Copyright Statement

The digital copy of this thesis is protected by the Copyright Act 1994 (New Zealand).

This thesis may be consulted by you, provided you comply with the provisions of the Act and the following conditions of use:

- Any use you make of these documents or images must be for research or private study purposes only, and you may not make them available to any other person.
- Authors control the copyright of their thesis. You will recognise the author's right to be identified as the author of this thesis, and due acknowledgement will be made to the author where appropriate.
- You will obtain the author's permission before publishing any material from their thesis.

To request permissions please use the Feedback form on our webpage. http://researchspace.auckland.ac.nz/feedback

General copyright and disclaimer

In addition to the above conditions, authors give their consent for the digital copy of their work to be used subject to the conditions specified on the Library Thesis Consent Form and Deposit Licence.
Leveraging Constraint-based Layout for User Interface Customization

Clemens Zeidler
May 2014

Supervisors:
Christof Lutteroth
Gerald Weber

A thesis submitted in partial fulfillment of the requirements of Doctor of Philosophy in Science
Abstract

In this thesis the usage of constraint-based layout in the field of user interface (UI) customization is explored. The constraint-based layout model is very powerful and can be used for many different kinds of layouts. However, it is also more complex than most other layout models, which makes it challenging for users to create sound layouts, i.e., layouts that are solvable and do not allow layout items to overlap. To leverage the constraint-based layout model for UI customization, we present methods that enable users to create and edit constraint-based layouts in a sound manner.

To motivate why the constraint-based layout model is used in this work, it is compared to other layout models. In a user evaluation the usability of the constraint-based layout model was compared to the grid-bag layout model, which is also very powerful and likely the most commonly used model. Another evaluation investigated the aesthetic aspects of how available space in a layout should be best distributed among widgets.

The first system for UI customization that we analyzed is Stack & Tile. It allows the user to stack and tile windows from a traditional desktop system into groups. Window groups are specified using the constraint-based layout model. A user evaluation showed that Stack & Tile substantially improves the work with multiple windows. Furthermore, we explored in a web survey how and if Stack & Tile is actually used by real users.

The second system for UI customization targets the editing of constraint-based layouts at application runtime. A set of edit operations is developed that makes it easy to edit constraint-based layout in a sound manner. To evaluate these edit operations, we implemented them in a graphical user interface (GUI) builder, the Auckland Layout Editor (ALE). In a user study participants performed significantly faster for layout creation and layout editing tasks compared to other layout builders.

Another contribution of this thesis is a new way to describe constraint-based layouts using a formal algebraic description for layout specifications. This algebra can be used to describe sound layout operations formally, i.e., operations that keep a layout solvable and non-overlapping. The edit operations used in Stack & Tile and ALE are then mapped to these algebraic operations.

To investigate if and where UI customization is useful, a user evaluation was conducted.
This study covered layout as well as functional customization. For this a functional customization prototype was developed that allows changing the functionality and behavior of an application. The evaluation showed that users are keen to customize applications to their needs.
At this point I thank everybody who made it possible for me to finish this thesis. First of all I thank my supervisors, Christof Lutteroth and Gerald Weber for their support and help. Their good advice and directions helped me heaps to keep me on track and finish my thesis. Moreover, they encouraged and aided me to successfully publish my work at various conferences. My work has been funded by the Royal Society of New Zealand under the Marsden grant UOA0919.

During my time as a PhD student I also worked with Wolfgang Sturzlenger, Johannes Muller and Alexander Gavruskin. I thank Wolfgang for the fruitful discussions to improve ALE. He certainly played a big part in getting the work on ALE published. Kudos to Johannes for introducing me into the art of R and giving me a superb example on how to work on a topic systematically. Furthermore, thanks for his countless help here and there. Alexander gave me many tips and much advice on the algebra chapter. Thank you for this. Thanks to Rishika Mukerjee and Keerthana Puppala for helping design and perform the comparison of ALE with a grid-bag GUI builder.

There are also many people who made my life as a PhD student fun and more pleasant. Many thanks to: Moiz, Noreen and Sudheendra for sharing the office; Stefan, Chris, Adam, Jamie, Glen and Ali for sharing the lunch; Paul, Denise and Tille for Wizard; Wei and Muna for the extended coffee breaks; Markus for the music; Rita for her billiard skills; Wenx for her social skills; the guys from volleyball for the side-rolls and the karate club for the side-kicks.

Last but not least, I thank my family who supported me to move to the other side of the world!
Contents

1 Introduction 1
   1.1 Research Questions ........................................... 3
   1.2 Overview ....................................................... 5

2 Constraint-based Layout 7
   2.1 Layout Management ............................................ 7
   2.2 Auckland Layout Model ........................................ 10
   2.3 Constraint Solving ............................................ 12

3 Usability of the Constraint-based Layout Model 17
   3.1 Grid-bag and Constraint-Based Layouts ......................... 18
   3.2 Related Work .................................................. 22
   3.3 Methodology .................................................... 23
   3.4 Results ......................................................... 27
   3.5 Discussion ..................................................... 32
   3.6 Conclusion ..................................................... 34

4 Layout Aesthetics 37
   4.1 Related Work .................................................. 39
   4.2 Aesthetic Problems of Constraint-Based Layout ................ 40
   4.3 Methodology .................................................... 42
   4.4 Results and Discussion ....................................... 45
   4.5 Conclusion ..................................................... 47

5 UI Customization in the Large: Stack & Tile 49
   5.1 Window Management with Stack & Tile ........................ 50
   5.2 Stack & Tile Operations ....................................... 53
   5.3 Related Work .................................................. 57
   5.4 Experimental Evaluation ..................................... 59
   5.5 Web-based Survey ............................................. 71
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.6</td>
<td>Conclusion</td>
<td>75</td>
</tr>
<tr>
<td>6</td>
<td>UI Customization in the Small: Layout Customization</td>
<td>77</td>
</tr>
<tr>
<td>6.1</td>
<td>Requirements for Layout Editing</td>
<td>78</td>
</tr>
<tr>
<td>6.2</td>
<td>Related Work</td>
<td>80</td>
</tr>
<tr>
<td>6.3</td>
<td>ALE Layout Edit Operations</td>
<td>82</td>
</tr>
<tr>
<td>6.4</td>
<td>Non-Overlapping Layouts</td>
<td>87</td>
</tr>
<tr>
<td>6.5</td>
<td>Evaluation</td>
<td>90</td>
</tr>
<tr>
<td>6.6</td>
<td>Conclusion</td>
<td>96</td>
</tr>
<tr>
<td>7</td>
<td>Algebra for UI Customization</td>
<td>99</td>
</tr>
<tr>
<td>7.1</td>
<td>Related Work</td>
<td>100</td>
</tr>
<tr>
<td>7.2</td>
<td>Tiling Algebra</td>
<td>100</td>
</tr>
<tr>
<td>7.3</td>
<td>Layout Edit Operations</td>
<td>110</td>
</tr>
<tr>
<td>7.4</td>
<td>Layout Editing Process</td>
<td>114</td>
</tr>
<tr>
<td>7.5</td>
<td>Applications</td>
<td>117</td>
</tr>
<tr>
<td>7.6</td>
<td>Conclusion</td>
<td>121</td>
</tr>
<tr>
<td>8</td>
<td>Outlook: An Evaluation of Advanced User Interface Customization</td>
<td>123</td>
</tr>
<tr>
<td>8.1</td>
<td>Related Work</td>
<td>124</td>
</tr>
<tr>
<td>8.2</td>
<td>Layout Customization</td>
<td>125</td>
</tr>
<tr>
<td>8.3</td>
<td>Functional Customization</td>
<td>126</td>
</tr>
<tr>
<td>8.4</td>
<td>Evaluation</td>
<td>129</td>
</tr>
<tr>
<td>8.5</td>
<td>Results and Discussion</td>
<td>136</td>
</tr>
<tr>
<td>8.6</td>
<td>Conclusion</td>
<td>142</td>
</tr>
<tr>
<td>9</td>
<td>Conclusion</td>
<td>145</td>
</tr>
<tr>
<td>9.1</td>
<td>Achievements</td>
<td>145</td>
</tr>
<tr>
<td>9.2</td>
<td>Future Directions</td>
<td>148</td>
</tr>
</tbody>
</table>
Co-Authorship Form

This form is to accompany the submission of any PhD that contains research reported in published or unpublished co-authored work. Please include one copy of this form for each co-authored work. Completed forms should be included in all copies of your thesis submitted for examination and library deposit (including digital deposit), following your thesis Abstract.

Please indicate the chapter/section/pages of this thesis that are extracted from a co-authored work and give the title and publication details or details of submission of the co-authored work.


Nature of contribution by PhD candidate: Study design, experimental work and editing.

Extent of contribution by PhD candidate (%): 80%

CO-AUTHORS

<table>
<thead>
<tr>
<th>Name</th>
<th>Nature of Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Christof Luttheroth</td>
<td>Study design and editing.</td>
</tr>
<tr>
<td>Gerald Weber</td>
<td>Study design and editing.</td>
</tr>
</tbody>
</table>

Certification by Co-Authors

The undersigned hereby certify that:

- the above statement correctly reflects the nature and extent of the PhD candidate's contribution to this work, and the nature of the contribution of each of the co-authors; and
- in cases where the PhD candidate was the lead author of the work that the candidate wrote the text.

<table>
<thead>
<tr>
<th>Name</th>
<th>Signature</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Christof Luttheroth</td>
<td>[Signature]</td>
<td>19/7/13</td>
</tr>
<tr>
<td>Gerald Weber</td>
<td>[Signature]</td>
<td>19/7/13</td>
</tr>
</tbody>
</table>
This form is to accompany the submission of any PhD that contains research reported in published or unpublished co-authored work. **Please include one copy of this form for each co-authored work.** Completed forms should be included in all copies of your thesis submitted for examination and library deposit (including digital deposit), following your thesis Abstract.

Please indicate the chapter/section/pages of this thesis that are extracted from a co-authored work and give the title and publication details or details of submission of the co-authored work.


