A constraint-based layout manager for Eiffel

Chair of Software Engineering

A constraint-based layout manager for Eiffel

Master Thesis

Emanuele Rudel
ETH Zurich
erudel@student.ethz.ch

November 4th, 2013 - May 4th, 2014

Supervised by:
Đurica Nikolić
Prof. Bertrand Meyer

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

inf | Informatik
Computer Science

Abstract

EiffelVision 2 is an object-oriented library for developing graphical user interfaces in Eiffel. It offers several strategies to organise widgets in windows, and each strategy has its own specific language and features.

The goal of this master thesis is to design and implement a more abstract language to develop user interfaces that is based on the mathematical model of linear programming. The resulting framework is seamlessly integrated in EiffelVision 2 and offers developers a valid alternative to existing layout managers.

Acknowledgments

I would like to express my sincere gratitude to my mentor, Prof. Bertrand Meyer, for giving me the opportunity of dedicating my master thesis to the topic of constraint-based layout managers and for letting me work at Eiffel Software in Santa Barbara, California.

I would like to thank my supervisor Đurica Nikolić for his valuable feedback and continuous guidance during the whole duration of this thesis. I would also like to thank Emmanuel Stapf for his insightful design decisions and suggestions for solving technical issues.

Finally, I wholeheartedly thank my family and friends who unconditionally supported me during my studies at ETH.

Contents

Chapter 1

Introduction

Computer programs offer a mean for user interaction through graphical interfaces. The actual representation of the program may vary across different devices, yet the purpose is to provide a clear, functional design and a pleasant user experience.

Developing the GUI is a process that primarily defines how the system is going to work — while aesthetics may also play a role in the design of a program, it is not our intention to discuss this subject. In order to create a user interface that suit the needs of a program, developers should be provided with the tools to achieve this goal in the easiest possible way.

From a user perspective, the graphical user interface (GUI) impacts the productivity in two ways: the initial learning curve to get a hold of the program and the time spent for executing a given task. An intuitive and functional user interface will speed up the latter and thus increase the efficiency of the end user.

We want to closely examine the tools at developers' disposal and come up with an alternative that is compensates for the shortcomings of the currently available solutions.

1.1 Motivation

EiffelVision — the framework for developing user interfaces in Eiffel — comes with a number of components that allow to layout interface elements on a window that is displayed to the user.

The goal of this work is to introduce a new component relying on a mathematical model that presents a simple API to develop user interfaces. Each of the existing components is meant to be used when a particular layout arises, yet it becomes easily cumbersome to combine them together in order to obtain a layout that cannot be expressed with one single component.

Our purpose is therefore to complement existing solutions with a component that simplifies the development of complex layouts. Being backed by a mathe-

matical model, the component has well-defined set of rules to follow in order to guarantee that the layout will be displayed as specified.

1.2 Contributions

The main contribution of this thesis is the implementation of a layout manager for EiffelVision. The new component does not require any modifications to existing applications using the framework and can be used together with all the other EiffelVision elements.

As the layout manager relies on the optimisation of linear programs, we also developed a reusable linear programming library that wraps `lpsolve` [1], a free solver written in C.

1.3 Structure of Thesis

The remainder of this thesis is organised as follows:

- **Chapter 2** gives an introduction to linear programming and highlights its important features.
- **Chapter 3** explains the relation between linear programming and its application to layout managers.
- **Chapter 4** illustrates how the layout manager API can be used in an application with EiffelVision.
- **Chapter 5** shows actual use cases of the layout manager in EiffelVision applications. The benchmarks measure the performance of the linear programming solver and of the layout manager.
- **Chapter 6** presents the conclusion of this thesis and proposes enhancements for future work.

[1] Available under LGPL license at http://lpsolve.sourceforge.net

Chapter 2

Linear Programming

A substantial part of this thesis revolves around linear programming and it is therefore necessary to first know its basics in order to understand how the constraint-based layout manager works. Linear programming is a generic and low level abstraction for defining user interfaces; the layout manager builds on top of it the abstractions to develop GUIs.

Section 2.1 focuses on the theoretical aspects of linear programming, while section 2.3 is dedicated to a brief description of its implementation as an object-oriented library for Eiffel.

2.1 Introduction

Linear programming is a technique to find the best outcome — either the maximum or the minimum — of a mathematical model constrained by a set of linear equations. Applications of linear programming span across a wide range of optimisation problems such as the traveling salesman problem, risk minimisation in investments and profits maximisation in finance.

The idea of using linear programming to define user interfaces has already been proposed, considering the fast-paced evolution in this field, a long time ago in [1] and a decade later in [2]. The similarity between a constraint and the natural description of a layout specification is remarkable: for instance, "this widget must be placed 10 pixels to the right of this other widget" or "this button must be at the center of its parent container" are very simple concepts that can be easily translated to constraints and that even people who do not have any programming experience can understand.

This concept has not seen a wide adoption until a few years ago, mainly because of the advent of mobile devices of different sizes and screen resolutions, it became apparent that linear programming would simplify the development of layouts.

A linear program is a mathematical model represented by a cost function, more commonly known as *linear objective function*, and a set of *constraints* that

consists of linear equalities and inequalities. The following example illustrates a simple linear programming problem.

Example Find values for x_1 and x_2 such that the linear objective function $f(x_1, x_2) := x_1 + x_2$ is minimised under the constraints

$$\begin{aligned} -x_1 + x_2 &\leq 1 \\ 2x_1 + x_2 &\leq 4 \\ x_1 + 3x_2 &\geq 2 \end{aligned}$$

where $x_1 \geq 0$ and $x_2 \geq 0$. The non-negative constraints on x_1 and x_2 are enforced because these values usually represent a cost or gain which would not have a useful meaning when smaller than zero.

Since the problem only has two variables, the constraints can be represented on a plane as follows:

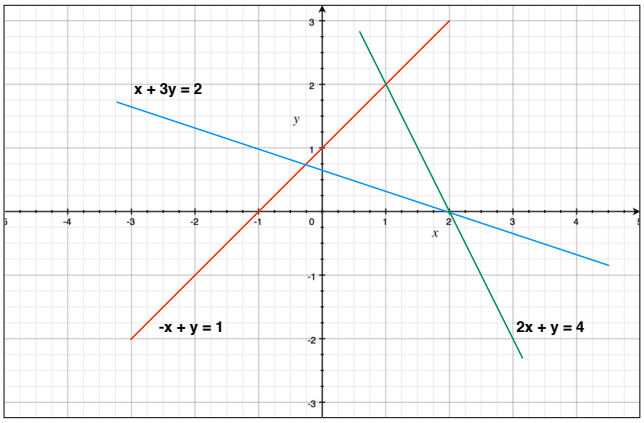

Figure 2.1: The set of constraints — shown as equalities for clarity — corresponding to the example problem.

The triangle enclosed by the three equations is the region where all constraints are satisfied, and thus the optimal value for the objective function must be in that area. It is easy to see that the minimum value for the objective function is attained at the intersection of the red and blue lines, that is $(\frac{-1}{4}, \frac{3}{4})$. However, the solution must be non-negative, i.e. x_1 and x_2 cannot assume negative values, hence the optimal value is reached at $(0, \frac{2}{3})$.

2.1.1 Minimum and Maximum Problems

The optimisation of a linear program can be performed by searching for either a maximum or a minimum value and the exact formulation varies slightly depending on the chosen direction. Although this section refers to minimisation problems only, it is important to keep in mind that the same principles also apply to maximisation problems.

A minimisation problem can be converted into a maximisation one and vicevversa, even though the transformation is non trivial and may introduce additional

constraints. Section 2.2.2 describes an alternative and more efficient way to switching from one problem to the other.

2.1.2 Terminology

Given two vectors $\boldsymbol{b} \in \mathbb{R}^m$, $\boldsymbol{c} \in \mathbb{R}^n$ and a matrix $\boldsymbol{A} \in \mathbb{R}^{m \times n}$, the compact form of the linear program is given by the following representation:

$$\begin{array}{ll} \textit{minimize} & \boldsymbol{c}^\top \boldsymbol{x} \\ \textit{subject to} & \boldsymbol{A}\boldsymbol{x} \geq \boldsymbol{b} \\ \textit{and} & \boldsymbol{x} \geq 0 \end{array}$$

Where $\boldsymbol{c}$ is the objective function, $\boldsymbol{x}$ contains the variables to be determined and $\boldsymbol{A}$ and $\boldsymbol{b}$ are the matrix and vector of coefficients representing the constraints — the left and right hand side, respectively.

