DIPLOMARBEIT

Space-saving Placement Using a Structural User Interface Model

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines Diplom-Ingenieurs unter der Leitung von

Univ.Prof. Dipl.-Ing. Dr.techn. Hermann Kaindl
und
Proj.Ass. Dipl.-Ing. Dr.techn. Jürgen Falb
als verantwortlich mitwirkendem Assistenten am
Institutsnummer: 384
Institut für Computertechnik

eingereicht an der Technischen Universität Wien
Fakultät für Elektrotechnik und Informationstechnik

von

Michael Leitner
Matr.Nr. 9926617
Löwenherzgasse 9 Tür 8, 1030 Wien

23.05.2010 _____

Kurzfassung

Im Zuge der automatischen Erstellung von *Graphical User Interfaces* (GUIs) durch die *Unified Communication Platform* (UCP) kommt es zu einem Zwischenschritt, bei dem ein abstraktes Modell des GUIs erstellt wird – das *structural User Interface* (*structural UI*) Modell. Dieses beinhaltet Informationen über den strukturellen Aufbau und den Inhalt des GUIs, die einzelnen Anzeigeelemente haben jedoch zunächst noch keine zugewiesene Größe und keine Layoutdaten. Diese Daten müssen nachträglich berechnet und an eine vorgegebene Bildschirmgröße angepasst werden. Um diese Aufgabe zu erfüllen, wurde im Rahmen dieser Diplomarbeit das *Layout Module* entwickelt. Es berechnet die erforderlichen Größen und Layout Daten und liefert ein an die vordefinierte Bildschirmgröße angepasstes *structural UI* Modell zurück. In weiterer Folge kann dieses *structural UI* Modell in Java Code transformiert werden. Die Berechnung der Layoutdaten wird durch die Einführung eines Rasters mit einheitlicher Zellgröße ermöglicht. Die benötigten Zeilen und Spalten pro Anzeigeelement werden berechenbar und mögliche Einfügepunkte können gefunden werden. Im Falle mehrerer möglicher Einfügepunkte werden Kriterien wie resultierende Platzverschwendung, Bildschirmgröße oder Proportion für die Entscheidungsfindung herangezogen. Ästhetik spielt dabei noch eine untergeordnete Rolle, kann aber in Form einer Optimierung bei der Entscheidungsfindung einbezogen werden, sofern die Komplexität des *structural UI* Modelles dies zulässt.

Abstract

During the automatic generation of graphical user interfaces (GUIs) using the Unified Communication Platform, it comes to an intermediate step where an abstract GUI model is created – the structural user interface (structural UI) model. This model contains information about the structure and the content of the GUI, but size and layout data are not set in advance. Considering a predefined target device's screen size, these data have to be computed afterwards. To fulfil this task, the Layout Module was created within the scope of this master thesis. It calculates the required size and layout data and returns a structural UI model that is tailored to the predefined target device's screen size. Afterwards, this structural UI model can be transformed to Java code. An introduced grid with equally sized cells allows the computation of the required layout data. Rows and columns required by each widget can be computed and possible insertion points can be found. In case of several possible insertion points for a single widget, decisions are made considering the resulting wasted space, screen size and proportion. Currently, aesthetics play a minor part for these decisions but can be subject for optimisations if the complexity of the structural UI model allows it.

Acknowledgements

"Immer einen Schritt nach dem Anderen...

...und nicht aufgeben. Hörst du?"

Das waren die Worte meiner Mutter, wenn es mal nicht so wollte mit der Studiererei. Und ich bin froh, dass sie sie mir diese Einstellung vermittelt hat. Es zählt nicht immer nur die Zeit, die man für etwas benötigt. Manchmal ist es einfach wichtiger, dass man die Geduld und den Willen hat etwas zu einem Ende zu bringen. Egal wie lange es dauert. Am Beginn meines Studiums der Elektrotechnik wusste ich nicht so genau, worauf ich mich da einlasse. Aber letztendlich ich habe das Gefühl, dass es eine gute Wahl war.

Vielen Personen in meinem Umfeld gebührt Dank. Personen, ohne deren Hilfe ich wohl jetzt nicht am Wiener Donaukanal sitzen und nach Worten suchen würde, um mich bei Ihnen zu bedanken. Allen voran möchte ich mich bei meinem Vater und meiner Mutter bedanken, die wohl eine ganze Menge an Geduld und Geld aufbringen mussten, um mich zu einem Abschluss zu begleiten. Danke, dass ihr immer an mich geglaubt habt. Aber auch meinem Bruder, meiner Schwester und ihrem Mann möchte ich für ihre Ratschläge und Motivationskünste danke sagen. Und nicht zu vergessen, meiner kleinen Nichte und meinem kleinen Neffen, die es immer wieder schaffen mir ein Lächeln ins Gesicht zu zaubern. Auch Dir ein Dankeschön, Susi, schön dass du da bist. Ich freu mich schon auf unseren Besuch in München. Danke dass ich Euch alle hab. Ich glaub ihr wisst, wie gern ich euch hab.

Jemanden, der sicher auch für mich das ein oder andere Gebet gesprochen hat, möchte ich hier auch erwähnen. Pater Anton Ringseisen, danke für Deine Freundschaft.

Herzlich bedanken möchte ich mich bei dieser Gelegenheit auch bei Univ.Prof. Dipl.-Ing. Dr.techn. Hermann Kaindl und Proj.Ass. Dipl.-Ing. Dr.techn. Jürgen Falb für die engagierte Betreuung. Danke auch an Euch, Roman Popp und Sevan Kavaldjian, für Eure Hilfsbereitschaft und Cafe bzw. Mittagessgesellschaft. Und Dir, David, ein großes Dankeschön für deine fast 24-Stunden Service-Line. Es hat Spaß gemacht mit Euch zusammenzuarbeiten.

Was wäre so ein Studium ohne die richtigen Freunde und Innen? Danke, dass ich meine Zeit mit Euch verbringen darf und auf Euch zählen kann, wenn es mal wo hakt. Da eigentlich alle meine Kollegen ja im Laufe der Zeit auch zu guten Freunden geworden sind, seid Ihr ja in den vorigen Sätzen auch gemeint.

Zu guter Letzt, Steffi, das letzte Jahr mit Dir war super. Ich hoffe auf Weitere. Und auf eine tolle Reise.

Widmen möchte ich diese Arbeit meiner Mutter, die leider seit fast schon drei Jahren nicht mehr auf dieser Welt ist. Gerne hätte ich Dein Gesicht gesehen, wenn ich Dir mein Diplom gezeigt hätte. Ich hoffe, Du machst dir jetzt keine Sorgen mehr um mich. Ich hab Dich lieb und Du fehlst mir. Danke für Alles.

Michi

Contents

Abbreviations and Acronyms

CSS	Cascading Style Sheet
DPI	Dots Per Inch
FWL	Flexible Widget Layout
GUI	Graphical User Interface
ICT	Institute of Computer Technology
LA	Layout Appropriateness
PC	Personal Computer
PDA	Personal Digital Assistant
UCP	Unified Communication Platform
UI	User Interface
URL	Uniform Resource Locator

Chapter 1

Introduction

Automatic generation of GUIs (Graphical User Interfaces) is a promising topic. Research started in the early 1980s and is still subject of current projects. The main concern is to make the generation of GUIs transparent to the designer in order to save time. Since time is closely linked to money and manually programming GUIs is a time consuming task, automatic GUI-generation would spare a lot of money.

The UCP (Unified Communication Platform) represents an approach that allows automatic generation of GUIs for multiple target devices (e.g. PC monitor, PDA, touch screen, etc.). The UCP has been developed by the Institute of Computer Technology (ICT) at the Vienna University of Technology. The UCP permits designers, even with limited programming skills, to create GUIs by modeling the interaction between two communication parties. A discourse model acts as basic model and is an abstract representation of the interaction between the two communication parties. This discourse model is transformed into a structural UI (structural User Interface) model, which is an abstract representation of the GUI and contains information about the contained widgets ("Windows + Gadget") and their hierarchy. Within this model-to-model transformation process, the widgets' size and layout attributes need to be set automatically. This is required to display the contained widgets properly on the target device's screen, in a non-overlapping manner. Furthermore, these size and layout attributes are necessary to calculate the size of each screen that is contained by a certain GUI. This information allows to check if the GUI fits into the used target device's screen size.

The goal of this master thesis was to automatically create a layout for any structural UI model generated by the UCP. Therefore, the Layout Module has been implemented to calculate size and layout-data (see Chapter 4). The Layout Module tries to create a layout that fits into a predefined target device's screen size while simultaneously presenting a maximum amount of information.

The widgets' size is generally obtained from a Cascading Style Sheet. For certain widgets, it is required to calculate size, or to obtain size from an alternative source. For example, the size of container widgets (e.g. panels) must be calculated considering the collocation and size of their contained widgets. The width of some other widgets (e.g. label, button, etc.) has to be set to a default value. This is required because the text to be displayed is not available in the course of the size and layout calculation.

The Layout Module's placement strategy tries to arrange the widgets within their container in a space-saving manner. After each placement, the entire unused space remains available for a

further placement. The next widget to be placed is inserted at a position that leads towards a minimum amount of wasted space. Therefore, the functionality of the GridBagLayout-Manager provided by Java is used. In order to make placement decisions computable, the grid-cells are set to a defined width and height for each container. The size of the single widgets defines the number of grid-cells they occupy in the grid. After each widget insertion, an image of the current container is created. This image – the grid – is implemented as a two-dimensional integer array that provides information about the current state of each grid-cell (e.g. occupied, free, forbidden, etc.). Based on this grid, it is possible to search possible insertion points and to compare them in terms of space consumption.

The text at hand starts with a short description of aesthetic characteristics and mathematical relationships in Chapter 2. These characteristics and relationships can help to make decisions where to place widgets in a pleasing manner. Furthermore several existing placement strategies are presented. Chapter 3 outlines basic information on the UCP framework, which is needed to comprehend the functionality of the Layout Module. It starts with a short description of the discourse model and the structural UI model. The model-to-model transformation that generates the structural UI model completes the chapter. In Chapter 4, the functionality and implementation of the Layout Module are outlined. The calculation of a widget's size and the layout algorithm are described in detail and some examples are given to illustrate possible outcomes that can be achieved with the aid of the Layout Module. Finally, some suggestions for an optimisation of the Layout Module are given in Section 5.

Chapter 2

State of the Art

Projects that aim to automatically generate graphical user interfaces can follow different approaches. Task descriptions or aesthetic characteristics can be either considered or not, and the models that are used to describe the users' tasks and to describe the structure of the graphical user interface (GUI) may differ in their level of abstraction. To give an overview, some of these approaches are listed below:

- **SUPPLE** represents a dynamic approach for automatic GUI generation for multiple target devices [GW04]. It considers the users' tasks and tries to minimize the estimated effort to perform these tasks. Furthermore, it generates GUIs that are tailored to the user's motor and vision capabilities (SUPPLE++) [GLW06, GWW07].
- **AESTHER** generates GUIs for the handheld device domain [YK09]. It rather considers aesthetic principles than task-description when creating the layout for an application.
- **FWL (Flexible Widget Layout)** generates GUIs that conform to the user's preferences [YNK09]. The user can assign desirabilities to widgets that have the same functionality but differ in their size (e.g. ListBox and DropDownListBox, Spinner and Slider, etc.). If the target device provides enough space, the preferred widgets are used.
- **LA (Layout Appropriateness)** represents an approach that generates layouts that are optimised according to a simple description of the tasks that can performed by a user [Sea93]. This simple task-description contains a description of each sequence of actions performed by a user, together with the frequency of the task's occurrence. A LA-optimal layout can be computed accordingly. Furthermore, a LA-value can be calculated for existing GUIs according to a corresponding simple task-description, allowing a comparison of different layouts for the same application in terms of the user's task performance.

However, all of these GUI generating approaches have one major problem in common. Each of them comes to a point, where the single widgets have to be placed on the screen. A placement strategy is required.

Bodart et al. state that any placement strategy has to answer the following three questions for each widget-placement [BHLV94]:

- **Localization**: Where should the widget be placed on the screen?
- **Dimensioning**: How large should the widget be placed on the screen?
- **Arrangement**: According to which order should the widget be placed on the screen?

In their paper, two placement-strategies are presented. A two-column based and a dynamic right/bottom strategy. Kim et al. present a placement strategy that arranges widgets with the aid of a decision-tree according to their shape and size [KF93].

This chapter starts with a short description of aspects that a placement strategy may consider in terms of user interface design (see Section 2.1). In Section 2.2 the three placement-strategies that were mentioned above are described in more detail.

2.1 User Interface Design

The current section outlines some principles of user interface design. In case of manually arranging the widgets, these principles can be satisfied as desired. If the layout has to be created automatically, the principles can be used to make decisions where to place the different widgets. However, it must be noted that these principles do not consider any content. They just propose a possible arrangement of widgets, and provide information about the visual appearance of a UI.

The section starts with a description of a set of aesthetic characteristic measures in 2.1.1. Moreover, several mathematical relationships that can help to improve practicability, applicability and workability are outlined in 2.1.2.

2.1.1 Aesthetic Characteristics

Various studies have been made concerning the appearance of user interface widgets (screen elements) on a screen. Vanderdonckt et al. describe a set of visual techniques how widgets should be arranged [VG94]. Depending on the involved widgets, these techniques can be difficult to apply. Ngo et al. state that an aesthetically well designed UI reasonably influences the UI's *usability* and its *acceptability* [NTB03]. Furthermore, they say that aesthetic UIs greatly increase *learnability*, *comrehensability* and *productivity*.

In order to measure the aesthetics of a collocation of widgets on a screen, they elaborated mathematical formulae for a set of characteristics, and present the methods that allow a computation of a measure for each characteristic. Altogether, they elaborated fourteen aesthetic characteristic measures, which are summarized in the following paragraphs. Detailed information about the computation of these characteristics can be found in the the paper [NTB03].

Assigning a constant weight to each measured characteristic to indicate its importance, an overall aesthetic measure can be computed. This can be achieved by calculating the weighted arithmetical average of the different computed characteristic results. With the aid of this overall measure, an existing UI can be evaluated. Furthermore, different UIs can be compared and rated in terms of their aesthetic appearance.

Aesthetic Measure of Screen Balance

The balance of a screen depends on the distribution of optical weight within the screen. The human eye senses each widget with a different weight. This weight depends on different factors that let some widgets appear heavier than others. For example, large widgets appear visually heavier than little ones. Color is visually heavier than shades of gray, black is heavier than white and irregularly shaped widgets appear heavier than regularly shaped ones do.

To be aesthetically balanced, the widgets of a screen must be arranged in the following way:

- **Horizontally balanced:** The top of the screen and its bottom must be equally weighted.
- **Vertically balanced:** The left and the right side of the screen must be equally weighted.

A screen's weight is computed considering its widgets' sizes and the distance from these widgets' centre-line to the respective centre-line of the screen (vertical or horizontal). Color as well as shape of the widgets is not considered here. At each side of the screen's centre lines (top, bottom, left and right), the weights are summarized and a total weight is computed for these parts of the screen. The difference between the left and the right total weight divided by their maximum results in the vertical balance. The difference between the top and the bottom total weight divided by their maximum results in the horizontal balance. Balance can finally be calculated depending on the horizontal and vertical balance.

Aesthetic Measure of Screen Equilibrium

In comparison to balance, which considers the visual weight, equilibrium considers the visual centre. It is a measure of the centre deviation of a collocation of widgets. If the centre of a collocation of widgets coincides with the screen's centre, perfect equilibrium is achieved. In this case, the screen is in a state of repose. Otherwise, there occurs a contradiction between the two different centers. The measure of equilibrium depends on the arithmetical average of the vertical and horizontal equilibrium.

Aesthetic Measure of Screen Symmetry

Symmetry is defined as axial duplication – a widget that appears on one side of a centre line is equivalently replicated on the other side. Perfect symmetry automatically causes balance, but balance does not necessarily cause symmetry. Concerning symmetry, a screen can be characterized by its deviation of vertical, horizontal and radial symmetry:

- **Vertical symmetry** states that the left and the right side of a screen consist of equivalent widgets and are arranged the same way.
- **Horizontal symmetry** states that the top and the bottom of the screen consist of equivalent widgets and are arranged the same way.
- In case of two or more axes, **radial symmetry** states that the partitioned areas consist of equivalent widgets and are arranged the same way.

A possibly existing deviation of the three itemized symmetries can be computed for each screen. The arithmetical average of these three deviations defines the measure of a screen's symmetry.

Aesthetic Measure of Screen Sequence

The measure of sequence of a screen depends on the arrangement of the the screen's widgets. Widgets can be placed in a certain sequence to facilitate the movement of the human eye when looking at the screen. For example, in the Western cultures the human eye is trained by reading. It starts at the upper left corner of a screen, moves forwards and backwards across the screen and ends at the lower right corner. Furthermore perceptual psychology revealed that the eye moves from big widgets to small ones. According to these facts, a measure of sequence can be calculated.

Aesthetic Measure of Screen Cohesion

Aspect-ratio is defined as the relationship between width and height of an object. Similar aspect-ratios provide a good aesthetic in terms of cohesion. Therefore, aspect-ratios should stay the same within a screen.

A measure of screen cohesion can be computed according to the two relative ratios:

- The aspect-ratio of the layout divided by the aspect-ratio of the screen.
- The relative arithmetic average aspect-ratio of the contained widgets divided by the aspect-ratio of the layout.

Aesthetic Measure of Screen Unity

Unity is the property of a set of widgets to seem related or even to be seen as one thing. To achieve unity within a screen, its widgets should be similarly sized and placed closer together. The space between the widgets should be less than the space that is left to the extents of the screen.

The measure of screen unity can be computed depending on the following three factors:

- The number of different sizes used.
- The free space within the layout.
- The free space within the whole screen.

Aesthetic Measure of Screen Proportion

A measure of proportion of a screen can be derived according to the proportional relationship of the screen's widgets and the screen itself. There exist numerous proportional relationships that are preferred in different cultures, and that are classified as aesthetically pleasing in all of those cultures. According to Marcus [Mar92], these pleasing proportional relationships are itemized below:

- **Square:** 1:1
- **Square root of two**: $1{:}\sqrt{2} = 1{:}1.414$
- **Golden rectangle:** 1:1,618
- **Square root of three**: $1{:}\sqrt{3} = 1{:}1.732$
- **Double square:** 1:2

In order to achieve an aesthetically pleasing screen, these proportional relationships should be used as much as possible. The more the widgets of a screen and the screen itself resemble these preferred proportions the better becomes the measure of screen proportion.

Aesthetic Measure of Screen Simplicity

Simplicity is given if the meaning of a collocation of widgets is easy to comprehend. The more widgets a screen contains, the more alignment points are required and simplicity declines. The measure of simplicity of a screen can be improved by minimizing the number of widgets and by minimizing the number of horizontal and vertical alignment points.

Aesthetic Measure of Screen Density

Density is an indicator of the relation between the total screen area and the area used by the screen's widgets. Ngo et al. claim that a screen density of 50% is the optimum relation. According to this optimum relation, an aesthetic measure of screen density can be computed.

Aesthetic Measure of Screen Regularity

Regularity analyses vertical and horizontal alignment points of a collocation of widgets. In comparison to simplicity, regularity is less sensitive to the number of these alignment points. It rather analyses the uniformity of the existing alignment points. The number of distinct horizontal and vertical distances between the alignment points is taken into account. To achieve regularity, a screen should have as few horizontal and vertical alignment points as possible and the number of different distances between them should be held at a minimum.

Aesthetic Measure of Screen Economy

The measure of economy serves to indicate how simple a message displayed to the user is. The fewer different sizes (of widgets) appear on a screen the better the measure of economy. Therefore, a perfect screen in terms of economy can be achieved by using the same size for each contained widget.

Aesthetic Measure of Screen Homogeneity

The measure of homogeneity provides the information of how evenly widgets are distributed among the quadrants of a screen. Perfect homogeneity can be achieved by placing the same number of widgets in each quadrant of the screen.

Aesthetic Measure of Screen Rhythm

The rhythm of a screen indicates how frequently patterns of changes of widgets are recurring on a screen. A screen with a recurring set of patterns of changes of widgets appears more exciting in terms of rhythm, than a complex screen whose widgets are not ordered systematically.

Aesthetic Measure of Screen Order and Screen Complexity

The measure of order of a screen depends on the aggregate of all measures that were mentioned in the previous paragraphs. It ranges from minimal order, which indicates high complexity, to maximal order.

2.1.2 Mathematical Relationships

As illustrated in Figure 2.1, each widget is characterized by the coordinates of its upper left (UL) and bottom right (BR) corners. According to the coordinate system of a computer screen, the positive branch of the y-axis in the figure is oriented downwards.

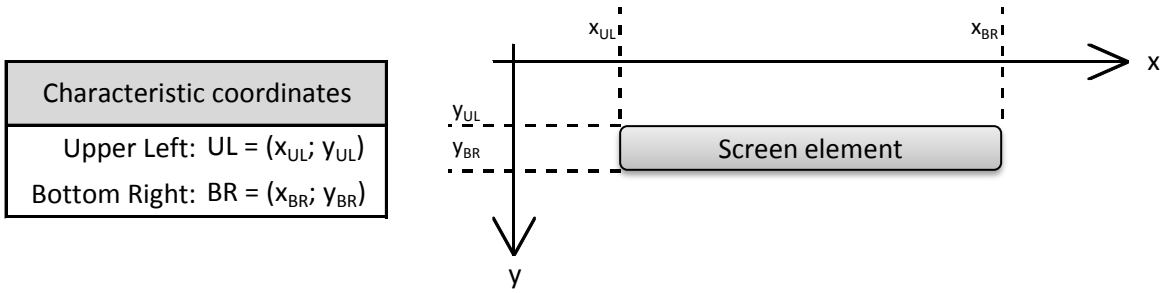


Figure 2.1: Characteristic coordinates of a widget.