<table>
<thead>
<tr>
<th>Nature of contribution by PhD candidate</th>
<th>Study design, experimental work and editing.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extent of contribution by PhD candidate (%)</td>
<td>80%</td>
</tr>
</tbody>
</table>

### CO-AUTHORS

<table>
<thead>
<tr>
<th>Name</th>
<th>Nature of Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Christof Lutteroth</td>
<td>Study design and editing.</td>
</tr>
<tr>
<td>Gerald Weber</td>
<td>Study design and editing.</td>
</tr>
<tr>
<td>Johannes Mueller</td>
<td>Study design, experimental work and editing.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Certification by Co-Authors

The undersigned hereby certify that:

- the above statement correctly reflects the nature and extent of the PhD candidate’s contribution to this work, and the nature of the contribution of each of the co-authors; and
- in cases where the PhD candidate was the lead author of the work that the candidate wrote the text.

<table>
<thead>
<tr>
<th>Name</th>
<th>Signature</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Johannes Mueller</td>
<td>[Signature]</td>
<td>07/10/2013</td>
</tr>
<tr>
<td>Christof Lutteroth</td>
<td>[Signature]</td>
<td>19/11/13</td>
</tr>
<tr>
<td>Gerald Weber</td>
<td>[Signature]</td>
<td>19/11/13</td>
</tr>
</tbody>
</table>
This form is to accompany the submission of any PhD that contains research reported in published or unpublished co-authored work. Please include one copy of this form for each co-authored work. Completed forms should be included in all copies of your thesis submitted for examination and library deposit (including digital deposit), following your thesis Abstract.

Please indicate the chapter/section/pages of this thesis that are extracted from a co-authored work and give the title and publication details or details of submission of the co-authored work.

Chapter "UI Customization in the Large: Stack & Tile" includes work from the paper "C. Zeidler, C. Lutteroth, and G. Weber. An evaluation of stacking and tiling features within the traditional desktop metaphor. INTERACT’13, to appear."

<table>
<thead>
<tr>
<th>Nature of contribution by PhD candidate</th>
<th>Study design, implementation, experimental work and editing.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extent of contribution by PhD candidate (%)</td>
<td>80</td>
</tr>
</tbody>
</table>

### CO-AUTHORS

<table>
<thead>
<tr>
<th>Name</th>
<th>Nature of Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Christof Lutteroth</td>
<td>Study design and editing.</td>
</tr>
<tr>
<td>Gerald Weber</td>
<td>Study design and editing.</td>
</tr>
</tbody>
</table>

**Certification by Co-Authors**

The undersigned hereby certify that:

1. the above statement correctly reflects the nature and extent of the PhD candidate’s contribution to this work, and the nature of the contribution of each of the co-authors; and
2. in cases where the PhD candidate was the lead author of the work that the candidate wrote the text.

<table>
<thead>
<tr>
<th>Name</th>
<th>Signature</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Christof Lutteroth</td>
<td>Lutteroth</td>
<td>19/7/13</td>
</tr>
<tr>
<td>Gerald Weber</td>
<td>Weber</td>
<td>19/7/13</td>
</tr>
</tbody>
</table>
This form is to accompany the submission of any PhD that contains published or unpublished co-authored work. **Please include one copy of this form for each co-authored work.** Completed forms should be included in all copies of your thesis submitted for examination and library deposit (including digital deposit), following your thesis Abstract.

Please indicate the chapter/section/pages of this thesis that are extracted from a co-authored work and give the title and publication details or details of submission of the co-authored work.


| Nature of contribution by PhD candidate | Study design, experimental work and editing. |
| Extent of contribution by PhD candidate (%) | 80 |

### CO-AUTHORS

<table>
<thead>
<tr>
<th>Name</th>
<th>Nature of Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Christof Lutteroth</td>
<td>Study design and editing.</td>
</tr>
<tr>
<td>Gerald Weber</td>
<td>Study design and editing.</td>
</tr>
<tr>
<td>Wolfgang Stuerzlinger</td>
<td>Study design and editing.</td>
</tr>
</tbody>
</table>

**Certification by Co-Authors**

The undersigned hereby certify that:
- the above statement correctly reflects the nature and extent of the PhD candidate's contribution to this work, and the nature of the contribution of each of the co-authors; and
- in cases where the PhD candidate was the lead author of the work that the candidate wrote the text.

<table>
<thead>
<tr>
<th>Name</th>
<th>Signature</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Christof Lutteroth</td>
<td>[Signature]</td>
<td>19/7/13</td>
</tr>
<tr>
<td>Wolfgang Stuerzlinger</td>
<td>[Signature]</td>
<td>July 19, 2013</td>
</tr>
</tbody>
</table>
This form is to accompany the submission of any PhD that contains research reported in published or unpublished co-authored work. Please Include one copy of this form for each co-authored work. Completed forms should be included in all copies of your thesis submitted for examination and library deposit (including digital deposit), following your thesis Abstract.

Please indicate the chapter/section/pages of this thesis that are extracted from a co-authored work and give the title and publication details or details of submission of the co-authored work.


<table>
<thead>
<tr>
<th>Nature of contribution by PhD candidate</th>
<th>Conceptional Ideas, Implementation and editing.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extent of contribution by PhD candidate (%)</td>
<td>80</td>
</tr>
</tbody>
</table>
Co-Authorship Form

This form is to accompany the submission of any PhD that contains research reported in published or unpublished co-authored work. Please include one copy of this form for each co-authored work. Completed forms should be included in all copies of your thesis submitted for examination and library deposit (including digital deposit), following your thesis Abstract.

Please indicate the chapter/section/pages of this thesis that are extracted from a co-authored work and give the title and publication details or details of submission of the co-authored work.


<table>
<thead>
<tr>
<th>Nature of contribution by PhD candidate</th>
<th>Study design and editing.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extent of contribution by PhD candidate (%)</td>
<td>80%</td>
</tr>
</tbody>
</table>

**CO-AUTHORS**

<table>
<thead>
<tr>
<th>Name</th>
<th>Nature of Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Christof Lutteroth</td>
<td>Study design and editing.</td>
</tr>
<tr>
<td>Gerald Weber</td>
<td>Study design and editing.</td>
</tr>
</tbody>
</table>

**Certification by Co-Authors**

The undersigned hereby certify that:
- the above statement correctly reflects the nature and extent of the PhD candidate's contribution to this work, and the nature of the contribution of each of the co-authors; and
- in cases where the PhD candidate was the lead author of the work that the candidate wrote the text.

<table>
<thead>
<tr>
<th>Name</th>
<th>Signature</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Christof Lutteroth</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
[Signature] | 30/10/13   |
| Gerald Weber     | 
[Signature] | 2013-10-22 |
Introduction

Since the modern human evolved people are using various kinds of tools. Before, in the very early times only simple tools like flint hand axes or sewing needles made of bones and wood were used, much more sophisticated tools have been developed since then. Now there are tools and machines for many different purposes. Some tools like knives can be used for a lot of different tasks while other tools, for example a typewriter, just serve one special purpose. Different people have different expectations of the tools and machines they are using, e.g., left handed people may prefer left handed scissors. However, some tools are so complex that a personalized version cannot be deployed for a particular user and thus these tools have to be customizable. For example, in a car one can change the seat and steering wheel positions, and the navigation and entertainment systems.

Probably one of the most remarkable and most flexible machines these days are computers. There are various tools (applications) for different tasks. Similar to physical tools, programmers have to design an application in a way so that it suits a large number of users. It is an old and well-known problem that applications may need to be customized to fit different needs [8, 43, 79]. Different users naturally have different requirements, e.g., depending on their role, preferred workflow, expertise, visual acuity and motor skills, and the devices they use. For example, professional users may want to streamline a user interface (UI) in order to make work more efficient. Developers cannot anticipate all the users’ needs, and changing a UI after release may not be possible or maybe too expensive. Giving users the ability to do their own customization can solve this problem and can
lead to a better user experience.

![Customization Power Diagram]

Figure 1.1: When acquiring new skills in customizing a system, users face steep learning slopes. There is a first barrier for normal users when they have to change settings of an application. The next barrier is when more advanced users face problems that can only be solved by programmers [79].

Already early work enabled users of different skill levels to customize a software system to their needs. MacLean et al. stated that there are steep slopes for skill acquisition when it comes to tailoring or customizing a system [79] (see Figure 1.1). Their approach tries to make it easier for users to learn customization skills by allowing users of different skill levels to customize a system. Application functions can be assigned to buttons, and normal users can rearrange the buttons on the desktop to simplify a certain task. Advanced users can edit parameters and attributes of the functions, while programmers can use a programming language to further customize or create completely new buttons. In this work we focus on advanced users that spend a significant amount of time with a system and are knowledgeable enough to do meaningful customizations. This could be a user who is familiar enough with an application to realize that the application may not be optimal for a certain workflow.

Various reasons have been identified that can trigger customization [78]. For example, external events may require users to customize their system to adapt to a new workflow. Furthermore, social factors can trigger customization, e.g., friends or colleagues may suggest a customization. Also software updates, the need to fix issues, or internal factors such as spare time to tinker with software can trigger customization. Another reason is that some applications are “bloated”, i.e., have more functionality than needed by the user [82]. In this case the user may like to simplify the UI by removing unused functionality. Users are more likely to customize if they are made aware of the customization features [8]. However, customizing a system by removing functionality can also lead to a loss of awareness of that functionality, potentially decreasing the performance for new tasks [33].
Most previous work allows customizing menus or toolbars, i.e., the user is able to rearrange, add or remove items within a menu or toolbar layout [24, 34, 39, 43]. However, the user cannot change the UI layout in an arbitrary way. This work tries to fill this gap and targets the customization of UI layout in general.

One of the most powerful layout models is the constraint-based layout model [5, 75]. Almost all graphical user interface (GUI) layouts can be specified using this layout model. To do so linear constraints are typically used to describe relations between the items in a layout. While the constraint-based layout model shows many promising theoretical advantages, it is unclear whether these advantages can be utilized when customizing constraint-based layouts. Editing a constraint-based layout can be quite challenging because the user typically has to understand the complex underlying constraint-based layout model. Furthermore, the underlying layout model does not prevent users from specifying conflicting constraints, which can lead to unsolvable layout specifications. More subtle problems can emerge when a layout is underspecified and layout problems are not directly visible, e.g., when two widgets are partially overlapping each other.