A linear program may or may not have a solution for a given problem, but it could also have an infinite number of solutions. A problem is therefore said to be *infeasible* when no solution exists and it is said to be *feasible* otherwise. For example, a linear program becomes infeasible whenever two constraints are contradictory.

When the linear program is feasible, it is *bounded* if the number of solutions is finite (i.e. the problem has at least one solution) and it is said to be *unbounded* when there are infinite solutions.

The *constraint set* is the space containing all the solutions for which the constraints are satisfied. It is important to note that not all solutions achieve the minimum value for the problem.

2.2 Theory

2.2.1 Geometric Interpretations

One often expresses a linear program just using its compact matrix form, however some of its properties and implications have an intuitive meaning when analysing the problem from a geometric perspective.

The intersections of all equality and inequality constraints form a domain where every point, including the ones on the boundaries, satisfies the constraints. In the general case where the optimisation problem has n variables, every constraint defines a hyperplane that divides the space in two parts, and is of the form:

$$a_1 x_1 + a_2 x_2 + \cdots + a_n x_n \star b, \star \in \{=, \leq, \geq\} \tag{2.1}$$

Due to the linearity of the constraint, the half-space is convex, and therefore the intersection of all half-spaces defines a convex polyhedron in a n-dimensional space called *feasible region*.

An interesting property of the convex polygon, called *the fundamental theorem of linear programming*, states that the maxima and minima of a linear function over a convex polygon — defined by $\boldsymbol{Ax} \leq \boldsymbol{b}$ — occur at the polygon's corners. Minima and maxima can also occur at two different corners and along the edge that connects them, i.e. there may exists more than one optimal solution.

Proof By contradiction, assume the optimal solution x^* lies inside the polytope $P = \{x \in \mathbb{R}^n : \boldsymbol{Ax} \leq \boldsymbol{b}\}$, i.e. $x^* \in int(P)$. Let $\epsilon > 0$ be an arbitrarily small radius of the ball centered in x^*, such that $B_\epsilon(x^*) \subset P$.

To show that there exists a better solution, take another point inside P, say

$$x^* - \tfrac{\epsilon}{2}\tfrac{c}{||c||},$$

and therefore

$$c^\top(x^* - \tfrac{\epsilon}{2}\tfrac{c}{||c||}) = c^\top x^* - \tfrac{\epsilon}{2}\tfrac{c^\top c}{||c||} = c^\top x^* - \tfrac{\epsilon}{2}||c|| < c^\top x^*.$$

Since x^* is not an optimal solution for any point inside the boundary, then x^* must be on the boundary of P. It may also be the case that the optimal solution can be defined as a linear combination of P's vertices, i.e. there exist multiple solutions on the edge between two vertices.

2.2.2 Duality

As already discussed above, linear programming problems can be either maximised or minimised. An interesting property of optimisation problems is that they can be viewed from different perspectives, which means a maximisation problem is representable as a minimisation problem and the other way around, without any necessary transformations to take place. The original problem is also called *primal problem* and its counterpart is known as *dual problem.*

The connection between the primal and the dual problem is that the latter provides a lower bound on the solution of the first — as in the previous sections, the minimisation problem is taken into consideration.

Referring to the problem in section 2.1.2, the primal problem

$$\begin{array}{ll} \textit{minimize} & \boldsymbol{c}^\top \boldsymbol{x} \\ \textit{subject to} & \boldsymbol{Ax} \geq \boldsymbol{b} \\ \textit{and} & \boldsymbol{x} \geq 0 \end{array}$$

has a correspondent dual problem of the form

$$\begin{array}{ll} \textit{maximize} & \boldsymbol{y}^\top \boldsymbol{b} \\ \textit{subject to} & \boldsymbol{y}^\top \boldsymbol{A} \leq \boldsymbol{c}^\top \\ \textit{and} & \boldsymbol{y} \geq 0 \end{array}$$

Depending on the formulation of the problem, the number of constraints and variables, the solver might choose to optimize the dual instead of the primal in order to improve the running time of the algorithm.

2.2.3 Augmented Form

Once the linear programming problem has been defined, it must be converted in the *standard form* for finding the optimal solution through the simplex method. A problem is transformed into the standard form by following these four rules:

1. if the objective function f is being minimised, then change the optimisation direction to $max\ -f$;
2. an inequality constraint of the form $a_{i1}x_1 + \cdots + a_{in}x_n \leq b_i$ is transformed to the equality constraint $a_{i1}x_1 + \cdots + a_{in}x_n + s_i = b_i$, where s_i is a non-negative variable called *slack variable*;
3. an inequality constraint of the form $a_{i1}x_1 + \cdots + a_{in}x_n \geq b_i$ is transformed to the equality constraint $a_{i1}x_1 + \cdots + a_{in}x_n - s_i = b_i$, where s_i is a non-negative variable;
4. if a variable x_j does not have any sign restrictions, replace it everywhere with the expression $x_j^{'} - x_j^{''}$, where both variables are non-negative.This does not usually happen when specifying layout constraints as the value of the variables must always be positive.

The original problem can be thus rewritten as:

$$max\ -f \quad subject\ to \quad \boldsymbol{Ax} + \boldsymbol{x_s} = \boldsymbol{b}$$

A constraint at position i that is already an equality constraint will have the corresponding slack variable x_{s_i} equal to zero.

The linear program in the standard form can be represented in the following block matrix form.

$$\begin{bmatrix} 1 & -\boldsymbol{c}^\top & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{A} & \boldsymbol{I} \end{bmatrix} \begin{bmatrix} z \\ \boldsymbol{x} \\ \boldsymbol{x_s} \end{bmatrix} = \begin{bmatrix} 0 \\ \boldsymbol{b} \end{bmatrix}$$

Solving the left hand side multiplication yields the objective function on the first row, while the second row represents the constraints in the augmented form (i.e. with slack variables).

2.2.4 Simplex Method

The simplex method is a well-known algorithm for solving linear programming problems and is at the core of the `lpsolve` library, on which the constraint-based layout manager relies.

Section 2.2.1 ensures that if the problem has a solution, then this will be on a corner of the convex polytope. Since the number of corners grows exponentially as the number of variables and constraints increases, the brute force approach

that compares the value of the objective function at each corner is too slow. The idea behind the algorithm can be easily expressed using once again a geometric interpretation of the linear programming problem and is broken down in three essential steps:

1. find an initial corner of the polytope. The problem is infeasible if no initial corner can be found;
2. compute the cost of each edge leading from the current corner to its neighbours;
3. if there is a decreasing cost with respect to the current one, move in that direction until encountering the next corner and go to the second step again. If there is no decreasing cost, then the solution is optimal and the algorithm terminates.

Essentially, the algorithm follows a path, starting from a random corner, where the cost of each path segment is always decreasing. The actual procedure for the algorithm is more formally described using two rules:

Rule 1 If all variables in the first row (the one describing the objective function) have a non-negative coefficient, the current solution is optimal; otherwise, choose a variable x_j with a negative coefficient in the first row, also called *entering variable*.

Rule 2 For each constraint containing the entering variable, compute the ratio between the right hand side and the coefficient of the entering variable. Choose the pivot row as the one with the minimum ratio.

The two steps are repeated until *Rule 1* guarantees that the solution is optimal. In the second step, after a column and row have been chosen, the algorithm performs a Gaussian elimination based on the selected pivot element.

2.2.5 Irreducible Infeasible Subsets

It is of real interest to analyse an infeasible linear programming problem in order to find the cause of the error. In a layout manager, for example, the user might unknowingly specify contradictory constraints which would prevent the solver from finding an optimal layout configuration.

A manual inspection of all constraints is not a viable option because when the number of constraints becomes too large, it is difficult to establish which ones are in conflict. In some cases it is even possible that for a large problem defining thousands of constraints, only a few are actually causing conflicts.

While the task of understanding and fixing conflicts is ultimately up to the developer, the computer can help speeding up the process by reducing the complexity of the linear program. There are in fact different algorithms that make it possible to find *irreducible infeasible subsets* (IIS). An irreducible infeasible subset is a subset of the original set of constraints that becomes feasible if any

of them is removed. For instance, if there is only one IIS with 2 constraints, one can remove either the first or the second constraint from the original linear program, which will now become feasible. This is particularly useful as it isolates a small number of conflicting constraints and allows the developer to quickly inspect and resolve the problem.

