed graphs).

**Tree** is an acyclic graph (no cycles are present) where any two vertices are connected by exactly one path.

**Root** is a selected vertex in a tree which in drawing acts as a topmost element.

**Leaf** is a tree vertex with degree of 1.

**Connected component**

A connected component in a graph is a subset of it's vertices and edges, where there is a *path* between any two vertices.

**Definition 1.1.3 (Subgraph).** A graph  $H = (V_H, E_H)$  is a subgraph of  $G = (V_G, E_G)$  (written  $H \subseteq G$ ) if  $V_H \subseteq V_G$  and  $E_H \subseteq E_G$ .

**Definition 1.1.4 (Vertex-induced subgraph).** Graph  $G_I = (V_I, E_I)$  is vertex-induced subgraph on graph  $G = (V_G, E_G)$  if  $V_I \subseteq V_G$  and  $E_I = \{e \mid e = (a, b) \in E_G \wedge a, b \in V_I\}$  — all edges from the original graph whose both endpoints are in the new vertex set  $V_I$ .

**Definition 1.1.5 (Edge-induced subgraph).** Graph  $G_I = (V_I, E_I)$  is edge-induced subgraph on graph  $G = (V_G, E_G)$  if  $E_I \subseteq E_G$  and  $V_I = \bigcup_{(a,b) \in E_I} \{a, b\}$  — vertex set of  $E_I$  endpoints.

## 1.2 Drawings

**Definition 1.2.1 (Curve).** Curve is a continuous mapping  $\gamma : [0, 1] \rightarrow \mathbb{R}^2$ , where  $\gamma(0), \gamma(1)$  are endpoints.

A curve is *closed* if  $\gamma(0) = \gamma(1)$ ; open otherwise.

A *Jordan curve* is a curve that has no repeated points (i.e. does not cross itself), except for the endpoints for closed curve.

**Definition 1.2.2 (Graph drawing).** A graph drawing  $\Gamma_G$  of a graph  $G = (V, E)$  is a mapping of each vertex  $v \in V$  to a point in a plane  $\mathbb{R}^2$ , and mapping of each edge  $e \in E$  to a *curve* with endpoints at the vertices incident to  $e$ .

**Definition 1.2.3 (Planar drawing).** A planar drawing is a drawing where each vertex  $v \in V$  is mapped to a distinct point, and each edge  $e \in E$  is mapped to an open Jordan curve. Additionally the following properties must be true:

- Drawings of any two distinct edges have no common points, except at common endpoints
- The drawing of an edge must not contain the drawing of a vertex

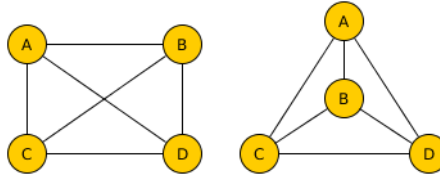


Figure 1.2: Planar graph with non-planar (left), and planar (right) drawing

**Definition 1.2.4 (Planar Graph).** A graph  $G$  is planar if and only if it has a *planar drawing*  $\Gamma_G$ .

It is easily observed that there is an infinite number of planar drawings for a planar graph (and drawings for any graph).

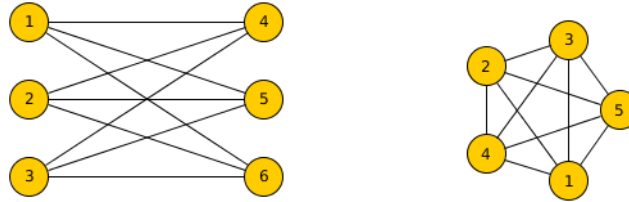


Figure 1.3: Kuratowski graphs  $K_{3,3}$  and  $K_5$  do not have any planar drawings

**Definition 1.2.5 (Planar Embedding).** A planar embedding of a graph  $G$  is a fixed ordering of edges incident to each vertex.

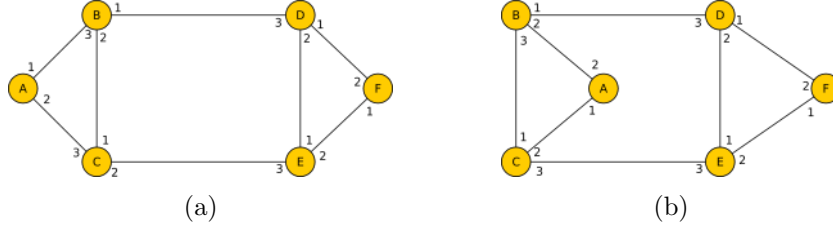


Figure 1.4: Same graph with two different embeddings (edge ordering is counted clockwise)

**Definition 1.2.6 (Face).** A face is a cycle in a *planar embedding* that has no edges going from the cycle into the region that is surrounded by the cycle. A face is adjacent to another edge if they share a common edge. The area not bounded by the embedding is called **outer face** (external face).

**Definition 1.2.7 (Upward Drawing).** A drawing of a *directed graph* is upward if the drawing of each edge  $e = (u, v)$  is monotonically nondecreasing in the  $y$ -direction from  $u$  to  $v$ .

**Definition 1.2.8 (Dual Graph).** Dual graph  $\overline{G}$  of a planar embedding of a planar graph  $G$  has a vertex for each face in  $G$ , and an edge for every edge in  $G$  that separates two edges. For every bridge in  $G$  the dual graph contains a self-loop.

We can also observe that two embeddings of the same graph will have different dual graphs.

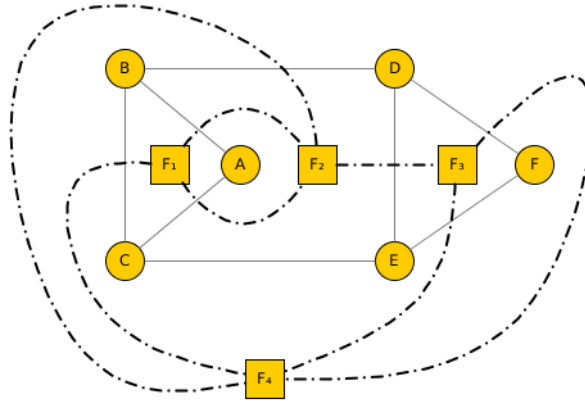


Figure 1.5: Dual graph of 1.4b

---

# Visual Aesthetics

Visual aesthetics is an important factor in the context of graphical layouting. The objective of achieving good visual aesthetics is to amplify human's natural cognitive abilities, while minimizing and mitigating factors that would impede them.

Good aesthetics can lead to a more understandable, concise, and perhaps even visually pleasing presentation, all while emphasizing valuable specifics of the particular domain. Bad aesthetics on the other hand can be confusing and even outright misleading.

In this chapter we will introduce the most commonly agreed and studied aesthetical criteria in the context of graph and diagram layouting[5] with special focus on criteria applying to UML diagrams[6][7], and especially UML class diagrams[8][9][10].

Still, aesthetics is a very vague term, and subjectivity to a certain extent is unavoidable. So although all the described criteria have *objectively measurable metrics* associated with them (such as measuring the total length of an edge), the applied value of any particular metric or combination of them is subjective and highly context-sensitive (“How long is *long* edge?”); all that in addition to the fact, that some criteria are directly competing with each other. For this reason the objective is often simplified to *minimizing* or *maximizing* particular metrics instead of reaching some specific targets.

## 2.1 Aesthetic Criteria

### 2.1.1 Area

The area of the drawing should be kept small enough, that it is possible to see the full drawing. The available area is determined mostly by the target medium. If UML diagrams are viewed mostly on computer screens, they should fit on the screen without the need of scrolling or zooming. If it is

common for other diagrams to be printed on paper, the diagram should fit there fully while being readable without any aids (magnifying glass).

### 2.1.2 Overlapping

No two elements in the drawing should overlap each other. Breaking this rule in many diagrams results in completely misleading visualization. For example if one vertex is hidden behind another or is visually connected to an edge, even though no actual connection exists in the graph.



Figure 2.1: (a) creates an impression that  $b$  is connected to  $a$  and  $c$ , while in fact it is not (b)

### 2.1.3 Edge Crossing

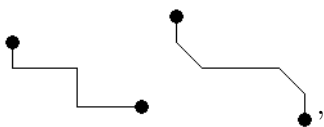
Similarly, edge crossings should be avoided, as it may not be obvious which way the edge continues. If a crossing is unavoidable, orthogonal crossings are preferred. Edges should never overlap each other as it makes them indistinguishable; in some special cases this however could be beneficially exploited, such as joint inheritance arcs 6.3.1. Some tools even add small visual *bridge* to further distinguish the lines.



Figure 2.2: An orthogonal edge crossing with a bridge

### 2.1.4 Edge Bends and Angles

Some notations permits the usage of polylines — edges composed of multiple straight segments. The number of such segments (edge bends between them) should be minimal as it breaks the natural flow of the line; it forces the user to manually trace the edge instead of judging the target by the line's initial direction. However, if edge bends are present, the angles between the segments should be uniform throughout the whole diagram. Most commonly used are right angles (orthogonal/rectilinear) and  $45^\circ$  (octilinear).



### 2.1.5 Orthogonal Drawing

Orthogonal drawings is the most commonly utilized and studied specialization of *Edge Bends and Angles* 2.1.4, where all the polylines in the diagram have orthogonal segments. As wide range of diagrams utilize this format it is also familiar to the reader. Furthermore vertical and horizontal lines are easier to trace visually than arbitrarily sloped ones.

### 2.1.6 Edge length

The overall length of edges should be kept small (cf. proximity 2.1.7). As edges in many diagrams represent secondary information (with vertices the primary one), the edge length tends to be the most often sacrificed criterion so other criteria (mainly edge crossings, bends, and overlapping) can be improved. Edge lengths also impact separation of objects.

### 2.1.7 Grouping and Proximity

Elements that are in close proximity or grouped together are perceived as related to each other. Therefore elements should be placed close to each other if they are related, especially if they are not visually connected, and unrelated elements in a group should be separated.

### 2.1.8 Similarity

Elements that are similar in some visual aspect are also considered related or similar. For example one can color related elements to add another dimension of relation between them.

### 2.1.9 Shape

Shapes are the heart of diagram visualizations. They allow to express properties and concepts of the underlying model visually without the need to resort to textual explanation. Generally the objective is to represent as much information as possible through pure visuals, although this is usually given by a diagram notation. Therefore this criterion should be achieved at the level of the diagram's notation rather than during layouting.

### 2.1.10 Symmetry

Symmetry produces a more visually pleasing drawing and tends to evenly distribute elements throughout the drawing.

### 2.1.11 Hierarchy

Elements visually placed higher are considered more important, therefore hierarchies should be represented as such. E.g. in UML class diagrams super-classes should be placed above their subclasses. In some cases the hierarchy can be expressed sideways, with the most important element on the left and spreading to the right. The left-to-right rule stems at least partially from our<sup>1</sup> left-to-right reading.

### 2.1.12 Mental Map

If new elements are added or removed from the drawing, application of the layout shouldn't diversify greatly the whole drawing. The user has already created a *mental map* of the diagram in his mind and can usually remember where a particular element is. Therefore after making a change and applying a layout, the user will expect previous elements at the same place as before. However, if the mental map is violated, the user is required to search for the elements, slowing the user down and increasing cognitive load.

### 2.1.13 Direction of flow

Many diagram notations benefit from uniform flow of edges.

Notably generalizations in UML class diagram are oriented upwards (cf. Hierarchy 2.1.11). This allows for easier identification when multiple types of edges are present.

---

A particular diagram notation can employ additional rules or ignore some; as such the criteria must be considered for each notation separately.

## 2.2 Graph Layouts

In this section we will explore several layouts available in the Roassal library from their visual perspective and value. The details of implementing Layouting algorithms are presented in the Layouting chapter<sup>3</sup>. For a complete overview of all Roassal layouts refer to the Layouting chapter of the AgileVisualization book[3].

---

We will start with random layout of a graph as an arbitrary layout. The vertices (nodes) are placed haphazardly and no sense can be gained from it.

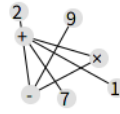


Figure 2.3: Haphazardly organized graph



Figure 2.4: An expression tree

But by applying a tree layout to the graph in 2.3 the following structure emerges:

While the first figure left us clueless, from the second one we can infer that we are dealing with an expression tree. This is the main objective of layouting; to present information in useful, visually pleasing, and human-cognition friendly format.

### 2.2.1 Vertex-based Layouts

More sparsely used, but still useful are vertex-based layouts. Such layouts do not use any information about edges (whether edges are present or not), and instead are focused on properties of the vertices themselves, such as their size, shape, or number.

The simplest example is line (column) layout. A line layout organizes all elements in a singular line (ditto for columns).



Figure 2.5: Fig: Layouting collection of elements in a single line

More advanced version is multiline (multicolumn) layout. In this layout the total width of the drawing is limited, either by the absolute width (in screen pixels), or by the number of elements. By default the layout will try to distribute the elements evenly in both (x, y) directions.

The last option is a *grid based layout*. All elements are placed in a grid of fixed size.

Good interface of such layouts should also provide the option to specify certain aspects such as the gap size between elements, overall size constraints, alignment of non-uniform elements, and other.

<sup>1</sup>Western; how are cultures reading right-to-left or top-to-bottom affected is unclear.

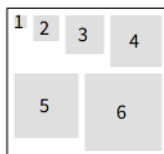


Figure 2.6: A multiline layout



Figure 2.7: Examples of an alignment

### 2.2.2 Graph Layouts

Graph layouts take both the vertices and the edges into consideration.

#### 2.2.2.1 Tree layout

For obvious reasons this is mostly used to visualize graph trees. The root is placed at the top of the drawing with children expanding downwards. It is useful for visualizing trees, hierarchies, and other acyclic structure. Roassal offers several variations of the layouting — top-down, left-to-right, and radial. Radial tree places the root in the center with children around in concentric circles. Radial tree is also the most space conservative.