In this work we investigate suitable solutions that allow users to customize constraint-based layouts. An important goal is to enable users to do GUI customization in a sound manner, i.e., the application and the layout must stay in a usable state. Hence the customization framework must only provide operations that are safe to use. While these operations must be sound, they must also be versatile enough to create all interesting layout configurations. They should be applicable to a wide range of applications and not just tailored to special use-cases.

1.1 Research Questions

The main research question in this thesis is: Can the constraint-based layout model be leveraged for UI customization? This can be answered by investigating the following more specific questions.

Q1: Is the constraint-based layout model a suitable model for UI customization? To target this question the constraint-based layout model is compared to the popular grid-bag layout model. Furthermore, aesthetic aspects for different constraint solving strategies are analyzed.

The grid-bag layout model is widely used in various applications, e.g., in GUI frameworks and for HTML tables. Hence it is an important question how the usability of the constraint-based layout model compares to this layout model. In a user evaluation we found that specifying a new layout can be done faster using the grid-bag layout model, but editing an existing layout specification is faster using the constraint-based layout
Moreover, the constraint-based layout model was preferred over the grid-bag layout model.

In general a resizable layout is not displayed at its optimal size and available space must be distributed among the items in the layout. Three different methods for distributing available space were analyzed in a user evaluation. We found that the use of a quadratic objective function in the constraint solver leads to aesthetically pleasant results for small and for large layout sizes [113].

Q2: How can a constraint-based layout be edited in a sound way, leaving the system in a usable state? The first constraint-based customization system we developed is Stack & Tile, which is an extension of a traditional window manager. Stack & Tile makes it possible to stack windows on top of one another or beside each other, forming groups. The user can easily customize the constraint-based layout of a Stack & Tile group by holding a special Stack & Tile key and dragging windows beside or on top of each another. A group can contain windows of different applications, and thus Stack & Tile can, for example, be used to group windows by tasks.

When working on Stack & Tile it became clear that, in order to ensure sound Stack & Tile groups, it is important that tiled windows in a group do not overlap each other, and that the resulting constraint-based specification is solvable. If a Stack & Tile group is active, stacked windows are either completely visible or completely occluded, and tiled windows do not overlap each other.

Stack & Tile as a customization system supports only simple edit operations (stacking, tiling, removing) and is limited to simple arrangements of windows. To investigate more sophisticated edit operations we developed a constraint-based GUI builder called the Auckland Layout Editor (ALE) [112]. We designed new drag and drop edit operations that allow a user to rearrange the widgets in a GUI layout fairly easily. Similar to Stack & Tile these operations hide the complexity of the constraint-based layout model. In order to generate sound layouts, again it is important that the edit operations leave the layout solvable and non-overlapping. We integrated ALE’s edit operations into a customization framework that allows users to edit constraint-based GUIs at application runtime.

Q3: Can UI customization systems that use the constraint-based layout model have a good usability? We evaluated the usability of Stack & Tile and found that it performs much better for multi-window tasks than traditional window management. Also switching between tasks that involve multiple windows can be done significantly quicker using Stack & Tile. Stack & Tile was released over two years ago as open-source software. In a web survey we found that people like Stack & Tile and actually use it for a variety of applications [115].
1.2 Overview

We also evaluated the usability of ALE, as its edit operations can also be used for layout customization. In two empirical studies we compared ALE against two major GUI builders: Visual Studio, which supports a grid-bag layout model, and Xcode, which supports the constraint-based layout model. The evaluation shows that ALE’s edit operations makes it easier to create new layouts as well as to edit existing layouts. In both cases we found that ALE performed significantly better and users enjoyed ALE more [111].

Q4: How can systems for constraint-based layout editing be described in a formal way? While Stack & Tile and ALE target different types of UI customization, they have some strong similarities. Both systems support layouts that contain layout items which are connected and adjacent to each other. Moreover, in both systems the layout must be solvable and non-overlapping. To describe the layout editing operations of Stack & Tile and ALE formally, an algebra was developed that can describe layout specifications, and includes edit operations for modifying these specifications. The algebra not only makes it possible to describe the edit operations more consistently, but also ensures soundness of the layout specifications. The Stack & Tile and ALE operations were both formalized using the algebra.

Q5: Are users able to use advanced UI customization systems and would they do so if such systems were available? It seems useful to provide powerful, advanced customization systems to end-users. However, it is not clear yet whether users are actually able to use them, and if they would do so in practice. In a user study we investigated two such advanced customization systems, which allow users to customize an application during runtime: the layout customization system based on ALE’s edit operations, and a prototype for functional customization that allows users to add, remove and modify the functionality of an application. The functional customization prototype gives an outlook on a different part of advanced UI customization that is not mainly covered in this thesis. Both customization systems were presented to participants, who were then asked to perform customization tasks and fill out a questionnaire. The study indicates that users with a certain level of experience are able to use the advanced customization systems, and would like to use them in practice [114].

1.2 Overview

This thesis is structured into the following chapters. Chapter 2 gives an overview of different layout models and describes the constraint-based layout model in detail. Chapter 3 compares the usability of the constraint-based layout model and the grid-bag layout model in a user evaluation, addressing Q1. Chapter 4 discusses how available space in a
layout is best distributed among the layout items and how this affects layout aesthetics, also addressing Q1. Chapter 5 describes the Stack & Tile system and how it leverages the constraint-based layout model (Q2). A user evaluation shows that Stack & Tile performs better than a traditional window manager (Q3). Moreover, in a web survey the actual usage of Stack & Tile is analyzed. Chapter 6 covers layout customization. A set of edit operations is presented that operates on constraint-based layouts (Q2). To evaluate these operations, the constraint-based Auckland Layout Editor (ALE) is introduced (Q3). Chapter 7 presents an algebra that can describe systems such as Stack & Tile and ALE (Q4). Chapter 8 addresses the question whether users are able and willing to use advanced customization systems for layout and functionality (Q5).
This chapter gives an introduction into different methods for GUI layout management (Section 2.1). While these methods are widely used in the field of user interfaces in general they can also be used for other applications where rectangular items need to be arranged in a 2-dimensional area. In this thesis the powerful constraint-based layout model is chosen to explore GUI customization. The Auckland Layout Model (ALM) is such a constraint-based layout model and the version used in this thesis is described in Section 2.2. How a constraint system of linear hard and soft-constraints can be solved is described in Section 2.3.

2.1 Layout Management

In early GUI frameworks, widgets such as buttons or text views had to be placed manually at a fixed position and with a fixed size, e.g., the Microsoft Foundation Class library (MFC) uses such an approach. This can become very tedious as soon as new widgets have to be inserted or the layout of existing widgets has to be modified. Furthermore, it is cumbersome to implement GUIs that are resizable. Modern GUI frameworks solve this problem by offering layout managers that allow developers to position the widgets in a user interface more abstractly.

A layout manager can automatically adjust a layout what has certain advantages. For example, when the user resizes a window the layout manager repositions GUI items
dynamically to adapt the layout to the new size. Rearranging and modifying a GUI can become much easier for the designer because individual widgets do not have to be rearranged manually. Another case where layout management becomes helpful is font changes or a change of the application language, e.g., from English to German, or even a change of the reading direction. Generally, in these cases the displayed text will change its size and thus requires the rearrangement of GUI items. Using a static approach makes it error prone for the developer to handle such cases. Moreover, the use of a layout manager often leads to more consistent GUIs since it can make sure that the widgets are well aligned and there is a consistent spacing between adjacent widgets. Some of the most prominent GUI frameworks that provide layout managers are Qt, Java AWT, Cocoa, Windows Forms, GTK+ and wxWidgets.

Anything that can be placed into a layout is called a layout item. The most important layout items are widgets, e.g., buttons or text labels. Other important layout items are spacers, which are invisible and can be placed between other layout items. A spacer can occupy a fixed amount of space or can act like a spring to push other layout items aside. In this way, a spacer can be used to refine a layout and give it the desired shape. In order to create nested layouts it is important to have an item that can hold another layout. This can easily be achieved by treating a layout as a special layout item. In the following we assume that layout items are rectangular.

How a layout manager lays out the layout items depends on the used layout model. Most layout managers provide a broad set of layout models for different types of layout problems, e.g., group layout model, grid layout model or flow layout model. Usually these special layout models can be combined by creating nested layouts. The most common layout models are described briefly in the following sections.

### 2.1.1 Group Layout Model

A group layout is a simple 1-dimensional layout that can hold items side by side in a single row or column. There are two main variants of this layout, a horizontal group layout that can hold a row of items and a vertical group layout that can hold a column of items.

By nesting horizontal and vertical group layouts many useful layout configurations can be created. However, this type of layout is not sufficient for more complex layouts, e.g., a link between layout items in two different group layouts is not possible.

---

2.1 Layout Management

2.1.2 Grid (Bag) Layout Model

Some of the shortcomings of a group layout model can be avoided by using a grid layout model, also known as a table layout model. Here, layout items can be placed in a 2-dimensional table. A layout item can occupy more than one cell in the grid what makes it possible to create complex layouts. Furthermore, it is possible to create a link between items not directly adjacent, e.g., by placing them in the same row or column. This makes the grid layout reasonably flexible and powerful. However, layout items are forced to stay in a fixed grid structure what may not be desired (see also Figure 2.1).

The grid layout model can be tuned by giving the rows and columns special weightings. This is useful in specifying which row and column should use more space compared to the other rows and columns.

2.1.3 Flow Layout Model

A flow layout is basically a horizontal group layout that can span over multiple rows if items do not fit into one row. This is comparable with a line of text in a word processor: if the end of the line is reached, the text is wrapped into the next line. An example of a flow layout is a button bar that becomes a multi-line button bar in case the window becomes smaller than the button bar width.

2.1.4 Constraint-Based Layout Model

In a constraint-based layout model, user interface layouts are specified mathematically as constraint problems. This makes it possible to create complex and flexible layout specifications, and calculate actual layouts using numerical constraint solving methods [5, 75, 94]. An example for a simple GUI constraint is the minimum width constraint of a widget:

\[
\text{widget}_{\text{right}} - \text{widget}_{\text{left}} \geq \text{min.}
\]

In general, there are two types of constraints: hard-constraints, which have to be satisfied, and soft-constraints, which can be violated if necessary. The way that soft-constraints are solved depends on the implementation of the constraint solver and can be used to control the final visual appearance of the layout. While in general any kind of constraints and also logical combination (AND or OR) of constraints are allowed [42], in the following only linear equality and inequality are considered.