We have chosen to combine two algorithms for finding an IIS because this solution provides an efficient performance in terms of execution time.

Elastic Filtering

The idea behind elastic filtering is the following: to every constraint is added a non negative variable called *elastic variable*, and the objective function to be optimised becomes the sum of the elastic variables. Conceptually, the objective function now represents the sum of constraints violations in the original formulation of the problem, which is often referred to as the *sum of the infeasibilities* (SINF). When the problem is solved — and of course it will now be feasible since there are elastic variables, there are two possible outcomes for each constraint:

- the values of the original variables remain the same because the constraint did not cause any conflicts, and therefore the elastic variable has a value of 0;
- the constraint was infeasible before, but the elastic variable made it possible to stretch it. The value of the elastic variable represents in fact *how far* is the constraint from being feasible and contributes to the final SINF value.

Every constraint whose elastic variable has a positive value is *enforced* by removing said variable; the constraint is then added to the IIS set. The linear programming problem is solved again and the constraints are enforced — which can be seen as a way of tightening the linear program step by step — until it becomes infeasible.

The method is efficient because it ignores constraints that are not part of an IIS, making it possible in practice to identify an irreducible infeasible set in just a few iterations. The resulting set contains all the enforced constraints, which may represent more than a single IIS. It is thus necessary to combine the elastic filter algorithm with a deletion filter in order to reduce the output to only one IIS.

Deletion Filter

The deletion filter is a brute-force approach for finding a single IIS in a set of infeasible constraints. As the name suggests, the algorithm temporarily drops one constraint at a time and checks whether the linear programming problem has become feasible. If the problem is not feasible it means that the dropped constraint does not contribute to the infeasibility and can therefore be discarded. The remaining constraints at the end of the algorithm represent exactly one IIS.

Algorithm 1 Elastic Filter

Input: { infeasible constraints }

```
1: procedure FILTER
2:     for all constraint in constraints do
3:         if constraint.sign is ≥
4:             add nonnegative elastic variable e_i then
5:         else if constraint.sign is ≤
6:             subtract nonnegative elastic variable e_i then
7:         else
8:             add elastic variable e_i'
9:             subtract nonnegative elastic variable e_i''
10:    repeat
11:            program.solve
12:            remove positive elastic variables
13:    until program is infeasible
```

Output: { constraints containing at least one IIS }

The algorithm is not very efficient compared to the elastic filtering because it requires the linear program to be solved at each iteration, i.e. as many times as the number of constraints. Some solvers can actually reuse the solution computed before and thus improve the running time, yet deletion filtering is still bounded by the amount of constraints.

If there are multiple IISs, the deletion filter will find the one whose first constraint (compared to other IISs) come last in the original set of infeasible constraints.

Algorithm 2 Deletion Filter

Input: { infeasible constraints }

```
1: procedure FILTER
2:     for all constraint in constraints do
3:         delete the constraint from the set
4:         solve the program
5:         if is feasible then
6:             add back constraint in the set
7:         else
8:             drop the constraint
```

Output: { constraints forming one IIS }

An irreducible infeasible set is found running the elastic filtering algorithm first, and the deletion filter afterwards. This approach reduces the execution time and yet is able to find a single subset.

2.3 Implementation

The linear programming library on which this work relies is written in C, which requires to adapt it to the Eiffel language through the so called *external mechanism*.

In the `lpsolve` library, a linear program is expressed in a matrix form where the first row defines the objective function and each subsequent row is a constraint (in)equality. Columns, on the other hand, represent the unknown variables. Optional variable bounds — i.e. minimum and maximum value, even if not directly stored in the matrix representation, are set and retrieved accessing the column index of the given variable.

Below is illustrated a simple linear programming problem written with `lpsolve`.

```
              C1    C2     C3    C4    C5    C6
Minimize       1     1      1     1     1     1
R1             0     0      0     0    -1     1    >=  86
R2             0    -1      1     0     0     0    >=  25
R3             0     0     -1     0     1     0    >=  55
R4             1     0      0    -1     0     0    >=  23
R5            -1     0      0     0     1     0     =  10
R6             0     1      0     0     0    -1     =  20

Type        Real  Real   Real  Real  Real  Real
upbo           0   457      0   436   Inf   Inf
lowbo          0   457      0   436     0     0
```

Figure 2.2: Output representation of a linear programming problem defined using `lpsolve`.

The `lpsolve` library also offers other features that are not relevant to the scope of this thesis — e.g. integer linear programming — and therefore have not been wrapped in the Eiffel library.

Although the conceptual mapping to an object-oriented library is relatively simple and straightforward, the actual implementation required a significant amount of time due to the lack of proper documentation and complex APIs. The class diagram 2.3 shows the relevant components of the Eiffel linear programming library.

As one can see from figure 2.2, each *cell* at position (i, j) in the matrix expresses the coefficient for the j-th variable at the i-th constraint. Instead of just storing the coefficients in the constraint in the Eiffel library, it makes sense to have a reference to both the variable and its coefficient. The `TERM` class therefore wraps a linear programming variable and its corresponding coefficient in the left hand side of constraints and also in the objective function.

An important aspect for the performance of the linear programming problem is to always use the a sparse representation of the constraints and the objective function: the terms only contain variables specified by the user. All the remaining variables have by default a coefficient equal to zero. As the number

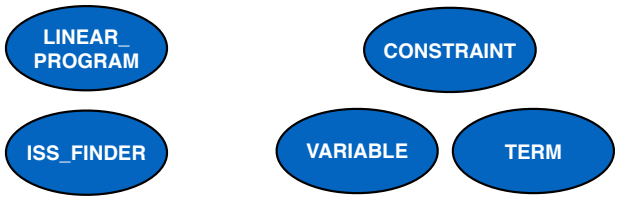

Figure 2.3: The class diagram of the Eiffel linear programming library.

of constraints and variables increases, this choice greatly impacts memory requirements since it avoids storing a large amount of unused terms. Inserting, updating or deleting constraints also become faster operations to perform, as modifications apply only to a small subset of the matrix.

Listing 2.1 shows the basic interface for the `LINEAR_PROGRAM` class. The creation of constraints and variables is deferred to their respective classes and is not presented here.

Listing 2.1: *Eiffel*: Linear Program class

```
1
2  class LINEAR_PROGRAM
3
4  feature -- Access
5
6    constraints: LIST [CONSTRAINT]
7
8    variables: LIST [VARIABLE]
9
10   objective_function: LIST [TERM]
11
12   last_result: INTEGER
13
14 feature -- Operations
15
16   maximize
17
18   minimize
19
20   solve
21
22 feature {NONE} -- Implementation
23
24   c_make_lp (a_rows, a_columns: INTEGER): POINTER
25     external
26       "C signature (int, int): EIF_POINTER use <lp_lib.h>"
27     alias
28       "make_lp"
29     end
30
31   -- Other C external calls
```

```
32
33 end
```

The IIS_FINDER class is responsible for running the elastic and deletion filters on a given linear program. Since it directly operates on the given argument and does not create a copy of the problem, one must pay attention when keeping external references to constraints and variables since the may be removed from the original linear programming problem. The objective function is modified as well, hence the linear program is not reused in the layout manager, but rather the execution terminates because a violation of the layout constraints is considered to be a programmer's error.

To ensure that the Eiffel linear programming problem has been correctly implemented, aside from using contracts, a series of tests taken from the NETLIB repository. The tests compare the results obtained with the Eiffel version and the C one. Section 5.1 describes more in depth the performances of the solver.

Chapter 3

Constraint-based Layout Manager

Chapter 2 focused on linear programming problems from a mathematical point of view; this chapter shows the transition from the low level and generic aspects of linear programming to a more abstract level, easier to understand and optimised to define layout specifications.

EiffelVision is the standard library for GUI programming in the Eiffel environment. This chapter discusses the different technologies that the framework offers for laying out interfaces and identifies the domains for which they are most suited in order to compare them to the constraint-based layout manager.