According to these two coordinates and the coordinate system of Figure 2.1, a set of simple mathematical relations between widgets can be established [BHLV94].

Horizontal sequencing

The two widgets E_1 and E_2 are horizontally sequenced if x_{BR,E_1} is less than x_{UL,E_2}. In this case, E_2 is placed next to E_1.

Vertical sequencing

The two widgets E_1 and E_2 are vertically sequenced if y_{BR,E_1} is less than y_{UL,E_2}. In this case, E_2 is placed below E_1.

Left justification

The two widgets E_1 and E_2 are left justified if x_{UL,E_1} is equal to x_{UL,E_2}.

Right justification

The two widgets E_1 and E_2 are right justified if x_{BR,E_1} is equal to x_{BR,E_2}.

Upper justification

The two widgets E_1 and E_2 are upper justified if y_{UL,E_1} is equal to y_{UL,E_2}.

Bottom justification

The two widgets E_1 and E_2 are bottom justified if y_{BR,E_1} is equal to y_{BR,E_2}.

Horizontal centering

The two widgets E_1 and E_2 are horizontally centered if their horizontal centre-lines coincide ($\frac{y_{UL,E_1}+y_{BR,E_1}}{2} = \frac{y_{UL,E_2}+y_{BR,E_2}}{2}$).

Vertical centering

The two widgets E_1 and E_2 are vertically centered if their vertical centre-lines coincide ($\frac{x_{UL,E_1}+x_{BR,E_1}}{2} = \frac{x_{UL,E_2}+x_{BR,E_2}}{2}$).

Horizontal equilibrium

Three or more widgets are horizontally equilibrated if the horizontal distances between the objects are equal.

Vertical equilibrium

Three or more widgets are vertically equilibrated if the vertical distances between the objects are equal.

Diagonal equilibrium

Three or more widgets are diagonally equilibrated if their centre-points are equally distributed and, therefore form a straight line when being connected.

Horizontal uniformity

Widgets are horizontally uniform if their lengths are equal.

Vertical uniformity

Widgets are vertically uniform if their heights are equal.

Proportional equilibrium

Three or more widgets are proportionally equilibrated if both of the following two conditions are applicable:

- The widgets are either vertically or horizontally uniform.
- The widgets are either vertically or horizontally equilibrated.

Total equilibrium

Three or more widgets are totally equilibrated if both of the following two conditions are applicable:

- The widgets are vertically and horizontally uniform.
- The widgets are either vertically or horizontally equilibrated.

2.2 Placement Strategies

The current section outlines three placement strategies in detail. It starts with a two-column based strategy and a right/bottom strategy proposed by [BHLV94]. The third strategy, proposed by [KF93], is based on shape and size of the single widgets.

2.2.1 Two-Column Based Strategy

Independently from the set of widgets, the static two-column based strategy always results in the same layout structure. As depicted in Figure 2.2, this layout structure consists of a title on the top, two columns for the widgets and a preserved space for buttons either on the bottom or on the right side of the screen.

The placement strategy tries to create two columns that ideally have the same height. Therefore, it places all widgets below each other, resulting in a stack with a certain height and width. This placement is performed considering some of the mathematical relationships that were mentioned above. More precisely, the four justifications, the two uniformities and vertical or horizontal equilibrium are considered when creating the stack. If the stack's half height is exactly between two widgets, the stack is divided at this position, resulting in two columns of the same height. If the stack contains a widget at its half height, the stack is divided at the next possible position, that results in a minimum of deviation of height between the two columns. Therefore, the stack is either divided at the top or at the bottom of the widget that is placed in the stack's middle. The two columns' widths depend on the maximum width of its contained widgets.

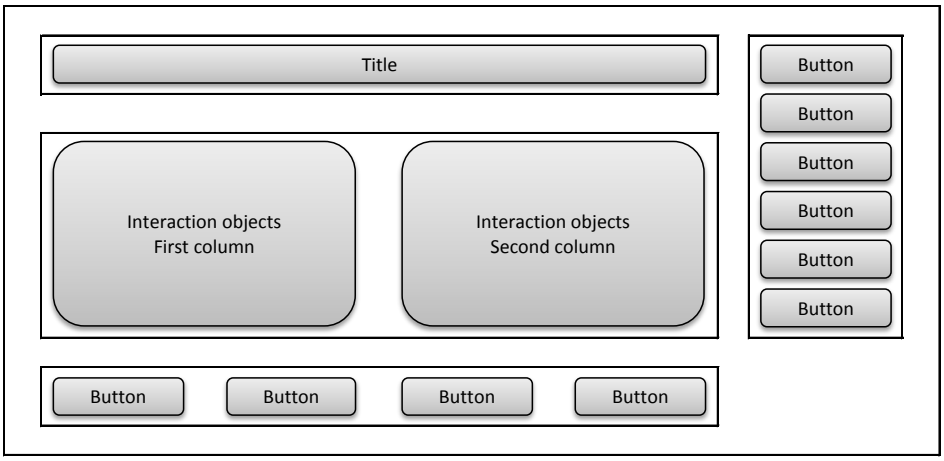


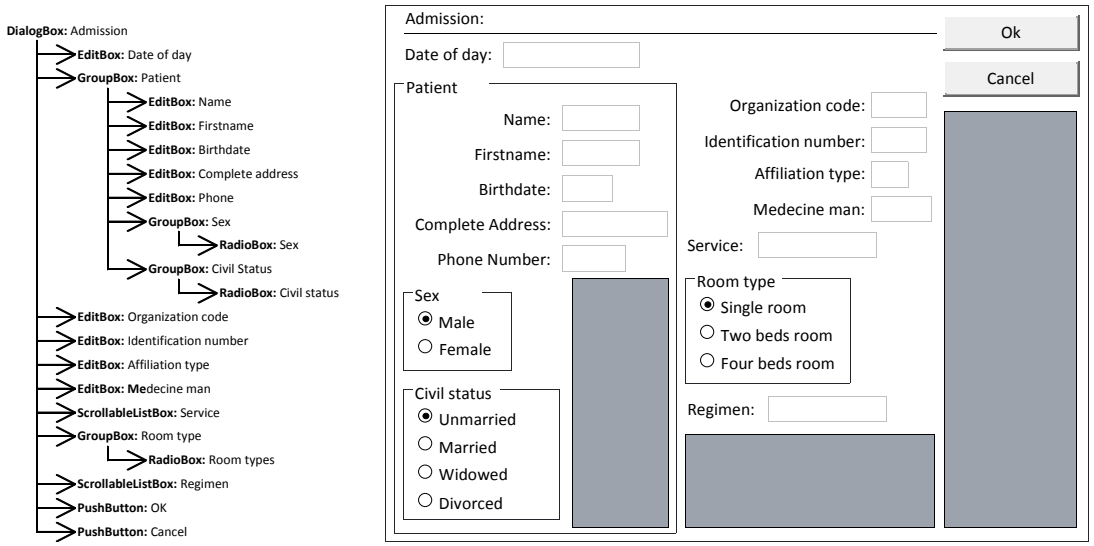
Figure 2.2: Grid structure of the two-column based strategy (redrawn according to [BHLV94]).

The choice of the location for buttons depends on the shape of the area occupied by the title and the two columns. If this area is shaped horizontally, the buttons are arranged horizontally on the bottom. If the constellation is shaped vertically, the space on the right side is chosen for the buttons. In both cases, the placement strategy tries to arrange the buttons considering proportional or total equilibrium. In case of horizontally arranging the buttons on the bottom, they are separated by an equal distance and get the same height. In case of vertically arranging the buttons at the right side, the buttons are equally separated as well and each button gains the same length.

The two-column based strategy automatically groups related widgets close to each other. Further, it justifies the widgets accordingly in order to increase unity. However, it has to be stated that the use of special space for the buttons provides consistency but requires a lot of additional space. The separation into two columns can possibly result in a huge amount of unused space as well.

To give an example, a set of widgets with a predefined hierarchy is illustrated in Figure 2.3(a). It represents a screen of an application for hospital admission. The result that can be achieved with the aid of the two-column strategy is depicted in Figure 2.3(b). One of the main disadvantages of this strategy is the huge amount of space that can possibly remain unused. This unused space is represented by the grayed regions in the figure. As shown in Figure 2.3(b), the two columns must not be of equal height. Therefore, the space at the bottom of the second column remains unused. Further unused space results from the fact that all widgets are arranged below each other before separating the stack. A possibly existing space next to a widget (e.g. next to the group boxes in the first column of the figure) is not available for further placement. The additional space for the buttons on the right side causes unused space as well.

The two-column based strategy most likely achieves a good score in terms of screen balance, symmetry and alignment. On the other hand, due to the large amount of required space in most cases, it would fail in terms of economy. Furthermore, a layout created by this strategy may result in a poor LA-score (Layout Appropriateness [Sea93]).



(a) User interface model. (b) Using the two-column based strategy.

Figure 2.3: Two-column based strategy – example (redrawn according to [BHLV94]).

2.2.2 Right/Bottom Strategy

The second placement strategy described in [BHLV94] is the right/bottom strategy. It is a strategy that dynamically places widgets considering the remaining space on the screen. In comparison to the two-column based strategy, the right/bottom strategy results in a layout that is optimised in terms of space consumption. If possible, the strategy places widgets with horizontal sequencing. Depending on the ratio between the height of the widget to be placed and the height of the previously placed widget, the strategy applies some of the mathematical relationships mentioned in 2.1.2. If both heights are equal, proportional uniformity is applied [BHLV94]. In case of different heights, top or bottom justification is applied depending on the available space. Furthermore, the strategy tries to square radio buttons as much as possible in order to preserve balance and unity. Widgets that own an identification label are either arranged with horizontal sequencing and bottom justification, or with vertical sequencing in conjunction with left justification.

Using the right/bottom strategy, the hierarchy of widgets that served as an example for the two-column based strategy (see Figure 2.3(a)) results in the GUI shown in Figure 2.4.

Since the right/bottom strategy tries to minimize the wasted space and to place widgets close together, it is most likely well rated in terms of LA. Although, in case of high screen density, it may appear in an unpleasant way.

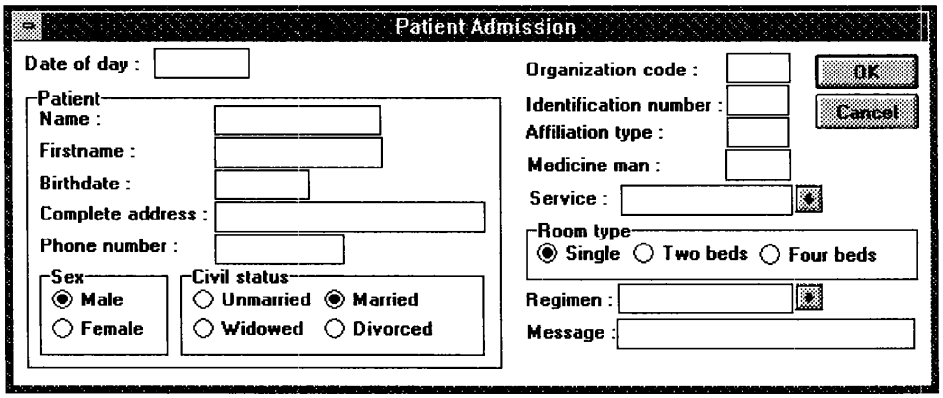


Figure 2.4: Result of the Right/Bottom strategy [BHLV94].

2.2.3 Shape- and Size-Analysis Based Strategy

A strategy that considers the shape and the size of widgets when arranging them is presented in [KF93]. A binary tree structure of bounds (extents) serves as a representation of the GUI. The leafs at the bottom of the tree represent the single widgets and, therefore have defined dimensions. The dimension of the group-bounds above, which represent the arrangement of their leafs, is initially unknown and are obtained within the process.

As depicted in Figure 2.5, this tree structure is processed from bottom to top. At each step, two bounds are analysed in terms of size and shape. According to this information, the two bounds are arranged with horizontal or vertical sequencing:

- If the maximum width of the two bounds is greater than their maximum height or if these two values are equal, vertical sequencing in conjunction with left justification is applied. The wider of the two bounds is placed on top.
- If the maximum width of the two bounds is less than their maximum height, horizontal sequencing in conjunction with upper justification is applied. The higher of the two bounds is placed on the left.

The created arrangement determines the extents of the next higher group-bound of the tree. A possibly resulting free area (remainder area) remains available for further placement. When encountering the root of the tree, the process ends and dimension and layout-data of all bounds is set.

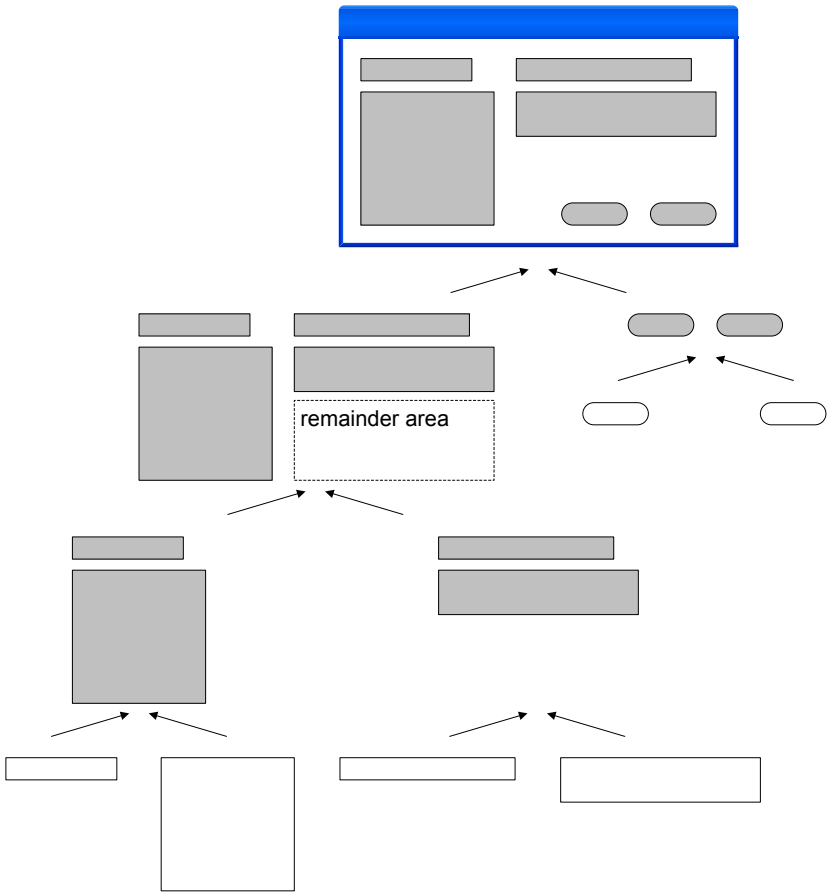


Figure 2.5: Placement process, working with the basic shape analysis (redrawn according to [KF93]).

In order to increase efficiency, the strategy can be extended in a way that more than two bounds are examined and arranged at once. As depicted in Figure 2.6, bounds with similar shape and size are grouped and arranged in the following way:

- If the majority of similar bounds is shaped horizontally, they are arranged with vertical sequencing in conjunction with left justification, starting with the broadest bound on the top.
- If the majority of similar bounds is shaped vertically, they are arranged with horizontal sequencing in conjunction with upper justification, starting with the highest bound on the left side.

If the given screen does not provide enough space for a certain arrangement, the strategy tries to apply *overflow solutions* (e.g. minimizing certain widgets; replacing widgets by smaller alternatives; minimizing margins and spacings; etc.) [Kim93].

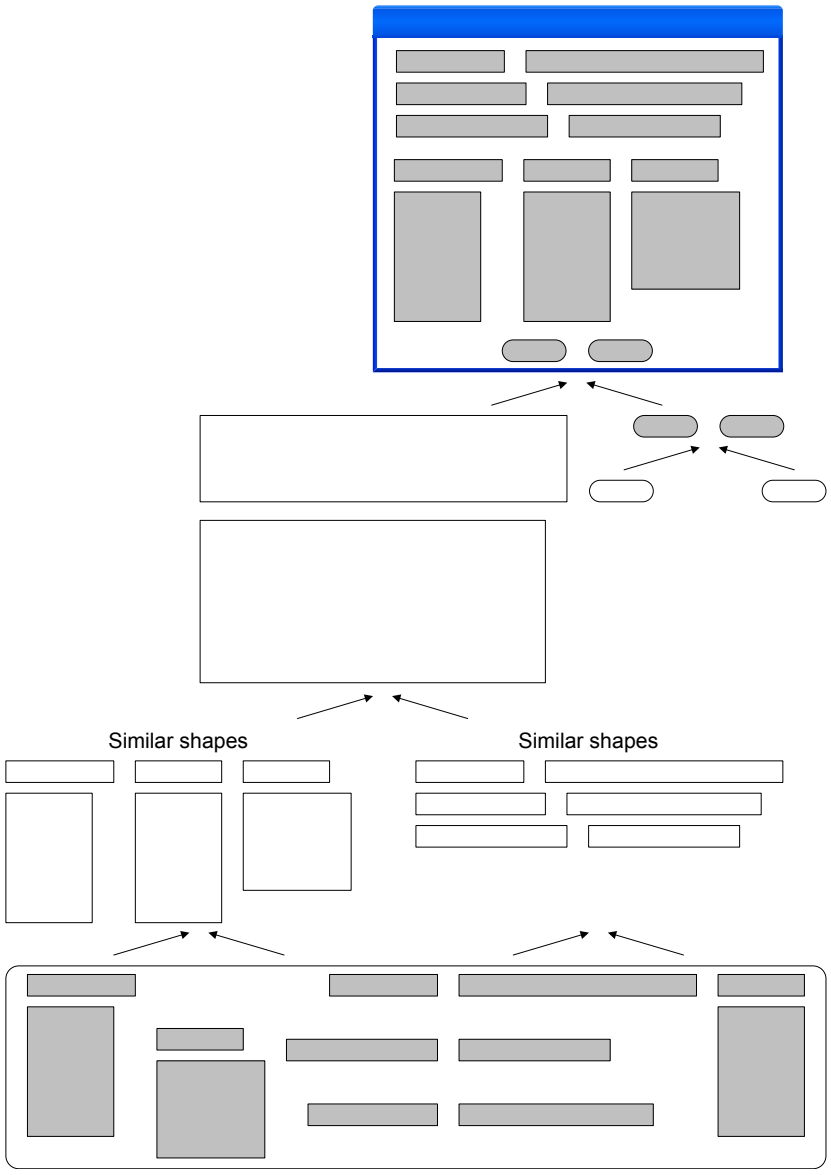


Figure 2.6: Placement process, working with the extended shape analysis (redrawn according to [KF93]).

Chapter 3

UCP – Unified Communication Platform

The UCP framework has the purpose to simplify the creation of applications. These applications can be dedicated to machine-machine and to human-machine interaction [Pop09]. It is a model-driven approach for generating applications tailored to various target devices (e.g. PC monitor, PDA, touch screen, etc.). A discourse model acts as underlying model. Based on human speech act theory, it describes the interaction between two communicating parties (human-machine as well as machine-machine) [KBFK08]. The UCP framework automatically transforms this discourse model into a structural representation of a graphical user interface (GUI). This GUI representation is an instance of the structural UI meta-model, which is described below. According to the chosen target device, the Layout Module, that is described in Chapter 4, assigns a layout to this structural GUI representation. If all relevant attributes of the structural representation of the GUI are set, a final GUI can be generated and displayed on the target devices screen.

Independent from the target device and the graphical toolkit used, a designer with limited programming skills can create various applications. Furthermore the designer does not have to deal with low-level machine-machine interaction protocols since the UCP framework takes over this responsibility. A created application does not differentiate whether it communicates with a human or another application. The possibility of creating systems for human-machine as well as for machine-machine interaction offers a wide field of application.

The current chapter outlines the basics of the UCP framework, that are needed to comprehend the Layout Module described in Chapter 4. It gives a short overview on the Discourse Model in Section 3.1 and on the Structural User Interface meta-model (Structural UI) in Section 3.2. The generation process of the Structural UI model out of the Discourse Model is described in Section 3.4. This model-to-model transformation is performed by matching certain patterns in the discourse model with a defined structure in the structural UI model using a predefined set of matching rules.

According to the current progress of the UCP framework, the definitions made in this chapter, and especially in Section 3.2, represent an update of the elaborations of [Ran08].

3.1 Discourse Model

The discourse model is a representation of the interaction of two parties (man-machine or machine-machine) on a high level of abstraction. It is based on human speech act theory and, therefore offers a far more intuitive alternative to common programming languages.

A detailed description of the discourse model and additionally on its underlying rhetorical structure theory (RST) is given in [Ran08]. The discourse model has low effect on the layout calculations. The only information that really concerns the layout calculations is the discourse relation to which certain structural UI elements are tracing back. The current section only gives an overview on the most important building blocks of the discourse model.