Constraint-based layout models are naturally powerful: with the notable exception of flow layouts, many other layout models, including gridbag layout, can be reduced to constraint-based layouts [107]. However, the constraint-based layout model supports even more complex layouts. For example, in a grid layout, layout items are always aligned to
Figure 2.1: Example constraint-based layout. The behavior of the middle row is independent from the top and bottom rows when resizing. This is impossible to achieve with a single gridbag layout.

an outer fixed grid while in a constraint-based layout a layout item can be aligned relative to another layout item and so is not bound to the fix grid as depicted in Figure 2.1. Many other layout models rely on a hierarchy of nested layouts to define more complex layouts. Within nested layouts, widgets typically cannot be aligned across different levels of the hierarchy. In contrast, constraint-based layout models greatly reduce the need for nested layouts, even for visually hierarchical layouts. Constraints can align widgets that are situated in different parts of a visually hierarchical layout [75].

2.2 Auckland Layout Model

The constraint-based layout model used in this thesis is heavily based on the Auckland Layout Model (ALM) [75]. ALM is a suitable representative of constraint-based layout in general. The model used here varies slightly from the earlier presented model and a description of the model used here is given in this section.

There are various implementations of a constraint-based layout model, the Java layout class SpringLayout or the layout model of the Mac OS Cocoa API, Auto Layout, is based on constraints. In Auto Layout, the programmer can specify linear constraints of the form $y = m \cdot x + b$ and $y \geq m \cdot x + b$ between two variables $x$ and $y$. These variables could, for example, be the width or the edge of a layout item. However, since these linear constraints only contain two variables this approach is not as powerful as general constraint-based layout models such as ALM, which allow specifying more complex constraints and make it possible to create layouts in a more abstract way. For example, constraints with multiple variables or variables not connected to any layout items are not possible.

A layout consists of various types of layout items, such as widgets and nested layouts. In the following and for brevity, all kinds of layout items are referred just as widgets. Each widget has an intrinsic minimum, and preferred and maximum size. The preferred size is the size the layout item should assume if there are no other constraints for the item size. This can be illustrated with a pinched sponge which, after releasing, expands to its original or preferred size. These intrinsic sizes are inspired by the Haiku layout manager that is used in this research. The intrinsic sizes can be explicitly overridden by
2.2 Auckland Layout Model

the developer.

From the intrinsic size values for each layout item in the layout, the corresponding size values of the complete layout can be calculated, i.e., the minimum, preferred and maximum layout sizes. Similar to a single layout item, the preferred size of a whole layout is the size it should assume if there are no other constraints. Notice that in a layout of minimal size, not all layout items may have their minimal size. This is because other larger layout items may be preventing the layout from shrinking further.

Variables in a constraint are called tabstops and represent horizontal or vertical grid lines. Other frequently used names for the same concept are aligners, guides, and snap or anchor lines. Each layout defines tabstops for its four borders, so that they can also be used for alignment. Tabstops are reference counted, to automatically remove unused ones.

![Diagram of layout with tabstops](image)

Figure 2.2: Left: Widgets are by default automatically centered in their layout area (light gray). The area is surrounded by a margin and four tabstops. Here, the button has a fixed size. Right: A combo box and button are connected to the same vertical tabstop on the left. As the combo box cannot shrink further, the button is again centered in its layout area.

Within the overall layout, each widget is associated with a rectangular layout area that is surrounded by four tabstops. By default and because most layouts use margins, there is a small margin between the layout areas and its tabstops, see Figure 2.2. When inserting a widget into a layout, some constraints are automatically derived from the intrinsic sizes of a widget: a hard inequality constraint for the minimum size and soft equalities for the preferred and maximum sizes. These constraints are defined both horizontally and vertically and are managed by the associated layout area. Consequently, users do not need to manually manage constraints for intrinsic sizes. The preferred size of a widget can be fine-tuned in the properties dialog. This is comparable to changing the “weight” of a row or column in a grid-bag layout. When setting up these constraints, both margins of the layout area $A$ are taken into account, i.e., twice the margin has to be added to the intrinsic sizes (e.g., the minimum width constraint becomes: $A_{right} - A_{left} \geq min + 2 \cdot margin$).

If the size of the layout area is between a widget’s minimum and maximum size, the widget uses the whole space of the area. If the widget’s maximum size is smaller than the size of the layout area, it is by default centered in the layout area (Figure 2.2). However,
the user also has the option to align the widget to any border or corner of the layout area.

Many useful layouts can be specified by just aligning layout areas with each other. Such layouts naturally reuse tabstops for multiple layout areas and thus need no additional constraints. See for example the right of Figure 2.2, where the top tabstop of the button is reused for the bottom tabstop of the combo box. Here the preferred size soft-constraints are enough to uniquely specify the sizes of the widgets and no other constraints are needed to connect both widgets.

A layout that solely consists of layout areas connected to other layout areas or layout borders is always solvable as long as there are no outer layout boundaries, i.e., the window containing the layout is large enough. This is because the maximum size of a layout area can be violated and thus no conflicting area connections can be created.

A layout item is always connected to two horizontal and two vertical tabs. The two horizontal tabs can naturally be regarded as a row and the two vertical tabs as a column. Multiple layout items sharing the same horizontal or vertical tabs also share the same row or column, respectively. In this way there can be interruptions in a row or a column, e.g., there could be another item between two items in a row that is only connected to one or even none of the horizontal row tabs. This is not the traditional definition of rows and columns but allows a simple grouping of the generally unordered tabstop system.

### 2.3 Constraint Solving

A suitable constraint solver for user interface constraints must be able to solve the hard-constraints and must also handle soft-constraints. This section only gives a brief insight into constraint solving; a broad overview of different techniques for solving GUI constraints is given in [5].

There are either equality or inequality hard-constraints. They can be written as

\[ A \cdot x = b \]
\[ A \cdot x \geq b. \]

Here \( A \) is the constraint matrix, \( x \) is the tabstop or variable vector and \( b \) is vector of constants.

Notice that this system of equations and inequalities is quite easy to solve if we assume that all variables are positive. This is certainly true for our kind of problem or at least the problem can be transformed into a problem with positive variables. Inequalities can be expressed as equalities using positive slack variables, e.g.,

\[ \sum_i a_i \cdot x_i \geq b_i \]
can be written as
\[ \sum_i a_i \cdot x_i - s = b_i \quad s \geq 0. \]

The resulting system with only positive variables can, for example, be solved with a method described in [3].

In ALM soft-constraints are specified in the same way as hard-constraints. How soft-constraints are violated is described using a scalar objective function. In general, this objective function is minimized while satisfying the hard-constraints at the same time. In the following two types of objective functions are described, a linear and a quadratic objective function.

### 2.3.1 Linear Objective Function

The simplest approach for an objective function is a linear objective function. When using a linear objective function, only a few soft-constraints are violated if necessary. Which soft-constraints are violated first can be specified by giving the soft-constraints different priorities. A high priority means that, if possible, other soft-constraints will be violated prior. A linear objective function is, for example, used in Apple’s AutoLayout as well as in the original version of ALM [75].

Technically, soft-constraints are implemented as hard-constraints of the form:
\[ \sum_i a_{soft,i} \cdot x_i + s_{shrink} - s_{grow} = b_{soft}. \tag{2.1} \]

Here, \(s_{shrink}\) and \(s_{grow}\) are two new positive slack variables that expresses that the soft-constraint can be violated in both directions. The goal is to keep both slack variables as small as possible to violate the soft-constraint as little as possible. The penalty factors \(p_{shrink}\) and \(p_{grow}\) can be used to prioritize a soft-constraint, with a large penalty factor meaning that growing or shrinking away from the optimal values is suppressed. This leads to the linear objective function, which is the weighted sum of all slack variables, and which must now be minimized while satisfying the hard-constraints [75]:
\[ \sum_{i \in \text{soft}} p_{grow,i} \cdot s_{grow,i} + p_{shrink,i} \cdot s_{shrink,i} \rightarrow \min. \tag{2.2} \]

Such a problem is commonly known as linear programming. A suitable solver for this purpose is lp_solve\(^5\) which uses the simplex algorithm [27].

One problem of a linear objective function is that minimizing Equation 2.2 generally leads to many valid solutions; the linear approach is non-deterministic. This means that

\(^5\)http://lp.solve.sourceforge.net
not all soft-constraints are violated in a uniform way, e.g., only a few constraints are violated and it is not clear which constraints are violated (see Figure 4.1).

2.3.2 Quadratic Objective Function

When using a quadratic objective function and there are conflicting constraints all soft-constraints are violated. To control how strong a certain soft-constraint will be violated a weight is assigned to each soft-constraint. A high weight means that the particular soft-constraint is only violated a little. This leads to a deterministic solution of the constraint system.

Minimizing a quadratic objective function can be compared with the method of least squares; the deviations from desired target values are minimized. For simple preferred size constraints, this is:

\[ \sum_{i \in \text{soft}} (x_i - \text{pref}_i)^2 \rightarrow \text{min}. \]  

(2.3)

More generally, the soft-constraints can be written in matrix form:

\[ A_{\text{soft}} \cdot x = b_{\text{pref}}. \]

Here \( x \) and \( b \) are vectors. Analogously to Equation 2.3, this leads to the general quadratic objective function

\[ \frac{1}{2} x^T A_{\text{soft}}^T A_{\text{soft}} x - b_{\text{pref}}^T A_{\text{soft}} x \rightarrow \text{min}. \]

Substituting

\[ A_{\text{soft}}^T A_{\text{soft}} = G \]

and

\[ -b_{\text{pref}}^T A_{\text{soft}} = g^T \]

this simplifies to:

\[ \frac{1}{2} x^T G x + g^T x \rightarrow \text{min}. \]

This is a known quadratic programming optimization problem and could, for example, be solved using the Active Set method [35]. To use the Active Set method, first a valid base solution for the hard-constraints has to been found (as described above). Continuing from that base solution, the Active Set method minimizes the quadratic objective function, while staying in the solution space of the hard-constraints.