3.1 Introduction

The goal of a layout manager is to arrange interface elements in a container, and it is usually the case that the container itself directly takes care of organising its children. There is a clear distinction between the two concepts though: a container is just the holder of its children, while a layout manager is the entity that adopts the strategy to lay out the children widgets. The tasks of a layout manager may include:

- positioning the widgets;
- spacing the elements from one another;
- spacing the elements from the container;
- adapting the size of widgets whenever the size of the container changes.

The EiffelVision framework offers several layout managers that revolve around the concept of rows and columns, which are well suited for displaying interface elements in a sequential order. There also exists a table layout manager that organises content in its adjustable cells.

All of these containers have different ways for specifying the layout interface due to their specific implementation. Unfortunately there is no universal language to express a layout and it is therefore required that the users know about each individual layout manager before opting for the most convenient container.

3.1.1 Boxes

A box is a container that displays widget elements in a linear fashion, that is either horizontally or vertically. Layout managers for box containers lay out widgets according to these rules:

- the order of the widgets on the x- or y-axis is determined by the order of insertion;
- the default space between two widgets is zero pixels, but it can be changed by the user. The spacing is the same among all widgets;
- the default space between a widget and its parent container is zero pixels, but it can be changed by the user. This spacing is applied to all four sides of the container;
- the minimum size — width and height for horizontal and vertical boxes, respectively — of the container is defined by the minimum size of the biggest element in the container, multiplied by the number of elements;
- whenever the size of the container increases, the widgets' sizes are evenly increased. It is also possible to disable resizing for single widgets, in which case the size remains always fixed to its minimum. The user is thus not allowed to set a custom size to a widget.

Box containers offer a very simple API to arrange widgets and is particularly useful for displaying an array of conforming elements.

However, it is challenging to arrange widgets both horizontally and vertically by just using boxes. Albeit creating a nested hierarchy of horizontal and vertical containers can be used as a workaround, there are several disadvantages with this approach. Nested hierarchies obviously require more code to set up the layout. This also goes at the expense of readability and understandability of the code because as the hierarchy deepens, one must take into account the different resizing behaviours. Additionally, the semantical grouping of widgets (in one row or column) might be lost in order to accomplish the desired layout.

As an example, a simple form with two labels and two text fields is created as follows: each form component, composed of a label and a text field, is contained in a horizontal box; each horizontal box is inserted in a vertical box. The labels must also be of the same width in order to show the text fields aligned, but the widgets belong to different containers, and creating a relationship between them is not very intuitive. One must in fact disable the two labels to automatically extend and set their minimum width to the maximum width between the two widgets. Listing 3.1.1 illustrates the code necessary to generate such the layout.

Listing 3.1: *Eiffel*: A form with two components using nested box containers.

```
create_form
    local
      name_label, location_label: EV_LABEL
      name_field, location_field: EV_TEXT_FIELD
      vb, hb: EV_HORIZONTAL_BOX
      min_width: INTEGER
    do
      create vb
      window.extend (vb)

        -- Initialize labels and text fields
      create name_label.make_with_text ("Project Name:")
      create location_label.make_with_text ("Project
          Location:")
      create name_field
      create location_field

      create hb
      hb.extend (name_label)
      hb.extend (name_field)
      hb.disable_item_expand (name_label)

      container.extend (hb)
      container.disable_item_expand (hb)

      create hb
      hb.extend (location_label)
      hb.extend (location_field)
      hb.disable_item_expand (location_label)

      min_width := name_label.minimum_width.max (
          location_label.minimum_width) + 10
      name_label.set_minimum_width (min_width)
      location_label.set_minimum_width (min_width)

      container.extend (hb)
      container.disable_item_expand (hb)
    end
```

3.1.2 Fixed Containers

An alternative to automatic layout management is provided by fixed containers. In this case the container simply organises its content as described by the developer, who has to specify size and position for every widget inserted. There are two main reasons for choosing a fixed container: first, it provides a really simple API to assign a widget a position and a size (one method call is often enough); secondly, as a consequence of the first reason, it allows to lay out complex and unconventional user interfaces.

Fixed containers, on the other hand, are known for lacking many features

that other layout managers have built-in:

- automatic resizing triggered when resizing the window;
- automatic resizing when translating software into another language. Labels and any other widgets containing text can potentially be truncated and result in a very bad user experience;
- adaptation to right-to-left languages. Boxes can for example flip their content to display the content right-to-left, but there simply exists no way to accomplish this task using fixed containers.
- inserting or deleting an item requires to recompute all the other widgets' attributes, which in the long run is clearly a problem for maintenance.

Fixed containers may be useful for example for presenting content that does not need — or does not want — resizing, such as videos, photos or even games. The constraint-based layout manager can express any layout defined in a fixed container, even though it may require a little more code to set up.

3.2 Mapping to Linear Programming

A linear programming problem can directly represent a layout interface, yet such connection would be very generic and also requires the user to understand the underlying theory.

This section abstracts from a generic linear programming problem and provides a language for expressing layout constraints that is simple to understand and at the same time powerful enough to build complex layouts.

3.2.1 Minimum Problem Formulation

Since the goal of a linear programming problem is to find an optimal solution that satisfies all the given constraints, it is therefore important to relate the concept of optimization with layout interfaces.

The size of the container defines the bounds in which items can be positioned. The bounds are defined, just as the other widgets, with four variables in the linear programming problem and they always have fixed values of $(0, 0)$ for the x- and y-position, while the width and height are set to be equal the size of the container. As the container resizes, the bounds always need to be updated to reflect its new size.

In terms of performance, it is better to set variable bounds rather than defining new constraints since the model will not increase in size.

The size of the widgets must be minimized in order to fit them in the container while satisfying the given constraints. The objective function is therefore defined as the sum of the widgets' attributes:

$$min \sum_{i=1}^{n} x_i + y_i + w_i + h_i \tag{3.1}$$

where n is the number of widgets in the layout manager. Note that since horizontal and vertical attributes are combined it is possible to define *ratio relationships* between widgets. A ratio relationship is a constraint that binds a horizontal widget's attribute with a vertical one.

3.2.2 Terminology

In a linear program, constraints are n-dimensional equations where the unknown variables are to be optimised. In the layout manager, each variable represents the x or y coordinate of a point in the container's area.

In order to represent the complete position for a widget is therefore necessary to define four variables in the linear programming problem. Recalling that EiffelVision uses a top-left coordinate system, (x, y) represents the top-left corner of a widget. The bottom-right corner can be expressed either by defining a *bottom* and *right* or *width* and *height* variables. There are two reasons for choosing the latter pair over the former:

1. the linear programming API is consistent with the EiffelVision API for managing the size of widgets. The variables `x`, `y`, `width` and `height` are in fact equal to the `EV_WIDGET` attributes `x_position`, `y_position`, `width` and `height`;

2. it is possible to set bounds for the minimum and maximum size of a widget. The advantage of this approach is the huge save of constraints — up to four per element — that would be needed when using the right and bottom sides.

This choice is purely technical and the user should still able to refer to any side of a widget. It is not ideal, though, that the variables of the linear program are directly exposed to the user. Variables can be represented in a more abstract way through *attributes*, which describe positions of the visual object making use of one or more variables. For example, right and bottom attributes are defined as

$$right\ attribute = left\ position + width \tag{3.2}$$

and

$$bottom\ attribute = top\ position + height \tag{3.3}$$

An attribute is thus represented as a linear combination of variables and their respective coefficients — implicitly set to 1 in this case. One can also define more complex attributes and, as shown later in section 3.3, the design of the layout manager allows developers to easily extend the list of attributes.

3.2.3 Fixed constraints

Sometimes it may be useful to define properties of a layout that should not adapt to changes, for instance when resizing the constraint-based container. Fixed constraints are especially used for spacing widgets in the container, often referred to as *padding* and *margin*.

In a box container, the margin represents the distance between the container itself and any other widget it contains, while the padding is the space among each item in the box. Margins are applied to all four directions — left, right, top and bottom — and they all assume the same value. Obviously, in the context of horizontal and vertical boxes, padding only applies to two out of four directions of the container. Figure 3.1 shows the difference between padding and margins.