Kavaldjian et al. declare communicative acts to be the most important ingredients of the discourse [KRP+10]. Those communicative acts are derived from human speech acts and model an utterance of a single communication partner that can either be an application or a user. Certain pairs of communicative acts form so-called adjacency pairs, that define a predefined sequence of utterances (e.g. question and answer).

By relating adjacency pairs to each other, a tree structure can be built. The relations that are needed for this purpose are called discourse relations. They can either be procedural relations (Condition, IfUntil or Sequence) or relations that are derived from rhetorical structure theory (RST relations). There are two different types of RST relations:

- Symmetric (multi-nuclear) RST relations relate similar subtrees that have the same temporal priority. The symmetric RST relations are the Alternative, the Contrast and the Joint relation.
- Asymmetric (nucleus-satellite) RST relations relate subtrees of different temporal priority. The nucleus links the main objective and the satellite provides additional information, supporting the nucleus. Those asymmetric RST relations are the Background, the Elaboration and the Result relation, whereby an Elaboration relation can be specialized into an Annotation and a Title relation.

Regarding to the Layout Module it is important to know from which discourse relation a certain structural UI element was created. If a discourse relation relates the communicative acts to each other as it is done in case of an asymmetric RST relation, the resulting part of the structural UI model gets its layout from predefined rules during its generation process. In case of a symmetric RST relation, the resulting part of the structural UI model needs to be layouted by the Layout Module. To give an example, a procedural Sequence relation results in a structural UI container element with children that have a strict order. In comparison, a symmetric RST relation results in a structural UI container whose children do not have a predefined order. Therefore they can be reordered according to different strategies. The structural UI outcome of a Sequence has to be layouted as well, but a reordering is not allowed in this case.

Kavaldjian et al. present the generation process of a structural UI model [KRP+10]. For that purpose, they have chosen a small excerpt of a discourse model for flight booking to serve as an example (see Figure 3.1). The yellow (or light gray) communicative acts in this example are uttered by the application, whereas the green (or dark gray) ones define utterances of the

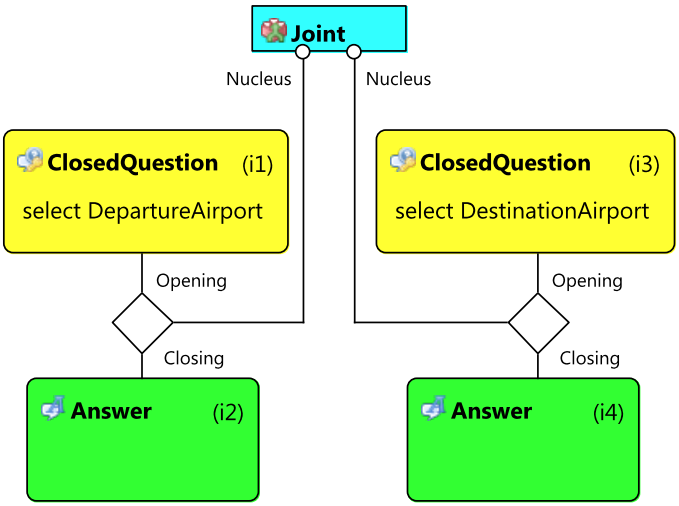


Figure 3.1: An excerpt of a FlightBooking discourse model [KRP+10].

customer. Each of the two ClosedQuestion-Answer pairs forms an adjacency pair, which is indicated by the diamonds in the diagram. Those adjacency pairs model the questions for a departure and a destination airport. Out of a list of available airports for each, the customer has to answer both questions. In this example, the two adjacency pairs are related to each other by the Joint relation. Being a symmetric RST relation, the Joint relation denotes that the adjacency pairs in its nucleus branches have the same priority and, therefore they are not ordered temporally. This means that the two ClosedQuestion-Answer pairs can be displayed in parallel or in sequence. Depending on the screen size, different rules determine the presentation of the two ClosedQuestion-Answer pairs as well as of the Joint relation, and the structural UI is created accordingly.

3.2 Structural User Interface Model

The UCP framework generates an abstract model that represents the structure and the content of the graphical user interface (GUI), that has to be created. This abstract model is called the structural UI model (see [Ran08]). Its generation is performed by matching certain patterns in the Discourse Model and mapping them to a defined structure in the structural UI model. It is represented by a tree structure of objects, that are instances of the abstract WIDGET class. If all attributes of the single Widgets are set, the structural UI model can be transformed to source-code, that creates a displayable GUI. Analogously to [Ran08], structural UI WIDGETS are written below in small capital letters to differentiate them from Java Swing widgets or widgets from other toolkits, that are just written in lower case.

The structural UI model serves as basis for the calculations provided by the Layout Module, which makes it worth mentioning here. In order to provide a structural UI model with a layout and to calculate the size of its FRAMES, the relevant layout and size attributes of its contained elements have to be calculated and set. Not all information provided by the structural UI model is used within the layout- and size calculations of the Layout Module.

Although not all information is relevant for the Layout Module, the currently existing WIDGET subclasses and their attributes are completely listed below. This is done with the intention of giving an overall description of the structural UI meta-model.

3.2.1 Widget Class

The definitions made in this section represent an update to [Ran08]. This is necessary because the structural UI meta-model has changed in the course of time and no current reference is available.

A WIDGET abstractly represents a single displayable element that can be found in a GUI. It does not provide any information about the toolkit that is used to create the resulting GUI. Due to this, it is possible to transform the structural UI model into source-code, using different toolkits (e.g. Web, Java Swing, etc.) [KBFK08]. For the time being, the size calculations that are necessary for the layout calculations described in Chapter 4, are tailored to Java Swing widgets only.

Since some of the WIDGET properties are used below, they are listed here:

visible **-** The value of this attribute defines whether the widget is initially visible or not.

style **-** Each widget can be assigned a certain style, which effects the way it is displayed. The style corresponds to an entry in a cascading style sheet (CSS), which is discussed in 3.3.

name **-** The name is used as a unique reference to the widget.

enabled **-** This property enables or disables the widget in the actual screen representation.

tracesTo **-** This field traces the widget back to a communicative act. This communicative act delivers the information needed to be displayed by the widget.

layoutData **-** Contains the data needed to set constraints imposed by a certain layout type (e.g. GridBagConstraints).

parent **-** In case the widget is contained by another widget, this field specifies the container widget.

content **-** This property field holds the data or a reference to the data that the widget should display.

contentSpecification **-** This property is just relevant for the mapping of the discourse model to the structural UI description. It specifies which data to put in the widget's content field by means of the Object Constraint Language(OCL).

text **-** If no content is specified, this field contains a static text to be displayed by the widget.

pointingGranularity **-** This attribute can either be set to FINE or COARSE. According to the chosen pointing granularity, the LISTWIDGET is rendered differently.

- **FINE** is selected if the user interface is operated using a mouse, or a pen on a touchscreen, for example.
- **COARSE** is selected if the user operates in a less exact way (e.g. finger on a touch screen).

width **-** Defines the widgets width in number of pixels.

height **-** Defines the widgets height in number of pixels.

getFrame **-** This method returns the FRAME, that contains the current WIDGET. to a window.

As depicted in Figure 3.2, the WIDGET class is specialized in either the PANEL, the INPUTWIDGET or the OUTPUTWIDGET subclass. These three subclasses inherit all characteristics from the WIDGET class and are further specialized in various other subclasses themselves. Unlike the OUTPUTWIDGET, that has the single purpose of presenting information, the INPUTWIDGET gathers it from the user. To give a brief overview, the three WIDGET subclasses as well as their according parts of the class diagram are described in the following subsections.

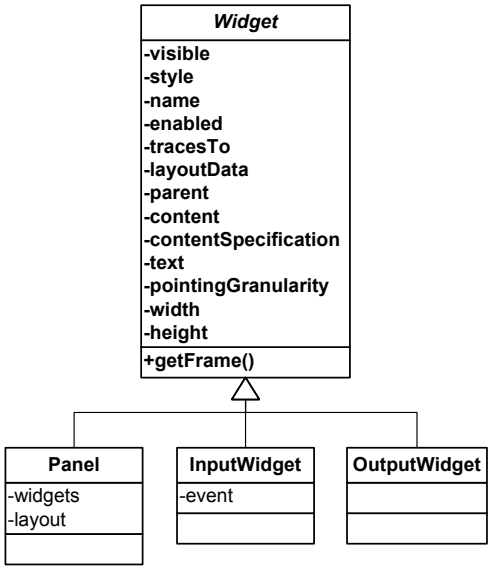


Figure 3.2: Widget class and its specializations.

3.2.1.1 Panel Class

The PANEL class is shown in the class diagram of Figure 3.3. It inherits its characteristics from the WIDGET class. It serves as a container for any other kind of WIDGET. Therefore, it extends the WIDGET class by introducing the attributes *widgets* and *layout*:

widgets **-** This property hosts the WIDGET objects, that are contained by the PANEL.

layout **-** Defines the layout-manager that is used to arrange the WIDGETS contained by the PANEL.

Furthermore, the PANEL class serves as a base class for other classes that contain WIDGET objects (e.g. LISTWIDGET).

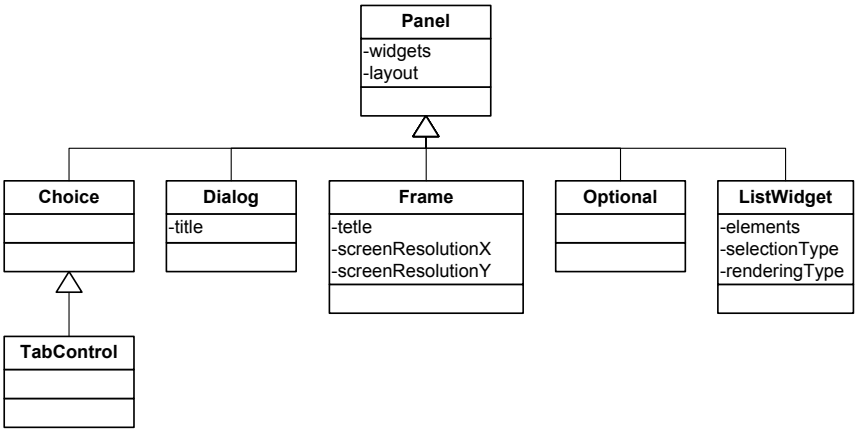


Figure 3.3: Panel class and its specializations.

Choice Class

This class offers the opportunity to choose exactly one out of its child WIDGETS. A CHOICE is the root of any structural UI model. In case of being the root of a structural UI model, the CHOICE's children are the windows to be generated (i.e. FRAMES). It may, of course also appear somewhere within the tree structure of the structural UI model. CHOICE is derived from PANEL. As PANEL is derived from WIDGET, this class inherits a WIDGET's characteristics as well.

TabControl Class

Being a specialization of a CHOICE, theTABCONTROL offers the possibility to the user to switch between different PANELS within the same screen. Concerning the inherited characteristics it is equal to CHOICE, differing only in the code that needs to be generated.

Dialog Class

This class represents a modal window. It extends the PANEL class by introducing the *title* attribute:

title **-** Defines the title text of the window.

Frame Class

The FRAME class represents a window of the generated application. More than one window can exist in a single application and, therefore in the corresponding structural UI model. FRAME can only exist directly below the root CHOICE of the structural UI model. A FRAME extends the PANEL class by introducing the following attributes:

***title* -** Defines the title text of the window.

***screenResolutionX* -** This property defines the x resolution of the window (in pixels).

***screenResolutionY* -** This property defines the y resolution of the window (in pixels).

Optional Class

This class offers the opportunity to choose any number of its child WIDGETS to be displayed at the same time.

ListWidget Class

The LISTWIDGET class represents a recurring set of widgets. It inherits all attributes from the PANEL class and from the INPUTWIDGET class. The LISTWIDGET class introduces the following attributes:

***elements* -** A reference to the elements that are contained by the LISTWIDGET.

***selectionType* -** This property declares, whether the LISTWIDGET provides single or multiple selection.

***renderingType* -** This property defines, how the LISTWIDGET is rendered. There are three different possibilities:

- If this property is set to **FOLDOUT**, the LISTWIDGET is displayed as a drop down widget. It can optionally have a submit button, and provides single selection only.
- If this property is set to **PANEL**, the LISTWIDGET is displayed as a panel that contains all element information. Each entry requires an INPUTWIDGET (e.g. a RadioButton for single selection, CheckBox for multiple selection).
- If this property is set to **LIST**, the LISTWIDGET is displayed as scrollable list. The single entries do not require an INPUTWIDGET, and single selection as well as multiple selection is provided.

3.2.1.2 InputWidget Class

The INPUTWIDGET class represents the counterpart to OUTPUTWIDGET, which is described below. Furthermore, it serves as a base class for all WIDGETS, that are used to collect information from the user. It extends the WIDGET class by introducing the property *event*, and can be specialized as shown in the class diagram of Figure 3.4.

- ***event*** - This property holds the specification of what needs to be done with the collected data.

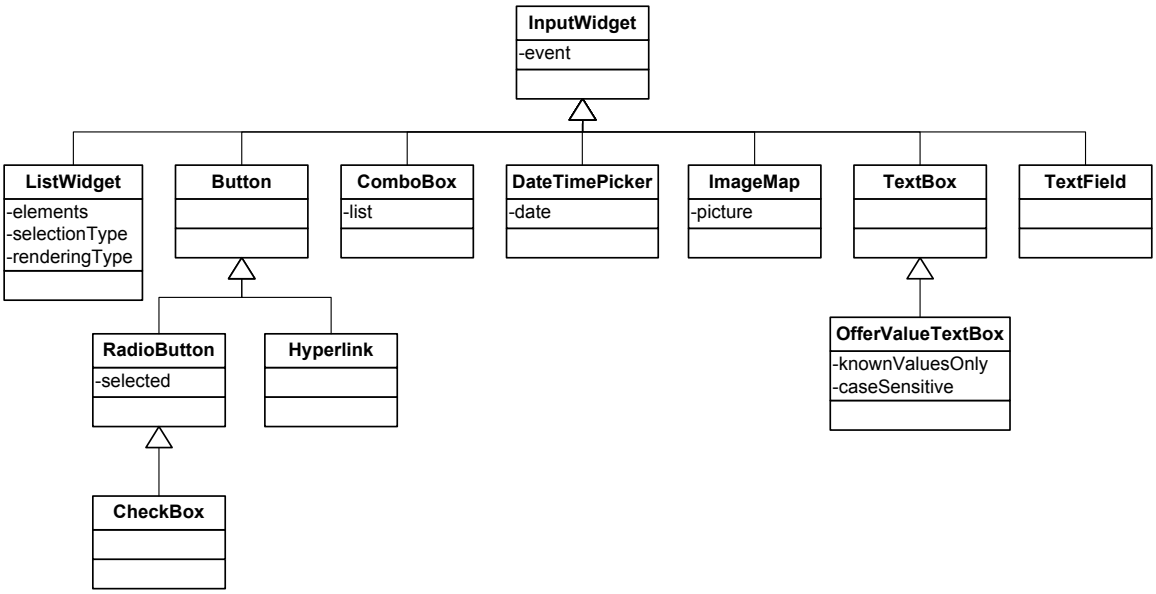


Figure 3.4: InputWidget class and its specializations.

ListWidget Class

The ListWidget also inherits all attributes from InputWidget. A detailed description can be found in 3.2.1.1.

Button Class

The Button class defines the abstract representation of a button widget and is derived from InputWidget.

RadioButton Class

The RadioButton represents a single selectable option in a list of two or more options that exclude each other. It inherits all its characteristics from the Button class and further extends it by introducing the *selected* attribute:

- ***selected*** - This property declares, whether the RadioButton is selected or not.

CheckBox Class

The CheckBox allows the user to set one or more options. It inherits all its characteristics from the RadioButton class.

Hyperlink Class

The Hyperlink is the abstract representation of a hyperlink and inherits all its characteristics from Button.

ComboBox Class

A ComboBox Widget represents the abstract class of a widget that allows the user to choose one item from an offered list. It extends the InputWidget class introducing the property *list*:

- ***list*** **-** This property contains the items to be added to the list that is offered to the user.

DateTimePicker Class

This class represents a widget that allows the user to choose a certain date as well as a certain time. It extends the InputWidget class by introducing the *date* attribute:

- ***date*** **-** This property declares the selected date and time.

ImageMap Class

This class belongs to the category InputWidget and is, therefore derived from this class. It represents a graphic, on which different sections can serve as a hyperlink. It extends the InputWidget class by introducing the *picture* attribute:

- ***picture*** **-** This property field carries the URL of the picture to be used.

TextBox Class

This class represents a user input field.

OfferValuesTextBox Class

This class is a representation of a user input field, that provides the user with a proposal of known input values to choose, while typing (auto-complete). It extends the TextBox class by introducing the following Attributes:

knownValuesOnly **-** This property declares, if only already known values are accepted as valid input, or not.

caseSensitive **-** This property declares, whether case sensitivity is activated, or not.

TextField Class

This class represents a user input field that accepts more than one line.

3.2.1.3 OutputWidget Class

The OutputWidget is derived directly from the Widget class and, therefore inherits all its characteristics. It serves as a base class for all widgets that have the purpose to display information to the user. The OutputWidget class can be specialized as shown in the class diagram of Figure 3.5.

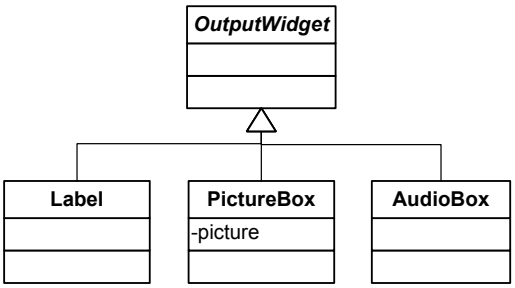


Figure 3.5: OutputWidget class and its specializations.

Label Class

This class is an abstract representation of the label widget. As it can only be used to display information, it is derived from the abstract OutputWidget class.

PictureBox Class

This class represents the abstract Widget used for the illustration of a picture. It extends the OutputWidget class by introducing the *picture* attribute:

- ***picture*** **-** this property field carries the URL of the picture to be used.

AudioBox Class

This class is an abstract representation of a simple audio player.

3.2.2 LayoutManager Class

In order to arrange the widgets that are contained by panels, a LayoutManager is assigned to each Panel. According to the Panels's LayoutManager, a certain kind of LayoutData is assigned to each contained Widget. There are three types of LayoutManager, that differ in the way of arranging widgets.

FlowLayout Class

The FlowLayout class is the abstract representation of the FlowLayout-Manager provided by Java. It arranges the widgets side by side. If there is not enough space left up to the right border of the container, the FlowLayout-manager starts with a new row. No additional data and, therefore no LayoutData object is needed.

XYLayout Class

The XYLayout class is the abstract representation of the XYLayout-Manager provided by Java. It offers the possibility to the designer to choose absolute coordinates, where to place the upper left corner of the widget. Additionally, a constant height and width can be assigned to the widget. To arrange the widgets correctly, an XYLayoutData object is assigned to each Widget that has to be arranged. An XYLayoutData object contains the following attributes:

- ***x*** - This attribute specifies the absolute x coordinate of the upper left corner of the widget (in pixels).
- ***y*** - This attribute specifies the absolute y coordinate of the upper left corner of the widget (in pixels).
- ***width*** - This attribute specifies the width of the widget.
- ***height*** - This attribute specifies the height of the widget.

GridLayout Class

The GridLayout class is the abstract representation of the GridBagLayout-Manager provided by Java. By setting the various LayoutData properties, widgets can be arranged in a dynamic grid. Therefore, a GridLayoutData object is assigned to each Widget, that has to be arranged. A GridLayoutData object contains the following attributes:

- ***row*** - Starting with zero as the uppermost row, this attribute specifies the row in the grid where the widget has to be placed.
- ***col*** - Starting with zero as the leftmost column, this attribute specifies the column in the grid where the widget has to be placed.
- ***rowSpan*** - This attribute specifies the amount of rows a single widget should cover. This value is set to one by default.
- ***colSpan*** - This attribute specifies the amount of columns a single widget should cover. This value is set to one by default.
- ***alignment*** - This attribute is used to determine, where the component should be displayed, if the area provided for this component is bigger than its actual size (e.g. CENTER, NORTH, EAST, etc.).
- ***weightx*** - The GridLayout-manager offers to influence the distribution of extra space. This attribute influences the distribution of extra space on the x-axis.
- ***weighty*** - The GridLayout-manager offers to influence the distribution of extra space. This attribute influences the distribution of extra space on the y-axis.
- ***fill*** - Apart from alignment, this value is used, if the display area is larger than the component's size. It determines, whether the remaining space should be filled or not and if so, in which way. Its value is defined by constants, and the possible selections are HORIZONTAL, VERTICAL, BOTH and NONE.

A well-defined grid, with equal sized cells, can be achieved with the aid of the following GridLayout attributes:

- ***rows*** - Defines the number of rows within the grid.
- ***cols*** - Defines the number of columns within the grid.
- ***rowHeight*** - Defines the number of pixels that define a row's height.
- ***colWidth*** - Defines the number of pixels that define a columns's height.