Soft constraints can be weighted by a weight \( w \). For the simple preferred size constraints, this leads to the objective function

\[ \sum_{i \in \text{soft}} w_i^2 \cdot (x_i - \text{pref}_i)^2 \rightarrow \text{min}. \]
Constraints with larger weights $w$ will be violated less than constraints with smaller weights.

In contrast to most other systems, ALM uses a quadratic optimization function with a quadratic active set solver [35]. This method is fast enough for GUI problems [18]. For the implementation used here it has been found that solving layouts with 30 widgets can be done in the order of 10 ms.

Soft preferred size constraints together with a quadratic optimization function ensure that all widgets have a well-defined size and that there is a unique solution. A linear optimization function cannot ensure this, as there are in general an infinite number of solutions for the soft-constraints, which leads to undetermined widget sizes (see Chapter 4).

### Soft Inequality Constraints

So far, only soft equality constraints can be specified using a quadratic objective function. However, soft inequality constraints can be constructed from a hard inequality constraint and a normal soft equality constraint. To do so, the soft inequality constraint is written as a hard inequality constraint including a positive slack variable $s$. For example,

\[
\sum_i c_i x_i \leq r_i \text{ becomes } \sum_i c_i x_i - s \leq r_i,
\]

\[
\sum_i c_i x_i \geq r_i \text{ becomes } \sum_i c_i x_i + s \geq r_i.
\]

This means $s$ can always be chosen to satisfy the hard inequality constraint. However, only when the soft inequality constraint must be violated because of other constraints, $s$ should be greater than zero. This can be achieved by adding the soft equality constraint $s = 0$ with a sufficiently high weight.

Notice that compared to a linear objective function there is only one weight per soft-constraint. Thus the developer cannot specify if a constraint is more likely to grow or to shrink. However, by combining two soft inequality constraints, e.g., $x \geq b$ and $x \leq b$, a similar growing and shrinking behavior can be achieved.

A quadratic objective function is used in the constraint-based layout model used in this thesis. Using a quadratic objective function leads to deterministic layouts and has aesthetic advantages as discussed later in Chapter 4.
Usability of the Constraint-based Layout Model

While the usability of GUI design methods has been studied in general, the usability of layout specification methods is largely unexplored. However, even though the constraint-based layout model is more powerful than the grid-bag layout model, this does not mean it is more usable. So far it is unclear whether either of them has any advantages in terms of usability. Hence, this chapter tries to answer the question how the constraint-based and the grid-bag layout models differ in terms of usability. Specifically, the following research questions are studied:

- How does the usability of the grid-bag and the constraint-based layout models differ for specifying layouts?
- How does the usability of the grid-bag and the constraint-based layout models differ for editing layouts?

Here just the bare layout models are analyzed and nested layouts are ignored.

To answer these questions a controlled experiment with postgraduate students of Computer Science and Software Engineering was conducted. In this study the abstract term usability is investigated as the efficiency and accuracy with which a task can be completed and the preferences users have. The efficiency is measured as task completion time, the accuracy as the number of errors made, and the preference with standardized items on a Likert-scale. The experiment was paper-based in order to abstract from a concrete
API implementations and directly compare the underlying layout models. Participants performed several specification and editing tasks on layouts.

Note that the existence of GUI Builders does not make the study of layout models less relevant. Although there are graphical GUI builders to support the design of GUls, many developers still prefer to specify a layout programmatically. This may have various reasons, apart from the personal developer preferences, for instance: no suitable GUI builder for the used toolkit or platform is available, used layout items are not supported by the GUI builder or layout specifications are changing at runtime. Furthermore, GUI builders are still using certain layout models and it is usually necessary for the developer to understand these models when using a GUI builder.

From the performed user evaluation it has been found that the grid-bag layout model is more efficient than the constraint-based layout model when specifying a layout from scratch. When it comes to editing an existing layout that is reasonably complex, the constraint-based layout model is significantly faster. With regard to the accuracy, the constraint-based layout model results in less errors when specifying and editing layouts. This becomes significant when editing more complex layouts. Overall, the participants preferred the constraint-based layout model, especially for editing a layout. The in this chapter presented evaluation has already been presented at the OZCHI’12 conference.

Section 3.1 gives an overview of how layouts are specified using the grid-bag layout model and the constraint-based layout model. Furthermore, this section describes the complexity for editing layouts using these layout models. Related work is discussed in Section 3.2. Section 3.3 describes the methodology of the study. The results are presented in Section 3.4. Section 3.5 discusses the results and analyses threats to validity. The conclusions are summarized in Section 3.6.

3.1 Grid-bag and Constraint-Based Layouts

The following section discusses how a layout can be specified using the grid-bag layout model and the constraint-based layout model. During the design process it is sometimes necessary to edit an already existing layout specification. This can, for example, happen if an additional button has to be inserted into a layout or the size of an item needs to be adapted. Editing a layout can have a different complexity for different layout models. To quantify this edit complexity the following simple definition is used. The edit complexity of a layout is the number of layout items that have to be updated during an edit operation for one single change.
3.1 Grid-bag and Constraint-Based Layouts

3.1.1 Grid-Bag Layout

One of the most prominent layout models is the grid-bag layout model. This layout model is, for example, used in HTML tables[^1] and almost all available GUI toolkits support this layout model.

In a grid-bag layout a layout item can be placed in a two-dimensional grid. Beside the row and the column that specifies where an item is placed, a *row-span* and *column-span* specifies how many rows and columns a layout item occupies. Using row- and column-spans for one item makes it possible to create complex layouts. Furthermore, it is possible to create a link between items not directly adjacent, e.g., by placing them in the same row or column. Figure 3.1 shows an example for a grid-bag layout. Here all layout items occupy one or more cells in the grid.

The grid-bag layout can be tuned by giving the rows and columns weightings for specifying which row and column should use more space compared to the other rows and columns.

![Figure 3.1: Layout using grid-bag layout specifications.](image)

**Editing a Grid-bag Layout** Whenever an edit operation causes the change of the row and column number of a grid-bag layout, other items in the layout might be affected and need to be updated. There are two cases that make this update necessary. First, items to the right or further down from the inserted layout item get a higher row or column value. Secondly, the row- or column-span of items that are intersecting in the vertical or horizontal direction have to be updated. Figure 3.2 illustrates these two cases, i.e., after the insertion of Button 2 the column number of Button 1 changes from 1 to 2 and the

column-span of the progress bar changes from 1 to 2. The simple case of an insertion into an empty cell does not change the row and column number and so does not affect any other layout items.

![Diagram showing grid_bag layout](image)

**Figure 3.2**: Grid-bag layout: Button 2 is inserted left beside Button 1. The existing Button 1 is moved from c1 to c2 and the column-span of the progress bar changes from 1 to 2.

### 3.1.2 Constraint-Based Layout

In the constraint-based layout model, the layout specifications are described by constraints using linear equalities and inequalities. Constraint-based layouts have attracted much attention in research and industry in recent years [5, 75, 94]. In the following the nomenclature of the constraint-based layout model ALM (Auckland Layout Model) [75] is used. However, in general any constraint-based layout model such as the Java SpringLayout[^1] is suitable for our usability study. It only has to provide an easy way to connect borders of layout items together.

The edges of each layout item are connected to two horizontal (top and bottom) and two vertical (left and right) tabstops. Through this items can be aligned by connecting them to the same tabstops. The rectangular layout area, i.e., panel or window, has four tabstops that are connected to the layout borders to facilitate alignment with the layout boundaries.

A simple constraint-based layout is shown in Figure 3.3. For example, to make sure that the two buttons stay on the right of the list view the left side of the buttons share the same tabstop $x1$.

**Editing a Constraint-Based Layout** While changes in a grid-bag layout can affect items anywhere in the layout, in a constraint-based layout changes are more local. This means only layout items that have to be connected to a newly added tabstop have to be updated. In the example in Figure 3.4 a list box is added between a text view and three

[^1]: Spring Layout API documentation, 2012 [http://docs.oracle.com/javase](http://docs.oracle.com/javase)
buttons. Here a new tabstop $x2$ is connected to the right of the text view and only this item has to be updated. The three buttons on the right do not need to be updated. From this example one can see that how many layout items have to be updated depends on where the new tabstop is inserted, i.e., if $x2$ had been inserted on the right side of the list view and $x1$ stayed at the right side of the text view all three buttons would need to be updated.

To specify the position of a layout item uniquely, the layout item has to be connected to at least one horizontal and one vertical tabstop that are directly or indirectly connected to a layout border tabstop. When indirectly connected to a layout border tabstop it means that there are some constraints that set a relative position between the tabstop and a border tabstop. This is necessary to specify the layout in a unique way. For example, if a layout item is not connected to any layout border tabstop, it is not clear where the item should be positioned within the layout; all positions are valid.

A layout specification is well-defined if all layout items have to be directly or indirectly connected to at least one horizontal and one vertical layout border tabstop. A well defined layout limits the edit complexity of a constraint-based layout. Usually a designer places a layout item either beside another layout item or between two layout items, or adjacent to a window border. In this way at least one horizontal and one vertical border of the layout item is connected to an existing tabstop, e.g., if a new button should be aligned to the top-right of another layout item, it is connected to the top and right tabstop of this item. Figure 3.4 shows an example where only a single tabstop ($x2$) has to be inserted. This limits the edit complexity to the number of layout items, that have to be connected to a maximum of two new tabstops.
3.2 Related Work

This study investigates the differences of the usability of two layout models: the grid-bag layout model and the constraint-based layout model. To identify such effects an often applied research method used in Software Engineering is that of controlled experiments [10]. In a systematic literature survey Sjøberg et al. identified a total of 1.9% of published papers in leading Software Engineering venues that conducted controlled experiments [97]. However, to the best our knowledge none of them investigates GUI layout.

Within the HCI community, a research stream of API usability evaluation is emerging, that argues that for programmers the API of frameworks, Software Development Kits (SDKs) and libraries are a user interface for computers. It has therefore to be designed according to some usability criteria [23, 29, 93]. Usability studies for APIs were, for example, conducted in the domain of Service Oriented Architectures [11], or specifically for the use of language constructs such as names [28], patterns [31] or the content of API documentation [69].