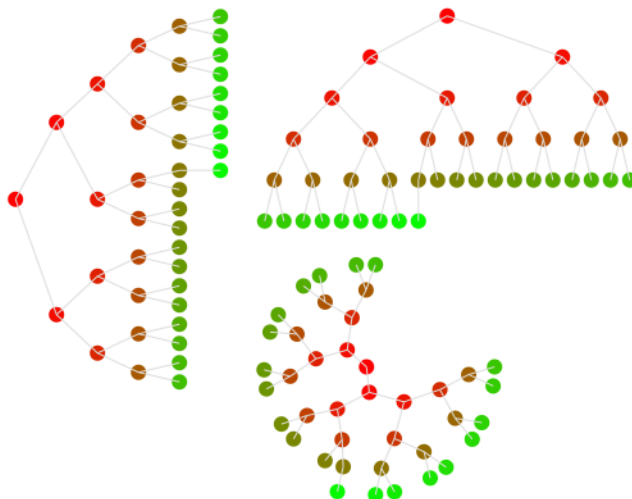


Figure 2.8: Horizontal, vertical, and radial tree layout

Sometimes even regular graphs can be visualized with tree. In such cases a *cycle removal* must be performed first. After the cycles are removed the layouting can be applied. After that the edges removed earlier are added back. This can be used for example for visualizing software dependencies, using *Dominance tree layout*[11]. As *good* dependencies are all uniformly oriented, any violation (dependency cycle) will be immediately visible by arrows (edges) going in the opposite direction.

### 2.2.3 Force-directed Layout

Force-directed layout is based on physics model of repulsion and attraction of particles. Each vertex is negatively charged and thus repulsing other nearby vertices. Edge between vertices works as a bond, an attractor. This method is especially effective for large and dense diagrams. The result is an aesthetically pleasing visualization that emphasizes clustering of elements. Note that since each vertex affects every other vertex, layouting for large graphs can be computationally demanding; thus it is often performed in multiple steps where each step gets closer and closer to force equilibrium. Also additional heuristics can be employed that limit the applied forces only to immediate vicinity[5][12].

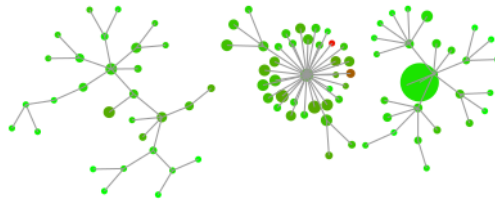


Figure 2.9: Force-directed layout applied on a hierarchy of classes

### 2.2.4 Sugiyama Layout

The Sugiyama layout[13] is an approach to visualizing hierarchies. It focuses on improving many aesthetical criteria such as edge crossing minimization, edges lengths, uniform flow and other.

The layouting algorithm assigns each node a layer based on their hierarchy within the graph. If a hierarchy spans several layers, dummy vertices are added to avoid overlapping. Then nodes in each layers are permuted to minimize edge crossings; finally, if possible, nodes are adjusted so dummy vertices are no longer needed.

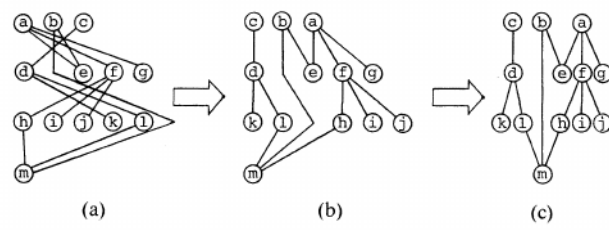


Figure 2.10: An example of improvements of a drawing

# Layouting

Three concepts will be discussed in this chapter:

1. Layouting of directed and undirected graphs using the topology-shape-metrics approach, and
2. Edge Labeling Problem
3. Integration with third-party layouting tools

Those choices were made to represent two distinct problems in the area of layouting and by no means attempt to cover everything, and in fact we have achieved our objectives only with limited success.

As Spönemann writes in his recent (2015) thesis [14]: “There is a striking contrast between the abundance of research results in the field of graph layouting methods and the current state of graphical modeling tools, where only a tiny fraction of the results are adopted.”, the author then follow with several explanations, quoted in full (our own notes are in italics):

- **P1** Users are skeptical about the quality of automatically generated layouts [...]
  - *We note that for static layouts the situation is often “make or break”; either the layout is fully self-sufficient or not useful at all.*
- **P2** Domain-specific layout constraints are not adequately addressed by the established methods. In particular, constraints on the positioning of *ports* (connection points of edges) are hardly considered.
  - *Researched layout algorithms often solve only parts of the associated problem, such as planarization and orthogonalization for graphs with vertices of degree 4 without addressing the real need of vertices or arbitrary degrees, ignoring needed requirements such as label placement, etc.*

- **P3** Graph layout algorithms are complex and thus their implementation requires considerable effort. For the same reason, commercial graph layout libraries are quite expensive.
- **P4** The integration of graph layout libraries written in C or C++, e.g. Graphviz [15] or OGDF<sup>2</sup> [16], into other platforms such as Eclipse is an intricate task.
  - *We will briefly elaborate on integration with the tools later.*
- **P5** Users are quickly overwhelmed by the multitude of graph layout methods and their configuration parameters. The usage of many parameters requires a detailed understanding of the underlying methods.

We fully agree with all the points as we were majorly impeded by the underlying problems, as stated in the introduction. Despite the limited success of the implementation we have gained deeper insight into the problems and can better accommodate and address them in the future.

For an extensive overview of layouting problems and algorithms refer to [17]. For a gentle introduction to the problematic of drawing graphs we recommend [18].

## 3.1 Topology-Shape-Metrics

The topology-shape-metrics approach[19] layouts a graph in three distinct phases, where each phase addresses a specific problem and set of aesthetic criteria.

### Topology (Planarization)

The objective of the topology phase is to determine where the graph nodes should be positioned in relation to each other. This step is therefore responsible for creating a *planar embedding* of the graph. An algorithm performing this step will attempt to *minimize edge crossings*.

### Shape (Orthogonalization)

Shape phase will determine the overall shape of the individual elements, edges and the diagram. Therefore *edge orthogonalization* is performed during this step. As the number of edge bends affect the shape, the algorithm performing the orthogonalization will attempt to minimize them.

### Metrics (Compaction)

The final phase focuses on the overall metrics of the drawing — length of edges, size of nodes, and the overall area required for the drawing. The associated algorithm will attempt to minimize them. Furthermore *dummy edges* introduced in the planarization phase are removed.

All three steps are NP-hard problems[19], and therefore heuristics must be used.

---

<sup>2</sup><http://ogdf.net>

## 3.2 Planarization

The objective of planarization is to transform a graph  $G = (V, E)$  into a graph  $G' = (V', E')$ ,  $V' \subseteq V$ ,  $E' \subseteq E$  that has a planar drawing.

As many graphs do not have planar drawings (e.g.  $K_5$ ) the planarization is usually performed as follows:

1. finding maximum planar subgraph of  $G$  by removing some edges  $F$  from  $E$
2. adding the edges  $F$  back into the graph and adding new vertices at crosspoints (Edge Insertion)

Finding *maximum planar subgraph* — a graph with maximum amount of retained edges (minimum amount of removed edges) is a NP-hard problem, however heuristics exist that solve similar problem — *maximal planar subgraph*.

A *maximal planar subgraph* is a planar subgraph  $G = (V, E' \subseteq E)$ , where adding any edge  $e \in E - E'$  would destroy the planarity. It follows that any *maximum planar subgraph* must be also *maximal*.

We will describe a planarization technique by Goldschmidt and Takvorian as described in [20] as it will be later used as a basis for *mixed upward planarization*. For more on planarization heuristics refer to [17] or [21].

### 3.2.1 Goldschmidt Takvorian two-phase planarization

The Goldschmidt Takvorian (GT) heuristic follows two phases; in the first phase an ordering  $\Pi$  of the vertex set of the input graph  $G = (V, E)$  is found. The vertices are then placed on a line according to the ordering. The ordering should maximize the number of edges between adjacent edges, as the choice of ordering impacts the quality of the planarization. The remaining edges are then drawn as arcs on one side of the line or another.

To describe the second phase, we first need to introduce *crossing with respect to  $\Pi$* .

**Definition 3.2.1 (Crossing with respect to  $\Pi$ ).** Let  $\pi(v)$  denote the position of vertex  $v \in V$  within the ordering  $\Pi$ . Then let  $e_1 = (a, b)$  and  $e_2 = (c, d)$  be two edges of  $G$ , such that, without loss of generality,  $\pi(a) < \pi(b)$  and  $\pi(c) < \pi(d)$ . The edges *cross with respect to  $\Pi$*  if  $\pi(a) < \pi(c) < \pi(b) < \pi(d)$  or  $\pi(c) < \pi(a) < \pi(d) < \pi(b)$ .

The second phase partitions the edge set  $E$  of  $G$  into subsets  $L$ ,  $R$  and  $B$  in such way that  $|L + R|$  is large (ideally maximum) and that no two edges both present in either  $L$  or  $R$  cross with respect to the ordering  $\Pi$  obtained from the first phase.

To partition the edges we create an *overlap graph* that has a vertex for every edge in  $G$ . If two edges cross with respect to  $\Pi$  we add an edge between the corresponding vertices.

To find the best assignments for  $L, R, B$  we would need maximal induced bipartite subgraph. As finding such subgraph is NP-complete, we rely on a simple heuristic: successively find two disjoint independent sets of the overlap graph, where the first set will be  $L$ , and second  $R$ .

After the second step we now have a graph  $G = (V, L \cup R)$  that is planar, and a set of edges  $B = E \setminus L \setminus R$  that would destroy the planarity. Adding the edge set  $B$  back into the graph will be discussed in Edge Insertion.

## 3.3 Mixed Upward Planarization

For oriented graphs where we are interested in keeping uniform edge orientation, the GT planarization method is insufficient, as it doesn't consider the orientation.

We will describe *An Approach for Mixed Upward Planarization*[20] that specifically addresses the problem in three parts:

1. Construct an upward planar subgraph
2. Determine an upward embedding of the subgraph
3. Insert the edges not contained in the subgraph, one by one

### 3.3.1 Directed Upward Planarization

The algorithm extends GT planarization described earlier to work also for directed graphs (Directed GT, DGT).

The original version of GT in the first phase created a vertex ordering  $\Pi$  that attempted to maximize number of edges between adjacent edges. In DGT we have additional constraint — maintaining proper topological ordering. Firstly we will address only purely directed graphs, and later we will discuss the mixed case.

The algorithm assumes the input graph does not contain cycles, otherwise

achieving uniform orientation is not possible.

**Data:** A directed graph  $G = (G_V, E_G)$

**Result:** A vertex ordering  $\Pi$

Select  $v_1$  from  $G$  with zero indegree and minimal outdegree;

$V = V \setminus \{v_1\}$ ;

$G_1$  = directed graph induced on  $G$  by  $V$ ;

**for**  $k = 2, \dots, |V|$  **do**

$U = \{v \in G_V \mid \text{indegree}(v) = 0 \text{ in } G_1\}$ ;

**if**  $v_{k-1}$  is connected to a vertex in  $U$  **then**

        select  $v_k$  as vertex in  $U$  adjacent to  $v_{k-1}$  with minimal degree in  $G_{k-1}$ ;

**else**

        select  $v_k$  as vertex in  $U$  with minimal degree in  $G_{k-1}$ ;

**end**

$V = V \setminus \{v_k\}$   $G_1$  = directed graph induced on  $G$  by  $V$ ;

**end**

**return**  $\Pi = (v_1, \dots, v_{|V_G|})$

**Algorithm 1:** Vertex order[20]

The algorithm starts with vertex  $v_{k-1} = v_1$  with no incoming edge (for acyclic graph there will be always one) and minimal outdegree. Then in each step it chooses a vertex  $v_k$  adjacent to  $v_{k-1}$  which is not the successor of an unchosen vertex. If that is not possible, it takes a vertex of minimal degree, which is also not the successor of an unchosen vertex. The result will be ordering  $\Pi$ .

The DGT algorithm's second phase remains principally unchanged from GT's second phase, only the line is drawn vertically (for regular GT the orientation doesn't matter).

From the sets  $L, R$  (obtained from GT's second phase) and the new ordering  $\Pi$  we obtain upward planar embedding:

For each vertex  $v \in V$  we sort the edges with source  $v$  in  $L$  decreasing according to  $\Pi$  and the edges with source  $v$  in  $R$  increasing according to  $\Pi$  and concatenate these two ordered lists to one. For edges with target  $v$  we do the opposite (decreasing in  $R$  and increasing in  $L$ ) and append to the list.

### 3.3.2 Edge Insertion

Edge insertion in upward drawings is more complex than for regular drawings because we cannot add edges independently of each other, as adding dummy nodes can create unexpected cycles[20].

To properly insert directed edges we create *routing graph*  $G_R$ . A layering  $l$  for a directed graph  $G = (V, E)$  is a mapping  $V \rightarrow \mathbb{N}$  such that  $l(v) > l(u)$  for each edge  $(u, v) \in E$ . Then a *routing graph*  $G_R$  is constructed as follows:

- We split the outer face into two faces  $f_l$  and  $f_r$  that are respectively to the left and to the right of the vertex embedding
- For each face  $f$  and each layer  $l$  that it spans add a vertex
- Every two vertices in neighboring layers and in the same face  $f$  are connected by an directed edge (in increasing layer order) of a weight 0
- Every two vertices of adjacent faces and at the same layer  $i$  are connected by an edge of weight 1 if the source vertex  $u$  of the edge  $e = (u, v)$  separating those faces is at the layer  $i$  or below it ( $l(u) \leq i$ ) and the target vertex  $v$  is above the layer ( $l(v) \geq i$ ).

Each edge of weight 1 represents a single edge crossing, thus choosing the shortest path will result in minimal amount of edge crossings.

To insert a single edge  $e = (u, v)$  into the planar graph  $G$ :

1. Find the shortest path from  $u$  to  $v$  in the routing graph  $G_R$
  2. For each crossing  $i$  we split the crossed edge  $e_c = (a, b)$  into  $(a, w_i)$  and  $(w_i, b)$  by introducing a dummy vertex  $w_i$
  3. Finally instead of adding edge  $e$  we add it's route
- $e = (u, v) \rightarrow (u, w_1, \dots, (w_i, w_{i+1}), \dots, (w_k, v)$  to the planar graph  $G$

It has to be noted that although this algorithm produces correct result (in terms of upward planarization), it doesn't necessarily produce optimal result, as each edge insertion degrades the graph (more crossings may be required for the edges added later). To find the optimal result we would need to test all options. [20] describes a heuristic that after each edge insertion chooses some edges and tries to reinsert them with potentially fewer edge crossings.