3.2.3 The Structural UI Tree

As depicted in Figure 3.6, the tree structure generally consists of a single Choice at its root. One level below that root element, at least one Frame is placed. Thus, the first two levels of a structural UI model are preserved for the Choice of a set of Frames. One level below the Frames, a Widget of any type can be placed, except the Frame element, which is just allowed one level below the root Choice. Furthermore, the number of Widgets that is contained by a single container Panel is not limited.

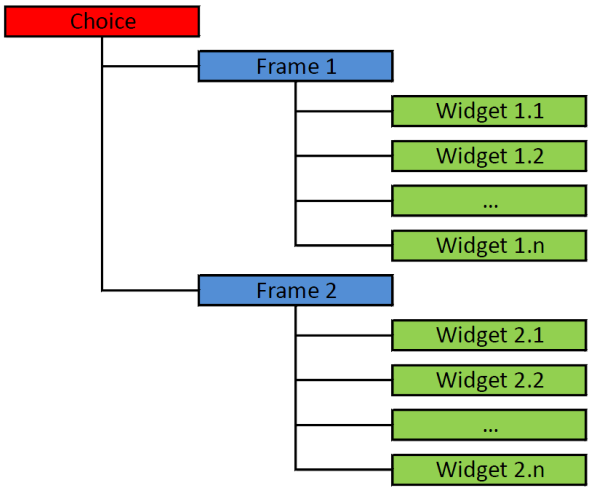


Figure 3.6: The structural UI tree.

After the generation of the structural UI model, the LayoutData of all Widgets is set in two different ways:

1. A Panel that contains at least one InputWidget or OutputWidget, has valid layouted children. The rule that transforms the corresponding part of the discourse model into the structural UI Panel and its children, contains valid LayoutData of the children as well. The Panel's children are arranged accordingly.
2. If there are only Panels inserted into a container Panel, there is no valid LayoutData set for these Panels. The corresponding rule does not contain valid LayoutData, because there is no information about the dimension of each inserted child Panel. Initially, the LayoutData for those child Panels are set to their default, resulting in

PANELS that are placed in the upper left corner of the container PANEL in an overlapping manner. LAYOUTDATA of those child PANELS has to be set afterwards. This is done with the aid of the Layout Module, which is discussed in Chapter 4 in detail.

3.3 Cascading Style Sheets

A range of styles are necessary to display the widgets on the screen in an appropriate way. A detailed description about the styles that can be set in a Cascading Style Sheet, can be found at http://www.w3.org/TR/DOM-Level-2-Style/css.html (March 2010). Moreover, they are needed to calculate the size of each single widget whether it is a panel or not (see Section 4.4).

The default size of each WIDGET can be defined in a Cascading Style Sheet (CSS). It can contain a WIDGET'S width and height as well as some other style data (e.g. font-name and font-style, border-thickness and padding, etc.). For certain applications it might be necessary to generate a number of GUIs for multiple target devices with different screen-resolution and screen size (e.g. PC monitor, PDA, touch screen, etc.). Therefore, it is most helpful to define a Cascading Style Sheet for each target device. These Cascading Style Sheets provide different styles for each WIDGET. This style can be assigned to a WIDGET in three ways, that differ in their scope by referencing:

1. Referenced by the WIDGET's unique name attribute.
2. Referenced by the WIDGET's style attribute.
3. Referenced by the WIDGET's type.

If a style is assigned by more than one of the enumerated references, the style attributes (e.g. text-size, padding, etc.) are combined with respect to the hierarchy of the enumeration. In other words, style attributes are summarized and, in case of being assigned twice, e.g. by WIDGET-name and WIDGET-style, the attribute with the higher priority is chosen.

3.4 Discourse Model to Structural UI Model Transformation Process

The basic transformation process is described in [KBFK08]. It results in a structural UI model, where all WIDGETS are arranged below each other. Introducing the Layout Module, this process can be optimised as specified in [KRP+10].

The Layout Module sets the LAYOUTDATA of all structural UI WIDGETS that do not have valid LAYOUTDATA, and simultaneously calculates the size of each contained WIDGET. As the size calculation concerns the FRAMES as well, information about the amount of screen space that would be necessary to display all FRAMES of a structural UI model, is available. Subsequently, it can be checked, if the structural UI model's FRAMES fit into a certain target device's screen size.

Since the Layout Module only plays a role within the optimised transformation process, the basic transformation process is not described in this section. Figure 3.7 illustrates this optimised transformation process. First, the discourse model is transformed to a structural UI model. This is done by matching certain patterns within the discourse model with a defined structure in the structural UI model, using a predefined set of transformation rules. The resulting structural UI model is layouted with the aid of the Layout Module and, thus it can be evaluated in terms of size. If the layouted structural UI model does not fit into the target device's screen size, the discourse model is transformed once again, by use of another set of transformation rules, that results in a smaller structural UI model. More precisely, a single rule is exchanged by a smaller counterpart at each iteration. Again, it is checked if the resulting structural UI model fits into the target device's screen size. This procedure is repeated until there are no more alternative rules left and, therefore no resulting structural UI model can be created for the chosen target device. As soon as a structural UI model is created that conforms to the target device's screen size, it is further transformed to a final UI and the transformation process ends successfully. If existing, this final UI represents the largest outcome that is possible for a given target device.

If the transformation process does not succeed in creating a fitting structural UI model, the smallest created structural UI model is taken. This structural UI model represents the outcome that requires a minimum amount of extra space. It exceeds the limits of the target device's screen and scrolling has to be enabled.

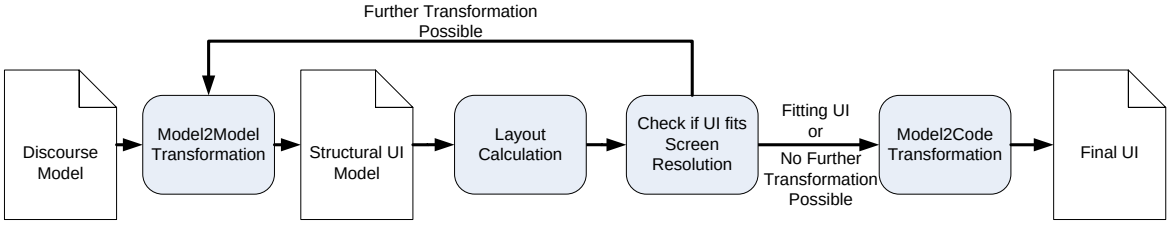


Figure 3.7: Transformation process.

At each iteration step, the set of transformation rules is changed, considering the following requirements:

- The resulting structural UI model should use as much of the available screen space as possible.
- The amount of navigation clicks should be held at a minimum.
- Scrolling should be avoided.

In [KRP+10], the discourse model shown in Figure 3.1 is transformed into a structural UI model for three different screen sizes. The rules that are necessary for this transformation process, are listed below:

- *Closed Question Rule*
  This is the standard transformation rule for a ClosedQuestion-Answer adjacency pair. It results in a structural UI Panel with layouted child Widgets. It contains a title

LABEL, a set of RADIOBUTTONS, that represent the content of the ClosedQuestion communicative act, and a BUTTON to submit the chosen RADIOBUTTON entry (see Figure 3.8(a)).

- *Small Closed Question Rule*
  This rule results in a smaller outcome of a ClosedQuestion-Answer adjacency pair. It results in a structural UI PANEL with layouted child WIDGETS as well. In contrast to the standard *Closed Question Rule*, it contains a title LABEL, a COMBOBOX, that represents the content of the ClosedQuestion communicative act, and a BUTTON to submit the chosen COMBOBOX entry (see Figure 3.8(b) and 3.8(c)).

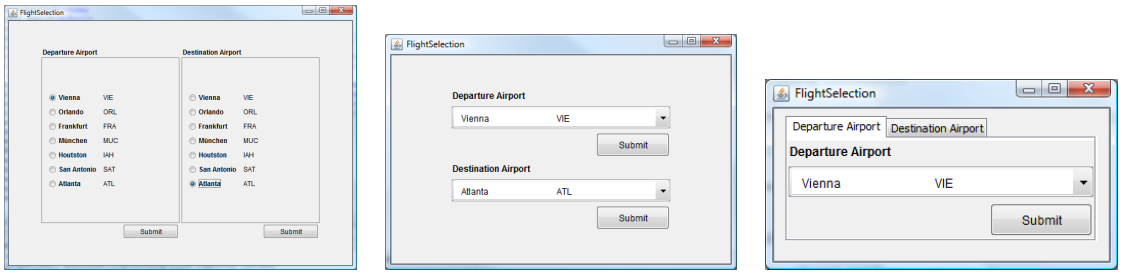
- *Joint Rule*
  This is the standard transformation rule for a Joint relation. The Joint relation links adjacency pairs that have the same temporal priority. It results in a structural UI PANEL that contains a set of child PANELS without valid LAYOUTDATA (see Figure 3.8(a) and 3.8(b)).

- *Small Joint Rule*
  This rule results in a smaller outcome than the standard *Joint Rule*. A TABBEDPANE is created as container instead of a PANEL. This TABBEDPANE contains a set of child PANELS, that do not have valid LAYOUTDATA (see Figure 3.8(c)).

In case of a screen size of 640 x 480, the transformation process starts with the *Closed Question Rule* and the *Joint Rule*. The Layout Module is able to arrange the two PANELS that are created by each *Closed Question Rule* next to each other, resulting in a layouted structural UI model, that fits into the given screen size. The transformation process ends, and the layouted structural UI model is further transformed into the GUI that is shown in Figure 3.8(a).

A screen size of 480 x 320 does not provide enough space for a valid outcome of the standard rules (*Closed Question Rule* and *Joint Rule*). Therefore, in the second iteration step of the transformation process, the *Small Closed Question Rule* is tried out instead of the *Closed Question Rule*. This modification results in a fitting structural UI model. The Layout Module arranges the two PANELS that are created by each *Small Closed Question Rule* below each other. The transformation process ends after this iteration. The resulting GUI is shown in Figure 3.8(b).

In case of a screen size of 320 x 180, the *Small Closed Question Rule* and the *Joint Rule* do not result in a fitting structural UI model. Subsequently, further transformation rule-sets have to be tried out. Finally, a fitting structural UI model can only be created by using the *Small Closed Question Rule* in conjunction with the *Small Joint Rule*. The corresponding GUI that is created from the generated structural UI model is shown in Figure 3.8(c).



(a) 640x480 (b) 480x320 (c) 320x180

Figure 3.8: Generated User Interfaces [KRP+10].

Chapter 4

Layout Module

The main issue when arranging WIDGETs on a screen with a predefined screen size is the simple question of how to place them. Does there exist any information about how different WIDGETs belong to each other by context and, therefore have to be placed closer together? Is it possible to fulfill any constraints concerning user interface design (see 2.1)? Is there a possibility to minimize the space needed?

WIDGETS that are conceptually related are arranged close together, since they are nested in the same structural UI PANEL. Subsequently, the hierarchy of the structural UI model determines the relationship between WIGETS for layouting, and a further evaluation concerning this task is not required.

The INPUTWIDGETS and OUTPUTWIDGETS themselves already get their layout during the generation process of the structural UI model mentioned in Section 3.4, using a predefined set of transformation rules. The same applies to PANELS that are nested together with INPUTWIDGETS or OUTPUTWIDGETS. Consequently, there is no need to decide where to place those widgets (e.g. a Button, Label, etc.), but still their size attributes have to be calculated and set (see Section 4.4). Constraints concerning user interface design can hardly be considered here, since INPUTWIDGETS and OUTPUTWIDGETS are already layouted within their parent PANEL. Layout calculation for a PANEL'S children is only required, if those children are all PANELS (e.g. CHOICE, OPTIONAL, etc.). However, optimisations concerning these constraints can be considered in future work as described in Chapter 5.

The Layout Module aims to calculate the size of all WIDGETS contained by a structural UI model and to calculate LAYOUTDATA of WIDGETS, that do not have valid LAYOUTDATA set after the transformation process. After its generation, a structural UI model contains a set of WIDGETS, whose size attributes are not set a priori. Subsequently, a size calculation is necessary for all WIDGETS to allow further layout calculations and to display the WIDGETS properly on the screen. As already said, the structural UI model may contain PANELS that have already layouted child WIDGETS. The corresponding LAYOUTDATA of the WIDGETS is set. WIDGETS that do not have valid LAYOUTDATA need to be layouted explicitly.

For the time being, the Layout Module is tailored to Java Swing widgets. If an application has to be generated for another toolkit, the size calculation needs to be extended accordingly.

To define size and format attributes of WIDGETS, Cascading Style Sheets are used, which can be edited by the designer. The Cascading Style Sheets may contain metric values such as

millimeter or centimeter values as well. When using such values, the designer does not need to make calculations concerning the screen resolution of different target devices when editing the Cascading Style Sheets. Furthermore, a single Cascading Style Sheet is needed for target devices of different screen resolutions. Nevertheless, in Java Swing a widget's size as well as other dimensions need to be set using pixel values and a conversion of possibly existing metric values in the Cascading Style Sheets is required. The StyleSheetConverter, which is a part of the Layout Module, converts the Cascading Style Sheets resulting in a converted Cascading Style Sheet that only contains pixel values. This conversion is described in detail in Section 4.3.

After the generation of the structural UI model, the contained WIDGETS do not have valid size attributes. Since no LAYOUTDATA can be calculated without knowing the size of each WIDGET, the size calculation for each WIDGET is one of the most important tasks provided by the Layout Module. Furthermore, a PANEL must be large enough for being able to display its child WIDGETS properly. The size calculation is described in detail in Section 4.4.

Section 4.5 describes the strategy of arranging the WIDGETS. In order to make this task computable, a grid with equal sized cells is created for each PANEL whose children need to be layouted. Step by step, all child WIDGETS are inserted into this grid. All inserted WIDGETS are marked in a model of the corresponding grid that is represented by a two-dimensional integer array. This enables a search for specific insertion points and their evaluation for further WIDGET insertion.

The calculation of the LAYOUTDATA that is assigned to each WIDGET is described in Section 4.6.

Finally, in Section 4.7 some structural UI model examples are used to illustrate the results that can be achieved with the aid of the Layout Module.

The current chapter starts with a description of the functionality of the Layout Module in Section 4.1 followed by its integration into the UCP framework in Section 4.2.

4.1 Integrated Size Calculation and Layouting Algorithm

The purpose of the Layout Module is the size calculation for each WIDGET and, if needed, the layout calculation of the WIDGETS that do not contain valid LAYOUTDATA. Furthermore, it is required to layout WIDGETS in a space-saving manner. The number of WIDGETS where LAYOUTDATA has to be set by the Layout Module, depends on the specific structural UI tree.

To give an example, the structural UI model sketched in Figure 4.1 is traversed recursively, bottom up (post-orderly). Table 4.1 lists the calculations that are made at each recursion step. INPUTWIDGETS and OUTPUTWIDGETS are merged here to IOWIDGETS in order to simplify the following description.

The algorithm starts with the size calculation of IOWIDGET 6, IOWIDGET 5 and IOWIDGET 4 at recursion steps one to three. Afterwards, at recursion step four, PANEL 3 is encountered. Its children already have valid size attributes as well as LAYOUTDATA. Using those data, the size of PANEL 3 can now be calculated. At the next three recursion steps, the Dimensions of IOWIDGET 3, IOWIDGET 2 and IOWIDGET 1 are calculated. Again, LAYOUTDATA of

these IOWidgets is already set. At recursion step eight, the size of Panel 2 is calculated according to its children. At step nine it is required to calculate the LayoutData of Panel 2 and Panel 3, before a size calculation for the encountered Panel 1 can start. Finally at recursion step ten, Panel 1 gets its LayoutData and the size of Frame 1 is calculated and set.

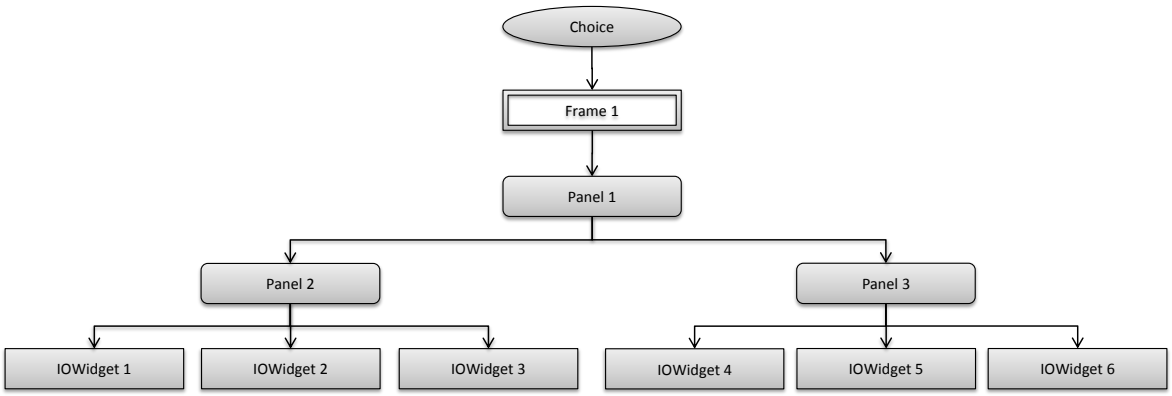


Figure 4.1: Structural UI tree.

Table 4.1: Structural UI calculation order.

Recursion step	Widget	Size calculation	Calculation of layout-data
1	IOWidget6	x	
2	IOWidget5	x	
3	IOWidget4	x	
4	Panel3	x	
5	IOWidget3	x	
6	IOWidget2	x	
7	IOWidget1	x	
8	Panel2	x	
9	Panel2		x
	Panel3		x
	Panel1	x	
10	Panel1		x
	Frame1	x	

The calculations can only succeed by following the constraints itemized below:

- The width and height attributes of each Widget must be calculated. The calculation of the size of a Widget is described in detail in Section 4.4.
- The size of each Panel can only be calculated if the width and height attributes of its contained Widgets as well as their LayoutData are already set. Therefore the structural UI model needs to be traversed recursively, bottom up. The calculation of the LayoutData is described in detail in Section 4.5.

As mentioned in the second item, the structural UI model has to be traversed recursively, bottom up. At each recursion step, the WIDGET type is checked. Encountering an INPUTWIDGET or an OUTPUTWIDGET, the layout engine assigns a width and a height to it. Encountering a PANEL the Layout Module prior sets the LAYOUTDATA of its children if needed. Afterwards it assigns a width and a height to the PANEL itself, considering padding and border-width of each child.

To set LAYOUTDATA on PANELS, and thus give them an order within its parent PANEL, the constraints itemized below are necessary to make decisions concerning the arrangement:

- The amount of free space that is wasted due to a single WIDGET insertion should be held at a minimum. Therefore it is necessary to try each possible insertion point, choosing the one that fits best (see 4.5.2).
- The frame width may not be exceeded when putting panels side by side.
- There is only one special case when this frame width limit can and must be exceeded. That is the case when at least one element in the structural UI model is broader than the given screen width itself.

The third item results in a layouted structural UI model, that does not fit into the target devices screen. The screen width that would be necessary to display all widgets is finally set in the structural UI FRAME attribute *width* to indicate this circumstance. A further discourse model to structural UI transformation is required or scrolling has to be enabled (see Section 3.4).

The data needed to calculate a WIDGET'S size is generally obtained by a converted Cascading Style Sheet (see Section 4.3). This converted Cascading Style Sheet does not necessarily need to contain all data required for the calculation. Therefore, the missing data has to be obtained by another source, to provide a successful size calculation and consequently a valid result of the layout algorithm. In this case, the alternative source is represented by a properties-file. Concerning the size calculation, it functions as some kind of fail-safe mechanism and provides the layout algorithm with the appropriate data.

Concerning the Layout Module, there are two different kinds of PANELS. They differ in the order of their children, which can be mandatory or not. The Layout Module treats the two PANEL types differently:

- The order of a PANEL'S children is relevant and, therefore mandatory, if the PANEL traces to the procedural Sequence relation (ProceduralRelation) or if the PANEL is an instance of a LISTWIDGET. A reordering of the child PANELS is not allowed in this case. Each child WIDGET has to be placed side by side, starting a new free line if there is not enough free space left to the end of the screen width. In case of a LISTWIDGET, a new free line is started if the given width does not provide sufficient space to place the next child WIDGET in the current row. Choosing the insertion point that leads towards a minimum space consumption is not possible due to the mandatory order of the child WIDGETS. Furthermore, the LISTWIDGET represents an exception. Its children are included in the layout process although they are no PANELS. They are treated like children of a PANEL that traces to a Sequence. The choice of insertion points is performed as described in 4.5.3.1.

- The order of a PANEL's children is not relevant, if the PANEL traces to an RSTMultiNucleusRelation. In this case, the child PANELS are reordered either by their size or their width considering padding and border-width at each side. This is done by using the common known Bubble Sort algorithm, which is sufficient in this case due to the limited amount of child PANELS. Whether the child PANELS are ordered by size or width and whether it is started with the largest or smallest PANEL is declared in the properties-file. The ordering ensures that similar sized PANELS are close together in the insertion queue. Consequently, free space can be saved more easily. The choice of insertion points is performed as described in 4.5.3.2.

If a PANEL is encountered that cannot be described by the two itemized cases, no calculation of LAYOUTDATA is required. This is the case if the PANEL traces to any other DiscourseRelation. Nevertheless it is necessary to calculate the size of such PANELS.