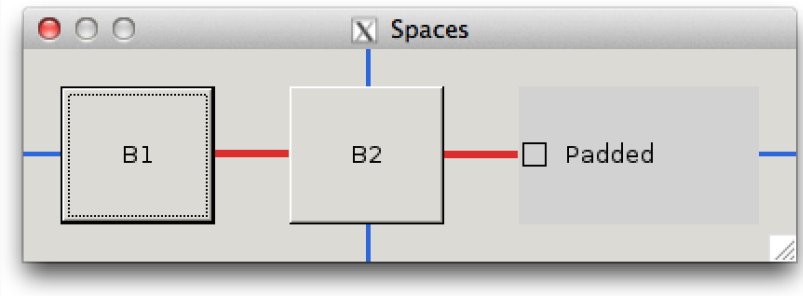

Figure 3.1: A horizontal box containing three elements. Margins are shown in blue lines, while padding is identified by the red lines.

A constraint-based layout manager enables users to define custom spacings between any two widgets in any direction as well as their distances from the parent container.

For instance, toolbars are a standard way to easily access actions and they are often organised in different groups of buttons. Instead of using the same padding for all elements (which would be the case when choosing a box container), a constraint-based layout manager can define a large padding between groups and a smaller one between widgets within the same group.

As the constraint-based layout manager does not make any assumptions about the location of widgets, it cannot infer automatically padding between them. On the other hand, since the container inherits from `EV_FIXED`, a default margin of zero pixels is applied to widgets — given that they are initially placed completely inside the container. The lack of information about relative positions thus requires the developer to specify a number of different constraints in order to space out widgets.

In the layout language, a fixed constraint can be expressed in the following ways:

$$width = constant \tag{3.4}$$

or

$$attribute_1 + constant = attribute_2 \tag{3.5}$$

Depending on the choice of attributes, the second form may actually also represent dynamic constraints, which are treated in the following section.

3.2.4 Dynamic constraints

We define constraints that adapt to changes as *dynamic constraints*. Rather than relying on constant values, dynamic constraints express a relationship between two or more attributes, which makes it possible to react to changes automatically. While dynamic constraints are mainly used to resize widgets, they can also define relations between widgets' attributes, i.e. that the width of one element be equal to the width of a second one. Relations enable the developer to create layouts that are quite difficult to achieve using the existing containers, for example:

- anchoring an element to an attribute of its container. One can place a widget to be pinned at the vertical center of the container even when the window is resized.
- maintaining proportions of an area. If the GUI is displaying an image, it is desirable that the ratio of the picture always remains the same in order to avoid stretching. A constraint that puts in relation the width and height attribute of the widget can easily define such a property.

A constraint describing the anchoring in the layout language might be defined as

$$container.vertical\_center = widget.vertical\_center \tag{3.6}$$

The abstraction of attributes proves to be particularly effective here because it maps a relatively complex constraint from the linear program to a simple, human-readable statement. The same exact constraint, for a container c and a widget w, is represented in a linear programming problem as

$$c_{top\_position} + \frac{1}{2}c_{height} - w_{top\_position} - \frac{1}{2}w_{height} = 0 \tag{3.7}$$

The constraints defined in the layout language are automatically rearranged to be in a suitable form for the linear programming problem.

3.2.5 Error Handling

Containers in EiffelVision make extensive use of Design by Contract in order to signal any incorrect uses to the programmer. Unfortunately in a constraint-based layout manager it is not possible to establish a priori (e.g. in a precondition) whether the given constraints are conflicting, thus generating an invalid layout. As a consequence, it is not possible to trace back the constraint(s) that caused the linear program to be infeasible; as the number of constraints grows, it increases the difficulty of finding and fixing the mistake. In addition, there is the chance for a constraint to be satisfied up until a certain condition, for

instance a resize event up to a given threshold, and to have no feasible value after that.

Section 2.2.5 describes in depth the process for recovering the smallest set of constraints from the original one. Whenever the linear programming problem does not find a feasible solution for the given constraints, the layout manager executes the elastic and deletion filters in sequence in order to find an IIS. The final step consists of raising an exception and displaying to the user the problematic constraints in the console log. The layout manager keeps a reference to all constraints defined by the user but does not preserve the semantics of the layout language: the layout attributes, in fact, do not keep a reference to the widget they represent. It is therefore not straightforward to translate the constraints of the irreducible infeasible subset back to a readable representation. Instead of displaying the conflicting attributes, the layout manager expresses the contradictory constraints in terms of the variables used in the constraint.

3.3 Implementation

This section describes the implementation details of the constraint-based layout manager: the first part is dedicated to the actual strategy for laying out widgets, while the second one describes the different approaches for defining layout constraints.

3.3.1 Autolayout

The constraint-based layout manager needs to be paired with a container. Since the fixed container, contrarily to box containers, does not perform any widgets arrangement, it is reasonable to choose it as the basis for building a new type of container. We will call it *autolayout* because of its ability to automatically recompute the optimal layout of widgets when it is needed, e.g. immediately after a resize event.

The `EV_AUTOLAYOUT` class inherits from `EV_FIXED` and defines methods for creating and removing constraints. The features for managing the children widgets are in fact already defined by the parent class. Although fixed containers allow the user to customise the position and size of widgets, the autolayout container will most likely immediately override the given values with the ones resulting from the linear programming solver.

An issue that needs to be addressed is mapping the variables of each widget in the container. In addition, one must make sure that variables are accessible (and thus are not void) only if a widget is contained in an autolayout. The justification for this requirement is that if the widget belongs to a box or fixed container, then the memory footprint should not increase and there should be no additional computations involved.

Each autolayout container has in its internal implementation exactly one linear program that is keeping track of all widgets' attributes. The appropriate time to create the four variables (`left`, `top`, `width` and `height`) is when the

widget is added to the container using the `extend` routine. Upon removing the widget, on the other hand, the container will remove the its variables. The developer is in charge of deleting any constraints referring to the pruned widget since the four variables have also been removed from the underlying linear program. In both cases the layout manager also takes care of updating the objective function, ensuring thus that all widgets' attributes are being optimised at each iteration.

Layout attributes are inherent properties of the widget and, following the convention of EiffelVision we define a `EV_LAYOUTABLE` class that lists the basic layout attributes. The property class also keeps track of the actual linear programming variables so that attributes can be immediately expressed through a combination of these variables, without having to query the autolayout container first. There are two ways a widget can represent its layout attributes: through inheritance or object composition. We analyse the advantages and disadvantages of each approach and then choose the one that best fits the performance criteria just defined above.

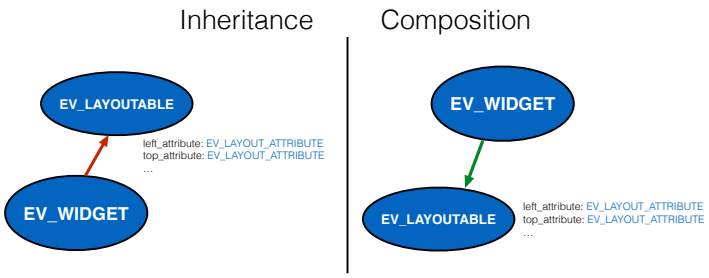

Figure 3.2: The two architectural choices for implementing layout attributes.

The main benefit of using inheritance is basically a matter of consistency: imitating property classes such as `EV_COLORIZABLE` and `EV_FONTABLE`, the widget gains a new whole set of attributes with very small (and backward compatible) changes to the `EV_WIDGET` class.

While at first it might seem a good idea to follow the inheritance strategy as other properties do, it must be taken into consideration that in the future one might want to add new layout attributes. If that is the case, then the only way to extend the set of attributes is by changing the `EV_LAYOUTABLE` class, which violates the Open/Closed principle (described in [3]). The implication is that external developers (with respect to the EiffelVision library) have to change the internal structure of the framework — a clearly undesirable side effect.

The approach that uses object composition does not suffer from this problem and therefore is preferred over the other. Custom widgets can now simply subclass `EV_LAYOUTABLE` to add new attributes and then redefine the `layout` property of `EV_WIDGET` to conform to it.

The only minor drawback of using object composition is that accessing layout attributes requires one more level of indirection, i.e. instead of

Listing 3.2: *Eiffel*: Accessing layout attributes using the inheritance strategy.

```
a_widget.left_attribute
```

one needs to first access the layout properties, and thus write

Listing 3.3: *Eiffel*: Accessing layout attributes using the object composition strategy.

```
a_widget.layout.left_attribute
```

Now that we have decided the structure of the constraint-based layout manager, we can examine more in detail the operations performed in the autolayout container. Figure 3.3 illustrates how to setup the autolayout container and also how it works internally.

