# Chapter 1

# Agile Graph Building

**Book Todo:**
TODO: introduce useInLayout

Graph, composed of nodes and edges, is one of the most popular visual representations for software dependencies. Advantages to represent dependencies as a graph are multiple. First, graphs are structures that are both intuitive and scalable. Second, graph modeling is well understood with a strong mathematical background. Numerous algorithms are available to carry out a wide range of analysis.

The importance of defining and manipulating graphs has lead to a profusion of languages to describe and specify graphs. However, it appears that most of these languages are unfit to cope with the large range of properties software components have to be visually associated with. Consider *Graphviz*, a popular software to visualize graphsfootnote{url{http://www.graphviz.org}}. Graphs are described in Graphviz using *Dot*, a domain-specific language that offers sophisticated constructions to define nodes, their connections and complex layouts. Consider the following example, an except from the official manual of Dot to represent a control flow (http://www.graphviz.org/pdf/dotguide.pdf):

```
digraph G {
    node [shape=box,style=filled,color=".5 .5 .5"];
    main -> execute;
    node [shape=box,style=filled,color=".2 .2 .2"];
    main -> init;
    node [shape=box,style=filled,color=".3 .3 .3"];
    main -> cleanup;
}
```

Each function is represented with a colored box. The shape of a node may be explicitly defined, as illustrated above, in case it has to be shaded using

a particular color. Dot suffers from a gap between *what* is represented and *how* it is represented. The connection between the graph (*i.e.,* the produced visualization) and the represented code (*i.e.,* the method main, execute) is not explicit in the Dot program: the code given above *draws lines between colored labeled nodes* whereas a practitioner wishes to visualize *dependencies and metrics between methods*. This gap has several serious consequences, including (i) verbose script containing duplicated code and (ii) jumping in both directions from the visualization to the code is costly in terms of manual human actions. Not properly addressing the gap between the visualization and the code inevitably leads to long and repetitive program description.

We propose *Graph*, a domain specific language to visualize software dependencies as a graph. Produced visualizations aim to assist software (re)engineers to carry out maintenance or comprehension analysis.

The key difference between *Graph* and traditional graph-description languages is the relation between visual elements and dimensions and the application to visualize. To visualize a software with Dot, one has to define a box with a label and numerical values to define its size and color. With *Graph*, one has to associate colors and size to software metrics and provide classes and methods as input. As a result, scripts written with *Graph* are short, concise and efficient.

## 1.1 Examples of Graph usage

As a running example to exhibit the characteristics of Graph, we use the code of Roassal itself. The visualization identifies dependencies toward Trachel, a low-level vectorial engine which is a subcomponent of Roassal. Roassal is composed of over 200 classes to implement shapes, layout, and many other things.

Figure 1.1 is a visualization of the Roassal application source code. This visualization shows the dependencies between classes as mapped on the class hierarchy. This visualization exhibits two typical requirements. First, it shows how more than one kind of relationships need to be visualized while still presenting an understandable structure. Second, it shows how to distinguish between structural relevant parts based on some heuristic (in this case, coloring nodes based on a convention).

Each circle is a class. The size of the class indicates the number of methods defined in the class. Gray lines indicate inheritance links. Colors indicate main components of Roassal: purple indicates Trachel, a low-level vectorial engine; green indicates visual shapes; yellow indicates graph layout algorithms. Classes that belong to other components are gray.