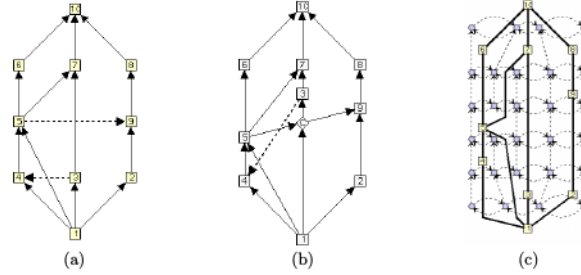


Figure 3.1: Edge insertion and the routing graph[20]

### 3.3.3 Mixed Upward Planarization

To solve the mixed upward planarization the authors of [20] directly use the Directed GT algorithm as described above by orienting the undirected edges in both ways. Although this produces correct result we note that it may degrade

the quality of the drawing as the choice of undirected edges still restricts the following choices. The authors suggest assigning weights to directed and undirected edges, but they note that this is still subject of further research.

Finally to insert edges in the mixed case the same edge insertion algorithm is used with undirected edges having same direction as in the GT vertex ordering.

### 3.4 Edge Labeling Problem

Edge Labeling Problem is concerned with the placement of labels associated with edges. The objective is to find an optimal placement such that the relation with the edge is obvious and other aesthetic criteria (overlapping) are upheld. As associations in UML class diagrams can have large number of labels, this problem is of great interested to us.

We will describe a Multiple Label Placement algorithm introduced by Kakoulis & Tollis in their work “Algorithms for the multiple label placement problem” [22]. This work expands their previous works [23][24] and presents a solution for multiple labels.

#### 3.4.1 Label Placement

The placement of a single label is seldom arbitrary. Usually the label describes a feature in relation to other objects, therefore the freedom of placement is restricted.

The basic rules for edge label placement[ELP] are[23]:

1. No overlaps of any kind
2. Each label must be easily identified with exactly one graphical feature
3. Each label must be placed in the best available position

The second rule ensures that we do not produce confusing layout that would label wrong objects, while the third ensures we select the best layout among all available.

#### 3.4.2 Terminology

We introduce terminology specific to the Multiple Label Placement (MLP) problem.

Let  $\Gamma$  be a drawing and  $F$  a set of edges<sup>3</sup> to be labeled in  $\Gamma$ :

- $M : F \rightarrow \mathbb{N}$  is a mapping that returns the number of labels associated with an edge.

---

<sup>3</sup>[22] uses the term *graphical feature* as their approach can be applied to arbitrary objects, we will however avoid introducing extra terminology.

### 3. LAYOUTING

---

- $\Lambda$  is the set of all label positions for all edges
- $\Lambda_f$  is the set of all label positions for  $f$  (its solution space).
- $\Gamma_f^i$  is the set of all label positions for the  $i$ th label of  $f$ .
- $\lambda : F \times \mathbb{K} \rightarrow \Lambda$  is a function that assigns to edge  $f$  in  $F$  a label position from  $\Gamma$ ;  $\lambda(f, i) = \lambda_f^i \in \Gamma_f^i$ ,  $\mathbb{K} \subseteq \mathbb{N}$ .
- $COST : F \times \mathbb{K} \times \mathbb{K} \rightarrow \mathbb{N}$  is the cost for assigning label  $k$  as  $i$ th label of edge  $f$ .

The cost itself is typically a linear combination of subcosts and permits modularity. [22] defines the cost broadly, therefore for our needs we have settled on these requirements:

- edge proximity — based on the distance between the label and other edges (i.e. to avoid ambiguity)
- element proximity — based on the distance between the label and elements
- ranking — how far is the label from its most desired position
- orientation — whether the label is above or below line

We have implemented the last two requirements as pluggable behavior, therefore it is possible to specify the needs for each label individually.

Since we have assigned a cost to each combination of label positions, we can solve the MLP problem as an optimization one:

$$\sum_{f \in F} \sum_{i=1}^{M(f)} \sum_{k \in \Gamma_f^i} COST(f, i, k) P(f, i, k),$$

where

$$P(f, i, k) = 1 \text{ iff } \lambda(f, i) = k, 0 \text{ otherwise}$$

and

$$\sum_{k \in \Gamma_f^i} P(f, i, k) = 1, f \in F \wedge 1 \leq i \leq M(f)$$

The  $P$  function guarantees that only labels that will be in the final assignment are considered.

To transform the geometric problem into a combinatorial one, we construct overlap graph of label positions in  $\Lambda$ . We then reduce and partition  $\Lambda$  into clusters of overlapping positions, such that each position is member of exactly one cluster. Finally we find the optimal matching of labels to the clusters.

*For brevity we do not include the precise definitions, and we use only simplified descriptions. For the full definitions refer to the used literature[23][24][22].*

An *overlap graph* has a vertex for every edge and every label position, edge for every relation between position and it's edge, and an edge for every two positions that overlap each other.

A *cluster* is a set of label positions, where each position in the cluster overlaps every other.

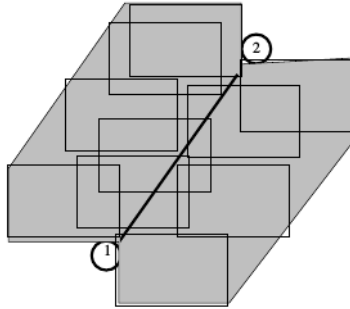
A *matching graph* has a vertex for each edge and every cluster, and an edge for every edge'

### 3.4.3 Algorithm

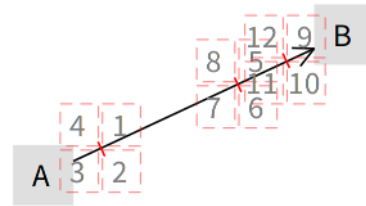
Given a set of edges  $F$  to be labeled and a set of labels  $\mathcal{L}$ , we solve the MLP problem as follows:

1. Find label positions  $\Lambda$  for each edge in  $F$  and each label in  $\mathcal{L}$
2. Construct an overlap graph  $G_o$  from edges in  $F$  and label positions  $\Lambda$
3. Partition the positions  $\Lambda$  into overlapping clusters
4. Construct a matching graph  $G_m$
5. Solve maximum cardinality minimum weight matching problem
6. Make the final assignment

To find the initial label positions we have to consider a *solution space*. A solution space is the set of all points that we can consider to place the label in. Usually such solution can be either expressed as an area (continuous solution space) or a set of discrete points.



(a) Continuous solution space[23]



(b) Discrete solution space

The size of the solution space directly impacts both the quality (positively) and performance (negatively) of the final assignment.

We have created a configurable discrete heuristic for the initial label positions:

1. Select equally spaced<sup>4</sup> points on the line.

<sup>4</sup>The spacing is configurable.

### 3. LAYOUTING

---

- The positions start (resp. end) slightly off the start (resp. end) of the line to avoid unnecessary overlapping later between the label and the element, and penalty for proximity to element.
2. For each of those points consider four corner rectangles, the rectangle has the size of the label we are trying to place.
  3. Remove all rectangles that overlap some feature in the diagram (feature is an edge or element, but not another label position rectangle).
  4. Use the centers of the remaining rectangles as the initial label placement set.

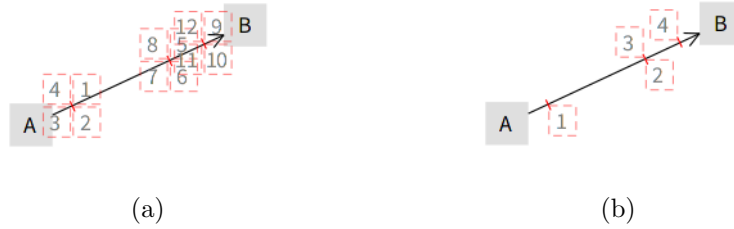


Figure 3.3: All considered label positions (a), and after removing overlaps (b)

The remaining steps of the algorithm, from creating the overlap graph all the way to the final assignment are straightforward and well described in their respective papers.

The main hurdle we have encountered is the fifth step — solving maximum cardinality, minimum weight matching graph. Although algorithms that can solve the assignment problem quickly have been invented, we are not aware of any free and open-source (or in fact any at all) library in Pharo that would implement them. Because the implementation is rather complex, and we do not even have graph library available, we have opted for implementing *Munkres assignment algorithm*[25]. The algorithm was simple to implement and enabled us to complete the Edge Label Placement, but unfortunately it has polynomial performance, therefore the whole solution is not usable for anything but the most trivial graphs.

When in the future a more performant algorithm will be implemented, the placement algorithm can make use of it. As a workaround we have implemented a fast localized labeling constraint in the upcoming chapter 5.6.

## 3.5 Interfacing with GraphViz

As we found it not possible to achieve meaningfully satisfiable layout without strong graph and layouting library, we have implemented a prototype inter-

face with the GraphViz library. GraphViz<sup>5</sup> is a popular library for visualizing graphs and software and has implemented several interesting layouting algorithms. The library uses its DSL format called *dot* to describe the graph structure. We use an existing Pharo library<sup>6</sup> that can generate a file in the *dot* syntax based on our programmable description. The dot file is then passed to graphviz executable that performs the layout. GraphViz offers a range of output formats, among other XML and SVG. We use the SVG output and extract the layouting information. Despite the implementation was trivial compared to attempting to implement proper layouting algorithms, the results are much better. Therefore we would recommend to further expand the interface before proper layouting library is developed. Interface with the OGDF<sup>7</sup> library was also considered, unfortunately the library is offered under the GPL license which is not compatible with the MIT license used by DynaCASE.

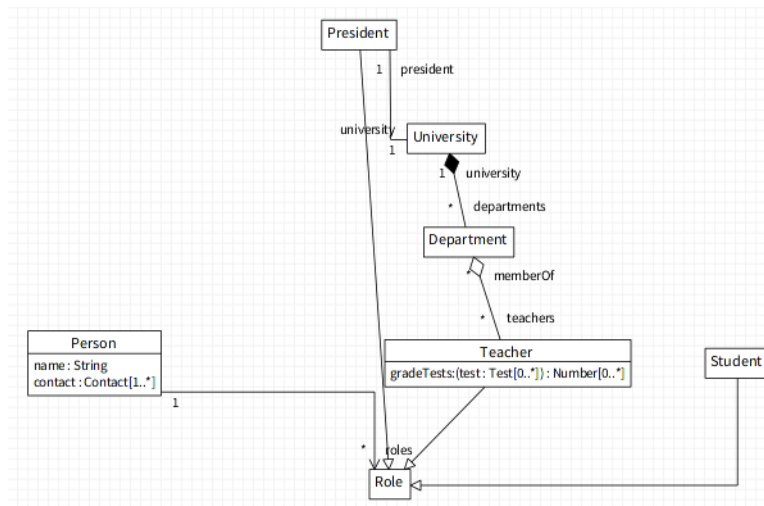


Figure 3.4: A diagram layouted through GraphViz

<sup>5</sup><http://graphviz.org/>(<http://graphviz.org/>

<sup>6</sup><http://smalltalkhub.com/#!/~hernan/GraphViz>

<sup>7</sup><http://ogdf.net/doku.php>



## Polylines

The used graphical library Roassal offers basic line shapes: straight lines, bezier lines, and multilines with a fixed pathing<sup>8</sup>. But to produce orthogonal drawings, or more complex drawings in general we need a more powerful solution. One with the option to arbitrarily add and remove bends, both programatically and interactively; none of which is provided by Roassal.

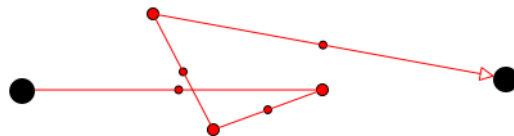


Figure 4.1: Polyline with activated handles

Polyline is a segmented line going through points on canvas defined by an ordered collection of points known as *waypoints* or *bends*. Each waypoint may introduce a new change in direction of the line. The purpose of such bent line is to navigate around existing objects on a canvas (so the line doesn't need to overlap them), or enabling orthogonal visualization, which is one of the aesthetic criterions<sup>2.1.4</sup>.

We have introduced a new polyline shape `DCRTConnection` that is capable of handling our requirements, along with several other classes that separate specific concerns of the implementation.

<sup>8</sup>Once the multiline is created the number of bends/segments is fixed

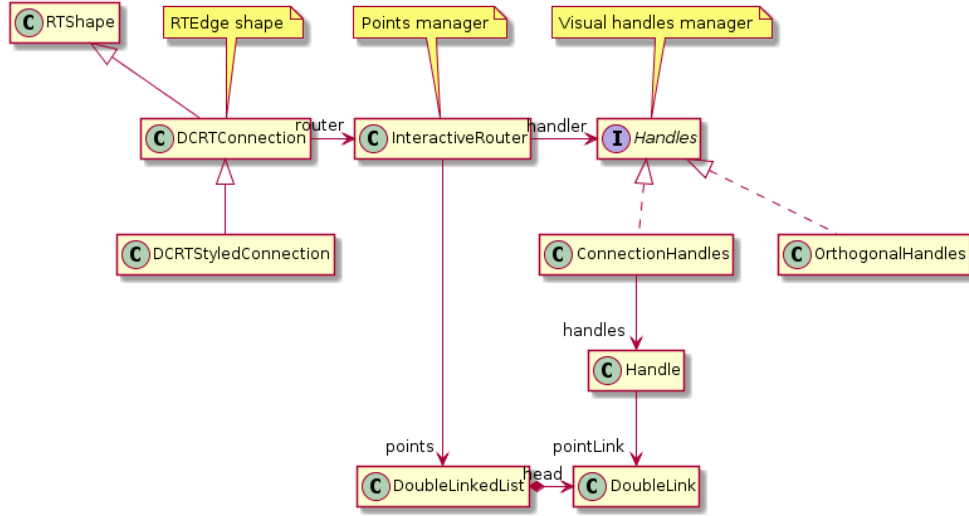


Figure 4.2: Diagram of classes which provide polyline connection

In 4.2 we see the following classes:

- **DCRTConnection** is a dynamic Roassal shape that is capable of adding and removing new line segments to the canvas
- **DCRTStyledConnection** is an extension that will properly manage line ends (arrows, diamonds, ...) and line styles (solid, dashed, ...)
- **InteractiveRouter** holds references to the used waypoints in a **DoubleLinkedList**. Using **DoubleLinkedList** simplifies addition and removal of waypoints at arbitrary positions
- **ConnectionHandles** manages **Handles** and user interaction
- **Handle** enables waypoint manipulation, such as dragging and removal

## 4.1 Managing waypoints

The handles management is separated from the polyline and waypoints. That way an automatic layouter can route the polyline as needed, and user interaction can be added later on need basis.