There is also some work in the program comprehension community related to API usability. Hou and Li conducted a case study about problems programmers have with the use of the Java Swing API [50]. They conducted an empirical evaluation of newsgroup posts and identified a set of API obstacles. Their focus is not on the underlying layout model but on the actual GUI API realization.

In the domain of GUI APIs little research has been done about usability so far. An experimental study about the usability of notations for XAML and Windows Forms was conducted by Kosar et al. [71]. Their experiment mainly focused on the understanding of the notations of the APIs and their usability. They did not analyze the usability
of layout models of both frameworks. Their study also differs from this study in the methodology. In these experiments, participants actively specify and change layouts. In their study, participants were only asked to evaluate existing GUI specifications in XAML and Windows Forms. Hence they only measured accuracy but not efficiency as it has been done here.

No work exists that directly compares the usability of layout models. Even though industry begins to adapt the constraint-based layout model from an empirical point of view, it is still unclear whether this model is more usable than well-established models such as the grid-bag layout model.

3.3 Methodology

First a pilot study was conducted to better understand the effects the constraint-based layout model has on usability. Form this experience concrete research hypotheses were formulated and a main study was designed.

3.3.1 Pilot Study

Based on our own experience with the constraint-based and grid-bag layout models, it was expected that specifying or editing layouts with a constraint-based layout would be faster and less error prone. When specifying a layout item in a grid-bag layout, the correct row and column-span has to be counted. Thus, the expectation was that if the row- or column-span of a layout item is high this should take more time and should be less accurate. Moreover, when editing a constraint-based layout, just the surrounding tabstops have to be considered, whereas in the grid-bag layout the whole grid has to be checked for changes (see Section 3.1).

To investigate this assumption a pilot study was designed that was primarily focused on layout specification and only secondarily on layout editing. The participants had to perform 12 tasks, once for the grid-bag layout model and once for the constraint-based layout model. Half of these tasks were training tasks similar to the main tasks. From the six main tasks, four tasks were to specify a layout and two tasks were to edit two of the previously specified layouts. The tasks were quite simple with only a few layout items to be specified, and only marginal changes in the layout specification were required when editing a layout. The evaluation was paper-based to reduce programming-related influences such as the factor of different programming skills, and to focus the study on the different underlying concepts of both layout models.

However, after a few participants performed the tasks it became clear that the usability of a constraint-based layout is no better with respect to all usability parameters, contrary
to the initial assumption. First, the measured task completion times were clearly in favor of a grid-bag layout. Secondly, it became clear that larger row- or column-span values are not a real problem when specifying a layout item; in most cases the values seemed to be too trivial to determine.

3.3.2 Hypotheses

Based on the experience gained in the pilot study, research hypotheses about the differences between both layout models, as potential answers to the research questions were formulated. The hypotheses are as follows:

- **H1** The grid-bag layout model is faster for layout specification.
- **H2** The constraint-based layout model is faster for layout editing.
- **H3** The constraint-based layout model is less error-prone for layout specification.
- **H4** The constraint-based layout model is less error-prone for layout editing.
- **H5** The constraint-based layout model is perceived as being easier to use.

To test these hypotheses the design of the pilot study has been refined for the main study.

3.3.3 Main Study

An insight from the pilot study was that the tasks were too easy and focused too much on layout specification to reliably discriminate between both layout models. Therefore in the main study the layout complexity of all tasks was increased and more layout editing tasks were included.

The experiment had a within-subject design [51], which means that each participant had to work with the constraint-based layout and the grid-bag layout, and was counterbalanced. Each participant had to conduct the same three tasks using a constraint-based layout and a grid-bag layout.

For each layout model, first some background was given and then the technique was explained. The participants then had to perform a training task in which they were allowed to ask clarification questions. The explanations of the experimenter were supported by a printed one page how-to, which remained as a reference for the participant during the whole experiment.

Afterwards two tasks followed for which the task completion time was measured by an experimenter. The experiment closed with a post questionnaire. To minimize order
bias, the participants alternately started either with the constraint-based layout or the grid-bag layout. The design of the study is depicted in Figure 3.5.

The tasks were subdivided into three subtasks. In the following a certain subtask is referred as Task.X.Y, where X is the number of the main task and Y the number of the subtask. The first subtask of each task was to specify a given layout with the means of the respective layout model. The layout was presented as a printed screenshot. The specification had to be done on a sheet of paper with a table for all layout elements similar to the tables shown in Figures 3.1 and 3.3. The layout items in the table were displayed in an iconized form to make it easy for the participants to identify them.

The following two subtasks were about editing the initial layout specification according to changes of the layout presented in a new screenshot. In the table the new layout item had an extra row that was then uncovered. In this way the participant was able to specify the new item and update the specification of the existing items in the same table. This mimics a real design process where a developer has done the initial layout specification and then edits this specification. To distinguish between the specifications of the different editing subtasks the participants used pens in different colors for each subtask. For each subtask the time that the participant needed to specify or edit a layout was measured.

The participants were asked to complete each subtask in two separate steps. First they had to become familiar with the layout they had to specify. That was done by sketching and labeling both, in the case of grid-bag layout, a suitable grid, or in case of constraint-based layout, the required tabstops into the given screenshot. In the second step they were asked to derive relevant parameters (e.g., the rows and columns of the layout items) from the layout and write them into a table that was provided.

In the following the experimental material (tasks and questionnaire) is described and how the experiment was conducted.
Tasks and Questionnaire

The training task was similar to the main tasks. It was designed to be sufficiently complex and to cover all interesting pitfalls the participants could run into while doing the main tasks, e.g., the training layout contained an empty area where it might be unclear where a tabstop or a row or a column should be inserted (see Figure 3.6 (a)).

Task 1 (Figure 3.6 (b)) was designed as a relatively simple task. The first insertion was a text view in the middle and the second insertion was a button between two existing buttons. For the first insertion subtask, the edit complexity for the constraint-based layout was two or three, depending on where the new tabstop was added (see Section 3.1.2). For the second inserting subtask the edit complexity was two. The edit complexity for the grid-bag layout was four for the first insertion subtask and five for the second insertion subtask.

Task 2 (Figure 3.6 (c)) was more complex than the previous one. It mimicked a window of a chat application. The editing subtasks were designed to be part of a possible development process. The first edit subtask was the insertion of a “video call” button beside an existing “audio call” button. This subtask had an edit complexity of one for the constraint-based layout and an edit complexity of six for the grid-bag layout. In the second insertion subtask another button had to be added with an edit complexity of two for the constraint-based layout and of eight for the grid-bag layout, respectively. This means that the second task, while still relatively simple, was more complex than the first task.

The questionnaire contained three demographics questions (gender, age and occupation), eleven 5-point Likert-scale questions with predefined answers and one open-ended question. The Likert-scale questions were:

Q1 I often use computers in my everyday life.
Q2 I understood the task and was able to perform it.
Q3 It was easy to specify and edit the layouts using grid-bag layout.
Q4 It was easy to specify and edit the layouts using constraint-based layout.
Q5 I have experience with designing user interfaces.
Q6 I have used or known grid-bag layout before, e.g., HTML tables.
Q7 I have used or known constraint-based layout before.
Q8 As a programmer I would like to use constraint-based layout in the future.
Q9 As a programmer I would like to use grid-bag layout in the future.
Q10 Specifying and editing a layout using constraint-based layout was difficult.
Q11 Specifying and editing a layout using grid-bag layout was difficult.

Each experimental session took approximately one hour and took place in a lab with closed door and blinds to minimize disturbance from external sources. There were two
3.4 Results

12 participants were recruited who were students of Computer Science at least at Masters level. The sample contains three female and nine male participants. Figure 3.7 summarizes the experience level of the participants. All participants were heavy computer users.
and most of them had experience with GUI design. Most of them were familiar with grid-bag layout, whereas more than a half of the sample was not familiar with constraint-based layout. They all understood the tasks they were asked to do.

3.4.1 Efficiency

To compare the efficiency for the specification and editing tasks the completion time difference

$$\Delta Time = t_{constraint} - t_{grid}$$

is calculated for each participant. It turned out that all the completion times of all tasks were unlikely normally distributed. Hence, a one-sided Wilcoxon rank sum test is applied to calculate the significance of the hypotheses. Table 3.1 depicts the mean of $\Delta Time$, the standard deviation $\sigma$ of $\Delta Time$ and the Wilcoxon test statistic $p_{wrs}$ value.

<table>
<thead>
<tr>
<th>Task</th>
<th>Time (s)</th>
<th>$\sigma$</th>
<th>$p_{wrs}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 (H1)</td>
<td>12.50 *</td>
<td>22.97</td>
<td>0.05</td>
</tr>
<tr>
<td>1.2 (H2)</td>
<td>-5.42</td>
<td>19.25</td>
<td>0.24</td>
</tr>
<tr>
<td>1.3 (H2)</td>
<td>-8.25</td>
<td>36.65</td>
<td>0.47</td>
</tr>
<tr>
<td>2.1 (H1)</td>
<td>16.50 **</td>
<td>16.00</td>
<td>&lt;0.01</td>
</tr>
<tr>
<td>2.2 (H2)</td>
<td>-25.00 **</td>
<td>25.28</td>
<td>&lt;0.01</td>
</tr>
<tr>
<td>2.3 (H2)</td>
<td>-37.00 **</td>
<td>28.46</td>
<td>&lt;0.01</td>
</tr>
</tbody>
</table>

The first hypothesis (H1) is that specifying layouts with the grid-bag layout model is faster than with the constraint-based layout model. Figure 3.8 depicts the boxplots of the task completion times for the specification tasks (CL means constraint-based layout and GB means grid-bag layout). For both tasks the task completion time was higher for the constraint-based layout model. From the resulting $p_{wrs}$-values for Tasks 1.1 and 2.1,
3.4 Results

Figure 3.8: Boxplots of the task completion times for the specification tasks

(Table 3.1) the H1 hypothesis can be confirmed. These results match with the experiences gathered in the pilot study.