Blue lines indicate dependencies between Roassal's classes toward Tra-

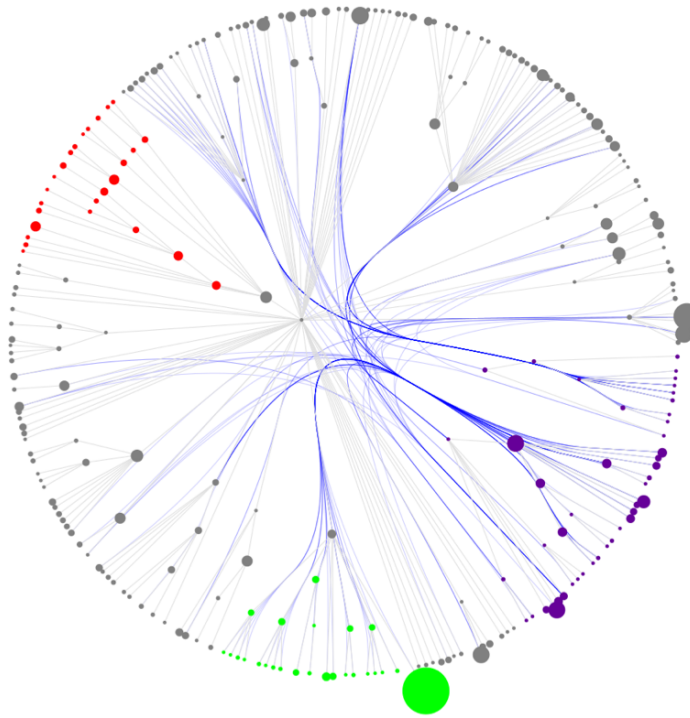

Figure 1.1: Visualization of a software system.

chel. Lines follow the class hierarchy, in which each control point is a superclass. Following a relevant hierarchical structure is effective at reducing edge cluttering. Edges have a transparency ratio of 0.2 to indicate dependency accumulation.

Figure 1.1 is produced by the following code:

```
b := RTGraphBuilder new.
b nodes if: [ :c | c inheritsFrom: RTShape ]; color: Color green.
b nodes if: [ :c | c inheritsFrom: RTLayout ]; color: Color yellow.
b nodes if: [ :c | 'TR*' match: c name ]; color: Color purple.
b nodes color: Color gray.

b edges
   connectTo: #subclasses;
   useInLayout.

b edges
   connectTo: #dependentClasses;
```

```
    follow: #superclass;
    if: [:from :to | ('RT*' match: from name) and: ['TR*' match: to name]];
    color: (Color blue alpha: 0.2).

b layout cluster.

b global normalizeSize: #numberOfMethods min: 5 max: 60.

b addAll: RTObject withAllSubclasses.
b addAll: TRObject withAllSubclasses.
b addAll: TREvent withAllSubclasses.
b build
```

Line 1 creates an instance of the class RTGraphBuilder and assigns it to the variable b. Line 2 is a node declaration to fill in the classes in green that inherit from the class RTShape. Line 3 colors all subclasses of a layout class in yellow. Line 4 colors all Trachel purple. A Trachel class begins with the two character TR. Line 5 defines the default color for nodes not previously matched. Lines 7-9 define a group of edges representing the superclass relation between classes. These edges are used for the layout, specified later on Line 17. Lines 11 - 15 visualize dependencies starting from Roassal (*i.e.,* classes with a name beginning with RT) and to Trachel. Edges follow the superclass relation between classes (Line 13). Line 19 defines a global production rule: each class has a size representing its number of methods. The class with the lowest number of methods is 5 pixels wide. The largest class is 60 pixels wide. Lines 21-23 feed the program with the classes to analyze. Line 24 renders the visualization.

## 1.2 Program Structure

This section describes and illustrates the domain-specific language (DSL) offered by *Graph*. A program $P$ is composed of 5 distinct parts: $P = N\ E\ L\ G\ I$. Where $N$ = node production rules, $E$ = edges production rules, $L$ = layout rules, $G$ = global rules, $I$ = input. The interpretation of the program occurs at each elementary feed using the keyword addAll:.

The input corresponds to a model of a software system to be visualized. *Graph* is not tied to a particular code model: this papers uses the Pharo code model and we regularly use *Graph* with FAMIX models. A code model typically describes (i) software structural entities, such as packages, classes, and methods, (ii) software metrics, and (iii) navigation functions. This papers uses the numberOfMethods metric and the dependentClasses and superclass navigation functions.

Each part of a program may be composed of zero, one or more *production*

*rules*. In our example, the node part is composed of four production rules and the edge part composed of two production rules. A rule assigns a visual attribute to a semantically related group of elements. For examples, all classes inheriting from RTShape are in green. Each rule may be accompanied with a condition that scopes the effect of the production rule.

## Nodes

A node represents a software element that represents a particular entity of the software system. Most of the time a node represents a structural element of the analyzed software, a package, a class, or a method. *Graph* supports alternative code models, which is key to have a reusable DSL.

Nodes are defined using the keyword nodes. A number of keywords may be employed to define the visual representation of a node and interactive actions the user may trigger. Two shapes are available rectangle and ellipse. Color, size, height and width may be set. Each node may be labeled.

### Visual shape

Shapes for nodes and edges reflect properties and metric values. A node is typically shaped as a rectangle or a circle. Edges are typically straight lines, bezier curves possibly arrowed. Instead of providing direct numerical values to boxes and lines, *Graph* allows metric functions to be used in place of numerical values. Metrics are then computed against elements between the software entity represented by the node or the edge.