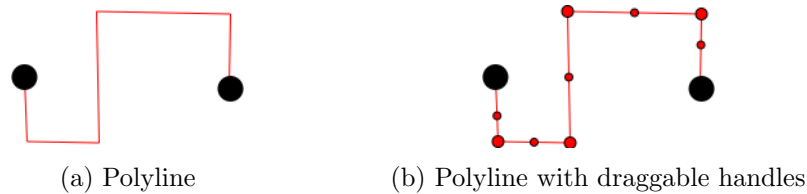


Figure 4.3

In 4.3 we can see two types of handles:

- larger red circles, which are managed by `Handle` instances
- smaller red circles, which are managed by `MidpointHandle` instances

`Handle` manages a waypoint, however we want to be able to add simply new waypoints. For this purpose we have `MidpointHandle`.

`MidpointHandle` is always placed between every pair of real handles, but by itself doesn't participate in the routing, nor is referenced from the waypoint list. However when user attempts to drag the midpoint, it is automatically converted into a real `Handle`. Since the waypoints are stored in a linked list, we can easily add the newly created waypoint at the correct position in list. Finally two new midpoint handles are added in the middle of the new segments.



Figure 4.4: Dragging the midpoint down (a) will create a new waypoint and two new midpoints (b)

In an alternative approach (used e.g. by UMLet<sup>9</sup>) one can create new waypoints by dragging on any portion of the polyline.

Removal works in a similar manner. When two proper handles are aligned within certain margin the dragged handle will be removed together with its neighboring midpoints. New midpoint will be placed at the appropriate place instead. Note that the real ends of the connection act as proper handles for the purposes of addition and removal.

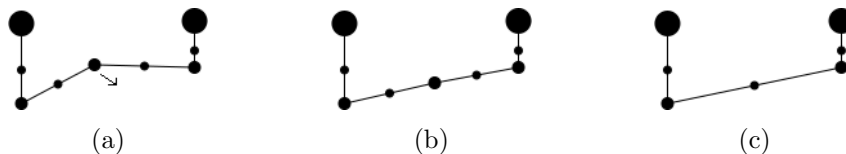


Figure 4.5: Dragging handle (a) to position (b), after which the handle is removed (c)

---

<sup>9</sup><http://www.umlet.com>



---

# Interactive Layouting

In the absence of a fully self-sufficient automatic layouting for a particular notation, or when no automatic layouting has been yet implemented, a user will always end up manually adjusting the layout. In fact a tool permitting arbitrary placement is more appropriate than a modeling tool enforcing only basic guidelines[8].

For this reason we deem it necessary to introduce collection of utilities that will assist the user in the manual layouting; their primary objectives are:

- **real-time performance** — the user must receive immediate feedback at all times
- **maintain mental map** — the user is already familiar with the diagram, breaking the mental map to achieve marginally better layout would be detrimental
- **increase precision** — create more precise drawings despite the less precise movement of the user
- **decrease effort** — the utilities should aid the user, thus the effort to use them should be minimal

## 5.1 Grid

Grid is a commonly used component not just in diagramming tools, but in drawing tools in general (cf. GIMP or Inkscape<sup>10</sup>). It gives the user a basic visual feedback on sizes and proportions, as well as visual step-stone for automatic snapping.

The grid is implemented under the name of `RTGridView` directly in Roassal and allows the user to manually specify the grid size.

---

<sup>10</sup>GIMP is a raster graphics editor, Inkscape is a vector graphics editor

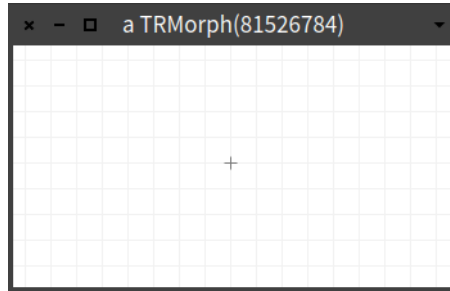


Figure 5.1: RTGridView

### 5.1.1 Principle

As the canvas the grid is drawn upon is potentially infinite, we cannot simply draw infinite lines. Instead, we focus only on the visible area:

1. Fill the visible area with equally spaced (`RTGridView>>gridSize:`) vertical and horizontal segments, with one additional line outside the visible area
2. Register mouse events that will move the grid upon canvas movement only by `gridSize` modulo
3. Register zoom events that will redraw the shape so the lines will always fully populate the visible area

The second step creates the illusion of an infinite grid: when the canvas is moved by a vector  $n \cdot \text{gridSize} + k$  (where  $k < \text{gridSize}$ ), we position the grid only by  $k$  and ignore the rest.

## 5.2 Snap To Grid

The purpose of a snap-to-grid functionality is to aid the user in creating better and more precise drawing with less precise user movement and in shorter amount of time.

Snap-To-Grid is the most basic utilization of a grid. Upon dragging a movable element, the element will move only in the multiplications of the grid size. Such movement can be further configured based on the needs of a particular element, that is: we may want to align the element's center on grid intersections, or the element's top left corner (or some other distinguished part of the element).

Finally, the Snap-To-Grid, implemented as `RTDraggableSnapToGrid` is independent of `RTGridView` and therefore can be configured differently, e.g. the movement being half of the grid, etc. By default Snap-To-Grid will use grid's configuration so one can specify it globally.

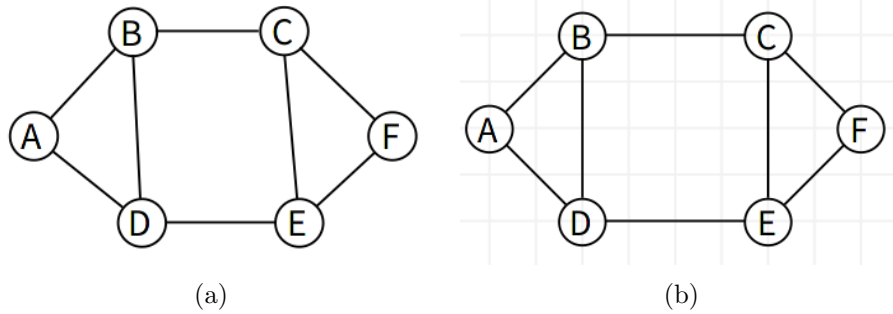


Figure 5.2: A simple graph created without (a) and with (b) snap-to-grid

### 5.2.1 Principle

When moving an element with snap-to-grid property, the *normal* (i.e. without movement restrictions) target destination is first computed based on the mouse movement, this position is then rounded to the nearest grid intersection point, where finally the element is visually positioned by it's center or top left corner (based on the configuration).

## 5.3 Alignment Snapping

Alignment snapping is an advanced form of snapping that relies on positions of other elements rather than on a grid. The objective is to give the user more degrees of freedom while maintaining consistent alignment between objects — in contrast grid will always force the user to adhere to the grid, whether it makes sense or not.

The most basic form of alignment is alignment between object edges in an orthogonal space.

### 5.3.1 Principle

Lines are projected from all *hot*<sup>11</sup> objects' edges, centers and possibly other points along the size of the canvas, when another object is being moved and approaches such projected line within certain distance, the moved object will automatically snap to the line, thusly being aligned with the projecting object.

## 5.4 Snap And Go

*Snap And Go* [26] addresses the problem of moving elements in close proximity to each other without the need of temporarily disabling the grid snapping.

<sup>11</sup>objects we are interested to be aligned with

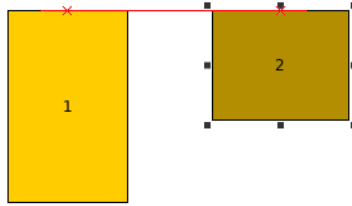
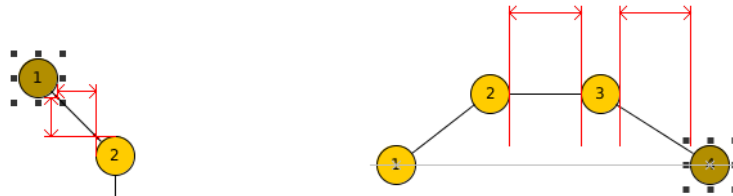


Figure 5.3: yEd editor utilizing alignment snapping (top line)

Especially interesting in situations where we want different precisions based on the surrounding context.

## 5.5 Constraint Snapping

The most advanced version of snapping discussed here is *constraint snapping*. In addition to aligning elements (like Alignment Snapping) a constraint engine would evaluate additional constraints applied to the drawing. Such constraints could ensure that elements are spaced equally, or they maintain some minimal distance.



(a) Object 1 is being placed in an equal distance in  $x$  and  $y$  distance from 2      (b) Object 4 is placed in an equal distance from 3, as is 3 from 2

Figure 5.4: Examples of automatic distance constraints between objects in the yEd graphical editor

Usually such constraints are either hard-coded in the implementation, or a special constraint language is used to describe them.

Finally, especially graphical editors can benefit from even more advanced snapping in any directions while utilizing rotation anchors, skewing, etc. However as such complex behavior is of no use in currently implemented models (BORM, UML), they were not explored further.

For an extensive study on snap-dragging see [27].

## 5.6 Edge Label Constraint

Edge Label Layouting discussed earlier<sup>3.4</sup> is attempting to produce a globally suboptimal solution to the problem. The solution there however has major drawbacks from interactive perspective. Due to complexity of the problem and performance of the implementation, even for smaller diagrams it is insufficient for any real-time recomputing. For this reason a more performant alternative was created.

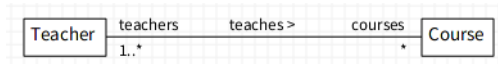


Figure 5.5: UML Association with five different labels

### 5.6.1 Basic Requirements

5.5 shows several different labels on a single edge. From this we can observe the first requirement: we must be able to programatically define the most preferred position — whether at the beginning, middle, or end of the line, whether above or below, and the minimum distance between the label and nearby objects.

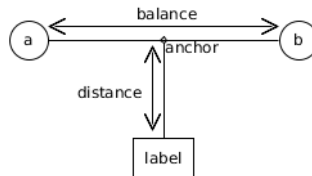


Figure 5.6: Properties of the constraint

We define the distance along the line from  $a$  to  $b$  as *balance*. This position ( $x \in \mathbb{R}$ ) can be defined either relatively  $x \in [0, 1]$ , where 0 is the source, 1 the target, and 0.5 the middle, or absolutely  $x \in (-\infty; 0) \cup (1, \infty)$  with positive numbers measured from the source, and negative numbers measured from the target.

Distance  $d \in \mathbb{R}$  specifies the minimal orthogonal distance that should be maintained between the *label* and the *anchor point*. If the distance is positive ( $d \geq 0$ ) the label will be above the line or right of the line (depending on the slope of the line), while negative distance positions the label below or to the left of the line.

### 5.6.2 Additional Requirements

Foremost the constraint must be able to handle polylines, which is something that none of Roassal's constraints are capable of as they've been introduced only as a part of this thesis.

Secondly a user must be able to move the element by hand if needed, even if such adjustment will reset when the edge moves. Reset after edge movement is to be expected as repositioning of objects in the diagram will likely change the optimal placements.

Finally an automatic *repulsion* is required that will nudge the label in an appropriate direction so it won't overlap neither it's edge, nor the edge's endpoints.

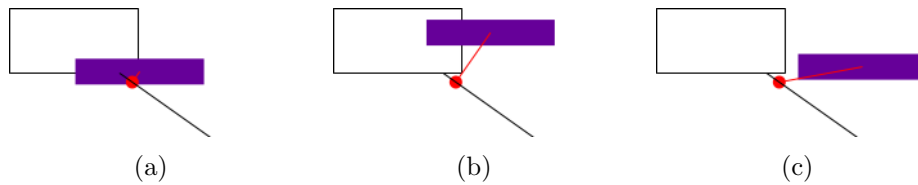


Figure 5.7: (a) initial placement of the label, (b) nudging from line orthogonally to it, (c) nudging from source element in the direction of the line

### 5.6.3 Results

The results show that the constraint works well for orthogonal drawings despite the label being nudged only from the associated edge and it's endpoints, while ignoring other elements and edges in the diagram. This is possible because a good overall layout, namely orthogonal planarized drawing, will leave enough space for the label placement.

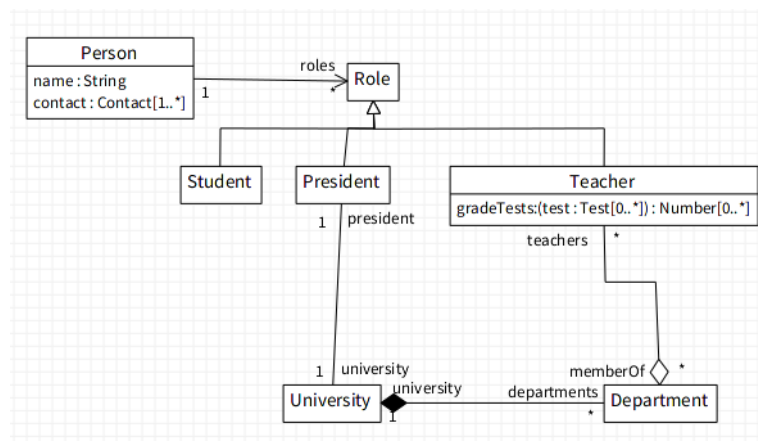


Figure 5.8: UML Class Diagram utilizing the edge label constraint

## 5.7 Rail Constraint

In some situations we may want to give the user (or semi-automatic layouting) more control over the placement of an edge label. Such example may be a BORM DataFlow.

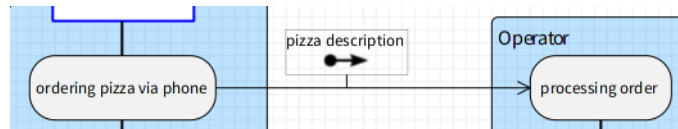


Figure 5.9: BORM *DataFlow* “pizza description” placed on a *Communication* line

For the DataFlow we are not interested in any particular position on the line (cf. UML association role names must be at the appropriate ends). Thus we can allow the user to freely move the label along the line after which we will remember and maintain the user-chosen position (both the *balance* and *distance* as described in 5.6).

Additionally the user may want to visualize both the current rail anchor and the joining *rod* between the rail anchor and the label. The rod will always be orthogonal to the current polyline segment.

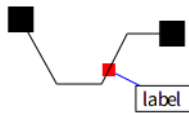


Figure 5.10: Rail constraint with visualized rail anchor and rod



# UML Class Diagrams Layouting

According to a study[28] UML Class Diagrams are with a large margin the most popular notation used for software visualization (used by 85% of respondents of the study, compared to activity diagrams at second place with less than 60%). Therefore implementing such notation, together with semi-automatic layouting would provide the highest potential benefit for users.