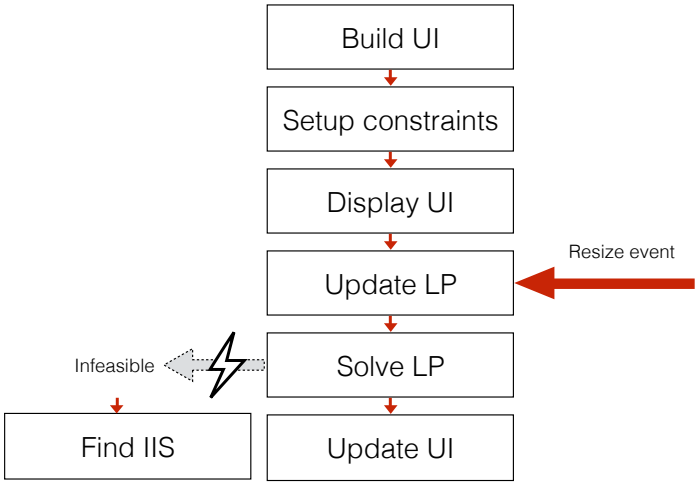

Figure 3.3: The autolayout container pipeline.

The first step to perform, as with any other container, is to create the container itself and then the items it should hold. After inserting the widgets in the container, the user can define layout relations between the different elements. It is fundamental that the insertion of widgets happens before defining constraints: recall that a widget does not have any layout attributes mapped to a linear program before it is assigned to an autolayout container. The last step, from a developer point of view, is to display the container to the end user. The rest of the process is handled by the autolayout container as follows:

1. the container receives an event change regarding its size. The variables' bounds in the linear program representing the container attributes are updated to reflect its current size;
2. the linear program is solved. There are two possible outcomes:
   - the program is feasible and the solver found a solution. For each widget in the autolayout container, its position and size are updated according to the new values of the corresponding variables in the

linear program. The user interface is then updated to reflect the changes;

- the program is infeasible and therefore no solution exists. In this case, the layout manager examines the given linear program with the infeasible irreducible subset finder. When the algorithm has completed, the program execution is interrupted and the error log is printed to the console.

The above steps are repeated as soon as an external event (e.g. user resizes a window) is triggered. Since the linear program always reuses the previous solution as a basis for the new problem, the simplex method only performs very few iterations to compute the optimal layout; this allows the autolayout container to achieve resizing in real time. Performances of the layout manager are further discussed in chapter 5.

3.3.2 Layout constraints

Layout constraints are simple wrappers of constraints as defined in linear programming problems. The motivation behind it is similar to the abstraction elaborated for widgets' attributes: the user should not directly interact with them and possibly compromise the linear program.

If the user wants to change the constraint, it is therefore necessary to remove it and add a new one with the updated values. An exception is made for the right hand side of the constraint, which can be adjusted at any time and the change will be reflected in the successive UI update. This allows users to change fixed constraints and dynamically fine-tune padding between widgets or the size of an element.

The main task of layout constraints is to convert a given linear equation in a suitable form for a linear programming constraint. The standard form of a linear equation is in fact given by

$$a\boldsymbol{x} + b = \boldsymbol{y}, \tag{3.8}$$

while a constraint is defined as having all the variables on the left hand side and a single coefficient on the right hand side. The equation 3.8 is therefore rearranged as

$$a\boldsymbol{x} - \boldsymbol{y} = -b \tag{3.9}$$

Both $\boldsymbol{x}$ and $\boldsymbol{y}$ are expressed as vectors since they represent layout attributes and may thus be formulated through more than just one variable.

We chose to create layout constraints using the form of equation 3.8, since it provides a more intuitive way of expressing layout specifications.

Listing 3.4: *Eiffel*: Creating layout constraints using the standard linear equation form.

```
class EV_LAYOUT_CONSTRAINT

create make
```

```
feature {NONE} -- Initialization

  make (a_program: LINEAR_PROGRAM; multiplier: DOUBLE;
      first_attribute: EV_LAYOUT_ATTRIBUTE; a_constant:
      DOUBLE; a_sign: INTEGER; second_attribute:
      EV_LAYOUT_ATTRIBUTE)
    local
      l_terms: LINKED_LIST [TERM]
      l_term: TERM
    do
      create constraint.make_with_program (a_program)

      create l_terms.make
      across first_attribute.terms as term loop
        create l_term.make (term.item.variable, multiplier *
            term.item.coefficient)
        l_terms.extend (l_term)
      end
      across second_attribute.terms as term loop
        create l_term.make (term.item.variable, -term.item.
            coefficient)
        l_terms.extend (l_term)
      end

      constraint.left_hand_side := l_terms
      constraint.sign := a_sign
      constraint.right_hand_side := -a_constant
    end

feature {NONE} -- Implementation

  constraint: CONSTRAINT

end
```

Given two widgets `w1`, `w2` and the autolayout container `autolayout` one can thus express a constraint like

Listing 3.5: *Eiffel*: Creation of a layout constraint using the standard equation form.

```
autolayout.create_constraint (1, w1.layout.right_attribute,
    20, {EV_LAYOUT_CONSTRAINT}.equal_sign, w2.layout.
    left_attribute)
```

to represent the layout specification

$$w1.right\_attribute + 20 = w2.left\_attribute. \tag{3.10}$$

We aim at simplifying the syntax of the constraint creation in 3.5; specifically, the procedure has too many parameters where one has to remember the exact form of the equation. It is possible to exploit Eiffel's *alias* mechanism

in order to introduce arithmetic symbols and facilitate the layout specification process. There is an exception for the equal symbol, which is reserved for the comparison of two objects and cannot be overridden, preventing us from using the `=` alias. We introduce the `EV_LAYOUT_EXPRESSION` class that is responsible for chaining layout attributes.

Layout attributes can be multiplied by a coefficient or can be added a constant in order to relate to another attribute. The application of one operation of `+`, `-` or `*` on a layout attribute gives back a layout expression, which is basically the same as a layout constraint but is not tied to an actual constraint in the linear program. Layout attributes can be directly compared to a constant with the $\leq$ and $\geq$ operators, while the feature `equal_to` allows to compare it to a layout expression. Although attributes and expressions are not conforming types, introducing a `convert` rule in `EV_LAYOUT_ATTRIBUTE` allows to use the former interchangeably with the latter.

In a similar fashion, layout expressions' operators `+` and `-` can be applied to other layout expressions; the same applies to sign operators as well. Table 3.3.2 summarises the allowed operations on layout attributes and expressions. For the sake of brevity, `EV_LAYOUT_ATTRIBUTE` and `EV_LAYOUT_EXPRESSION` are replaced by `ATTRIBUTE` and `EXPRESSION`, respectively.

Type	Feature	Alias	Argument	Result type
ATTRIBUTE	plus	+	REAL_64	EXPRESSION
	minus	-	REAL_64	
	multiply	*	REAL_64	
	greater_equal	$\geq$	REAL_64	
	less_equal	$\leq$	REAL_64	
	equal_to		EXPRESSION	
EXPRESSION	plus	+	EXPRESSION	EXPRESSION
	minus	-		
	greater_equal	$\geq$		
	less_equal	$\leq$		
	equal_to			

Table 3.1: The list of operators for layout attributes and expressions.

Taking advantage of the new operations defined above, expression 3.5 can thus be rewritten as

Listing 3.6: *Eiffel*: Creation of a layout constraint using layout expressions.

```
autolayout.create_constraint ((w1.right_attribute + 20).
    equal_to (w2.left_attribute))
```

The addition of layout expressions and syntax sugar aliases helps tidying up the code for defining layout constraints, with the goal of reducing the learning curve of the autolayout container API.

Defining a series of similar constraints in a row, e.g. aligning widgets on one axis, is not an uncommon occurrence and it would be useful to be able

to express such layouts in a concise way. The current solution for describing constraints requires the developer to write one constraint at a time, sensibly extending the number of lines of code necessary to set up the user interface. For this reason we introduce a second API that allows to establish multiple constraints simultaneously in a visually straightforward style.

The alternative method for describing layout constraints, instead of using expressions, consists of representing them with strings. Defining a small grammar, the user interface can be simplified to an ASCII representation that gives a strong visual cue of how the layout will result at runtime. A string, however, can only represent the layout of one axis — i.e. either horizontal or vertical — because there is no intuitive way to describe a two-dimensional interface using a string of one single line.