Each node is associated with a model element given as input of the program. Visual parameters of a node may reflect numerical value or metrics computed on the model element.

### Scoping

Being able to carefully select nodes and edges that matter for a particular software analysis task is crucial. Scalability of the visualization, especially when dealing with software dependencies, is a major obstacle. Being able to precisely define the scope of an analysis is crucial.

The scope of node production rule may be set using the if: keyword. This keyword takes a block function as argument to indicate whether or not the model element has to be considered by the production rule. In the example given at the beginning of the chapter, each colored node has a particular scope, subclasses of RTShape or RTLayout, or classes named after TR*.

## Edges

An edge represents a directed relationship between two nodes, typically representing a dependency between two software entities. Dependencies in an object-oriented language may be diverse. For example, a class $B$ may depend on another class $A$ by being a subclass, using $A$'s methods, or simply having duplicated source code duplication.

The edges production rule links nodes to each other. A proper visual aspect of an edge is driven by many different parameters. This section describes the linguistic constructs for edges.

### Connecting

The two edge extremities are specified using the connectFrom: and connectTo: keywords. For each element, the starting point of an edge is given by connectFrom: and the ending point by connectTo:. These two keywords accepts a function as argument that returns some software entities.

```
b := RTGraphBuilder new.
b nodes color: Color gray.
b edges
   connectFrom: #superclass;
   useInLayout.
b layout tree.
b addAll: (RTShape withAllSubclasses).
b build
```

In the example above, nodes are classes, subclasses of RTShape. Each node is connected to its superclass. The expression [:cls | cls superclass] is applied to each node, which designates the superclass node. The example shows the shorthand superclass. Note that the instruction connectFrom: #superclass, which connects each class to its superclass, has the same visual effect as connectTo: #subclasses, which connects each class to its subclasses.

### Directed line

The direction of an edge is typically indicated with an arrow. Dedicated keywords are available for classical direction indicator (diamond, arrow, arrowHead). In addition, our DSL offers the possibility to have curved lines to indicate edge orientation. The following example indicates dependencies between shapes of Roassal:

```
b := RTGraphBuilder new.
b nodes color: Color gray.
b edges directed; connectTo: #dependentClasses; useInLayout.
b global
```

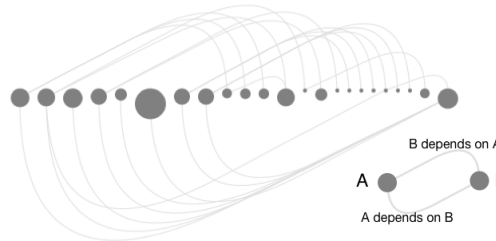


Figure 1.2: Directed line.

```
    normalizeSize: #numberOfMethods min: 5 max: 40 using: #ln.
b layout horizontal.
b addAll: (RTShape withAllSubclasses).
b build
```

Classes are in gray and horizontally lined up. The size of each class indicates its number of methods. A line indicates dependencies between classes. The keyword dependentClasses, when sent to a class *c* returns the list of classes *c_i* that depend on *c* (*e.g.*, *c_i* may be a subclass of *c*, at least one of *c_i*s method reference to $c$). The lines indicates that many classes depend on the right-most class.

**Follow**

Lines may follow a path of controlling elements to form a bundle. The follow: keywords specify the path for both the starting and ending points. Assuming the following:

- *e_s* and *e_e* are the starting and ending elements, respectively
- *f* the function provided to the follow: keyword

The control elements are formed up to the common ancestor element (if any) from the starting and ending point. Consider the following example:

```
b := RTGraphBuilder new.
b nodes if: [ :c | '*Line*' match: c name ]; color: Color red.
b nodes color: Color gray.

b edges
    follow: #superclass;
    connectTo: #dependentClasses;
```

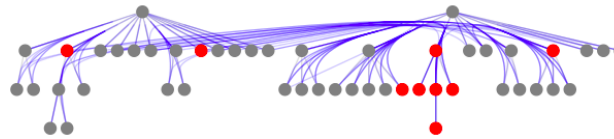

Figure 1.3: Bezier following a given path.

```
    color: (Color blue alpha: 0.2).

b edges connectFrom: #superclass; useInLayout.
b layout tree.
b addAll: TRShape withAllSubclasses, RTShape withAllSubclasses.
b build.
```

Figure 1.3 shows two class hierarchies, Trachel and Roassal. A superclass is above its subclasses, and inheritance is indicated with gray lines. A class is colored in red if it contains the word "Line" in its name. Blue lines indicate dependencies and follow the class inheritance links.