4.2 Integration of the Layout Module

The Layout Module is called by the UCP framework after each generation of a structural UI model (see the box named Layout Calculation in Figure 3.7). A generated structural UI model is layouted by the Layout Module for a certain target device. For that purpose, all LAYOUTDATA and WIDGET dimensions are set by the Layout Module. Afterwards, the generation process checks if the layouted FRAMES fit into the target device's screen. Width and height of the FRAMES are compared to the predefined screen size of the target device. If the size of a FRAME exceeds the predefined screen size, the generation process generates another structural UI model using the same Discourse Model as used before. The generation is now performed with another set of rules that result in a structural UI model with smaller WIDGETS or split screens. The generation process of a structural UI model is described in Section 3.4 in more detail.

To deliver appropriate LAYOUTDATA as well as WIDGET dimensions, the Layout Module requires a set of data for its calculations:

- **Screen Resolution:** The DPI of the target device's screen.
- **Screen Size:** The width and height of the target device's screen.
- **Cascading Style Sheet:** Contains styles and default sizes for WIDGETS.

Since a single pixel can be of different size depending on the target device's screen resolution, the Layout Module allows the designer to define distances like width and height in a Cascading Style Sheets by using metric values. The use of metric values has the effect that a single Cascading Style Sheet can be used for different target devices independently from their screen resolution. Nevertheless, these distances have to be set in the WIDGET'S attributes as pixel values and, therefore have to be converted appropriately. As depicted in Figure 4.2, the Layout Module primarily converts all metric values that occur in the Cascading Style Sheet into pixel values. The StyleSheetConverter described in Section 4.3 is responsible for this conversion. The designer does not have to care about the target device's specification when editing the Cascading Style Sheet.

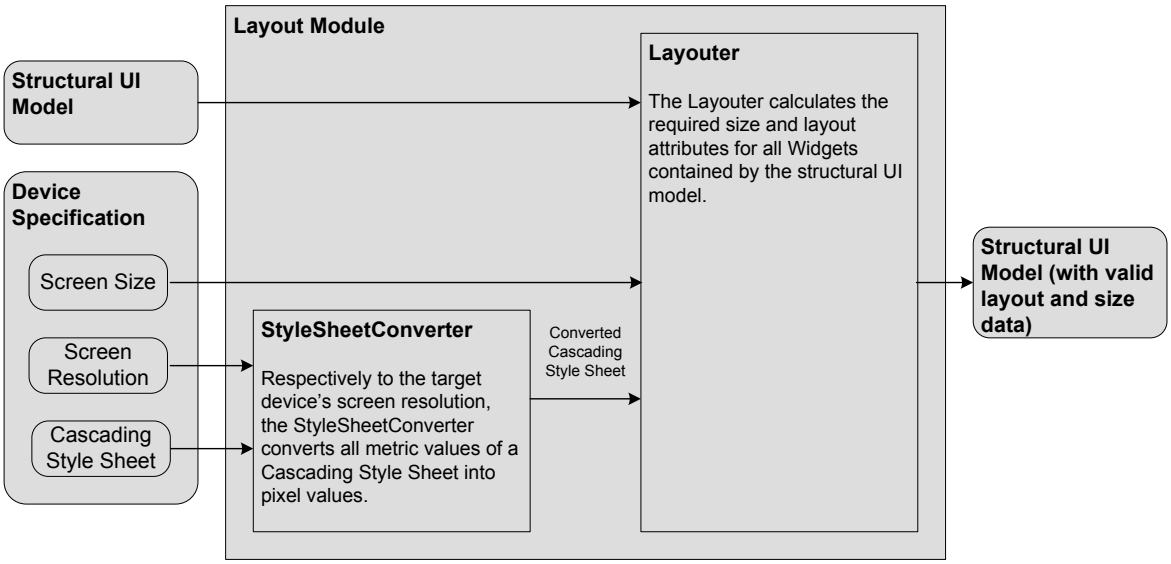


Figure 4.2: The Layout Module.

The Layout Module primarily uses the StyleSheetConverter to ensure that the Cascading Style Sheet contains only valid pixel data. Afterwards, the Layouter calculates LayoutData as well as size attributes for all Widgets contained by the structural UI model.

The functionality of the two mentioned classes is explained more precisely below.

4.3 StyleSheetConverter

The values required to set a Widget's style are generally stored in a Cascading Style Sheet. This style values are mostly given in unfavourable pixel values. The problem is, that target devices can have different resolutions. The target devices' resolution, given in Dots Per Inch (DPI), determines the dimension of its pixel. To give an example, 100 pixels correspond to 2.65 cm in case of 96 dpi and to 3.53 cm in case of 72 dpi. This can easily be derived by using Equation 4.1. The factor 2.54 is needed to transform inches into centimetres.

$$value_{cm} = \frac{value_{px} * 2.54}{DPI} \tag{4.1}$$

Vice versa, a certain length given in centimetres corresponds to a different number of pixels, that depends on the target devices' screen resolution. Consequently, if an application has to be rendered for different screen resolutions, a separate Cascading Style Sheet is required for each screen resolution.

If the screen resolution of the target device is known a priori, it is possible to convert values of different units (e.g. mm, cm, in, pt, etc.) to pixel values. Subsequently, the width and height of a widget, for example, can be set by using centimeter values in the corresponding Cascading Style Sheet. Therefore, the widget is displayed at exactly the same size on any target device, independent of the target device's screen resolution. In contrast, a pixel value in the Cascading Style Sheet would lead to differently sized widgets on target devices that vary in their screen resolution.

The Equations 4.2, 4.3, 4.4 and 4.5 provide a conversion of the most relevant units into pixel values. To guarantee as much pixels as required for each widget, the calculated pixel values are rounded up to the next higher integer value.

- Millimetre → Pixel:
$$value_{px} = \left\lceil \frac{value_{mm} * DPI}{25.4} \right\rceil \tag{4.2}$$
- Centimetre → Pixel:
$$value_{px} = \left\lceil \frac{value_{cm} * DPI}{2.54} \right\rceil \tag{4.3}$$
- Inch → Pixel:
$$value_{px} = \lceil value_{in} * DPI \rceil \tag{4.4}$$
- Point → Pixel:
$$value_{px} = \left\lceil \frac{value_{pt} * DPI}{72} \right\rceil \tag{4.5}$$

Using those equations, the StyleSheetConverter converts the relevant values into pixel values. The style attributes that have to be converted into pixel values by the StyleSheetConverter are itemized below:

- **width**: Defines the widget's width.
- **height**: Defines the widget's height.
- **padding**: Defines the number of pixels that have to be left free at each side of the widget.
- **padding-top**: Defines the number of pixels that have to be left free above the widget.
- **padding-bottom**: Defines the number of pixels that have to be left free below the widget.
- **padding-left**: Defines the number of pixels that have to be left free at the left of the widget.
- **padding-right**: Defines the number of pixels that have to be left free at the right of the widget.
- **border-width**: Defines the number of pixels that are reserved for the widget's border at each side. It can be chosen out of three different border-widths, that are THIN, MEDIUM and THICK. Their corresponding number of pixels can be derived from the properties-file.

The functionality of the Layout Module allows the designer to edit metric values, such as centimetre or millimetre values, when creating a Cascading Style Sheet. According to the screen resolution of the target device, an application is rendered for, the StyleSheetConverter converts those metric values to pixel values. Consequently, a single Cascading Style Sheet can be used to define widget styles for target devices of the same type (PDA, Touch Screen or PC Monitor) but with different screen resolutions.

4.4 Size Calculation

The layout algorithm described in Section 4.5, is based on the size attributes of each WIDGET. Since these size attributes are not set after the generation of the structural UI model, they have to be calculated explicitly according to the WIDGET'S type. As previously mentioned, this is done for each WIDGET, traversing the structural UI model recursively bottom up. At each recursion step, the layout algorithm examines whether the current WIDGET is a PANEL or not. If it is not a PANEL, only a size calculation is required. This is done by one of the three approaches below with decreasing priority:

1. Width and hight attributes of the WIDGET are defined in the appropriate converted Cascading Style Sheet and, therefore can be derived from the style sheet (see Section 4.3).
2. According to a WIDGET whose size has to be calculated, a so-called Java Swing dummy widget is created temporarily. Its style attributes (e.g. font-style, font-name etc.) are either obtained from the appropriate converted Cascading Style Sheet, or by the properties-file that contains those data as well. The WIDGET'S height attribute is set according to the height of the resulting dummy widget. The WIDGET'S width must be obtained from the properties-file. This is necessary, because the text that has to be displayed by the WIDGET and, therefore determines the WIDGET'S width, is not available when size calculation is processed.
3. Width or height attributes of the widget are defined in the properties-file, that stores various default values.

The resulting size is just the width and height of the WIDGET itself. Padding and border-width do not affect the size of the WIDGET itself and are not considered in the size calculation.

4.4.1 Size Calculation for InputWidgets and OutputWidgets

Since size calculation is performed differently depending on a WIDGET'S type, this calculation is explained in detail for each INPUT- and OUTPUTWIDGET type. Note, that the WIDGETS are matched to Java Swing widgets. To provide a size calculation for other toolkits, other dummy widgets have to be used.

- BUTTON
  The Layout Module tries to obtain width and height by reading the converted Cascading Style Sheet (see Section 4.3). If there is no relevant entry in the converted Cascading Style Sheet for either of them, a Java Swing JButton is created as a dummy widget. The various font attributes for this dummy are obtained as itemized below, whereby the converted Cascading Style Sheet gains the higher priority:
  - **Font-name** and **font-size** are either obtained by reading the converted Cascading Style Sheet, or by reading the properties-file.
  - **Font-weight** and **font-style** are either obtained by reading the converted Cascading Style Sheet, or set to zero for plain text.

Since information about the text to be displayed is not available when size calculation is processed by the Layout Module, the created dummy can only determine the height attribute of the BUTTON. The width attribute is set to a standard pixel value that can be obtained from the properties-file as well.

- HYPERLINK
  The size calculation for a HYPERLINK works analogously to the size calculation for a BUTTON. A Java Swing JLabel is used as a dummy widget.
- RADIOBUTTON
  The size calculation for a RADIOBUTTON works analogously to the size calculation for a BUTTON. A Java Swing JRadioButton is used as a dummy widget.
- CHECKBOX
  The size calculation for a CHECKBOX works analogously to the size calculation for a BUTTON. A Java Swing JCheckBox is used as a dummy widget.
- COMBOBOX
  The size calculation for a COMBOBOX works analogously to the size calculation for a BUTTON. A Java Swing JComboBox is used as a dummy widget.
- TEXTBOX
  The size calculation for a TEXTBOX works analogously to the size calculation for a BUTTON. A Java Swing JTextBox is used as a dummy widget.
- TEXTFIELD
  A TEXTFIELD derives its width and height either from the converted Cascading Style Sheet or the properties-file. The converted Cascading Style Sheet gets the higher priority.
- IMAGEMAP
  An IMAGEMAP derives its width and height either from the converted Cascading Style Sheet or the properties-file. Here again, the converted Cascading Style Sheet gets the higher priority.
- DATETIMEPICKER
  A DATETIMEPICKER derives its width and height either from the converted Cascading Style Sheet or the properties-file. Here again, the converted Cascading Style Sheet gets the higher priority.
- LABEL
  The size calculation for a LABEL works analogously to the size calculation for a BUTTON. A Java Swing JLabel is used as a dummy widget.
- PICTUREBOX
  A PICTUREBOX derives its width and height either from the converted Cascading Style Sheet or the properties-file. Here again, the converted Cascading Style Sheet gets the higher priority.
- AUDIOBOX
  An AUDIOBOX derives its width and height either from the converted Cascading Style Sheet or the properties-file. Here again, the converted Cascading Style Sheet gets the higher priority.

4.4.2 Size Calculation for Panels

The size calculation of Panels is performed differently, since the size of Panels depends on the child Widgets they contain. Encountering a Panel at a single recursion step, the size of its child Widgets is already set in the previous recursion step, but their LayoutData does not necessarily need to be set at this time. In this case, the LayoutData of the child Panels has to be set before in order to enable a size calculation of the Panel itself.

Depending on the Panel type, LayoutData is set differently (see Section 4.6) and size is calculated as follows:

- Choice
  In case of a Choice, one of the child Panels is chosen to be placed at the space reserved by the Choice's size. Therefore, the Choice's size is set according to the maximum extents of its child Panels. Padding and border width of the child Panels are considered in the maximum extents as well. This ensures that each child Panel fits into the space reserved for the Choice.

- TabControl
  Each child Panel of a TabControl is placed in a single tab of the TabControl. The TabControl must be large enough to be able to display all contained tabs. Furthermore, the additional space that is claimed by the tabs' header has to be considered. Therefore, a Java Swing JTabbedPane is created as a dummy TabControl. Representatively for each child Panel, Java Swing JPanels are inserted into the JTabbedPane, acting as single tabs. The preferred size of the JPanels is set according to the corresponding child Panels size, their padding and their border-width. Padding and border-width of the child Panels are either derived from the converted Cascading Style Sheet or set to zero. The inserted JPanels determine the size of the resulting JTabbedPane and the size attributes of the TabControl are set accordingly.

- ListWidget
  If the ListWidget's *renderingType* attribute is set to LIST or PANEL, its size is not affected by the child widgets' size. In this case, it derives its width and height either from the converted Cascading Style Sheet or the properties-file. Again, the converted Cascading Style Sheet gets the higher priority. If the *renderingType* attribute of the ListWidget is set to FOLDOUT, the height is derived from a Java Swing ComboBox, that functions as a dummy and has exactly the hight for displaying a whole entry of the ListWidget. The width is derived from the converted Cascading Style Sheet or by the properties-file.

- Panel and Optional
  When the layout part is completed for each child Widget, the size calculation for the current Panel or Optional can be started. A Java Swing JPanel is created to function as a container dummy. Representatively for each child Panel, Java Swing JPanels are inserted into the JPanel container dummy. The preferred size of the inserted JPanels is set according to the corresponding child Widget's width and height attributes, enlarged by their padding and border-width at each side. Furthermore, the inserted JPanels are layouted according to the corresponding child Widgets' LayoutData. Padding and border-width of each child Widget are derived from the converted Cascading Style Sheet. If an according entry in the converted Cascading Style Sheet does

not exist, these values are set to zero. After all JPanels are inserted, the size of the resulting JPanel container dummy represents the size of the PANEL or OPTIONAL under test. Its width and height attributes are adopted.

- FRAME
  The size calculation for a FRAME works analogously to the size calculation of a PANEL or an OPTIONAL. In order to consider the additional space for the FRAME'S title-bar, a Java Swing JFrame is used as a container dummy.

Furthermore, in case of PANELS, this size calculation is not just needed to make decisions about where to place them in the next recursion step, but also to ensure that its contained child WIDGETS can be displayed properly on the screen. For example, a single widget W, that is nested in a panel P, can only be displayed properly, if P is large enough. The width and height of panel P, that contains the widget W, must fulfill the following constraints:

- The panel's width needs to be greater or equal than the widget's width, summarized with its padding and border-width at the left and the right side (see Equation 4.6).

$$P_{width} \geq W_{width} + W_{paddingLeft} + W_{paddingRight} + W_{borderWidthLeft} + W_{borderWidthRight} \tag{4.6}$$

- A panel's height needs to be greater or equal than the widget's height, summarized with its padding and border-width at the bottom and the top (see Equation 4.7).

$$P_{height} \geq W_{height} + W_{paddingBottom} + W_{paddingTop} + W_{borderWidthBottom} + W_{borderWidthTop} \tag{4.7}$$

This condition is similar for a collocation of several widgets. To be displayed properly, the layouted widget collocation, with respect to the padding and the border-width of all widgets, must entirely fit into the panel. Therefore, it is of highest importance to know the exact number of pixels (vertical and horizontal extents), claimed by a collocation of widgets.

4.5 Layout Algorithm

When widgets with different width and height values have to be arranged, the question arises which layout-manager to choose. The LAYOUTMANAGERS provided by the UCP framework were already explained in 3.2.2. The Layout Module uses the most customizable GRIDLAYOUT-manager (Java Swing GridBagLayout-Manager) and assigns the appropriate layout-data (GRIDLAYOUTDATA of the structural UI meta-model) to the PANELS, that need to be arranged. The FLOWLAYOUT-manager (Java Swing FlowLayout-Manager) is not used here because of its low flexibility.

A short example illustrates the problems that can occur by using the GridBagLayout-Manager. A set of Java Swing JPanels is arranged. Figure 4.3 shows three panels P1.1, P1.2 and P1.3, whose preferred size attributes are set. The three panels have to be arranged in their parent panel P1. The preferred size attribute of P1 is not set in this example. Therefore, widgets of any size can be inserted into P1, which is automatically resized afterwards to ensure enough space for its inserted widgets.

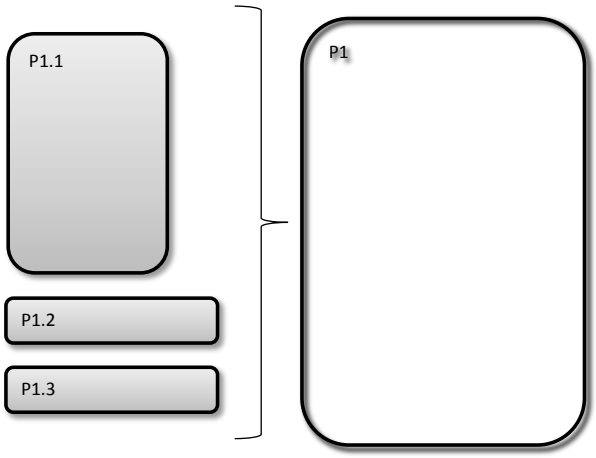


Figure 4.3: panels to be arranged (1).

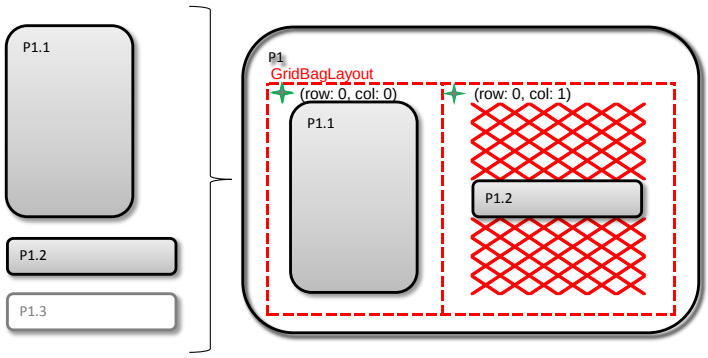
Using the GridBagLayout-Manager, the three panels can be arranged in various ways. In this example, the first panel to be placed in P1 is P1.1, followed by P1.2 and finally by P1.3. Figure 4.4(a) illustrates the special case, where P1.1 and P1.2 are arranged next to each other, and P1 is resized according to the current collocation of the inserted panels. Colspan and rowspan of the two placed panels are set to one in this example. P1.1 is placed at row zero and column zero. Therefore, two possible insertion points for P1.2 exist. These are below P1.1 at row one and column zero, and to the right of P1.1 at row zero and column one. The second one is chosen in the current example. Fully aware of the fact that arranging them below each other would save space, they are arranged next to each other in this example, to reveal some problems when using the GridBagLayout-Manager.

As shown in Figure 4.4(a), the red marked space above and below P1.2 is not available any more. For further panel inserting, another insertion point has to be found. This is done in Figure 4.4(b). Again, plenty of space is wasted. To make the wasted area under P1.2 accessible for P1.3, the rowspan of P1.1 has to be set to a value greater than one. In Figure 4.4(c) a rowspan of two is chosen, and a second row is created. P1.1 uses both of them at column zero. P1.2 can stay at row one and column zero. P1.3 can be placed at row one and column one, as shown in Figure 4.4(c). That way, P1.2 and P1.3 are placed in the same column, each in a single row and the wasted space is reduced.

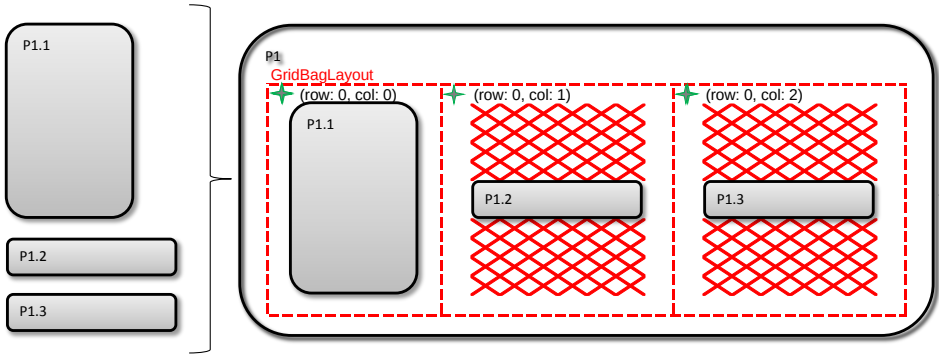
Each cell can have different width and height, because it adapts its dimension to the widgets that are placed on it. If several widgets of different widths are placed below each other in a single column, the column's width is automatically resized according to the maximum width of the widgets. The same characteristics are valid when several widgets are placed next to each other in a single row. A row adapts its height to the highest widget, it contains. This behavior gets more and more confusing, when single widgets obtain more than one row or column. A precise declaration of the different cell dimensions cannot be made by reasonable effort.