As part of this thesis we have implemented basic UML Class Diagram notation in the DynaCASE tool, performed review state-of-the art of UML Class Diagram layouting and implemented prototype layouting algorithms applicable to them.

## 6.1 Diagram, Subject, Model, Meta-model

As the terms *diagram*, *model*, *meta-model* (*metamodel*), etc. are often used interchangeably to a great confusion of the reader, we follow this terminology:

**Subject** is something we are interested in, whether it is a piece of software code, network topology, or even a meta-model.

**Model** is a representation of a *subject* that captures some specific properties of it, while ignores or simplifies other ones. A subject can have many, often overlapping, models each with a specific purpose.

**Meta-model** provides a description of a terminology that gives semantic meaning to elements of a model. E.g. a meta-model of a *graph* would describe concepts such as *vertex*, *edge*, etc.

**Meta-...-meta-model** describes a level below itself. There is no limit to how many meta-\* layers we can add, however it is not uncommon for a *meta-meta-model* to be self-described — it acts as its own meta-model.

**Diagram** is a visual representation of a model. It applies a *diagram notation* to a model.

**Diagram model (Diagram notation)** A *diagram* can have its own meta-model that describes how a particular subject's meta-model element

## 6. UML CLASS DIAGRAMS LAYOUTING

---

should look. E.g. graph diagram model would say that a vertex (meta-model element) should be visualized as a circle with a label inside.

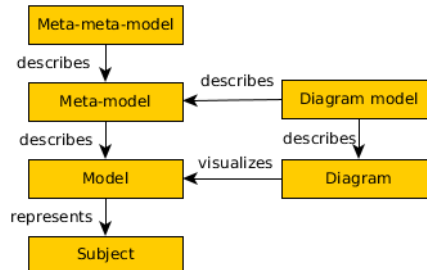


Figure 6.1: Roles of the subjects

## 6.2 UML Class Diagram Notation

Implementation of the UML metamodel[29] is daunting task given its sheer complexity. Even if we focus only on the subset of elements used for Class Diagram models we would require implementation of the core of UML infrastructure, which is still very extensive. For this reason as a temporary solution we have chosen to use the FAMIX meta-model[30] and FAME meta-meta-model[31]. Both libraries are part of the Moose Platform<sup>12</sup>.

The FAMIX metamodel is used by Moose to reengineer source code; the core is very similar to UML infrastructure, therefore we deemed it a sufficient basis for our own UML metamodel. The model is also well tested with both tests and in practice. However to model our needs we had to extend to model with binary associations, multiplicities, aggregations, value specifications, default values and additional UML features.

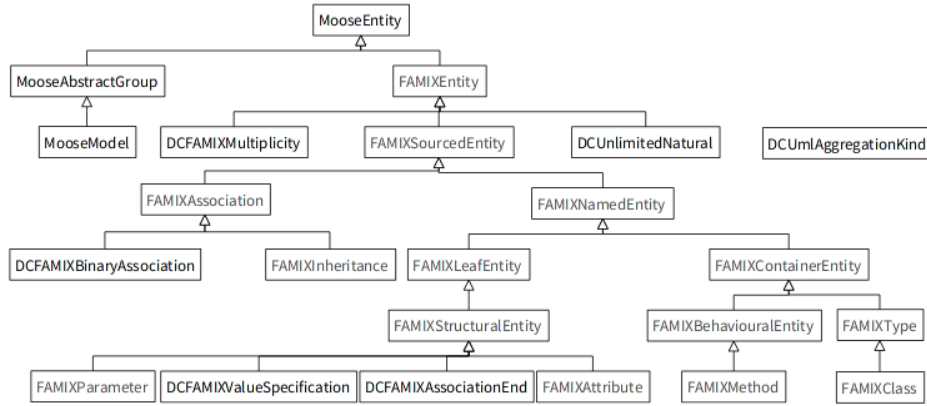


Figure 6.2: Model of a FAMIX metamodel including DynaCASE extensions

Having UML metamodel is not sufficient to visualize diagrams, for that purpose UML utilizes Diagram Definition[32] and Diagram Interchange[29].

“The Diagram Definition (DD) specification provides a basis for modeling and interchanging graphical notations, specifically node and arc style diagrams as found in UML [...]” [32].

Practically speaking DD/DI describes how UML model elements should look (*diagram model*), e.g. instead of a `Method` with `isAbstract = true` we would work with `UMLLabel` with `UMLStyle` having `fontItalic = true`. Additionally it describes properties specific to UML drawings. For example a *UMLEdge* representing a *UML Association* may have a cross at the endpoint to represent non-navigability, or when the association is navigable in both ways it is common to not draw arrows at all. As this information is purely

<sup>12</sup><http://www.moosetechnology.org>

## 6. UML CLASS DIAGRAMS LAYOUTING

visual it is not expressed in the UML model, instead it is represented in the DD model.

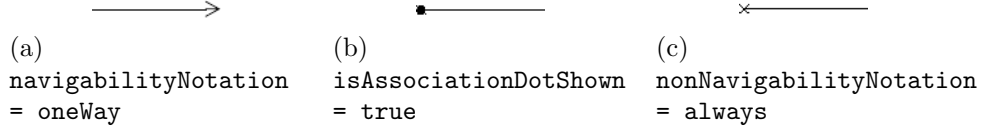


Figure 6.3: UMLEdge with various diagram settings

DD also describes mechanism for managing styles of shapes such as colors, font styles, etc.

Apart from implementing parts of DD necessary to visualize class diagrams we have also extended the DD element classes, so they are able to draw themselves on Roassal using Roassal shapes.

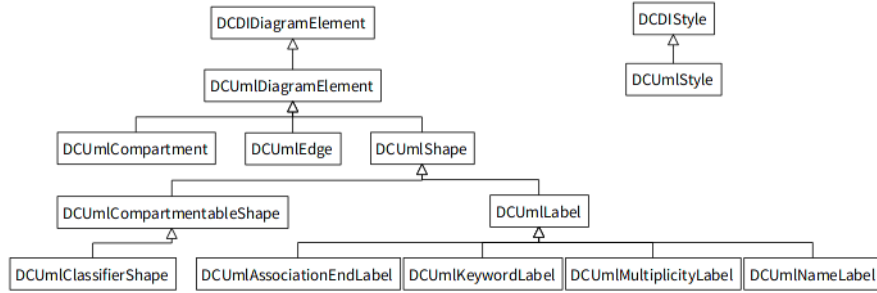


Figure 6.4: Portion of DD/DI implemented in this thesis

Notation	Diagram elements	Ref.
<div style="display: flex; justify-content: space-between;"> <div> <b>end1</b>  0..1 {redefines...} </div> <div> <b>Association1</b>  {modifier1} </div> </div>	UMLEdge (Association) - UMLAssociationEndLabel (Property) - UMLNameLabel (Association) - UMLMultiplicityLabel (Property) - UMLRedefinesLabel (Property) - UMLLabel (Property)	<a href="#">11.5.4, B.3.4</a> Fig 7.12, 9.12, 11.5, 11.26, 11.28, 11.32

Figure 6.5: Notation and diagram elements used to visualize an Association element[29, B.2]

Introducing DI as an intermediate representation between controllers and Roassal provided a great aid in both simplified implementation and mental management of the architecture as it allowed us to focus in controllers only at conceptual problems associated with the manipulation of UML shapes, while in the DI elements we had to worry only about mapping simple shapes onto the Roassal view without worrying about what the shapes represent.

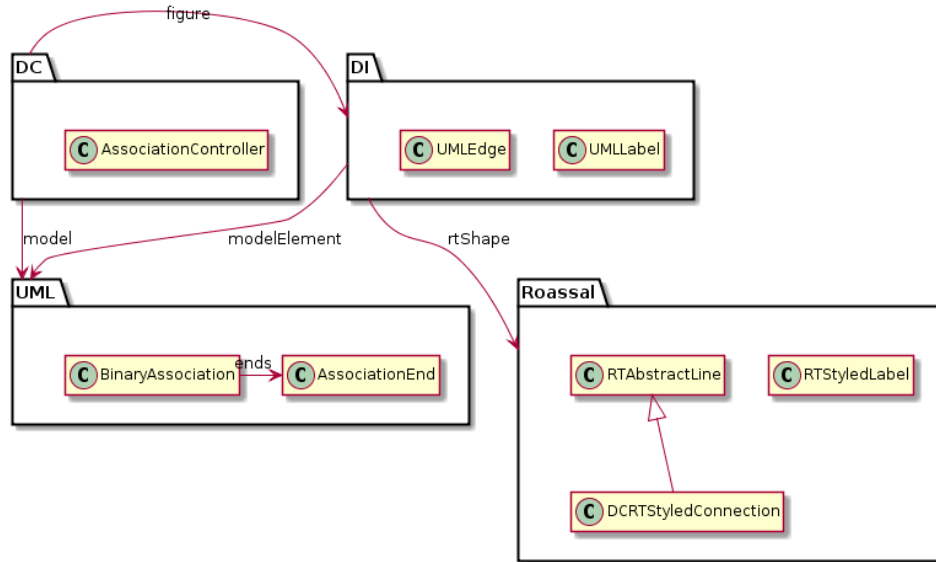


Figure 6.6: Relations between model and its visualization

## 6.3 UML Class Diagram Layouting

Although Class Diagram layouting has been a well studied problem for some time, the practical impact on the tools used in the industry is limited, as the many tools do not adhere to any aesthetic principles, or only very basic ones[9]. The reasons for this are discussed in greater detail in the Layouting chapter.

In this section we will briefly explore the various needs and improvements specific to class diagrams, however majority of the work of both the description and implementation of algorithms used for class diagram layouting is in the Layouting chapter.

### 6.3.1 Aesthetic Requirements

Earlier we have explored a variety of aesthetic criteria<sup>2</sup>; in this section we will select the most important to class diagrams, and introduce some more.

Eichelberger, et al.[33] lists the following aesthetic criteria:

1. Enforce hierarchy as the most appropriate ordering criterion for edges in a class diagram.
2. Respect spatial relationships to encode coupling, cohesion and importance of parts of the diagram.

3. Visualize the natural clustering of nodes according to semantical reasons like containment, n-ary associations and patterns.
4. Avoid crossings and overlappings of model elements.
5. Center position of selected nodes (n-ary associations, pattern nodes).
6. Respect the vicinity of association classes, notes and constraints.
7. Clearly assign adornments to edges and reflective associations to the connected classes.
8. With the minimum priority respect other graph drawing criteria.

Many other sources[19][8][34][10] also agree, based on several empirical studies, on the following additional criteria:

- crossing minimization
- bend minimization
- orthogonality
- horizontal labels
- joined inheritance arcs
- drawing generalizations in the same class hierarchy always in the same direction
- avoiding nesting of class hierarchies

The lists also include aesthetics not discussed earlier:

*Joint inheritance arcs* are an important aesthetic rule that joins generalization arcs of several classes that are a specialization of the same class. Joining them together creates an impression of close relatedness. Doing so is also essential for generalization sets as it makes it obvious visually what elements are in the set.

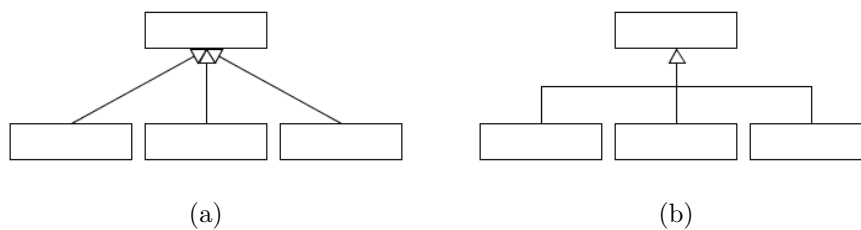


Figure 6.7: Inheritance drawn without (a), and with (b) arc joining

*Horizontal labels* rule requires labels to be always drawn horizontally irrespective of the line they are on. Doing so makes it easier to read them, however it imposes a new constraint on the layout. The layouting of edge labels has been discussed in previous chapters in both the static global variant as Edge Labeling Problem, and as interactive one in Edge Label Constraint section.

### 6.3.2 Algorithm

Devising an algorithm that would be capable of dealing with all the criteria poses a special challenge as it is not clear what is the most appropriate relation between the criteria and their relative value. Some authors may circumvent this problem by introducing parameterizable algorithms. However such solution doesn't solve the problem, instead it delegates the problem to the person using the algorithm, who is in turn required to understand deep implementation details to be able to properly assign criteria values.

Algorithms proposed by various authors can be generally split into two categories:

- hierarchy-based approach
- topology-shape-metrics approach

The first approach always considers the hierarchy as the most important factor of all, which in many instances is indeed true. As the hierarchy is always acyclic<sup>13</sup> creating a layout is a relatively simple task. Either simple tree layout, or a more advanced Sugiyama[13] layout created specifically for visualizing hierarchical systems can be used. An extension of Sugiyama layout specifically for class diagrams was also proposed[35].

Hierarchy-based approach is however inherently flawed as it places too much importance on the hierarchy. This becomes problematic in situations where the modeler is interested in studying relations between objects, or there is a lack of hierarchies, additionally programmers should generally prefer composition over inheritance[36]. Therefore the disadvantage of using hierarchy-based approach is major.

The second approach — *topology-shape-metrics* is the current state-of-the-art of UML Class Diagrams visualizations[34][19][33]. The approach is three phased, where each phase is tasked with solving a specific problem. *Topology* with topological organization of nodes, *shape* with general shape of nodes, edges, and the diagram, and *metrics* with sizes of objects (edges, diagram area, ...). As the topology shape doesn't place any conceptual restrictions, this approach can be even used as direct hierarchy-based approach replacement given appropriate *topology* heuristics, but unlike hierarchy it is not constrained by it. To our knowledge the most extensive work focused on UML class diagrams is the dissertation "Automatic layout of UML class diagrams: a topology-shape-metrics approach"[37].

The discussion and implementation of topology-shape-metrics algorithm has been presented in the previous Layouting chapter.

---

<sup>13</sup>In some meta-instances a cycle may exist, such as Pharo's ProtoObject subclasses itself; however this is a single case as normal hierarchy doesn't permit cycles.



---

# BORM Object Relation Diagrams Layouting

In this chapter we will explore BORM diagrams. Unlike UML Class Diagrams<sup>6</sup> where we already have a wealth of research into the layouting, to our knowledge there is no prior research assessing the layouting needs of BORM diagrams; therefore our primary objective will be such assessment.

## 7.1 BORM ORD Model

Business Object Relation Modelling (BORM) is a complex method for systems analysis and design that utilizes an object oriented paradigm in combination with business process modelling.[38][39]

Although the BORM method encompasses several different model notations, we will focus only on the most used one — Object Relation Diagrams (ORD).