## Layout

Nodes have to be properly spatially located to communicate a particular situation. A large number of layouts exists, for which some of them are driven by edges.

*Graph* supports a number of well-known layouts (force, flow, circle, grid, cluster, vertical, horizontal, tree, sugiyama, translateBy:, pushAway:). However, it often happens that complex visualizations require more than a "simple" layout to visualize graphs. It has been shown that implementing a complex layout is a difficult programming activity. The layout production rule partially addresses this problem by featuring two properties:

- *Partitioning* : A particular layout may be applied to a graph subset. Partitioning is useful when the graph is too large to be efficiently laid out or when the graph is composed of semantically distinct clusters.
- *Composition* : A succession of layouts may be applied to achieve a particular space distribution.

The following example illustrates the graph partitioning and layout composition mechanisms:

```
b := RTGraphBuilder new.
b nodes
```

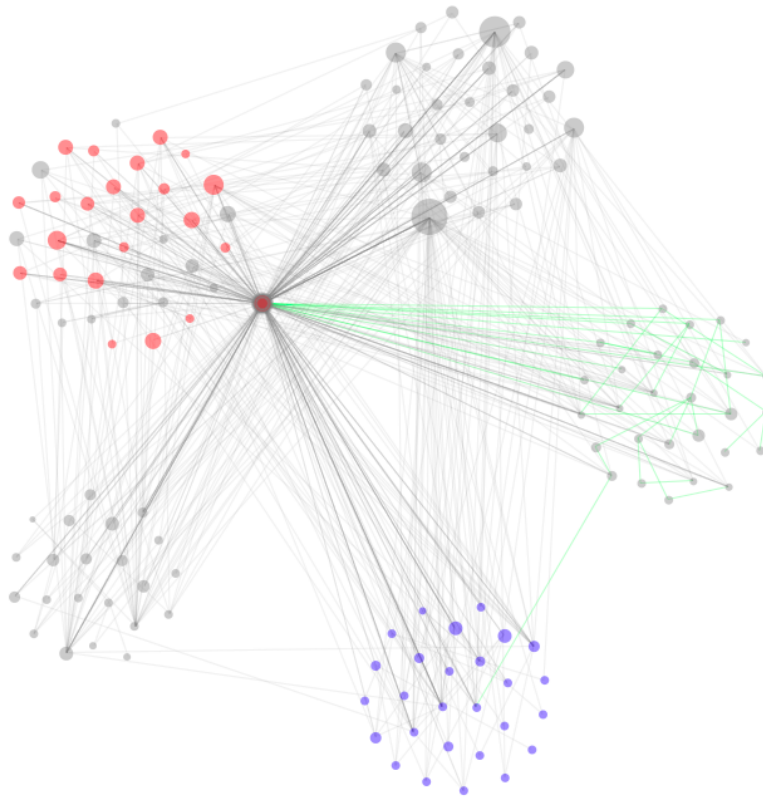

Figure 1.4: Visualization of a software system.

```
    color: Color gray;
    if: [ :c | c inheritsFrom: RTShape ] color: Color blue;
    if: [ :c | c name endsWith: 'Builder' ] color: Color red.
b edges
    connectTo: #dependentClasses;
    color: (Color gray alpha: 0.1);
    if: [ :f :t | f inheritsFrom: RTLayout ] color: (Color green alpha: 0.2).

b layout
    partition: [ :c | c inheritsFrom: RTLayout ];
    partition: [ :c | c inheritsFrom: RTShape ];
    partition: [ :c | c inheritsFrom: RTInteraction ];
    partition: [ :c | c inheritsFrom: RTBuilder ];
    partition: [ :c | '*Example*' match: c name ];
    force.
```

```
b layout circle radius: 250.

b global
   alphaColor: 0.4;
   normalizeSize: [ :c | c dependentClasses size ] min: 5 max: 30 using: #sqrt.
b addAll: (RTObject withAllSubclasses).
b build
```

The script contains two layout rules. The first layout rule partitions nodes of the graph into 5 semantically different sets. Classes not matching any partition: instruction are all located at the same point, center of the visualization. A force-based layout is applied on each defined partition. The second layout rule uses a circle layout to locate the partitions.