Not only the task completion time for layout specification, but also the effect of the constraint-based layout model on the task completion time for editing tasks is interesting. Hypothesis H2 is that the constraint-based layout model allows faster layout editing. However, from the measured completion times shown in Figure 3.9 this is not quite clear. Task 1.2 seems to have been completed faster with the grid-bag layout model, but all other tasks seem to have been completed faster when using the constraint-based layout model. Table 3.1 shows the results for editing tasks. The tests show that Tasks 2.2 and 2.3 were completed faster with the constraint-based layout model on a 0.1 % significance level. The results for Task 1.2 and 1.3 are less clear, here the null hypothesis can only be rejected with less statistical significance. The more complex Tasks 2.2 and 2.3 support H2 whereas this is not so clear for the easier Tasks 1.2 and 1.3.
3.4.2 Accuracy

The accuracy is another important factor that indicates how usable a layout model is. For both, the specification and the editing task, each item that has not been specified correctly is counted as one error. This means multiple wrong values for a single layout item, e.g., wrong row and wrong column span are counted as just one error. For the editing subtasks, values specified wrongly in the preceding specification or editing task were not counted as mistakes for the current subtask.

The accuracy is high if the number of layout specification errors \( E \) in a certain task is low. The accuracy for the specification and the editing tasks is measured to test \( H_3 \) and \( H_4 \). Similar to the efficiency, the difference between the error rate of the constraint-based layout model and the grid-bag layout model \( \Delta E = E_{\text{constraint}} - E_{\text{grid}} \) is calculated and a Wilcoxon rank sum test is applied (Table 3.2).

<table>
<thead>
<tr>
<th>Task</th>
<th>( \Delta E )</th>
<th>( \sigma )</th>
<th>( p_{\text{wrs}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task 1.1 (H3)</td>
<td>0.08</td>
<td>0.67</td>
<td>0.72</td>
</tr>
<tr>
<td>Task 1.2 (H4)</td>
<td>-0.42</td>
<td>0.79</td>
<td>0.09</td>
</tr>
<tr>
<td>Task 1.3 (H4)</td>
<td>-0.25</td>
<td>0.97</td>
<td>0.20</td>
</tr>
<tr>
<td>Task 2.1 (H3)</td>
<td>-0.25</td>
<td>0.45</td>
<td>0.07</td>
</tr>
<tr>
<td>Task 2.2 (H4)</td>
<td>-0.33</td>
<td>0.65</td>
<td>0.09</td>
</tr>
<tr>
<td>Task 2.3 (H4)</td>
<td>-1.08 *</td>
<td>1.68</td>
<td>0.03</td>
</tr>
</tbody>
</table>

Hypothesis \( H_3 \) says that when specifying layouts using the constraint-based layout model the error rate is lower. This hypothesis can be tested using the observations from Task 1.1 and 2.1. The boxplots in Figure 3.10 give no clear picture. The specification subtask of Task 1 contains more errors for the constraint-based layout, but that outcome changes for Task 2 where the grid-bag layout model seems to generate more errors. From Table 3.2 it can be seen that the \( p_{\text{wrs}} \)-value of 0.07 is quite low but still not significant. Therefore, \( H_3 \) cannot significantly be confirmed but the data indicates that the hypothesis is true.

Hypothesis \( H_4 \) claims that the error rate is lower when editing a constraint-based layout than when editing a grid-bag layout. This hypothesis is tested with the observed errors of the editing tasks (Figure 3.11). Here the boxplots give a rather clearer picture. The error rate seems generally higher for the grid-bag layout model. However, conducted statistical tests in Table 3.2 confirm this assumption only for the most complex editing Task 2.3. Here the null hypothesis can be rejected to a 5 % significance level and \( H_4 \) can be accepted.
3.4 Results

3.4.3 Preference

With hypotheses H1 – H4 the actual behavior of the participants was studied. Hypothesis H5 analyzes the subjective impression of the usability of the constraint-based layout model. For that several questions were asked for which the frequencies of the answers are depicted in Figure 3.12.

![Figure 3.10: Boxplots of the error rates for the specification tasks](image)

![Figure 3.11: Boxplots of the error rates for the editing tasks](image)

![Figure 3.12: Frequencies of answers to impression questions](image)
To test $H_5$ two question pairs (“is easy” and “is difficult”) were included into the questionnaire. The results of a Wilcoxon rank sum test are depicted in Table 3.3. Here $L$ is the value from the Likert scale that goes from -2 for strongly disagree to 2 for strongly agree. According to these results, for the first question pair the null hypothesis can be rejected on a significance level of 1% and for the second question pair on 5% significance level. These results are also backed up by many comments from the participants who said that editing a constraint-based layout is much easier and faster than editing a grid-bag layout.

### Table 3.3: Test statistics for H5

<table>
<thead>
<tr>
<th>∆L</th>
<th>σ</th>
<th>$p_{wrs}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q3 - Q4</td>
<td>-0.83</td>
<td><strong>0.94</strong></td>
</tr>
<tr>
<td>Q11 - Q10</td>
<td>0.67</td>
<td><em>1.15</em></td>
</tr>
</tbody>
</table>

3.5 Discussion

For the effectiveness the data shows that specification is faster for the grid-bag layout model. For Task 1.1 it was on average 12.5s faster and for Task 2.1 it was on average 16.5s faster (Table 3.1). There are two possible explanations for that behavior. First, when specifying a constraint-based layout for all layout items, the four surrounding tabstops have to been found. Similar to that, a layout item in a grid-bag layout is specified by the four values column, row, row-span and column-span. However, in a grid-bag layout items often have a small row- and column-span that makes it very easy to determine the span values, e.g., a layout item in a single cell has a row- and column-span of one. Secondly, the grid-bag layout model is more familiar to the developers (Figure 3.7) and thus they need less time to perform this task.

However, for the editing tasks the picture changes. For the easier tasks the constraint-based layout was slightly, but not significantly, faster. When the tasks became more complex (Task 2.2 and 2.3) editing a layout was clearly faster in the constraint-based layout model. The main explanation for that observation is that the edit complexity for the constraint-based layout remained almost constant whereas the edit complexity for the grid-bag layout increased. This is mainly because changes in a constraint-based layout are local: changing a layout item only affects the surrounding layout items, while for the grid-bag layout in the worst case the whole layout has to be updated (Section 3.1).

The measured accuracy showed a similar behavior. For Task 1 the results are not clearly in favor of the constraint-based layout model, but for the more complex Task 2 the error rates for specification and editing are dropping considerably. For Task 2.3 the
error rate was significantly lower. Again, the main reason that was identified is that the edit complexity is smaller for a constraint-based layout than for a grid-bag layout.

The objectively measured data for efficiency and accuracy matches the subjective impressions the respondents reported in the questionnaire. The participants perceived the constraint-based layout model as easier to use. This is interesting because most of the participants had known the grid-bag layout model before. Even in a short training session the participants became comfortable with the new layout model and experienced its advantages.

The results of the experiment suggest that the constraint-based layout model can compete with well-established layout models such as the grid-bag layout model, and can outperform them for more complex editing tasks. These interpretations have to be seen in the light of some possible threats to their validity.

Internal Validity

Since a within-subject experimental design was applied, one could face an order bias. To minimize this possible threat the order in which the layout models were used was alternated.

Another threat could be that because only two main tasks were used for the evaluation not all interesting layout configurations were covered. Since the experiments already took about an hour it was not possible to introduce another task. Since both tasks show the same pattern (specification is slow with constraint-based, editing is fast) one can assume that the tasks capture the general effects and are hence adequate to study the differences in usability of both layout models. Furthermore, the subtasks were designed in such a way that they became successively more complex, hence covering a range of different complexities.

Many of the participants already knew the grid-bag layout model. That could have introduced an advantage for the grid-bag layout model. This threat was minimized by abstracting from a concrete API implementation to a paper-based experiment and with an extensive training phase.

A final internal threat could be social desirability bias that could stem from the fact that some participants knew about this work in the field of constraint-based layout models. They could therefore have been positively biased in the questionnaire answers about the constraint-based layout model. This threat was minimized by including these questions twice, but with other formulations in the questionnaire. Even though this measure can help reduce the bias, one has to be aware of this problem in the interpretation of the results.
External Validity

The study poses some threats to its external validity. First, it can be an issue that the abstraction from concrete API implementations did not reflect the usability of a real-world API. However, the objective of the study was to understand the differences in the usability of the layout models. A specialization to a concrete implementation is not intended and may in fact endanger the generalizability of the results. Nevertheless, it can be assumed that differences in usability of layout models translate to similar differences in usability of well-designed APIs that implement these layout models.

Another threat is the selection of the sample that consisted solely of students. However, only students at least at Master’s level were selected. For them one can assume sufficient knowledge in Computer Science to consider them as software developers. This assumption is also supported by a recent study about the representativeness of students for professional software developers [101].

Finally, the task design is an issue. The tasks were rather small compared to real-world GUI layouts. However, the tasks were designed in such a way that they reflect typical problems of real-world development situations (e.g., empty cells in a layout or introducing new layout items). Layouts in practice may replicate these problems several times, but the cognitive process to handle them is the same. Therefore it should be possible to extrapolate the results to much more complex cases.

3.6 Conclusion

Two of the most versatile layout models are the grid-bag layout model and the constraint-based layout model. While the grid-bag layout is quite popular, the constraint-based layout is lesser known although it is the more general and flexible layout model. An experimental study comparing the usability of these two layout models was conducted. The usability was measured considering the factors efficiency, accuracy and preference. These factors were measured in various tasks, which included the specification of new layouts and the editing of existing layouts with different layout and edit complexities.

It has been found that it is quicker to specify new layouts with the grid-bag layout model. However, when it comes to editing the previously specified layouts, the constraint-based layout model outperforms the grid-bag layout model. Especially for more complex layouts, constraint-based layout is significantly faster and has a significantly higher accuracy. From the questionnaire, the participants preferred the constraint-based layout and, consistent with the experiment, found it much easier for editing a layout.

The results of this study are quite promising and are clearly in favor of the more powerful constraint-based layout model. A next step would be to test the usability on widely used API implementations. For that purpose, even more complex programming
tasks could be designed.