Table 3.2 defines the grammar to express layouts using an ASCII representation, referred to as *visual format* from now on.

visual_format	::=	`direction` (`start-container spacing`)? `widget` (`spacing widget`)* (`spacing end-container`)?
direction	::=	`H` ‖ `V`
start-container	::=	{
spacing	::=	ϵ ‖ `-` (`number -`)?
widget	::=	`[widget_name(size)?]`
end-container	::=	}
size	::=	`(number)`
widget_name	::=	a name identifier that only contains alphanumeric characters
number	::=	a non-negative integer

Table 3.2: The visual format grammar

As an example, the horizontal layout of the application in figure 3.1 can be represented with the following layout format:

Listing 3.7: *Eiffel*: Visual format for a user interface that displays three widgets horizontally aligned.

```
H{-[button1]-20-[button2]-20-[checkbox]-}
```

It is easy to see the close resemblance of the actual user interface to the string representing it. Moreover, the visual format string defines four constraints in one single line, thus satisfying the initial goal of offering a clear and concise API alternative. Obtaining the same appearance with layout expressions would require four different instructions.

In the example, the padding between widgets and the parent container is implicitly set to be the default distance defined by the layout manager. It

is also important to note that widget names can be arbitrarily chosen since they only act as placeholders for the actual widget, which must be specified as a separate argument. The API requires in fact that along the visual layout string, the developer also gives the list of widgets involved in the constraints creation.

The visual format only covers a subset of the layout constraints that can be achieved through layout expressions, and in particular it does not allow to define:

1. size relationships between widgets, e.g. declare that two widgets have the same width;
2. the alignment of widgets to the center of their parent container.

The use of visual format is generally useful and preferred when positioning multiple objects next to each other, yet it has its own disadvantages with respect to layout expression. The first and most relevant problem is the lack of type checking at compile time: the visual format string can only be verified by the layout parser during the execution of the program. Note that there is a difference between an invalid specification and an infeasible layout: the former describes an incorrect formulation of one or more layout constraints, while the latter refers to a semantically correct set of constraints that does not have any solutions — the linear program is infeasible.

`EV_AUTOLAYOUT_PARSER` is the class responsible for parsing visual format strings and creating the constraints specified. In case a visual format is not valid, the parser collects and reports a list of the errors encountered. In the future the parser may be extended to support size relationships between widgets and the alignment to the parent container, although aside from the implementation, one must also in finding a meaningful and simple way to express these concepts only using ASCII characters.

Chapter 4

Usage

This section briefly describes the steps necessary to install and run the constraint-based layout manager in the Eiffel Verification Environment (EVE). The second part gives an overview of the available API to manage layout constraints through a sample application. The complete source code is available in Appendix A.

4.1 Installation

The constraint-based layout manager relies on the linear programming library for Eiffel described in chapter 2, which in turn relies on the C `lpsolve` library. The first component is platform independent; the Eiffel linear programming library, on the other hand, requires different files for Windows and Unix operating systems in order to interact with `lpsolve`. The license of the library allows us to integrate the original headers and the compiled libraries in the project. The Eiffel linear programming library can thus be treated as a standalone component to be integrated in EVE.

The constraint-based layout manager extends the EiffelVision framework and the whole implementation is therefore contained in the `vision2` library folder. The project is available in the main EVE repository[1].

4.2 Using EV_AUTOLAYOUT

Using the constraint-based layout manager in EiffelVision is relatively simple. The class `EV_AUTOLAYOUT` inherits from `EV_FIXED`, which already provides the necessary features for inserting, removing and accessing widgets in the container. Additionally, `EV_AUTOLAYOUT` defines the following routines to manage layout constraints (contracts and implementation are omitted):

[1] https://svn.eiffel.com/eiffelstudio/branches/eth/eve

Listing 4.1: *Eiffel*: The set of API to manage constraints in the autolayout container.

```
class EV_AUTOLAYOUT
...
  create_constraint (a_expression: EV_LAYOUT_EXPRESSION)
    do
      ...
    end

  create_constraints_with_format (a_layout_format: STRING;
      a_widgets: ARRAY [EV_WIDGET])
    do
      ...
    end

  update_constraints
    do
      ...
    end

  last_constraint: detachable EV_LAYOUT_CONSTRAINT

  constraints: LINKED_LIST [EV_LAYOUT_CONSTRAINT]

  remove_constraint (a_constraint: EV_LAYOUT_CONSTRAINT)
    do
      ...
    end
  ...
end
```

The `update_constraints` routine allows the developer to immediately reflect UI changes whenever a constraint has been modified, e.g. the padding between two widgets was updated. In order to illustrate the usage of the API, we have developed an application that demonstrates a layout that can be easily achieved using layout constraints, but would be rather complicated to implement using box containers.

The goal of the demo application is to show the main features of the constraint-based layout manager, including:

- specification of padding using the layout visual format;
- use of advanced layout attributes like horizontal and vertical alignment;
- set up size relationships between widgets inside the same container;
- dynamically changing existing layout constraints.

The application will display one button anchored to the top-left corner of the container and a checkbox anchored to the bottom-right corner of the container. A text field is placed at the horizontal and vertical center of the container, maintaining its position even when the window is resized. Moreover, all three

widgets should maintain the same width. Clicking the button at the top should increase its width of 10 pixels, and therefore cause the other two widgets to resize as well.

Figure 4.1 shows the application implementing the requirements above. The project can be replicated by creating a new graphic application (using EiffelVision 2) and replicating the source code in Appendix A.

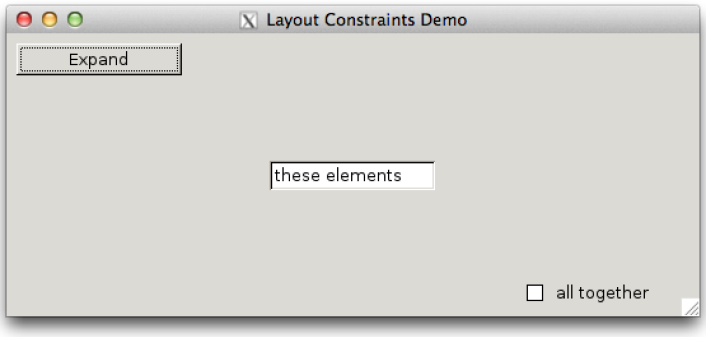

Figure 4.1: A screenshot of the sample application demonstrating the use of the autolayout container APIs.

Chapter 5

Use Cases

This chapter analyses the different performance aspects of the constraint-based layout manager. Section 5.1 solely focuses on the performances of the solver, while section 5.2 measures execution times of the autolayout container in an application that makes heavy use of layout constraints.

5.1 Linear Programming Tests

Since the optimal layout is computed solving a linear programming problem, it is of interest to measure the performance of the solver in order to predict possible limitations that may arise when formulating a complex layout specification.

The time necessary for solving a linear programming problem only depends on the external C library `lpsolve`, however one must also take into account the time needed to build the model. Despite this is a one time operation — assuming all constraints are all defined at the same time, it may have a non-negligent cost.

5.1.1 Benchmarks

The performance tests performed are based on a subset of linear programming problems freely available at NETLIB[1]. The problems differ in size — defined by number of constraints and variables — and table 5.1 shows the execution times for each of them (loading and solving time are computed separately). All tests have been performed on a 2.4 GHz Intel Core i7 processor with 8 GB of memory.

We want to understand what are the limits in terms number of widgets and layout constraints in an autolayout container. First, it is necessary to convert the number of variables in number of widgets: since each widget has 4 variables, it suffices to divide the number of variables by that number. This means that even for a layout containing about 1350 widgets and 1150 constraints, the total

[1] http://www.netlib.org

Problem name	Variables	Constraints	Load time (ms)	Solve time (ms)
AFIRO.SIF	32	27	0.153	0.174
SC105.SIF	103	105	0.391	1.299
VTP-BASE.SIF	203	198	0.901	2.847
AGG2.SIF	302	516	3.031	9.016
FORPLAN.SIF	421	161	2.604	16.389
SCAGR25.SIF	500	471	1.635	28.486
PILOT4.SIF	1000	410	3.511	67.748
MAROS.SIF	1443	846	5.616	279.110
STOCFOR2.SIF	2031	2157	7.220	291.441
SCTAP3.SIF	2480	1480	7.362	117.763
CZPROB.SIF	3523	929	8.920	363.623
SHIP08L.SIF	4283	778	10.149	138.440
SHIP12L.SIF	5427	1151	12.952	286.829

Table 5.1: Execution times for linear programming problems of different sizes, sorted by the number of variables.

time necessary to compute the optimal solution is less than a third of a second. It is usually the case, however, that for each widget there is more than one just constraint for each widget.