Result of the script is given in Figure 1.4. Each circle is a class and its size reflects the amount of methods defined in the class. Shape classes are blue and builder classes are red. Edges starting from a layout class are in green.

## Gloabl rules

Global rules may be set to either avoid code duplications between production rules or to perform color or size normalization.

### Global rule

Size and colors of nodes may be globally set: alphaColor: set transparency of all nodes and expect a value between 0.0 and 1.0; minSize: and maxSize: set the minimum and maximum node pixel size, respectively. This is relevant in the case that elements are extraordinary small or large, as it often happens with software metrics.

Algorithms may be globally applied, for example: colorCycles and colorBranches color nodes that are in cycles and accessible from a particular branch, respectively. This is useful to understand the ramification of depending elements.

### Normalization

Nodes may be compared to each other by their shape and color. The ct{global} rule provides two composed keywords:

- normalizeSize: metricBlock min: minValue max: maxValue using: transformation normalizes the size of nodes. Each node has a size ranging from minValue to maxValue. The size is computed using the function metricBlock on each object model. As it often happens, values may have

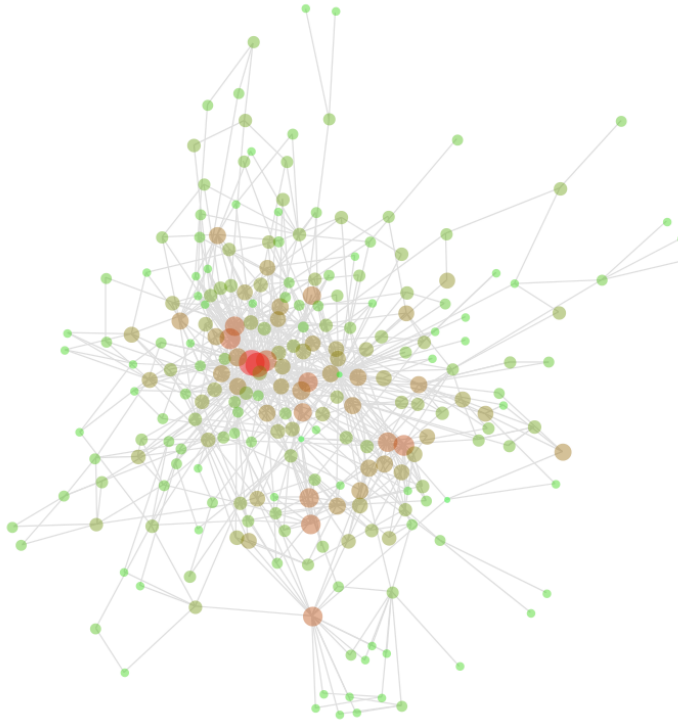

Figure 1.5: Use of normalization and force layout.

to be transformed to be meaningfully compared, using the function transformation.

- normalizeColor: metricBlock using: colors using: transformation assign a range of colors to nodes. The colors value is an array of colors, for which the the function metricBlock determines which colors have to be picked for a node. The first color of colors is assigned to the node with the lowest value of metricBlock. The last color in colors is assigned to the node with the highest value.

The argument metricBlock takes as argument a node and returns a numerical value. It is known that software system often follow power laws}. As a consequence, metric values often have to be projected to better exhibit differences. The transformation field is a simple transformation function. Typically a logarithm or square root. Illustrated. Figure 1.5 is the result of:

```
b := RTGraphBuilder new.
b edges connectTo: #dependentClasses; useInLayout.
```

```
b layout force charge: −80.

b global
    minSize: 15;
    normalizeSize: [ :c | c dependentClasses size ]
        min: 5 max: 25 using: #log;
    normalizeColor: [ :c | c dependentClasses size ]
        using: { Color green . Color  red } using: #log;
    alphaColor: 0.4.

b addAll: RTObject withAllSubclasses, TRObject withAllSubclasses.
b build
```

The size and color of a class indicates its amount of dependent classes. The class with the smallest number of dependencies is green and 5-pixels wide. The class with the most dependencies is red and 25-pixels wide. Other classes have their metric value projected with the logarithm function.

## 1.3 Chapter summary

*Graph* is a specialized language to render graphs. One of the key aspects of *Graph* is to provide a small and consistent language to seamlessly map software code into a partitioned graph. Such features is non-trivial to obtain when considering other domain-specific languages.