Object Relation Diagram is an object-oriented approach with semantics based on finite state automata (FSA) augmented with communications between them.

The topmost view available in ORD is a *diagram*. It represents an artifact of the modeled domain, usually focused on a specific bounded problem. Diagram itself is further composed of *participants*.

Participant is a representation of a software or business process of a specific entity that can have a state and is capable of engaging in communication with other participants. For example user (Person participant) requesting information about her bank account (System participant). Internally each participant has its own FSA that tells us in what state the participant currently is, as well as any activity or communication engaged by the participant. Each participant has an initial state, at least one final state, and any number of intermediate states. Transitions between the states are enabled by input-

## 7. BORM OBJECT RELATION DIAGRAMS LAYOUTING

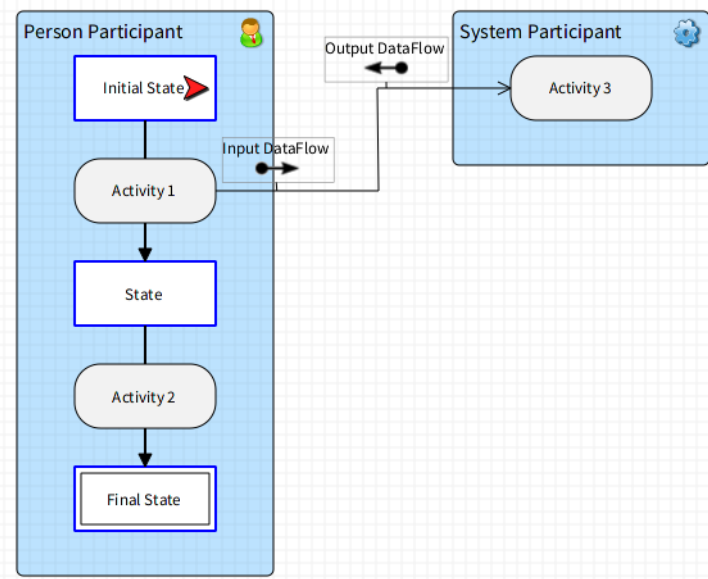


Figure 7.1: ORD diagram depicting various elements

output conditions and guarded by constraints. During transitions additional behavior — *activity* — can be performed.

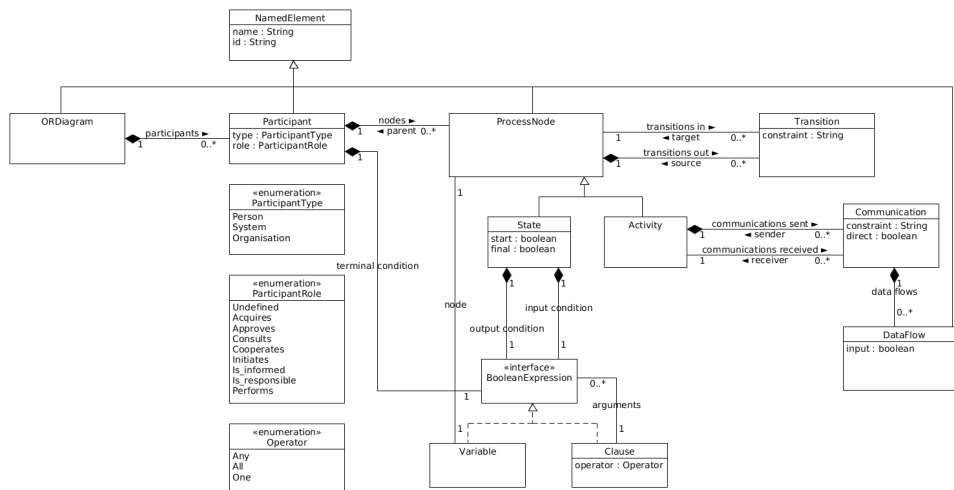


Figure 7.2: Metamodel of BORM Object Relation Diagram

## 7.2 Diagram overview

Diagram is the topmost view currently available.<sup>14</sup> The diagram itself is visually only a blank canvas that contains participants, and partially communications.

Participants from diagram’s perspective are an ordered collection of connected rectangles. It is customary to organize those boxes in such manner that the most important participant — usually the initiator of the diagram — is the leftmost one. This convention allows the reader of such diagram to locate the diagram’s beginning point without searching. Furthermore, participants can then be positioned accordingly to their relation to the primary (secondary, ...) participant; those relations are represented by communications, and particularities of communications will be discussed later.

The leftmost rule dictates the overall left-to-right organization of the participants, however there are other factors that can be leveraged to further improve the layout. Foremost the size of each participant is given by the number of states and activities it contains, therefore it will vary in size. Additionally the connections (communications) between participants will often originate from different places on the sides of the participant, allowing us in some cases to place secondary participants on top of each other. Such vertical stacking has multiple benefits from aesthetical perspective: the overall area is smaller, edges are shorter, and there are fewer edge bends.

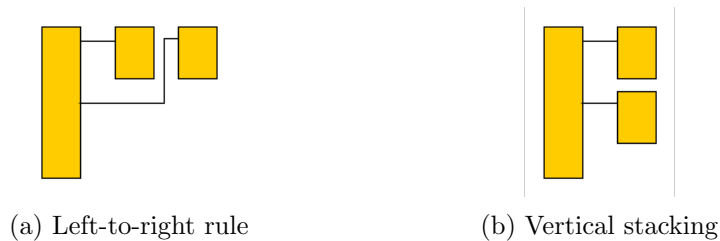


Figure 7.3: Comparison of participants organization

Yet another option is to place a participant to the left of the initiating one. However one needs to exercise caution when doing so in order to avoid confusion about which participant is the primary one. This can be resolved by placing the participant in question distinctly below the starting horizontal line of the primary one. Also being smaller in size will help with visual distinction.

As this violates the left-to-right rule it should be done only if it provides significant benefits to the overall diagram, such as freeing valuable space and removing larger number of line crossings or bends. This is up to the judgment of the modeler.

Finally, an automatic layout generally shouldn’t change user-defined ordering of participants. Often the ordering is context-sensitive and user would

<sup>14</sup>In the future we would like to enable expanded diagram overview.

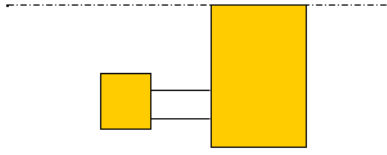


Figure 7.4: Participant placed to the left of the primary one

prefer to keep it that way — the proximity aesthetic criterion takes precedence over any other.

Automatic layouting of participants is however still valuable in situations where the layout is not yet available, such as when the diagram is generated from a DSL.

### 7.3 Participant’s finite state automaton

Problem of participant’s content will be split into three parts: layouting of FSA without back edges, layouting of back edges, and service-oriented participants.

The non-FSA parts of participants have fixed position — name in the top left corner, and icon of a type in the top right corner.

#### 7.3.1 Layouting without back edges

Participant’s FSA is composed from states and activities. Each state is represented by a rectangle (which contains the state’s name). Transitions between states are then accommodated with directed polylines. Additionally each transition has an associated activity. An activity is a rectangle with rounded corners. Visually the transition leads from the source state to the activity as a undirected edge, and then from activity to the target state as a directed edge. From the perspective of the layouting we can however treat both activities and states equally with the exception that an activity is associated with a single transition, and therefore it can contain only single incoming and outgoing edge.

Process represented by FSA is traditionally flowing from top to the bottom. This complements the left-to-right flow of participants and therefore they together evenly populate the area of the diagram.

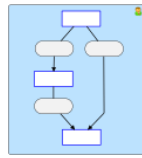
To layout the automaton we can take advantage of several factors. Because we are representing process, we know that it will contain a beginning and an end with flow generally going in single direction (down). This direct flow should be in fact enforced, as it enables the modeler or reader at a glance determine the relative position within the process or the overall state of it. Thus even if vertical space could be saved by creating upward transition, such

compromise would impede the overall readability as it would create confusion with real back edges.

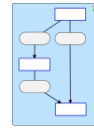
So if we for the moment ignore back edges, the whole process is a directed acyclic graph and can be organized with topological ordering.

From analyzed set of ORD diagrams<sup>15</sup> it was also observed that the average vertex degree is very low – usually two to four. The main reason is that most modeled business decisions are of boolean character — accepted/refused, available/unavailable, etc. Outgoing degree higher than two is mostly found in enumeration of several different equal cases (buying a family car, a truck, an SUV, ...), however such situations are relatively uncommon. With such low vertex degree and directed acyclic graph the automaton can be organized with topological ordering without *any* edge crossings at all and with minimal edge lengths and bends.

Finally it is noteworthy to determine the appearance of edges that span several layers in the flow, or they present fork/join situations.



(a) Symmetrical fork



(b) Side-track fork

In figures 7.5a and 7.5b are applied two competing rules — symmetry versus minimal edge length and bends. However each of those cases can tell a different story.

By maintaining symmetry in figure 7.5a we make both paths visually equal. This is beneficial if the choices are indeed of equal importance.

The figure 7.5b on the other hand shows one direct path and one *side-track*. Such layout could be used if the side-track is a marginal case that is taken only occasionally or under special circumstances, and which most of the time the reader is not interested in. Example could be an employee asking a receptionist in a building to open turnstile. This would be an edge case if every employee is expected to have entrance card that allows him to pass the turnstile directly, without the need to ask the receptionist (and proving his identity to him).

Unfortunately choosing one option over another in automatic layouting is problematic, as the weight of the choices is not semantically defined in the represented model. Thus we feel that user should have the option to choose the appropriate case and the layouting algorithm should honor and remember the choice. If not specified, the *equality* choice should be the default one as a safer option — choosing unnecessary evaluation of minor case over accidental undervaluing of an important one.

---

<sup>15</sup>Mostly student works from Process engineering university class.

### 7.3.2 Layouting of back edges

As mentioned before, back edges often represent exceptional behavior resulting in a *rollback* of the process' progress; this is visualized by an upward flow of the transition.

Semantically back edges can only end in one of the previous state nodes. If the endpoint was an activity node, it would result in incorrect behavior — an activity node can be connected only to one and only one incoming edge. Thus if we have back edge into an activity node, there would have to be already an incoming edge, otherwise this wouldn't be an back edge situation. There is visually one possible exception, it will however be dismissed in the next section 7.3.3.

The back edge itself is visualized as an orthogonal polyline; unlike regular transition which always starts from the bottom of one node and ends in top of another, the back edge can freely start on an appropriate side of both the source and target nodes. The side should be chosen based on surrounding elements to minimize possible edge crossings. Given that most processes are planar, we can route the back edge alongside the border of the participant's container box, where the only possible crossing would be with communications or other back edges.

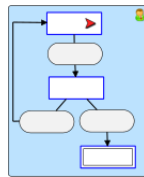


Figure 7.6: A participant with a back edge

### 7.3.3 Service-oriented participants

Service-oriented participants are a visual shortcut useful for representing a service with several independent use case scenarios. For example an ATM could provide the option to both withdraw, and to deposit money. Instead of creating full automaton with initial and final state, transitions, and possible constraints, all unnecessary parts are hidden, which results in simpler and more readable diagram. This is however purely visual change and doesn't affect the model itself (cf. diagram settings of UML diagrams 6.3).

As mentioned in previous section, this simplification could create the impression that it is possible to create back edge into an activity node since there is no visible incoming edge. Such impression would be incorrect as there is incoming edge already present, only hidden from the visualization.

From layouting perspective, service-oriented participants give the layouting more freedom, as the activity nodes could be moved around freely without

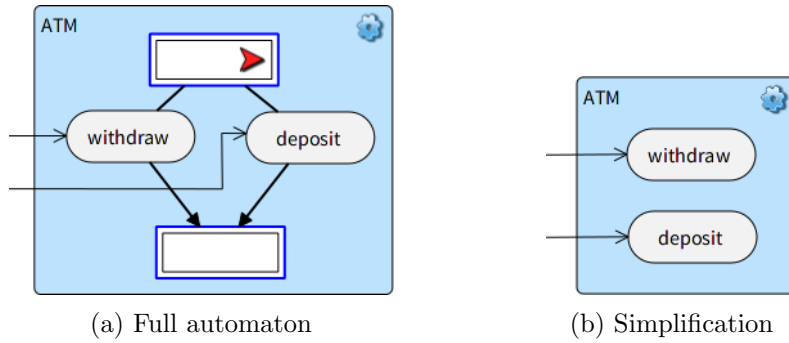


Figure 7.7: Possible visualizations of a service-oriented participant

worrying much about the rest of the participant's content.

## 7.4 Communications and data flows

Communication in ORD model is a communication scenario between two participants. Because communication is an active action, it is always performed between an activity node of the initiating participant and an activity of the target participant.

Communication within a participant is not permitted, because such situation would lead to a deadlock; participant has to be in such state that it can both send and receive this transmission. Thus it would require it to be in two states at the same time, which is not possible.

The notation for communication is a directed line from the initiating activity to the target one. The communication itself however can be bidirectional, thus one participant can ask a question and receive an answer within the same communication frame. If such question or answer, or more generally any object is of significance exchanged during the communication, it can be modeled with *data flows*.

Data flow is an object exchanged during communication, it can be any meaningful material or conceptual object such as confirmation request, money payment, restaurant order, meal, etc. The graphical notation is a box with label describing the content and an arrow denoting its type. This arrow can be either oriented in the same direction as the communication line — an *input* data flow sent from the initiator, or in the opposite direction which is an *output* data flow sent from the target and received by the initiator.

Because participants can be moved freely around, the orientation of a communication can change (i.e. swapping position of two communicating participants). Therefore the data flow arrows must also dynamically update to correctly preserve their meaning.

The exact position of data flows on the communication line is not defined, however we recommend placing them close their source participants (input

close to the source participant and output close to the target one). This enables the reader to quickly determine what the participant has to send without the need to trace the whole communication, which is especially important for communications that spans larger portions of the graph or route around other objects (such as 7.11).

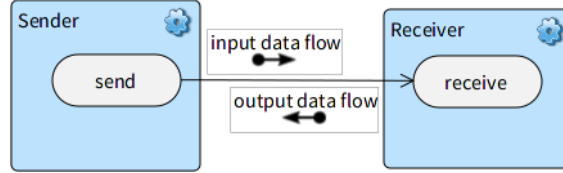


Figure 7.8: Communication with two data flows

### 7.5 Layouting Algorithm

As we have described the layouting needs and visual representation, we can now propose a layouting procedure for the diagram.