These facts complicate further calculations. It is difficult to make decisions about the widgets' rowspan and colspan, and to calculate the free space that is available to place further widgets. The question of how to find an optimum number of rows or columns that should be allocated to a single widget, is not a trivial one. The cells that are occupied by other widgets, have to be remembered as well as their width and height. They are not available for further placement.

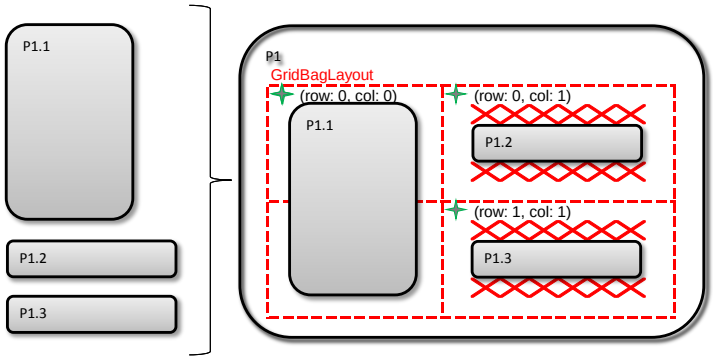
The search for insertion points is confusing as well, because of the different cell dimensions and the fact, that their dimensions can alter when further placement is done.



(a) P1.2 is placed to the right of P1.1, wasting the marked space above and below.



(b) Since the marked space above and below P1.2 is not available any more, P1.3 has to be placed beside, wasting space itself.



(c) Introducing a second row leads to a lower amount of wasted space, but still some of it remains. Increasing the number of rows would improve space saving.

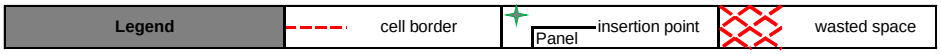


Figure 4.4: Panels to be arranged (2).

To solve this problem, the Layout Module defines a constant amount of pixels for rows and columns. This results in a grid with equally sized cells that can be represented by a two-dimensional integer-array. Colspan and rowspan as well as insertion points can easily be calculated. Furthermore, debugging is simplified since the grid is made visible through its introduced representative integer-array.

4.5.1 The Grid

As already mentioned, the Layout Module defines a constant width for all rows and a constant height for all columns. This is done for each PANEL whose children have to be layouted. Therefore, the attributes of the corresponding GRIDLAYOUT are set accordingly. The *rowHeight* and *colWidth* attributes are defined in the properties-file. The numbers of a PANEL'S rows and columns, which are set with the aid of the *row* and *col* attributes, depend on the collocation of the PANELS' children. Resulting in a grid with equally sized cells, this well defined grid offers the possibility to calculate rowspan and colspan, that are required for each WIDGET. In order to guarantee this functionality, the WIDGET'S (W) dimension and LAYOUTDATA must fulfill the Equations 4.8 and 4.9:

$$colspan = \left\lceil \frac{W_{width} + W_{paddingLeft} + W_{paddingRight} + W_{borderWidthLeft} + W_{borderWidthRight}}{columnWidth} \right\rceil \tag{4.8}$$

$$rowspan = \left\lceil \frac{W_{height} + W_{paddingBottom} + W_{paddingTop} + W_{borderWidthBottom} + W_{borderWidthTop}}{rowHight} \right\rceil \tag{4.9}$$

To guarantee enough space for the widget within the allocated grid-cells, colspan and rowspan are rounded up to the next larger integer value. If these equations are satisfied, a defined number of equally sized cells is allocated to each widget. This offers the possibility to represent a PANEL by introducing a temporary two-dimensional integer-array, that is called grid in the further writing. The bounds of the grid are calculated as described in the Equations 4.10 and 4.11, where C represents the children of the PANEL.

$$maxCols = \frac{\sum C_{colspan}}{columnWidth} \tag{4.10}$$

$$maxRows = \frac{\sum C_{rowspan}}{rowHight} \tag{4.11}$$

Equations 4.10 and 4.11 guarantee that all child WIDGETS that have to be placed can be arranged next and below to each other in the grid if needed.

Child WIDGETS that are already processed are marked in the parent PANEL'S grid representation. This is done by assigning defined integer values to the corresponding grid-cells. The different values that can occur in this grid are listed in Table 4.2. The resulting grid offers the possibility to search for patterns that represent possible insertion points . These insertion points are evaluated in terms of size consumption and one of them is chosen for a further WIDGET insertion.

Table 4.2: Grid values.

Marker	Name	Description
0	free	A 0 denotes a cell that can be used to place a further WIDGET.
1	occupied	A 1 denotes a cell that is occupied by a WIDGET. It is restricted for further use.
2	start point	A 2 denotes a cell that is occupied by the upper left corner of a WIDGET. It is restricted for further use.
4	forbidden space	A 4 denotes a forbidden cell in the grid. No WIDGET may occupy this cell. This value is only necessary in case of a LISTWIDGET or a PANEL that traces to a Sequence.
9	out of frame	A 9 denotes a cell that exceeds the FRAME'S width.

Example

To give an example, the LAYOUTDATA of the PANELS P1.1, P1.2 and P1.3, as they are arranged in Figure 4.4(c), are mapped to the grid with a constant column-width and row-height of 10 pixel each. According to the size and LAYOUTDATA displayed in Table 4.3, the three PANELS are inserted into the grid.

Table 4.3: Calculation of colspan and rowspan for the panels P1.1, P1.2 and P1.3.

Calculation of ColSpan							
Panel	**panel width**	**padding**		**border width**		**total width**	**colspan**
		left	**right**	**left**	**right**		
P1.1	30 px	4 px	4 px	1 px	1 px	40 px	**4 colums**
P1.2	40 px	4 px	4 px	1 px	1 px	50 px	**5 colums**
P1.3	40 px	4 px	4 px	1 px	1 px	50 px	**5 colums**
Calculation of RowSpan							
Panel	**panel hight**	**padding**		**border width**		**total hight**	**rowspan**
		top	**bottom**	**top**	**bottom**		
P1.1	50 px	4 px	4 px	1 px	1 px	60 px	**6 rows**
P1.2	10 px	5 px	5 px	1 px	1 px	22 px	**3 rows**
P1.3	10 px	5 px	5 px	1 px	1 px	22 px	**3 rows**

As shown in Figure 4.5, PANEL P1.1 is assigned a rowspan of six and a colspan of four. It is placed at row zero and column zero. PANEL P1.2 as well as PANEL P1.3 are both assigned a rowspan of three and a colspan of five. In order to be displayed completely and not being overlapped by PANEL P1.1, both PANELS (P1.2 and P1.2) start at column four (note that the grid indices are starting with zero). Vertically, PANEL P1.2 starts at row zero and P1.3 at row three since the first three rows are occupied by PANEL P1.2.

The wasted space results from the fact that the total height of P1.2 and P1.3 is not exactly an integer multiple of the row width. According to their layout-data, the WIDGETS are aligned within the three rows. In the current example, the two PANELS are centered within the three

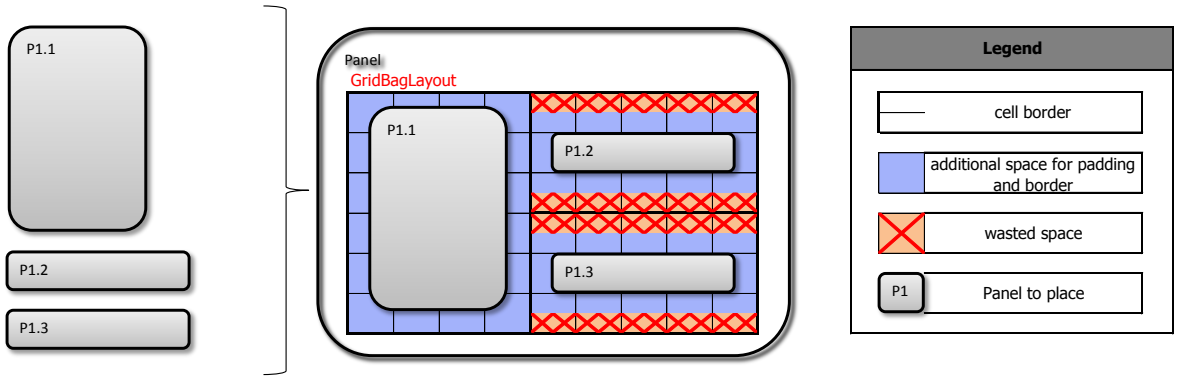


Figure 4.5: Arranging P1.1, P1.2 and P1.3 in a grid whose cells are set to squares with a side length of ten pixels.

rows, resulting in waste of four pixel at the top and at the bottom. This can be optimised by lowering the height of each row.

Defining a column-width and a row-height of one pixel for each would lead to the characteristics of an XYLayoutManager. Consequently, row and column values would represent the x and y attributes in this case. Width and height attributes would be determined by the widgets' dimensions.

The grid representation of the current example is shown in Figure 4.6. It serves as fundamental instrument to calculate further insertion points, and to validate them in terms of space consumption.

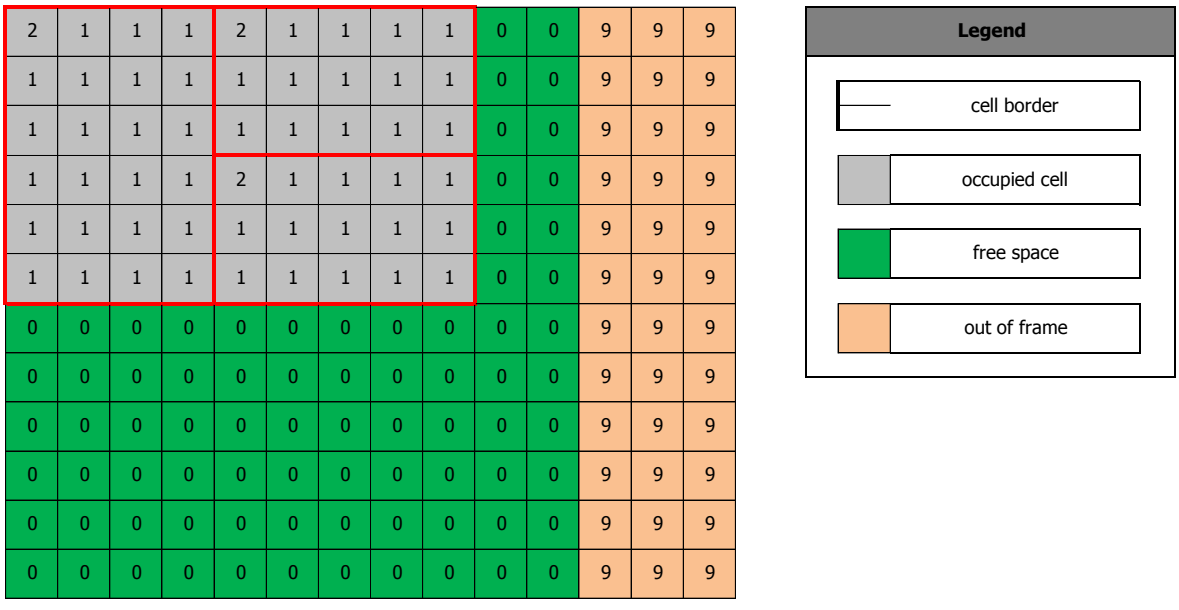


Figure 4.6: P1.1, P1.2 and P1.3 indicated in the representative grid of P1.

4.5.2 Insertion Points

Each Panel whose children have to be layouted is assigned a corresponding grid. The limits of this grid are derived from the child Widgets (see Equations 4.10 and 4.11). This offers

the possibility to calculate insertion points for each child. These insertion points define the upper left corner of the corresponding grid cell.

Initially, the grid is empty and, therefore all grid-cells are set to 0 and potentially 9 to indicate cells that exceed the right frame border. In case of a ListWidget, the 9 values indicate the right border that is defined by the ListWidget's width. In an empty grid, the upper left grid cell defines the first insertion point by default. For a possible second child Widget, two insertion points result: one next to the first placed Widget and one below to it.

In case of a large queue of child Widgets, many more insertion points can result at each child Widget placement. One of them has to be chosen for each child Widget, and the grid must be updated accordingly (see Section 4.6).

To find insertion points, the grid is examined for certain patterns. The search algorithm does not differentiate between the grid values that denote an already occupied or forbidden cell. Therefore, it treats a 1, a 2, a 3 and a 4 just the same way. The insertion points in this chapter are illustrated by using only 0, 1 and 2 values.

In the figures below, the insertion points are marked with a bold (red) 0, and the corresponding pattern is displayed in the dark-grayed boxes. Furthermore, the irrelevant zero values are grayed, too. To illustrate the patterns for possible insertion points, the grid is filled with two of the three Panels that were already illustrated in Figure 4.3. In the following figures, these Panels are displayed in light gray. In order to achieve different insertion points that result for a placement of the third Panel, the size of the first two Panels is varied in each case. The gray down arrows at the bottom indicate that the grid does not end at the corresponding row. It may contain further empty rows that are not displayed here.

The insertion points that can occur in an existing grid are only applicable if the Widget that has to be placed at this insertion point does not encounter grid values greater than zero. In other words, the Widget must fit into the free region next to the insertion point and below it.

Empty Grid

In case of an empty grid, the upper left grid cell is always set to 0. As shown in Figure 4.7, this grid cell is used as insertion point. If the grid is filled with at least one Panel, the value of this grid cell is set to 2 on any case.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	**0**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	9
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	9
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	9

Figure 4.7: Insertion point in an empty grid.

First Row

In the first row, a 1 horizontally followed by a 0 defines the pattern for an insertion point. As depicted in Figure 4.8, the bold (red) 0 defines the resulting insertion point.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	2	1	1	1	2	1	1	1	**0**	0	0	0	0	0	0	0	0	0	0	0	9
1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	9
2	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	9
3	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	9
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	9
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	9
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	9

Figure 4.8: Insertion point in the first row.

First Column

In the first column, a 1 vertically followed by a 0 defines the pattern for an insertion point. As depicted in Figure 4.9, the bold (red) 0 defines the resulting insertion point.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	2	1	1	1	2	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	9
1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	9
2	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	9
3	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	9
4	**0**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	9
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	9
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	9

Figure 4.9: Insertion point in the first column.

Inside - 1

Inside the grid, there exist two possible patterns for insertion points. The first one to mention is depicted in Figure 4.10. The pattern to search for is shown in the grayed cells. the bold (red) 0 at the bottom right of the pattern denotes the insertion point.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	2	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	9
1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	9
2	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	9
3	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	9
4	2	1	1	1	**0**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	9
5	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	9
6	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	9
7	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	9
8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	9
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	9

Figure 4.10: Insertion point inside the grid (1).

Inside - 2

The second possible insertion point pattern that can be found anywhere inside the grid but not in the first column or row is shown in Figure 4.11.

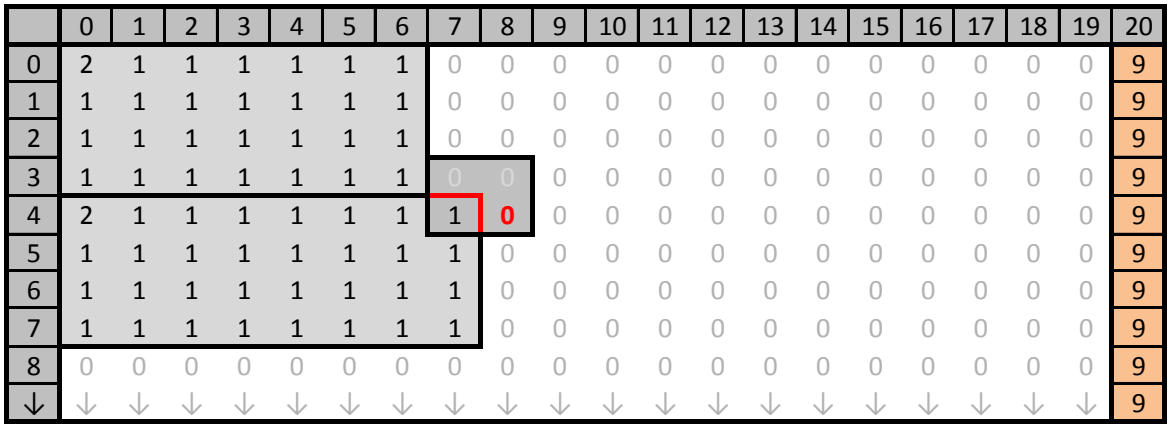


Figure 4.11: Insertion point inside the grid (2).

New Column

As shown in Figure 4.12, the upper cell of the leftmost free column defines the insertion point at a new free column.

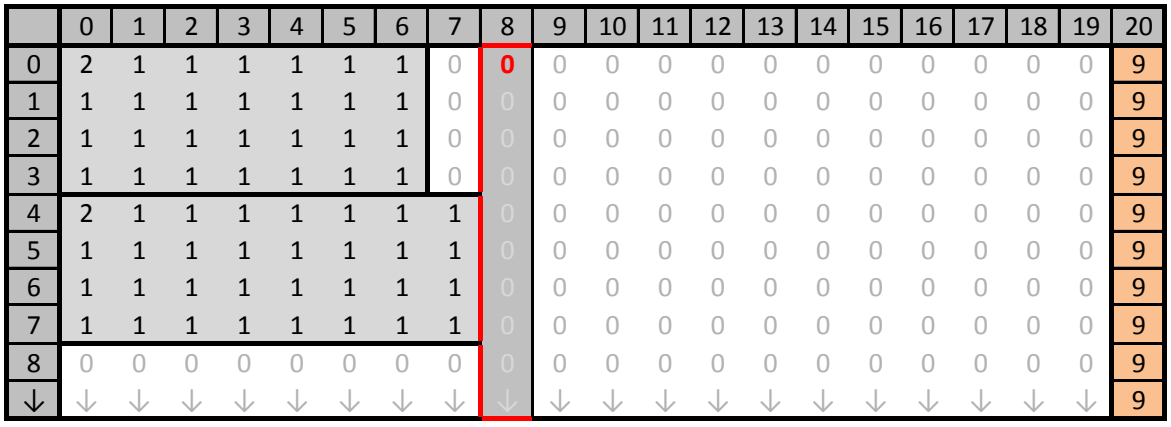


Figure 4.12: Insertion point at a new free column.

New Row

As depicted in Figure 4.13, the first cell of the topmost free row defines the insertion point at a new free row.

If no other valid insertion points were found, these insertion points functions as failsafe. Even if the WIDGET to place exceeds the maximum width, this insertion point is applicable. This is required to continue the calculation. In this case, a structural UI model can be created but scrolling has to be enabled.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	2	1	1	1	1	1	1	1	2	1	1	1	0	0	0	0	0	0	0	0	9
1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	9
2	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	9
3	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	9
4	0	0	0	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	9
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	9
6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	9
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	9

Figure 4.13: Insertion point at a new free row.

4.5.3 Choice of the Insertion Point

Choosing the best insertion point is a delicate issue within the Layout Module. A certain strategy may lead towards a well looking GUI for a given set of Widgets. But for another set of Widgets, this strategy may lead towards a so-called worst case scenario. The placement-strategy of the Layout Module aims to minimize the used area.

As already mentioned in the previous subsection, an insertion point is only applicable if the Widget that has to be placed at the corresponding cell fits into the free cells next to this point and below to it. A Widget that exceeds the limit of the screen represents an exception. It is automatically placed at the insertion point at the new free row. All other possible insertion points are discarded for this Widget.

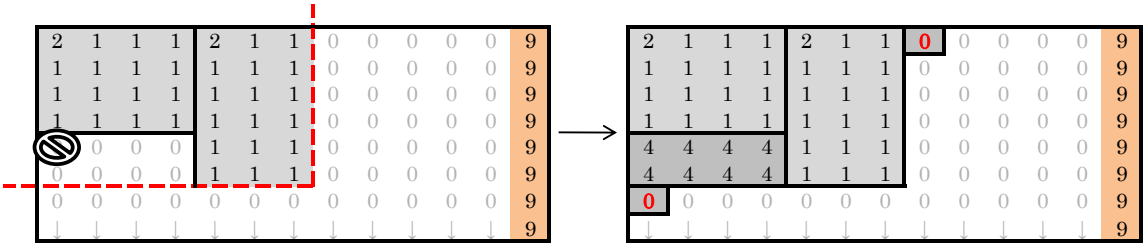
Before starting the algorithm that searches insertion points, some grid modifications may be necessary. Whether these modifications are required or not depends on the order of the corresponding Panel's children. If the Panel's children do not have a mandatory order, no modifications are required for the corresponding grid. If the order of the Panel's children is mandatory, the corresponding grid has to be modified after each Widget insertion. The modification ensures that the given order is kept. The two cases need to be explained separately, because the strategy of choosing the insertion point differs.

4.5.3.1 Choice of Insertion Point for Child Widgets with a Mandatory Order

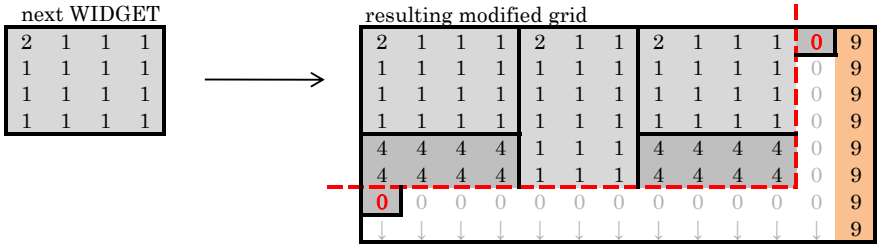
In this case, the order of the child Widgets needs to be kept. Therefore, the grid is modified to prevent the Layout Module to choose insertion points that harm this order. The modifications are illustrated in Figure 4.14.