The number of variables is not the only determining factor for solving a linear programming problem. The `STOCFOR2` problem, for example, clearly shows that a great number of constraints also impact the time needed for finding the optimal solution.

If we want to obtain real-time performances when resizing a window, and assuming that the application frame rate is 30 Hz, then the autolayout container must find an optimal layout and apply it to the widgets in less than 33 milliseconds. According to table 5.1, the autolayout container can therefore render in real time a layout that has at most 125 widgets and about 4 constraints for each widget.

It is important to remember that if a linear program is modified, the successive solution can be computed using as a basis the previous one, thus speeding up considerably the solving time. Section 5.2 also measures subsequent executions of the solve function.

5.2 Grid Application

The grid application is a sample project exclusively developed to evaluate performances of the autolayout container. It creates a grid of $n \times n$ widgets, where n is an integer variable assuming values between 5 and 10. Widgets are randomly spaced one another — both vertically and horizontally — and the ones on the left, top and right borders are anchored to the parent container. All widgets are set to have the same width, while the height stays constant by default and thus there is no need to specify additional constraints.

5.2.1 Benchmarks

We want to understand to what extent the number of widgets — i.e. variables — and constraints impact the autolayout container and possibly prevent it from achieving real-time performances. The size of the grid is gradually increased, starting from 25 widgets up to 100 elements, and, using the EiffelStudio profiler, the average time needed to execute one complete pass (solving the linear program and updating the widgets values) of the autolayout container is measured. Table 5.2 shows the results for the different number of widgets and constraints. All tests have been performed on a 2.4 GHz Intel Core i7 processor with 8 GB of memory and the application was run in finalised mode.

Widgets	Constraints	Time (ms)
25	79	0.25
36	113	0.34
49	153	0.41
64	199	0.61
81	251	0.64
100	309	0.70

Table 5.2: Average time for one pass of the autolayout container main algorithm.

The average time for solving a linear program is about two orders of magnitude smaller with respect to the time for the first execution of the algorithm. Reusing the previous solution to find the new optimal solution for the updated problem is of fundamental importance to allow the autolayout container to update the user interface in real time. Contrary to the expectations in the previous section, the autolayout container should still be able to manage hundreds of widgets without negatively affecting the user experience.

The profiler also highlighted an unusual timing of the `EV_FIXED` routines for setting the position and size of widgets. It appears that the runtime checks performed in the routines slow down their execution of about 100%; removing these checks reduces in fact the execution time from 4 ms to 2 ms. While this change does not strongly impact the efficiency autolayout container, it is still important to consider the overall performances when executing code on the UI thread in order to maintain a pleasant user experience.

Chapter 6

Conclusions

6.1 Conclusions

The introduction of linear programming in the context of user interfaces is an endeavour to improve and simplify the development of layouts. In practice, complex appearances defined by means of containers offered in the EiffelVision framework are often more clear and concise to build using layout constraints.

In this master thesis we developed a new widgets container for the EiffelVision library. It relies on a layered architecture that abstracts from the mathematical model of linear programming to a higher level language, throughout this work referred to as layout language. The layout language aims at resembling as close as possible to the natural language for expressing constraints in a user interface. We have successfully used the constraint-based layout manager in a series of sample applications and tests to measure performances, showing therefore that is possible to choose the autolayout container as a valid alternative to the existing ones.

The recent adoption of linear programming as a mean to describe interfaces in different environments, in particular — but not exclusively — in the area of mobile devices, emphasises the effectiveness of this technique and the need for a simple way to build user interfaces. We believe that further development of the constraint-based layout manager for Eiffel could simplify and speed up the creation of user interfaces even more.

6.2 Future Work

Although the constraint-based layout manager is already capable of representing relatively complex layouts, it is possible to further extend its flexibility. The layout constraints defined in an autolayout container are called *hard*, because they must be satisfied at all times. If this is not the case, then the layout is infeasible and the program execution terminates.

There are some occasions, however, in which constraints do not necessarily need to be satisfied — or can be partially satisfied — and provide therefore more tolerance to conflicting layout specifications. For instance, one might want to set the width of a widget to be twice the width of another one, but only if there is enough space available (i.e. the container is wide enough). On the other hand, an insufficiently wide container will still expand the widget as much as possible and mark the violated constraint with a *penalty coefficient*, a value in the linear programming problem that expresses the deviation from the optimal solution. Soft constraints always guarantee that a solution exists and are therefore useful when dealing with conflicting constraints in linear programming problems. The `SOFT_CONSTRAINT` class in the linear programming library defines an initial implementation of soft constraints, however it is not supported yet in the constraint-based layout manager.

A graphical tool to create, modify and inspect constraints should be considered a high priority goal in the future development of the constraint-based layout manager. Even though defining the layout through the autolayout container APIs is relatively straightforward, an application that lets the developer see the actual interface before even running the program definitely speeds up and eases the GUI development. The existing EiffelBuild application enables developers to develop user interfaces using box, table and fixed containers, though the interaction is limited by the specific behaviour of each container: widgets cannot be freely dragged around or anchored arbitrarily to the parent container, limiting thus the possibilities for creating highly customised layout interfaces.

Appendix A

Layout Constraints Demo

Listing A.1: *Eiffel*: Complete source code of the layout constraints demo application's root class.

```
class
  LAYOUT_CONSTRAINTS_DEMO_APPLICATION

inherit
  EV_APPLICATION

create
  make_and_launch

feature {NONE} -- Initialization

  make_and_launch
      -- Initialize and launch application
    do
      default_create
      prepare
      build_interface
      launch
    end

  prepare
    do
        -- create and initialize the first window.
      create first_window
      first_window.wipe_out
      first_window.show

      create container
      first_window.extend (container)
    end

  build_interface
    local
      button: EV_BUTTON
```

```
      field: EV_TEXT_FIELD
      checkbutton: EV_CHECK_BUTTON
    do
      create button.make_with_text ("Expand")
      create field.make_with_text ("these elements")
      create checkbutton.make_with_text ("all together")

      container.extend (button)
      container.extend (field)
      container.extend (checkbutton)

      container.create_constraints_with_format ("H{-[button]
          ", << button >>)
      container.create_constraints_with_format ("H[check]-}"
          , << checkbutton >>)
      container.create_constraint (container.layout.
          horizontal_center_attribute.equal_to (field.layout.
          horizontal_center_attribute))
      container.create_constraint (container.layout.
          vertical_center_attribute.equal_to (field.layout.
          vertical_center_attribute))

      container.create_constraints_with_format ("V{-[button]
          ", << button >>)
      container.create_constraints_with_format ("V[check]-}"
          , << checkbutton >>)

      container.create_constraint (button.layout.
          width_attribute.equal_to (field.layout.
          width_attribute))
      container.create_constraint (field.layout.
          width_attribute.equal_to (checkbutton.layout.
          width_attribute))

      container.create_constraint (button.layout.
          bottom_attribute + 20 <= field.layout.top_attribute
          )

      container.create_constraint (button.layout.
          width_attribute >= 100)
      width_constraint := container.last_constraint

      button.select_actions.extend (agent do
        width_constraint.constant := width_constraint.
            constant + 10
        container.update_constraints
      end)

    end

feature {NONE} -- Implementation

  first_window: MAIN_WINDOW
      -- Main window.
```

```
  container: EV_AUTOLAYOUT

  width_constraint: EV_LAYOUT_CONSTRAINT

end -- class APPLICATION
```

Bibliography

[1] Greg J. Badros, Alan Borning, and Peter J. Stuckey. The cassowary linear arithmetic constraint solving algorithm. *ACM Trans. Comput.-Hum. Interact.*, 8(4):267–306, December 2001.

[2] Christof Lutteroth, Robert Strandh, and Gerald Weber. Domain specific high-level constraints for user interface layout. *Constraints*, 13(3):307–342, September 2008.

[3] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1st edition, 1988.