From previous sections we can summarize that there are several distinct areas — layouting of participant’s content, organization of participants in the diagram, and routing of communications.

Thus the most natural option is application of multistage layouting algorithm. Each stage should act as an independent algorithm working on the result of the previous one. This in principle follows the topology-shape-metrics technique, including all it’s benefits and limitations.

#### 7.5.1 Participant’s content

The first stage is layouting of the automaton. The reason for selecting this as first is simple; the layout of the automaton changes the size of the participant’s container and thus affects diagram organization and communications routing.

As stated earlier, the automaton can be in most cases drawn as planar downward directed acyclic<sup>16</sup> graph. Two possible candidates to layout such graph emerge here: Sugiyama hierarchical layout and dominance tree layout, both already available in the Roassal library. Even though there are no direct disadvantages to either of them, we assert that dominance tree would be a more appropriate choice as it maintains node layers.

The dominance tree layout is already provided by Roassal, however there are extra steps required. Firstly the data fed to the layouter needs to be prepared by removing back edges from the graph. After the application of the layout bending of some lines might be required.

---

<sup>16</sup>Acyclic assuming back edges are ignored.

As Roassal’s layouts do not understand the notion of polylines, we need to add bends explicitly. The bend points should be introduced in such a way that symmetry is preserved. If there is a fork/join scenario with uneven number of elements, the lines should still maintain symmetry by adding additional bend points.

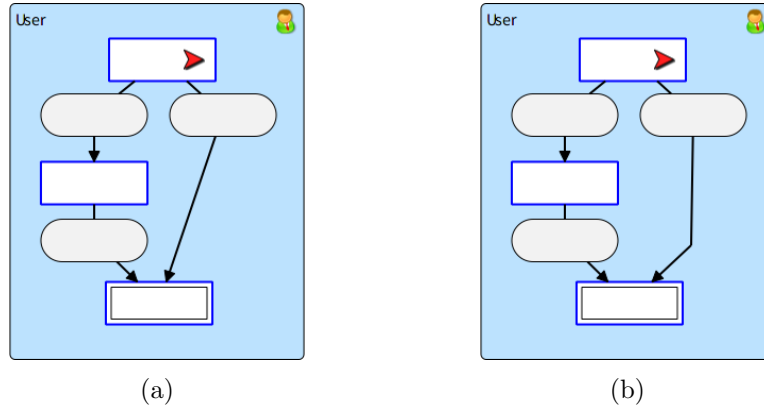


Figure 7.9: Addition of a line bend point to maintain line symmetry

#### 7.5.1.1 Back-edges

In all analyzed cases a back edge can be created with exactly three bend points. The goal is to route the back-edge to the side of the participant, up towards the target, and finally to the side of the target. The first and last step can result in edge crossing with other transitions. However this is not of major concern as such situations are rare<sup>17</sup> and the back edge is horizontally oriented whilst regular transition is oriented downwards, so there is no confusion. Repositioning of nodes could be considered in the future to further improve the layout. The crossing between several back edges is not addressed as such situations are rare, but the main concern should be avoiding complete overlapping of the lines.

The last step of participant’s layouting is adjusting the size of the container box so all elements fit in. This includes also the label and type, and an extra inner margin.

#### 7.5.2 Organization of participants

The second stage of the overall layout is organization of participants. The default chosen layout, as discussed earlier is left-to-right flow. The order of participants is based on the order of communications — the target of the

<sup>17</sup>As we’ve pointed out earlier, the automaton’s graph is usually very simple and allows for planar drawings.

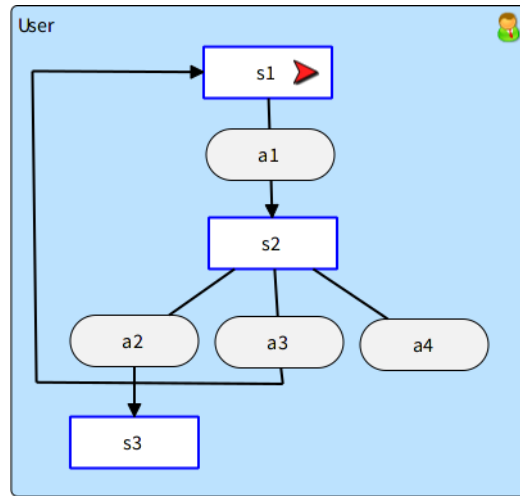


Figure 7.10: Back edge with a crossing

primary participant's first communication is also the next participant, and so forth. If the layout was manually altered we do not recommend reorganization of the participants, as it would break user's mental map and the ordering is context-sensitive.

### 7.5.3 Layouting of communications

Firstly we define the position of data flows. Generally we place data flows next to their respective source. Thus input data flows are next to the sender/initiator of the communication and output data flows next to the receiver. The vertical position (above or below) is not important, however crossing with the line should be avoided.

With data flow's positions determined, the participants should be nudged apart so there is enough space between them for both the data flows and communication routing.

Finally the connections are routed with an orthogonal router and data flows are placed accordingly.

### 7.5.4 Summary

To summarize all steps of the layout:

1. Participant's automaton layouting
  - Layouting of the graph without back edges with dominance tree layout
  - Adding line bends to achieve symmetry
  - Routing the back edges
  - Adjusting container size

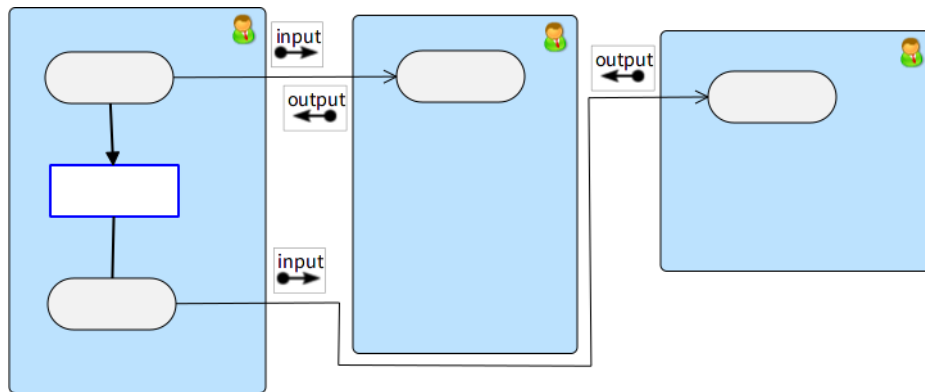


Figure 7.11: Multiple participants with communications and data flows

2. Ordering participants (optional)
3. Layouting communications
  - Determining data flow positions
  - Increasing spacing between participants
  - Routing communications
  - Placing data flows

As with topology-shape-metrics approach, this phasing enables us to solve one problem at a time without worrying about the rest, however just like with TSM we may deprive ourselves of the best layouts as we would need to consider all properties at once, not in sequence, to achieve them.

## 7.6 Conclusion

In this chapter we have introduced the notation of BORM Object Relation Diagrams, assessed the needs of layouting such diagrams and proposed algorithmic solution. As the quality of the algorithm's steps rests on quality achieved in the Layouting3 chapter, we have achieved only partial success and more work is required. Furthermore we have implemented the necessary notation and BORM ORD model, none of which were discussed here in detail, as they would be of little value to the reader, and the model is already described in previous literature[39][38]. Finally we recommend considering devising a specification of the diagram notation in the likes of UML DI/DD in the future, as the current implementation is ad-hoc.



---

# Testing

To ensure certain level of quality several techniques were employed. Although hard metrics such as number of tests, code coverage , etc. exists, their interpretation is often very subjective. Furthermore there are many other hardly measurable ways to further improve and ensure quality such as alpha/beta testing, *eating your own dogfood*<sup>18</sup>, etc. Finally this is all is amplified by the fact that not all code produced requires the same level of quality. As the DynaCASE tool is of experimental nature with many parts unexplored, exploratory coding and prototyping is commonplace. For this reason we had no specific quality level that we would aim for.

That being said, we have written 127 unit tests to test the implementation. At the time of writing, all tests are passing.

We will also briefly explore several tools and concepts that were used during the implementation to achieve higher level of product quality, especially in the instance of testing layouting, which is by its very nature hard to unit test.

## 8.1 SUnit Tests

Unit tests have a long tradition in Smalltalk as the SUnit framework<sup>19</sup> is the “mother of all unit testing frameworks”. Therefore large portion of code written is covered by tests (both unit and integration tests).

Additionally we are currently exploring the Hapao Test Coverage Tool<sup>20</sup>. Hapao is capable of both calculating code coverage as well as visualizing it (using the Roassal library). Although we have discovered this tool relatively late in the implementation, we consider it a valuable tool for future works as

---

<sup>18</sup>Directly using your own product for its intended purpose (contrasted by producing software for third party without ever having the pleasure or horror of actually using it).

<sup>19</sup><http://sunit.sourceforge.net/>

<sup>20</sup><http://objectprofile.com/Hapao.html>

it gives us better insight into overall test coverage and guides us to parts that are poorly covered.

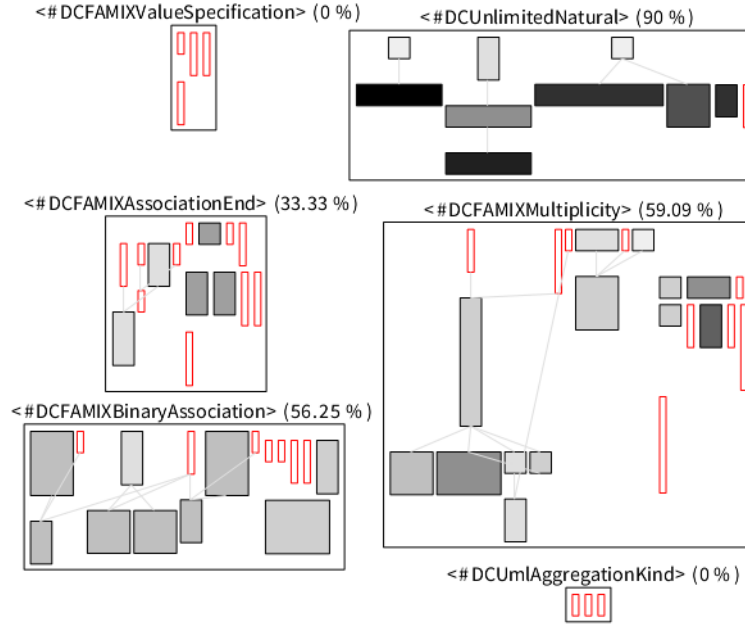


Figure 8.1: Hapao test coverage visualization applied on our FAMIX extensions

Figure 8.1 shows an example Hapao visualization applied on one of the packages implemented here. Each colored box represents a method (the big container is a class), the height is lines of code of the method, width the number of senders and color coverage (red = not tested, black = well tested, gray = tested). From the visualization we can also see that some classes (e.g. ValueSpecification) would require more tests (as it has been tested only by use or integration tests of other packages depending on it).

## 8.2 Continuous Integration

One of the tasks of continuous integration (CI) is running the complete test suite of the whole DynaCASE package. As running the whole suite may be time and resources demanding a programmer wouldn't usually do it after each modification. Using another system to run the tests also adds another environment on which the tool is tested (and thus partially mitigating "works on my machine" problem). The CI is also capable of creating builds which can be downloaded with a single click and used by users without the need to perform complex building/package loading operations.

DynaCASE is using Jenkins CI hosted on Inria<sup>21</sup> servers. This hosted CI is also used by Pharo itself, the Moose platform, as well as large portion of Pharo Community. Although it serves its purpose, we are finding it increasingly tedious to use for building many components of DynaCASE separately (as we have only one job available). Furthermore it relies on availability of the servers (which stop working from time to time). If however one is building a Pharo project we would recommend it.

An alternative solution is currently emerging — Travis CI<sup>22</sup>. As the DynaCASE project is a composition of several independent projects (DynaCASE core, BORM model, BORM editor, UML model, UML editor, UML DSL, ...) we found it not possible to test each project in isolation with Jenkins. For this reason we have opted to also use Travis in addition to Jenkins. In this way we can have a separate CI job for each of the projects. As the project, named smalltalkCI<sup>23</sup> is still in early development there are things to be desired, however even in this stage we found it very beneficial.

## 8.3 Visual Debugging

Although many parts of DynaCASE were written in the *test driven development* (TDD) style, when implementing layouting we have very quickly discovered that unit testing, although important and used, is not of much help for understanding the core of the problem, as even small scale graphs were producing large amount of data. This is a result of inherent complexity of the layouting problems, as large solution spaces have to be evaluated. To solve this problem we were prototyping the layouting algorithms with visual debugging in mind — at each step we considered a good way to visually represent the data.

Figure 8.2 shows us the striking difference between simply presenting list of data (which would be the main produce of unit testing) and a visualization of the same. Such visualization is priceless during development and debugging, and can even be used later to better communicate the purpose of each step of complex layouts.

---

<sup>21</sup><https://www.inria.fr/>

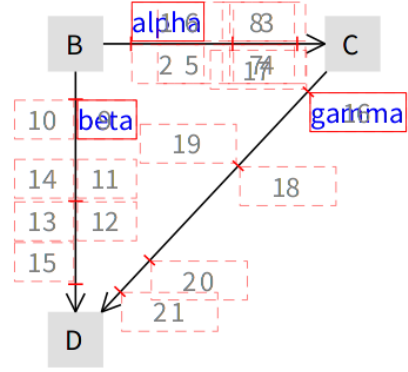
<sup>22</sup><https://travis-ci.org/>

<sup>23</sup><https://github.com/hpi-swa/smalltalkCI/>

## 8. TESTING

---

1	(51.0@-2.0) corner: (90.0@19.0)
2	(51.0@21.0) corner: (90.0@42.0)
3	(106.0@-2.0) corner: (145.0@19.0)
4	(106.0@21.0) corner: (145.0@42.0)
5	(65.0@21.0) corner: (104.0@42.0)
6	(65.0@-2.0) corner: (104.0@19.0)
7	(100.0@21.0) corner: (139.0@42.0)
8	(100.0@-2.0) corner: (139.0@19.0)
9	(21.0@51.0) corner: (53.0@72.0)
10	(-13.0@51.0) corner: (19.0@72.0)
11	(21.0@63.0) corner: (53.0@104.0)
12	(21.0@106.0) corner: (53.0@127.0)
13	(-13.0@106.0) corner: (19.0@127.0)
14	(-13.0@83.0) corner: (19.0@104.0)
15	(-13.0@128.0) corner: (19.0@149.0)
16	(147.0@47.0) corner: (199.0@68.0)
17	(93.0@34.0) corner: (145.0@45.0)
18	(109.0@87.0) corner: (161.0@108.0)
19	(55.0@64.0) corner: (107.0@85.0)
20	(61.0@138.0) corner: (113.0@159.0)
21	(45.0@155.0) corner: (97.0@176.0)



(a) List of all considered label positions

(b) Visualization of the considered positions

Figure 8.2: Considered positions during edge labeling

---

# Conclusion

## Faced Challenges

During the literature exploration, and specially during attempts to properly implement usable layouting algorithms we have realized the sheer complexity of the task. For example the dissertation [37] focuses only on the topology-shape-metrics approach. Similarly a bachelor thesis “Implementing an Algorithm for Orthogonal Graph Layout” [40] also focuses solely on researching and implementing the same algorithm, while already having a strong support of its underlying platform and libraries (Eclipse, KIELER<sup>24</sup> respectively).