Figure 4.14(a) illustrates a grid that represents a Panel which is already filled with two Widgets. For further insertion, the order of the child Widgets has to be kept. Placing the third Widget at the crossed out position in the grid at the left side would break the order of the Widgets. To prevent the search algorithm from choosing this insertion point, the grid has to be modified. Therefore, the grid is trimmed according to the maximum extents of the two already inserted Widgets, illustrated by the broken line in Figure 4.14(a). The included free cells (0-values) are changed to forbidden cells (4-values) in order to indicate forbidden areas. Only the two insertion points illustrated in Figure 4.14(a) remain available for the next Widget, next to the second Widget and in a new free row. This guarantees

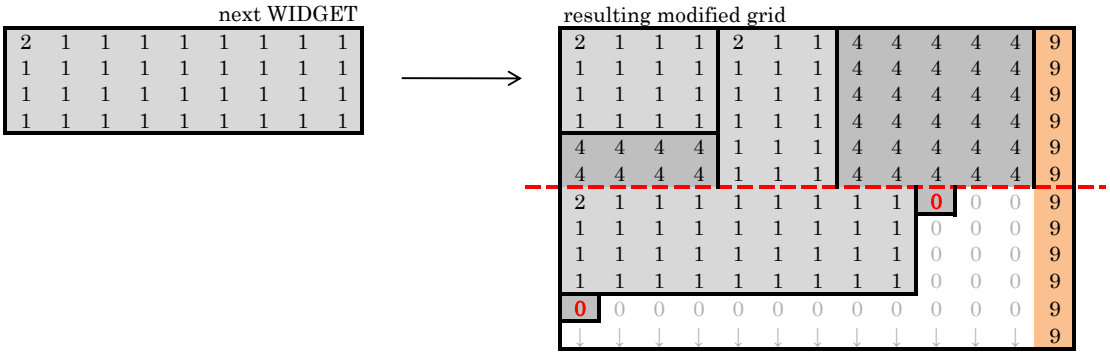
that the predefined order can be kept. Regardless of the wasted space, the insertion point next to the second WIDGET gets the higher priority in this case. As already mentioned in the previous subsection (see 4.5.2), this insertion point is only applicable if the WIDGET that has to be placed fits into the remaining space. If this requirement is not met, a new free row has to be started. In any case, further grid-modifications are required to guarantee the order of the widgets.



(a) Modification of a grid, that contains two PANELS.



(b) The WIDGET to be placed fits into the free space next to the second WIDGET.



(c) The WIDGET to be placed does not fit into the free space next to the second WIDGET. Therefore, the WIDGET has to be placed in a new free row.

Figure 4.14: Grid modifications for a LISTWIDGET as well as for PANELS, that are tracing to a Sequence.

In Figure 4.14(b), the third WIDGET is small enough to be placed next to the two other WIDGETS. The grid again has to be modified in the same way as before. It is trimmed according to the maximum extents of the already inserted WIDGETS (dashed line in Figure 4.14(b)), and afterwards the included 0-values are changed to 4-values. Keeping the WIDGET order, this again results in the two possible insertion points for a further WIDGET as depicted

in Figure 4.14(b).

Figure 4.14(c) illustrates the case, when the remaining space next to the first two WIDGETS is too small for the next WIDGET to place. To keep the WIDGET order, the grid is modified again to provide an insertion point at the next free row. The remaining 0-values above the dashed line in Figure 4.14(c) are changed into 4-values, to avoid a placement of further WIDGETS in this region which would destroy the predefined order. The remaining space above becomes a forbidden region. A further WIDGET would be placed next to the third WIDGET or below it.

For any further WIDGET, the grid modification just affects the grid starting at the actual row. Furthermore, all grid modifications are made permanent.

To sum up, each WIDGET can be either placed right beside the previous WIDGET or below in a new free row.

4.5.3.2 Choice of Insertion Point for Child Panels Without a Given Order

All possible insertion points can be used, and a modification of the grid is not required in this case. To put WIDGETS of similar size closer together, and to lower the complexity of the resulting WIDGET collocation in the grid, the child WIDGETS are reordered according to an entry in the properties-file. Depending on this entry, the child WIDGETS are ordered by area or width, starting with the smallest or the largest.

For each WIDGET insertion, the search algorithm offers a certain set of insertion points. These insertion points have to be evaluated in terms of the free space that is wasted after an insertion of the current WIDGET. If only a single applicable insertion point exists, it is chosen without any evaluation. In contrast, for more insertion points some calculations are necessary. A temporary copy of the current grid is created for each applicable insertion point. Further, the WIDGET is inserted into each copy of the current grid at one of the calculated insertion points. Afterwards, the grid copies are trimmed according to the extents of their containing WIDGETS – the currently inserted WIDGET included – and the included zero values are counted. The insertion point whose WIDGET insertion results in a minimum amount of zero values in the corresponding trimmed copy of the grid represents the insertion point, which leads towards a minimum waste of free space. This insertion point is finally chosen, and the current WIDGET is inserted into the original grid at this position. If two or more insertion points result in the same amount of zero values in the corresponding trimmed grids, the insertion point that minimizes scrolling is chosen.

Figure 4.15 illustrates this procedure. The WIDGET displayed on the top (next WIDGET) has to be inserted into the grid illustrated on the right. Two other WIDGETS are already inserted into the grid, leading to the three insertion points that are marked by a bold (red) zero in the gray cells. One in the first column, one in the new free row and one in the first row.

The resulting grid that is created temporarily for each insertion point is illustrated below. The one at the left side is created for the insertion point in the first column. It results in a trimmed grid with 11 cells that are set to zero. The two other grids contain 38 and 23 cells that are set to zero. Therefore, the insertion point in the first column leads to a minimum of waste space in the current example. It is finally chosen for the current WIDGET, and the

original grid is updated according to this choice. Any remaining free area – even an area that is surrounded by other WIDGETS at each side – is available for further WIDGET insertion.

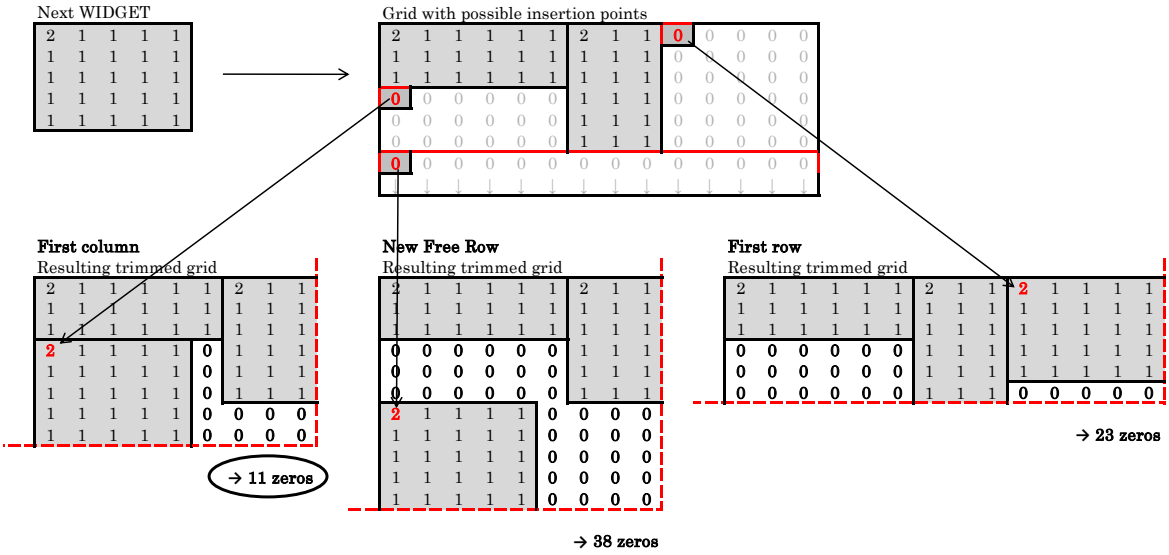


Figure 4.15: Insertion point evaluation for a WIDGET. The PANEL's children do not have a mandatory order and were sorted by area, smallest first.

4.6 Calculation of the LayoutData

To set a layout upon a structural UI model, the LAYOUTDATA of all of its WIDGETS need to be set. The attributes, required for each WIDGET, depend on the currently used LAYOUTMANAGER(see 3.2.2).

In case of using the GRIDLAYOUT-manager, GRIDLAYOUTDATA has to be set. Only the *row* and *column* as well as the *rowspan* and *colspan* need to be calculated. Furthermore, the GRIDLAYOUT attributes *rows*, *cols*, *rowHeight* and *colWidth* have to be set to ensure equally sized cells within a PANEL's grid. Decisions concerning *fill*-type, *alignment*, *weightx* and *weighty* can not be made when the layout algorithm of the Layout Module is processed. As itemized below, these attributes are set to default values for each WIDGET that is layouted by the Layout Module:

- In order to retain the calculated size of the WIDGETS, the *fill* type is set to **NONE**.
- In order to left align WIDGETS that are placed in the same column, and to top align WIDGETS that are placed in the same row, the *alignment* attribute is set to **NORTH_WEST**.
- In order to retain the calculated size of the WIDGETS, the *weightx* attribute is set to 0.
- In order to retain the calculated size of the WIDGETS, the *weighty* attribute is set to 0.

Using the XYLAYOUTMANAGER, the x and y coordinates of the WIDGET's upper left corner as well as its *width* and *height* need to be set in the corresponding XYLAYOUTDATA.

As the number of pixels of a row and a column are known, a WIDGET's GRIDLAYOUTDATA can be easily converted into XYLAYOUTDATA. The GRIDLAYOUTDATA's *row* and *col* attribute can be converted to absolute coordinates. Together with an offset that results from the GRIDLAYOUTDATA's *alignment* attribute, the x and y coordinates that are required by the XYLAYOUTDATA to determine the upper left corner of the WIDGET can be calculated. The *width* and *height* attributes are obtained from the WIDGET itself. This provides a layout calculation for both of the mentioned LAYOUTMANAGER, using the same layout algorithm (see Section 4.5).

Depending on the PANEL type, the calculation of the LAYOUTDATA varies. In case of a CHOICE or a TABCONTROL, the layout algorithm, and especially the search for insertion point using the grid, is not necessary. Due to the characteristics of these PANELS, the GRIDLAYOUTDATA attributes *row* and *col* of all child WIDGETS are set to zero.

Encountering one of the following PANELS, the LAYOUTDATA is filled according to the results of the layout algorithm:

- PANEL that traces to a Sequence or an RSTMultiNucleusRelation
- FRAME that traces to a Sequence or an RSTMultiNucleusRelation
- OPTIONAL that traces to a Sequence or an RSTMultiNucleusRelation
- LISTWIDGET

The further paragraphs describe the calculation of the child WIDGET's LAYOUTDATA for the different PANEL types.

Choice

Since there is only one child WIDGET placed in the space reserved at the same time, each child WIDGET is placed at the upper left corner of its containing PANEL, the CHOICE. There is no need to calculate any further LAYOUTDATA.

The two different LAYOUTDATA values are set as follows:

- GRIDLAYOUTDATA: *Row* and *col* of each child WIDGET are set to zero. By default, *rowspan* and *colspan* are set to one, and *alignment* is set to **NORTH_WEST**.
- XYLAYOUTDATA: The x and y coordinates are set to zero, *width* and *height* is adapted to the WIDGET's dimension.

TabControl

Each child PANEL is placed in a single tab. LAYOUTDATA is set in the same way as it is done in case of a CHOICE.

Frame, Panel or Optional that is tracing to a Procerural Sequence Relation or an RSTMultiNucleusRelation

The layout algorithm calculates *rowspan* and *colspan*, as well as an insertion point for each child WIDGET. The calculated data can be mapped to the data that is required by the used LAYOUTMANAGER.

- GRIDLAYOUTDATA: *Row* and *col* of each child WIDGET are filled according to the calculated insertion point. The insertion point's x value represents the column, and the y value represents the row. *Rowspan* and *colspan* are set according to the default grid width and height, the child WIDGET'S *width* and *height*, its padding at each side and its border-width (see Equation 4.9 and 4.8).
- XYLAYOUTDATA: The x and y coordinates are set according to the calculated insertion point and the default grid width and height. *Width* and *height* are set according to the child WIDGET'S *width* and *height* attributes, its padding at each side and its border-width.

ListWidget

The layout algorithm calculates *rowspan* and *colspan*, as well as an insertion point for each child WIDGET. The calculated data can be mapped to the data that is required by the used LAYOUTMANAGER. This is done in the same way as described for a FRAME, A PANEL or an OPTIONAL that is tracing to a Sequence or an RSTMultiNucleusRelation.

4.7 Results of Using the Layout Module

Using the Layout Module, a user interface can be created that provides both, a minimum amount of scrolling as well as a minimum waste of free space on the display area. If a target device provides enough screen space, the widgets are arranged in a way that scrolling can be avoided.

Depending on the chosen cell-dimension, the distances between single PANELS can vary in size. Additionally to the predefined padding of a single PANEL, this distance might be enlarged by the remaining number of unused pixels of the corresponding row or column. This problem has already been mentioned in 4.5.1, and is sketched in Figure 4.4. Being aware of this inaccuracy it can be held at a minimum by choosing small grid-cells. A choice of one pixel for height and width for each grid-cell would solve this problem. The dimension of the grid-cells is defined in the properties-file. During the development of the Layout Module, the grid-cells were set to squares with ten pixels for each side. This grid-cell dimension was chosen to simplify debugging. For the final integration into the UCP framework, the grid-cell dimension has been set to one pixel for best performance.

In the following paragraphs, an excerpt of a structural UI model, that results from the CommRobShopping discourse, is chosen to illustrate the operating principle of the Layout Module. It is handled twice, according to the strategies of ordering the WIDGETS. Furthermore, the OnlineShop discourse delivers an example for the outcome of a LISTWIDGET whose *renderingType* is set to **FOLDOUT**. It is presented according to two different WIDGET *widths*.

The evaluation and selection of insertion points within the following examples is performed as described in 4.5.3.

CommRobShopping – Largest First

Figure 4.16 illustrates the strategy where large WIDGETS are inserted first. The structural UI model shown on the top of the figure is already ordered accordingly.

As can be seen in the figure, the FRAME *Frame_CommRobShopping* contains six PANELS. The largest PANEL in this list is called *Panel_ShoppingLists*. It contains two further PANELS that are ordered largest first, too. This is indicated through the green line in the grid representation shown in the figure (middle). Their children are not illustrated in the structural UI model, since this would go beyond the scope of this example. Furthermore, LAYOUTDATA and size attributes of all children of the six PANELS are assumed to be set in previous recursion steps. Since the six PANELS that are contained by the FRAME are already ordered by their size, a corresponding size calculation is assumed to have been performed previously as well. Subsequently, the size attributes of the six PANELS, together with their padding and border-width, are used to calculate the number of grid-rows and grid-columns (colspan and rowspan) that are required to display the included WIDGETS properly. Padding and border-width can be obtained from the Cascading Style Sheet, or are set to zero as described above (see 4.5.1). Afterwards, the insertion of the single WIDGETS is performed, beginning with the PANEL *Panel_ShoppingLists*. PANEL *Panel_ProductLists* is the next in line to be inserted. Since there is not enough space available for an insertion next to the first PANEL, it is placed at the single alternative insertion point below. The resulting grid offers plenty of space for the next three Panels (*Panel_ManageShoppingList*, *Panel_Resume* and *Panel_FollowMe*) to be placed at the right. Finally, the last PANEL, *Panel_returnTrolley*, is placed below of the second PANEL according to the constraints of minimizing the used display area and minimizing scrolling. The resulting grid of this insertion procedure, together with the final user interface representation, is shown in Figure 4.16 as well.

The LAYOUTDATA that are necessary to display the PANELS the way they are illustrated in Figure 4.16 are listed in Table 4.4. This LAYOUTDATA can be easily reconstructed with the aid of the representative grid, shown in the figure. By default, *alignment* is set to **NORTH_WEST**, and *FillType* is set to **NONE**.

Table 4.4: Calculated LAYOUTDATA for the CommRobShopping example (largest first).

Panel	Insertion Point		RowSpan	ColSpan	Alignment	FillType
	Row	Column				
Panel_ShoppingLists	0	0	13	24	NORTHWEST	NONE
Panel_ShoppingAndDestinationList	0	0	13	16	NORTHWEST	NONE
Panel_InCartList	0	16	13	8	NORTHWEST	NONE
Panel_ProductLists	13	0	5	13	NORTHWEST	NONE
Panel_ManageShoppingList	13	13	2	7	NORTHWEST	NONE
Panel_Resume	15	13	2	7	NORTHWEST	NONE
Panel_FollowMe	17	13	2	7	NORTHWEST	NONE
Panel_ReturnTrolley	18	0	2	7	NORTHWEST	NONE

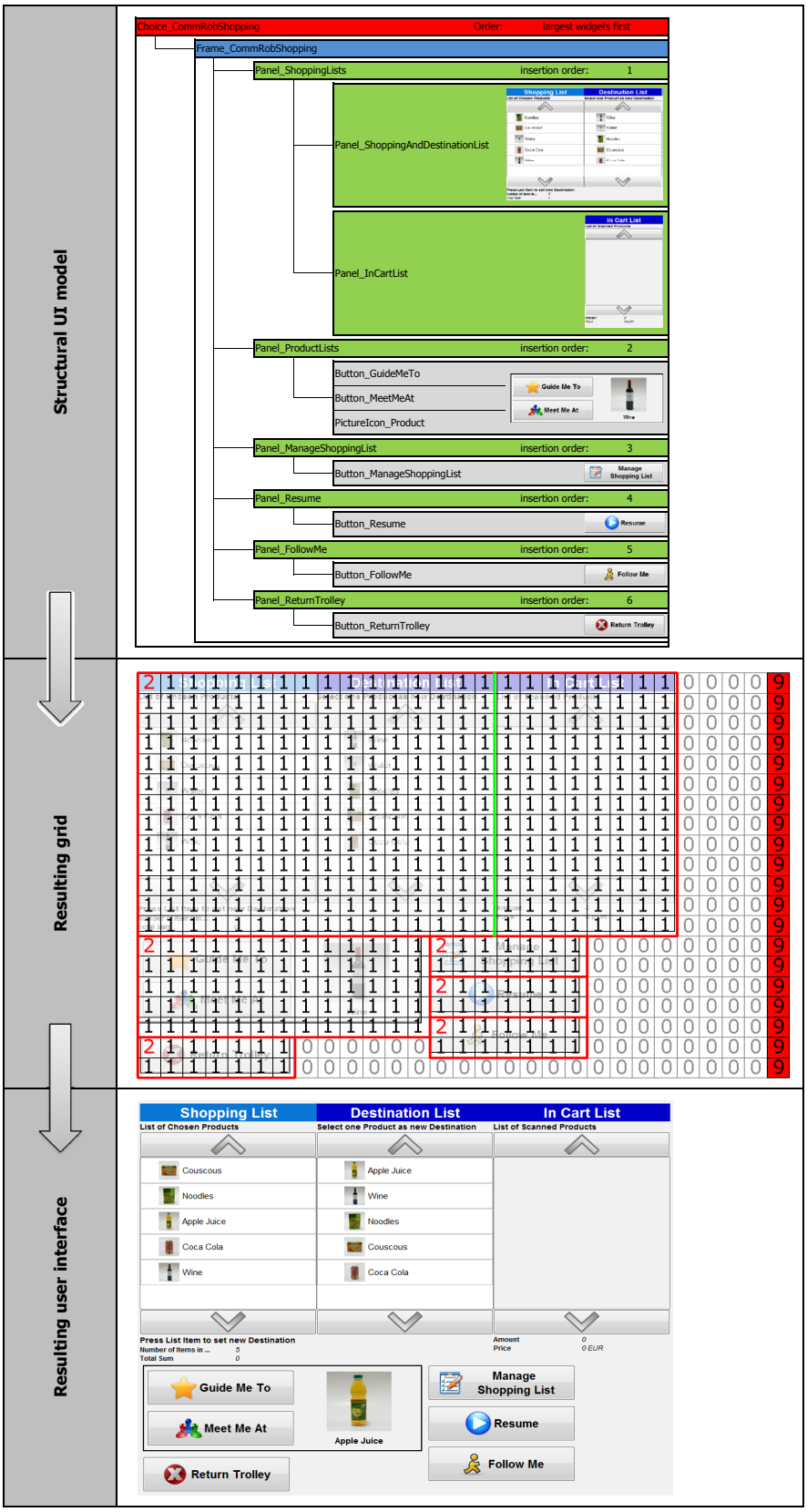


Figure 4.16: Result of the commRobShopping discourse, when large WIDGETS are inserted first.

CommRobShopping – Smallest First

In comparison to the previous example, the same structural UI model is now ordered the other way round, placing the smallest WIDGETS first. Figure 4.17 illustrates this example.