But the focus of this thesis was much broader — implementing two different notations (BORM ORD, UML Class Diagram), researching and implementing associated layouting (for BORM ORD there is no prior literature); and that all in an experimental platform DynaCASE on non-mainstream technologies Pharo and Roassal without any preexisting graph or layouting libraries to start from.

Considering all these factors — the complexity of layouting combined, breadth of the assignment, use of non-standard technologies (or complete lack thereof) we consider it not possible to achieve usable layouts in the extent laid out in the beginning, therefore we were forced to reach for compromises.

## Achieved Results

Although the implementation of layouting algorithms was met with less than mixed success, we cannot completely dismiss the achieved results.

Firstly, we have explored the broad area of layouting. This gives us much better standing in the future as we can now better estimate the amount of work required to implement a particular layouting, we have obtained deeper understanding of many of the concepts in the area of layouting, and we have

---

<sup>24</sup>Kiel Integrated Environment for Layout Eclipse Rich Client

better insight into what foundations will be required to implement good layouting — namely the development of an extensive graph library.

Secondly, we have analyzed the layouting needs of BORM Object Relation Diagrams. Unlike Class Diagrams, where a large amount of existing literature already exists, to our knowledge there was no prior analysis of ORD, and thus we can consider our findings new.

Thirdly, we have implemented a variety of utilities that support interactive layouting, such as grid, grid snapping, local edge labeling, or polylines. So although we haven't achieved fully self-sufficient automatic layouting, implementation of those auxiliary utilities will provide great aid in speeding up and improving manual layouting.

And lastly, we have successfully implemented all the necessary models and notations for both UML Class Diagrams, and BORM Object Relation Diagrams.

## Future Work

As we actively continue the development of the DynaCASE platform, having automatic layouting to the extent we were initially hoping for remains of high interest to us. To achieve this, we will begin work on a dedicated graph library that would provide us with a systematic way of graph manipulation, a wide variety of common graph algorithms (finding shortest paths, set partitioning, vertex cover, ...), and a set of performant advanced algorithms such as assignment solvers, minimum cost maximum flow algorithms, etc. Using one of Pharo's FFIs<sup>25</sup> and interfacing with some existing C or C++ library is also a solution that could be considered.

Among other common notations, we originally planned to also include DEMO (Design & Engineering Methodology for Organizations), however after a consultation with my supervisor we have concluded that the implementation of DEMO model, notation and layouting would be far more complex. We believe that such task would even exceed the extent of bachelor studies altogether, as the implementation would require a lot of programming experience, as well as lot of experience with DEMO modeling, which is part only of master's curriculum. Currently we have master students that are interested in implementing the DEMO methodology.

Finally, other work is being done around DynaCASE platform. For both BORM ORD and UML Class Diagrams we have developed textual DSLs. As the platform is capable of generating models (and diagrams) from those DSLs, having good layouting would be even more valuable. And lastly a source-code generator from UML Class Diagrams to Pharo Smalltalk was developed as part of external contract aimed at support of agent-based modeling.

---

<sup>25</sup>Foreign Function Interface

---

# Bibliography

- [1] Uhnák, P. DynaCASE modeling platform. [online; cit. 2016-01-11]. Available from: <https://dynacase.github.io>
- [2] Black, A.; Ducasse, S.; Nierstrasz, O.; et al. *Pharo by Example*. Square Bracket Associates, 2009, ISBN 978-3-9523341-4-0. Available from: <http://pharobyexample.org>
- [3] Bergel, A. Agile Visualization, [To be published est. mid 2016.]. Available from: <http://agilevisualization.com/>
- [4] Wilson, R. J. *Introduction to graph theory*. Prentice Hall, fourth edition, 1996, ISBN 978-0-582-24993-6.
- [5] Dubé, D. *Graph layout for domain-specific modeling*. Master's thesis, McGill University, 2006. Available from: [http://digitool.Library.McGill.CA:80/R/?func=dbin-jump-full&object\\_id=97943](http://digitool.Library.McGill.CA:80/R/?func=dbin-jump-full&object_id=97943)
- [6] Fuhrmann, H.; Spönemann, M.; Matzen, M.; et al. Automatic Layout and Structure-Based Editing of UML Diagrams. In *Proceedings of the 1st Workshop on Model Based Engineering for Embedded Systems Design (M-BED'10)*, Dresden, March 2010.
- [7] Galapov, A.; Nikiforova, O. UML Diagram Layouting: the State of the Art. *J. Riga Technical University*, volume 44, 2011: pp. 101–108. Available from: <http://dx.doi.org/10.2478/v10143-011-0027-0>
- [8] Eichelberger, H.; Schmid, K. Guidelines on the aesthetic quality of UML class diagrams. *Information & Software Technology*, volume 51, no. 12, 2009: pp. 1686–1698. Available from: <http://dx.doi.org/10.1016/j.infsof.2009.04.008>

- [9] Eichelberger, H. UML class diagrams – state of the art in layout techniques. In *Proc. VISSOFT 2003 – 2. Int. Workshop on Visualizing Software for Understanding and Analysis, IEEE*, 2003, pp. 30–34.
- [10] Sun, D.; Wong, K. On Evaluating the Layout of UML Class Diagrams for Program Comprehension. In *IWPC*, IEEE Computer Society, 2005, ISBN 0-7695-2254-8, pp. 317–326. Available from: <http://dx.doi.org/10.1109/WPC.2005.26>; <http://doi.ieeecomputersociety.org/10.1109/WPC.2005.26>
- [11] Falke, R.; Klein, R.; Koschke, R.; et al. The Dominance Tree in Visualizing Software Dependencies. In *Proceedings of the 3rd International Workshop on Visualizing Software for Understanding and Analysis, VISSOFT 2005, September 25, 2005, Budapest, Hungary*, edited by S. Ducasse; M. Lanza; A. Marcus; J. I. Maletic; M.-A. D. Storey, IEEE Computer Society, 2005, ISBN 0-7803-9540-9, pp. 83–88.
- [12] Tamassia, R. *Handbook of Graph Drawing and Visualization (Discrete Mathematics and Its Applications)*. Chapman & Hall/CRC, 2007, ISBN 1584884126.
- [13] Sugiyama, K.; Tagawa, S.; Toda, M. Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man, and Cybernetics*, volume 11, no. 2, 1981: pp. 109–125.
- [14] Spönemann, M. *Graph layout support for model-driven engineering*. Number 2015/2 in Kiel Computer Science Series, Department of Computer Science, 2015, ISBN 9783734772689, dissertation, Faculty of Engineering, Christian-Albrechts-Universität zu Kiel.
- [15] Gansner, E. R.; North, S. C. An open graph visualization system and its applications to software engineering. *SOFTWARE - PRACTICE AND EXPERIENCE*, volume 30, no. 11, 2000: pp. 1203–1233. Available from: <http://www.graphviz.org>
- [16] Chimani, M.; Gutwenger, C.; Jünger, M.; et al. *The Open Graph Drawing Framework (OGDF)*. CRC Press, 2012, to appear.
- [17] Di Battista, G.; Eades, P.; Tamassia, R.; et al. Algorithms for drawing graphs: an annotated bibliography. *Computational Geometry*, volume 4, no. 5, 1994: p. 235–282.
- [18] Kaufmann, M.; Wagner, D. (editors). *Drawing Graphs: Methods and Models*. London, UK, UK: Springer-Verlag, 2001, ISBN 3-540-42062-2.
- [19] Eiglsperger, M.; Gutwenger, C.; Kaufmann, M.; et al. Automatic layout of UML class diagrams in orthogonal style. *Information Visualization*,

- volume 3, no. 3, 2004: p. 189–208, ISSN 1473-8716, 1473-8724, doi:10.1-20057/palgrave.ivs.9500078.
- [20] Kaufmann, M.; Eiglsperger, F.; Eppinger, M. An Approach for Mixed Upward Planarization. *Graph Algorithms and Applications 4*, volume 4, 2006: p. 203.
- [21] Resende, M. G.; Ribeiro, C. C. A GRASP for graph planarization. *Networks*, volume 29, no. 3, 1997: p. 173–189.
- [22] Kakoulis, K. G.; Tollis, I. G. Algorithms for the multiple label placement problem. *Computational Geometry*, volume 35, no. 3, Oct 2006: p. 143–161, ISSN 09257721, doi:10.1016/j.comgeo.2006.03.005.
- [23] *An Algorithm for Labeling Edges of Hierarchical Drawings*. Lecture notes in computer science, 1997.
- [24] Kakoulis, K. G.; Tollis, I. G. A UNIFIED APPROACH TO AUTOMATIC LABEL PLACEMENT. *International Journal of Computational Geometry & Applications*, volume 13, no. 01, Feb 2003: p. 23–59, ISSN 0218-1959, 1793-6357, doi:10.1142/S0218195903001062.
- [25] Munkres, J. Algorithms for the Assignment and Transportation Problems. *Journal of the Society of Industrial and Applied Mathematics*, volume 5, no. 1, March 1957: pp. 32–38.
- [26] Baudisch, P.; Cutrell, E.; Hinckley, K.; et al. Snap-and-go: helping users align objects without the modality of traditional snapping. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, ACM, 2005, pp. 301–310. Available from: <http://dl.acm.org/citation.cfm?id=1055014>
- [27] Bier, E. A.; Stone, M. C. Snap-dragging. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '86, New York, NY, USA: ACM, 1986, ISBN 0-89791-196-2, pp. 233–240, doi:10.1145/15922.15912. Available from: <http://doi.acm.org/10.1145/15922.15912>
- [28] Hutchinson, J.; Whittle, J.; Rouncefield, M.; et al. *Empirical assessment of MDE in industry*. ACM, 2011, p. 471–480. Available from: <http://dl.acm.org/citation.cfm?id=1985858>
- [29] OMG. OMG Unified Modeling Language 2.5. Technical report, Mar 2015. Available from: <http://www.omg.org/spec/UML/2.5>
- [30] Ducasse, S.; Anquetil, N.; Bhatti, M. U.; et al. *MSE and FAMIX 3.0: an Interexchange Format and Source Code Model Family*. Nov 2011. Available from: <https://hal.inria.fr/hal-00646884>

- [31] Kuhn, A.; Verwaest, T. *FAME, a polyglot library for metamodeling at runtime*. 2008, p. 57–66. Available from: <http://core.ac.uk/download/pdf/1549752.pdf#page=57>
- [32] OMG. Diagram Definition (DD) 1.1. Technical report, Jun 2015. Available from: <http://www.omg.org/spec/DD/1.1>
- [33] Eichelberger, H.; von Gudenberg, J. W. *Uml class diagrams-state of the art in layout techniques*. Citeseer, 2003, p. 30–34. Available from: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.220.8667&rep=rep1&type=pdf#page=36>
- [34] Gutwenger, C.; Jünger, M.; Klein, K.; et al. *A new approach for visualizing UML class diagrams*. ACM, 2003, p. 179–188. Available from: <http://dl.acm.org/citation.cfm?id=774859>
- [35] Seemann, J. *Extending the Sugiyama algorithm for drawing UML class diagrams: Towards automatic layout of object-oriented software diagrams*. Springer, 1997, p. 415–424. Available from: [http://link.springer.com/chapter/10.1007/3-540-63938-1\\_86](http://link.springer.com/chapter/10.1007/3-540-63938-1_86)
- [36] Gamma, E.; Helm, R.; Johnson, R.; et al. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995, ISBN 0-201-63361-2.
- [37] Eiglsperger, M. *Automatic layout of UML class diagrams: a topology-shape-metrics approach*. Dissertation thesis, Universität Tübingen, Nov 2003.
- [38] Podloucký, M.; Pergl, R. Towards Formal Foundations for BORM ORD Validation and Simulation. In *ICEIS (2)*, edited by S. Hammoudi; L. A. Maciaszek; J. Cordeiro, SciTePress, 2014, ISBN 978-989-758-028-4, pp. 315–322. Available from: <http://dx.doi.org/10.5220/0004897603150322>
- [39] Knott, R. P.; Merunka, V.; Polák, J. Process Modeling for Object Oriented Analysis Using BORM Object Behavioral Analysis. In *ICRE*, IEEE Computer Society, 2000, ISBN 0-7695-0565-1, pp. 7–16. Available from: <http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=6907>; <http://www.computer.org/csdl/proceedings/icre/2000/0565/00/index.html>
- [40] Claussen, O. *Implementing an Algorithm for Orthogonal Graph Layout*. Bachelor thesis. Christian-Albrechts-Universität zu Kiel, Sep 2010.

## Acronyms

**BORM** Business Object Relation Modeling

**BORM ORD** BORM Object Relation Diagram

**UML** Unified Modeling Language

**DEMO** Design & Engineering Methodology for Organizations

**DC** DynaCASE

**DynaCASE** Dynamic Computer-aided Software Engineering

**DD** Diagram Definition

**DI** Diagram Interchange

**ORD** BORM Object Relation Diagram

**FSA** Finite State Automat(on/a)

**DSL** Domain Specific Language



## Contents of enclosed CD

	readme.txt .....	the file with CD contents description
	exe .....	the directory with executables
	pharo-vm .....	directory containing pharo virtual machine executables
	dynacase.image .....	pharo image file containing the implementation
	dynacase.changes .....	pharo changes file containing the implementation
	dynacase.sh .....	dynacase shell launcher
	src .....	the directory of source codes
	repository .....	export of the source code in the filetree format
	thesis .....	the directory of L <sup>A</sup> T <sub>E</sub> X source and build codes of the thesis
	text .....	the thesis text directory
	thesis.pdf .....	the thesis text in PDF format
	thesis.ps .....	the thesis text in PS format