This way, it is possible to place the four equally sized PANELS side by side, in the first row. The next in line is the PANEL *Panel_ProductLists*, which has to be placed below of the first four PANELS since there is not enough space at the right. Furthermore, this PANEL would be placed at this insertion point anyway, since the area which is occupied by the first fife PANELS would be minimized. The last PANEL in this insertion queue is the PANEL *Panel_ShoppingLists*. There is not enough space available to place it next to PANEL *Panel_ProductLists* and, therefore it needs to be inserted at a new row. If there were enough space to place it at the right, the two insertion points would be evaluated in terms of space a resulting constellation would occupy. The insertion point that results in a smaller resulting PANEL is chosen.

Similarly to the example above, the LAYOUTDATA of the single PANELS are listed in Table 4.5. Again, this LAYOUTDATA can easily be reconstructed with the aid of the representative grid, shown in Figure 4.17. By default, *alignment* is set to **NORTH_WEST** and *FillType* is set to **NONE**.

Table 4.5: Calculated LAYOUTDATA for the CommRobShopping example (smallest first).

Panel	Insertion Point		RowSpan	ColSpan	Alignment	FillType
	Row	Column				
Panel_ManageShoppingList	0	0	2	7	NORTHWEST	NONE
Panel_Resume	0	7	2	7	NORTHWEST	NONE
Panel_FollowMe	0	14	2	7	NORTHWEST	NONE
Panel_ReturnTrolley	0	21	2	7	NORTHWEST	NONE
Panel_ProductLists	2	0	5	13	NORTHWEST	NONE
Panel_ShoppingLists	7	0	13	24	NORTHWEST	NONE
Panel_InCartList	0	0	13	8	NORTHWEST	NONE
Panel_ShoppingAndDestinationList	0	8	13	16	NORTHWEST	NONE

In the first of the two examples above, WIDGETS are ordered largest first. The resulting GUI is smaller than the one that results when using smallest first order. Both versions have the same height but the smallest first version is broader. As can be seen in Figure 4.16, where larger WIDGETS are inserted first, smaller WIDGETS (Buttons on the bottom) can be placed in the remaining free area. In Figure 4.17, where smaller WIDGETS are inserted first, using the remaining free area for larger WIDGETS is not possible and the three large WIDGETS have to be placed in a new free row on the bottom.

Ordering WIDGETS by their size can minimize complexity of the resulting GUI. However, it is hard to decide which order to take (largest first or smallest first). Which of them achieves better results depends on the predefined screen size and on the set of WIDGETS that have to be layouted. In most cases, a smaller GUI outcome can be achieved by using the largest first order. In order to save space, the largest first strategy of ordering the Widgets is chosen by default. The smallest first strategy can be enabled by changing the corresponding entry in the properties file.

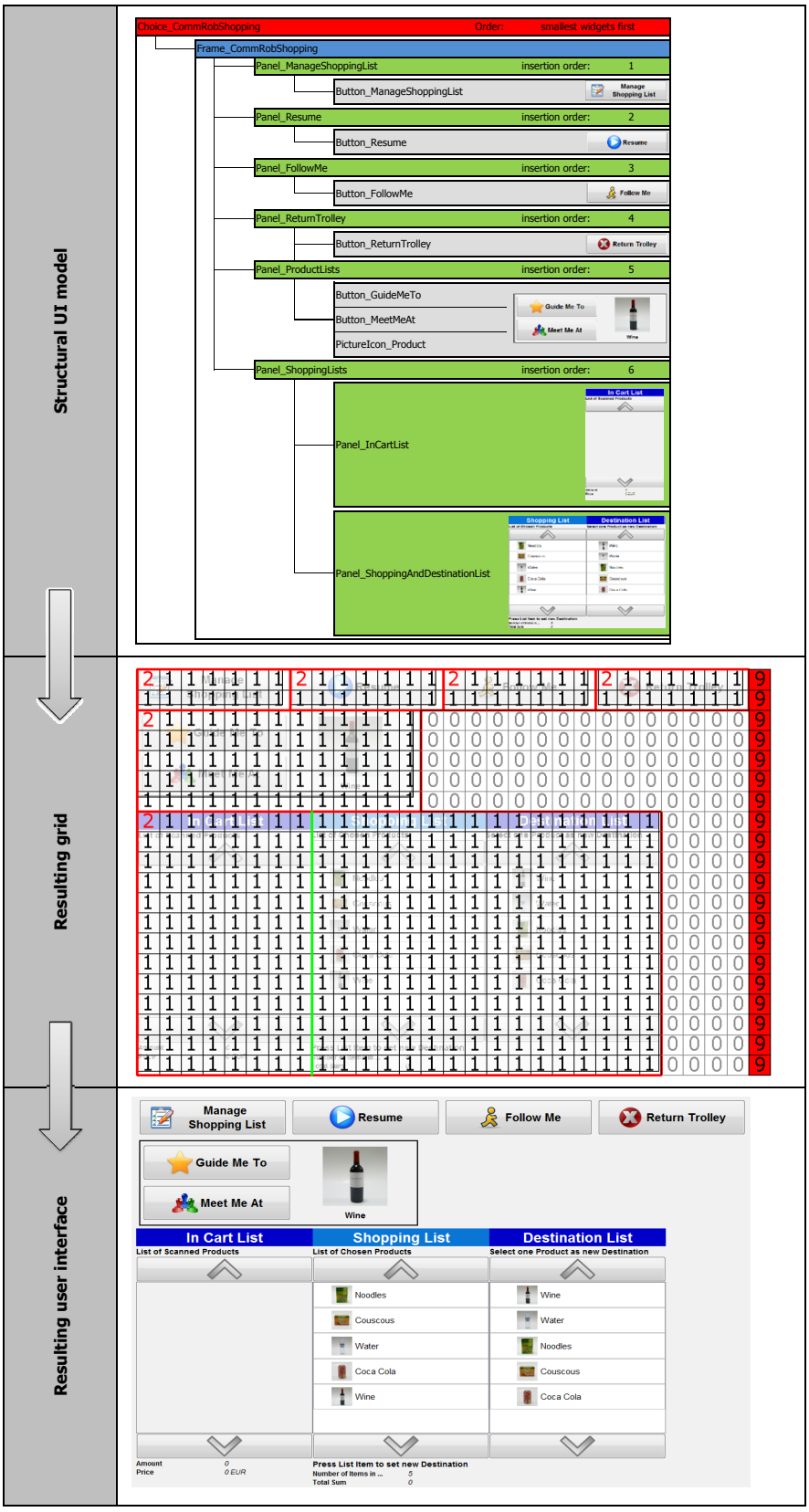
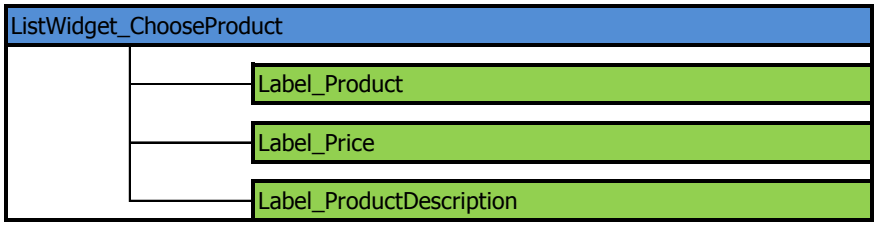


Figure 4.17: Result of the CommRobShopping discourse, when small WIDGETS are inserted first.

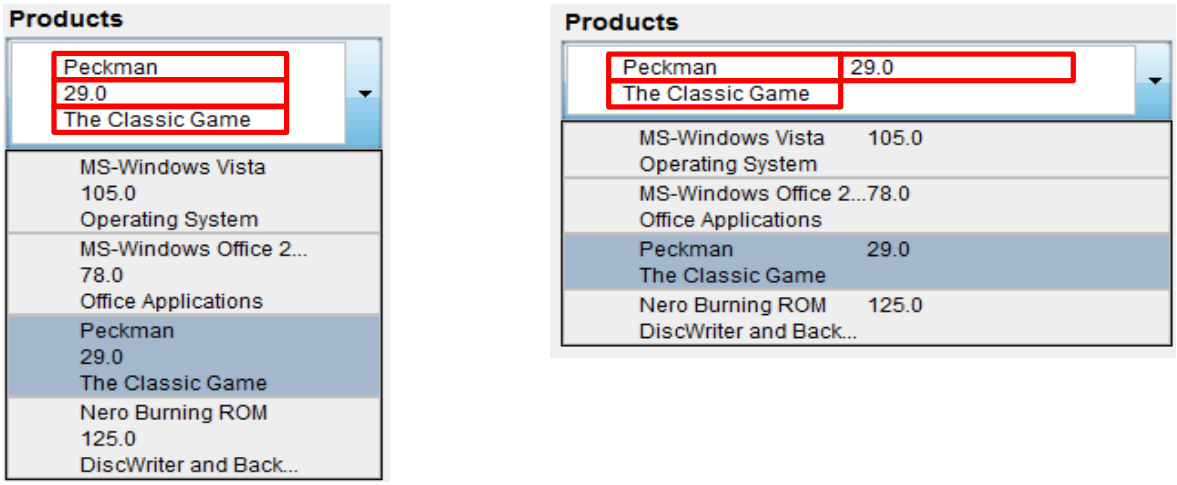
ListWidget Example

In terms of size-, and layout calculation, ListWidgets are handled in a special way (see Sections 4.4 and 4.5). Furthermore, they are rendered differently, according to their *renderingType* attribute. A ListWidget whose *renderingType* is set to **LIST** is rendered the same way as the three lists depicted in Figure 4.16 and 4.17 (Shopping List, Destination List and In Cart List). If the *renderingType* of a ListWidget is set to **FOLDOUT**, it is rendered as a drop down widget as shown in Figure 4.18. A *renderingType* set to **PANEL** results in a ListWidget that is displayed as a panel that contains all element information.

Note, that the width of a ListWidget is either set to the corresponding value of the Cascading Style Sheet, or to the corresponding entry in the properties-file. This is done independently from the ListWidget's *renderingType*. By contrast, the ListWidget's height depends on this *renderingType* attribute. In case of being set to **PANEL** or **LIST**, the height is set according to the corresponding entry in the Cascading Style Sheet or in the properties-file as well. In case of being set to **FOLDOUT**, the height depends on the arrangement of the ListWidget's inserted child Widgets. As shown in Figure 4.18, the ListWidget's height is set according to this arrangement.



(a) Structural UI model excerpt.



(b) Resulting ListWidgets.

Figure 4.18: OnlineShop: The renderingType of the ListWidget is set to FOLDOUT. It is rendered for two different widths.

The illustrated ListWidget results from an OnlineShopping discourse and its *renderingType* is set to **FOLDOUT**. As shown in Figure 4.18(a), an entry of the ListWidget consists of three Labels. These three Labels are arranged with respect to the predefined width of the

ListWidget. According to this layout, the height of the ListWidget is set. Figure 4.18(b) illustrates the same ListWidget rendered for two different widths. The broader outcome offers enough space to place the first two Labels next to each other and the third below in a new free row. The narrower outcome just offers the possibility to arrange the three Labels below each other. The height of the two examples is set according to a single corresponding entry. This ensures that the ListWidget can display a whole entry when it is not in fold out state.

Chapter 5

Conclusion

The Layout Module has been successfully integrated into the UCP framework. It succeeded in providing any generated structural UI with appropriate size and layout-data. Furthermore, it enabled the optimised discourse model to structural UI model transformation (see Chapter 3.4).

Chapter 4 describes the Layout Module at the time of its integration into the UCP framework. At this time, only the right border of a screen has been considered when choosing insertion points. Even if there were enough unused space on the screen, WIDGETS could have possibly been inserted somewhere below the screen – only accessible by using scrollbars. This problem concerns the choice of insertion points and has been solved as described in the paragraph below.

In case of a PANEL whose children do not have a predefined order (see 4.5.3.2), an insertion point that exceeds the screen's height is only chosen if no alternative insertion point exists within the displayable screen area. Therefore, another value has been introduced into the list of grid-values of Table 4.2. When a grid is created for a single PANEL, an 8 represents grid-cells which exceed the lower limit of the target device's screen. For each insertion point, the algorithm that chooses the WIDGETS' insertion point checks if a placement at the corresponding location is possible without exceeding the lower limit of the screen. Therefore, encountering grid-cells that are set to 8 when placing a WIDGET into a grid, is only tolerated if this WIDGET does not fit into the remaining free grid-cells.

The computable characteristics mentioned in 2.1.1 can be a valuable assistance for programs that automatically generate GUIs. Nevertheless, these characteristics can only be used to make decisions concerning widget placement if each widget of a screen initially has a predefined size. In case of the UCP framework, the size of certain widgets – more precisely of panels – depends on the collocation of their contained widgets and has to be calculated accordingly afterwards. Consequently, the contained widgets have to be arranged at a time, when size and arrangement of further potentially existing widgets are unknown. Furthermore, it is even unknown where the current collocation is finally placed on the screen. Therefore, it is hard to say how its widgets have to be arranged. These facts complicate an integration of aesthetic characteristics within the UCP framework and especially in its automated placement strategy for arranging widgets. Currently, the placement strategy of the UCP framework considers two of the aesthetic characteristics described in 2.1.1:

- **Measure of screen proportion:** A predefined proportion is considered.
- **Measure of screen simplicity:** The set of possible insertion points results in a low amount of horizontal and vertical alignment points.

In most cases, the mathematical relationships described in 2.1.2, can hardly be integrated in the UCP's placement strategy. The relevant attributes required to apply most of the mathematical relationships are defined in a Cascading Style Sheet(e.g. widget height and width, padding, border-width, etc.), and the placement strategy has no influence on that data. Still, upper and left justification are achieved through the particular choice of possible insertion points and a standard alignment of NORTH_WEST for each widget.

The Layout Module aims to create a layout that requires a minimum amount of screen space. More precisely, it has the purpose to create a layout that completely fits into a predefined screen. Contrary to the placement strategies mentioned in Section 2.2, the free screen area remains entirely available for further widget placement. Furthermore, horizontal as well as vertical screen extents are considered when searching possible insertion points.

Since each existing system offers the possibility for optimisation, the subsequent paragraphs present several further approaches that could improve the performance of the Layout Module.

Optimisation of an Arranged Grid

After a PANEL is processed by the Layout Module, the layout-data of its children are set and the corresponding grid is filled accordingly. The created layout represents a first attempt that results in a layout that fits into the target device's screen. A possibly existing free area within this grid offers the option for optimisation. If a child WIDGET fits into this free area, a rearrangement of the WIDGETS may lead to a smaller amount of the used area.

Furthermore, the collocation within an automatically arranged PANEL can be optimised according to the aesthetic characteristics mentioned in 2.1.1. A setup could be implemented that allows the designer to choose the aesthetic characteristics to be considered. Moreover, the importance of the chosen characteristics could be set as well by assigning a weight to them.

Note, that this optimisation can only be processed for PANELS whose child WIDGETS do not have a mandatory order.

Optimisation of the Placement Strategy for Child Widgets with a Mandatory Order

The choice of insertion points for child WIDGETS that have a mandatory order is described in 4.5.3.1. The algorithm can be optimised by allowing a further possible insertion point if it does not harm the WIDGETS' order. This insertion point is depicted in Figure 5.1 (encircled insertion point). A further WIDGET can be placed at this insertion point without breaking the WIDGETS' order if it fits into the free area that is surrounded by the dashed line. Furthermore, a possibly remaining forbidden area (e.g. the area under the first WIDGET in Figure 5.1) can be allocated to the WIDGET above. Instead of wasting this space, the height of the WIDGET above can be adapted to the available height or the WIDGET can be vertically centered.

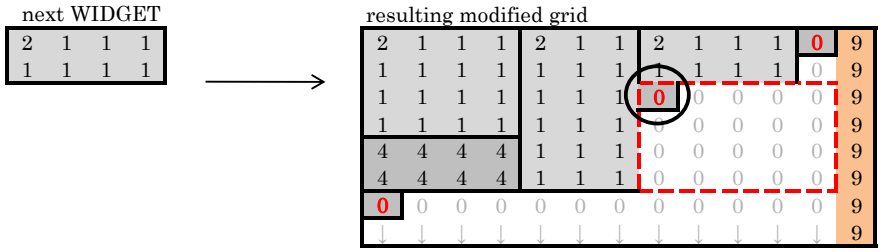


Figure 5.1: Optimised placement strategy for child WIDGETS with a mandatory order.

Placement Strategy for InputWidgets and OutputWidgets

Currently, INPUTWIDGETS and OUTPUTWIDGETS get their layout-data during the generation process of the structural UI model according to the corresponding transformation rule. Children of a LISTWIDGET represent an exception to this rule (see 4.5.3.1). If no layout-data for INPUTWIDGETS and OUTPUTWIDGETS is defined by the transformation rules, the layout-data of these WIDGETS have to be set automatically as well. For this purpose, knowledge how these WIDGETS belong to each other would be necessary. A solution could be one of the approaches for placement-strategies mentioned in Section 2.2.

Calculation of the Widget's Width

Currently, the text to be displayed by certain INPUTWIDGETS and OUTPUTWIDGETS is not available when size calculation is performed by the Layout Module. If this information were available at this time, the width of the single WIDGETS could be calculated according to this text. The default width, which is stored in the properties-file for the different WIDGETS, would not be required any more. For example, the width of each LABEL contained by a PANEL could be obtained according to this text. Furthermore, in order to provide horizontal uniformity, the width of each contained LABEL could be adapted to the broadest contained LABEL.

List of Figures

List of Tables

Bibliography

[BHLV94] François Bodart, Anne-Marie Hennebert, Jean-Marie Leheureux, and Jean Vanderdonckt. Towards a dynamic strategy for computer-aided visual placement. In *AVI '94: Proceedings of the workshop on Advanced visual interfaces*, pages 78–87, New York, NY, USA, 1994. ACM. 3, 8, 10, 11, 12, 13, 66

[GLW06] Krzysztof Z. Gajos, Jing Jing Long, and Daniel S. Weld. Automatically generating custom user interfaces for users with physical disabilities. In *Assets '06: Proceedings of the 8th international ACM SIGACCESS conference on Computers and accessibility*, pages 243–244, New York, NY, USA, 2006. ACM. 3

[GW04] Krzysztof Gajos and Daniel S. Weld. SUPPLE: Automatically generating user interfaces. In *Proceedings of the 9th International Conference on Intelligent User Interface (IUI '04)*, pages 93–100, New York, NY, USA, 2004. ACM Press. 3

[GWW07] Krzysztof Z. Gajos, Jacob O. Wobbrock, and Daniel S. Weld. Automatically generating user interfaces adapted to users' motor and vision capabilities. In *UIST '07: Proceedings of the 20th annual ACM symposium on User interface software and technology*, pages 231–240, New York, NY, USA, 2007. ACM. 3

[KBFK08] Sevan Kavaldjian, Cristian Bogdan, Jürgen Falb, and Hermann Kaindl. Transforming discourse models to structural user interface models. In *Models in Software Engineering, LNCS 5002*, volume 5002/2008, pages 77–88. Springer, Berlin / Heidelberg, 2008. 16, 19, 28

[KF93] Won Chul Kim and James D. Foley. Providing high-level control and expert assistance in the user interface presentation design. In *INTERCHI '93: Proceedings of the INTERCHI '93 conference on Human factors in computing systems*, pages 430–437, Amsterdam, The Netherlands, The Netherlands, 1993. IOS Press. 4, 10, 13, 14, 15, 66

[Kim93] Won Chul Kim. *Knowledge-based framework for an automated user interface presentation design tool*. PhD thesis, Washington, DC, USA, 1993. 14

[KRP+10] S. Kavaldjian, D. Raneburger, R. Popp, M. Leitner, J. Falb, and H. Kaindl. Automated optimization of user interfaces for screens with limited resolution. In *Proceedings of the MDDAUI'10 Workshop on Model Driven Development of Advanced User Interfaces*, 2010. 17, 18, 28, 29, 31, 66

[Mar92] Aaron Marcus. *Graphic design for electronic documents and user interfaces*. ACM, New York, NY, USA, 1992. 7

[NTB03] David Chek Ling Ngo, Lian Seng Teo, and John G. Byrne. Modelling interface aesthetics. *Information Sciences*, 152:25 – 46, 2003. 4

[Pop09] Roman Popp. Defining communication in soa based on discourse models. In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 829–830, New York, NY, USA, 2009. ACM. 16

[Ran08] David Raneburger. Automated graphical user interface generation based on an abstract user interface specification. Master's thesis, Technical University of Vienna, 2008. 16, 17, 18, 19

[Sea93] A. Sears. Layout appropriateness: A metric for evaluating user interface widget layout. volume 19, pages 707–719, Piscataway, NJ, USA, 1993. IEEE Press. 3, 11

[VG94] Jean Vanderdonckt and Xavier Gillo. Visual techniques for traditional and multimedia layouts. In *AVI '94: Proceedings of the workshop on Advanced visual interfaces*, pages 95–104, New York, NY, USA, 1994. ACM. 4

[YK09] Yeonsoo Yang and Scott R. Klemmer. Aesthetics matter: leveraging design heuristics to synthesize visually satisfying handheld interfaces. In *CHI '09: Proceedings of the 27th international conference extended abstracts on Human factors in computing systems*, pages 4183–4188, New York, NY, USA, 2009. ACM. 3

[YNK09] Takuto Yanagida, Hidetoshi Nonaka, and Masahito Kurihara. Personalizing graphical user interfaces on flexible widget layout. In *EICS '09: Proceedings of the 1st ACM SIGCHI symposium on Engineering interactive computing systems*, pages 255–264, New York, NY, USA, 2009. ACM. 3