The empirical study substantiates the effort of the industry to implement constraint-based layout models in their GUI APIs (e.g., Cocoa’s Auto Layout or Java’s SpringLayout). This model has clear advantages over common layout models, and it is worth to give it a try. Only time will tell if the constraint-based layout model will be widely adopted — at least it has the potential.
Layout managers provide an automatic way to place controls in a graphical user interface (GUI). With the wide distribution of fully GUI-enabled smartphones, as well as very large or even multiple personal desktop monitors, the logical size of commonly used GUIs has become highly variable. A layout manager can cope with different size requirements and rearrange controls depending on the new layout size. However, there has been no research on how additional space or the lack of it can be best distributed onto the controls in the layout and how this distribution of space affects the aesthetics of the whole layout.

To setup a GUI using a special layout model, the developer has to specify a set of layout specifications. To keep things as simple as possible for the developer, the layout specifications that are required to define a layout with good visual appearance should be small. This means that layout specifications usually do not specify every single detail about a layout, but leave some of the details to the layout model. The cases are interesting when the available space in a GUI is insufficient or when there is more space available than needed. Both cases happen regularly when windows are resized, e.g., when adjusting a GUI to the size of the available screen space. For example, consider a simple layout containing just two controls in a horizontal row, spanning the complete window size (Figure 4.3). Depending on the window size, the layout model has to decide what control width is the best to yield a visually appealing result. An important hint for that decision can be the control’s preferred size, which describes the size preferred by the control in the absence of any other constraints.
In this chapter strategies to distribute available space in a visually appealing way are described and compared. All strategies are modeled with the constraint-based layout model, since such a layout model can be used to describe a wide range of layouts. Some aesthetic problems of the constraint-based layout model have been identified and solutions have been provided.

The following overall research question is addressed: how the space available in a GUI should be distributed among the layout items? To answer this question, one has to consider what layouts are perceived as aesthetically pleasing, as well as the solving strategies that determine the layouts. Therefore, a minimal deviation strategy to distribute the available space is motivated. In a constraint based layout system this can be implemented by using a quadratic solving strategy. As shown later this solving strategy is superior to a linear solving strategy. To answer the research question the focus is on a subproblem of this general question. Here only a very restricted class of layouts and GUIs is analyzed, namely GUIs that are comprised solely of buttons. This is worthwhile since it allows us to isolate possible cross-influencing factors. The here presented restricted experiments yield interesting results that can now be tested in other more general classes of layout.

In the field of typesetting and document layout much research has been done about how to create accessible and clear document layouts using automated layout systems. However, no study has been undertaken as to how different solving methods compare to each other with respect to aesthetics. The compared solving strategies are equal distribution, weighted distribution and minimal deviation. All solving strategies were implemented and evaluated using the Auckland Layout Model (ALM), a constraint based layout model. However, the results presented in this chapter are not limited to ALM and can be applied to other layout models as well. A constraint based layout system is able to model all interesting layouts and solving strategies for this work. Furthermore, pitfalls of the solving strategies are analyzed and solutions for them are presented.

In a user evaluation three solving strategies, equal distribution, weighted distribution and a minimal deviation have been compared. As a result, the minimal deviation approach seems to be a good strategy for large and small layout sizes. The minimal deviation and the equal distribution strategy is best for large layout sizes while the weighted distribution approach seems to perform better for small layout sizes. Furthermore, the evaluation shows that layouts with a high degree of symmetry are clearly preferred by the users. In this chapter presented evaluation has already been introduced at the CHINZ’12 conference.

The next Section gives an overview of related work, including how other layout models distribute available space and how Gestalt principles are used in other fields to get visually appealing results. Section compares the effect of linear and quadratic objective functions on the visual quality of the layout. In Section a user evaluation is presented.
4.1 Related Work

The scientific field of Gestalt psychology \cite{70} covers principles about the perception of shapes and groups of shapes. For example, the law of equality states that similar shapes are perceived as a group, and the law of proximity states that shapes that are placed close to each other are perceived as a group as well. These findings can be transferred to user interfaces, where aligned controls are perceived as a group. The law of equality can be applied when items are placed in the same row or column and share the same height or width. When items are aligned close to one another the law of proximity can be applied. This can be used to group related controls \cite{47} to achieve a clear layout appearance. Gestalt psychology is the basis for many fields and is used in most aesthetics related papers.

Gestalt principles can also be applied to conventional typography \cite{68} as well as to web documents \cite{16}. Here they can help to structure a document to make it easier to read and understand. Another application is the pagination problem, which targets the question how to best distribute content over multiple pages, e.g., how to place figures and text in a complex document to produce visually pleasing results \cite{91}. In the field of graph layout, a set of similar layout aesthetics has been used to optimize graphs \cite{26}.

The knowledge of Gestalt principles can help to layout UI objects in a more pleasant way \cite{16}. Layout mangers often make it easier to set up good layouts, without having to define the final layout in all its details. However, they do not apply Gestalt principles alone. Whether or not Gestalt principles are used depends on how a GUI designer specifies a layout, not on the layout model.

Also other approaches have been tried in adapting a layout to different sizes. SUPPLE is an automated system that can automatically adapt layouts to different layout sizes, in particular to different device screen sizes. This system supports discrete changes of layout items, i.e., it rearranges or changes widgets depending on the available space \cite{108}. Optimizations of more experimental layouts have been studied for the GADGET framework \cite{36}. For example, GADGET targets problems like how a GUI can be automatically generated using certain optimization rules. In this work the focus is strictly on the optimization of resizing of GUI components, and in this area the focus is on rather subtle changes.

Generally there is a size discrepancy between the actual size and the preferred size of a layout or layout item. Most layout models distribute available space in a simple way; however, different layout models use different methods of distribution. Exactly how
available space is distributed is neither well documented nor is there any explanation why a particular method of distribution has been chosen.

For example, the Grid Bag Layout from the Java AWT framework distributes the layout size discrepancy using weights which can be assigned to columns and rows (weighted distribution). This means generally that an item grows or shrinks by

$$\Delta \text{size}_\text{item} = \text{discrepancy} \cdot \text{weight}_\text{item} / \sum_{i \in \text{items}} \text{weight}_i.$$  

The Qt toolkit follows a different approach and distributes or takes available space equally from all items in the layout (equally distribution):

$$\Delta \text{size}_\text{item} = \text{discrepancy} / \#\text{items}$$

Another approach is implemented in Haiku\footnote{The Haiku Operating System, 2011, www.haiku-os.org}. Here, for all items in a group layout the sum of the quadratic item discrepancies is minimized (minimal deviation). This is similar to the approach already described in in Section 2.3.2.

4.2 Aesthetic Problems of Constraint-Based Layout

Describing soft-constraints using a linear or a quadratic objective function leads to different behaviors when distributing the size discrepancy to the layout items. In this section the behavior of preferred size soft-constraints is discussed from an aesthetic point of view.

4.2.1 Linear Objective Functions Cause Non-Determinism

An example for a simple homogeneous layout is a layout containing just three buttons with exactly the same properties. Figure 4.1 a) shows the resulting layout solved by lp_solve (linear objective function). As expected all hard-constraints are satisfied. The soft-constraints for the first two buttons are matched exactly, meaning the height is equal to the preferred size of the buttons. The only violated soft-constraint is the preferred height of the third button.

From the aesthetic point of view, this layout configuration looks odd. Because all buttons have the same properties, one would expect that all buttons take the same amount of space. The height ratio between the different buttons is not specified by the layout, but is a result of how the solver solved the constraints. It is theoretically even possible for the ratio to change non-deterministically during resizing. Following Gestalt principles, the three buttons with identical properties should be perceived as a group, which is not the case here.
4.2 Aesthetic Problems of Constraint-Based Layout

One solution to this problem is to manually specify a size relation between the related items, but this is extra work for the developer. A better solution is to leverage the advantages of a quadratic objective function, which minimizes the deviation to the preferred item size for each item, not just the sum of deviations of all items. Figure 4.1b illustrate how a quadratic objective function leads to the desired uniform result.

Proportionality Scale Variables

The problem of the linear objective function, as described above, could be partially solved by introducing one or more additional free global scaling variables $s$. Such a scaling variable can be used to make layout item sizes proportional to their layout item’s preferred size. To do so, the preferred size soft constraints $c_i$ have to be rewritten to:

$$c_i : \quad x_i - x_i,\text{pref} \cdot s = 0.$$  

However, the approach can only be used in special cases, i.e., when all layout items can be sized proportional to their preferred size. In case this requirement is no longer satisfied, the soft-constraints have to be violated again which leads to the same problem of non-determinism as described previously.

4.2.2 Spring Effects

One disadvantage of minimizing the deviation to the preferred sizes is that this sometimes leads to an unwanted spring effect. This is an analogy between preferred size constraints and mechanical springs, pulling or pressing an item to its preferred size.

The problem occurs when multiple “springs” are coupled and thus their strengths are combined to a resulting force. Such a coupling could, for example, be observed in a multi-row layout as shown in Figure 4.2. In the first row three “springs” and in the second row two “springs” are coupled. Because each button has the same preferred height, this
results in the same descriptive spring force $F_s$ for each button. Thus the first row has a “spring” force of $3 \cdot F_s$ and the second row a force of $2 \cdot F_s$. For the solved layout this means both rows have a different height.

![Diagram](https://via.placeholder.com/150)

**Figure 4.2**: Spring effect: Three buttons in the first row pull more strongly to their preferred size than the two buttons in the second row.

This is certainly a limitation of constraint-based layout because since all items have the same properties one would expect, according to the equality Gestalt law, two equal sized rows. In such a case a relation between rows has to be specified explicitly, e.g., by applying a hard-constraint that keeps the height of both rows constant. Another more general solution is to define preferred size constraints on whole rows and columns only.

Remember that a row or column is defined by two tabstops and at least one layout item between these tabstops. Rows are bordered by horizontal tabstops and columns by vertical tabs. Layout items connected to the same two tabstops are associated with the same column or row (Section 2.2).

To solve the spring effect problem, the preferred size constraint is only applied for rows and columns and not for each layout item. Since there could be multiple items in a row or a column, the weighted average of the preferred sizes of the items is employed, using the soft constraint weights as weights. In the upper example, this has the effect that both rows have the same preferred height, and thus get the same height when solving the layout.

### 4.3 Methodology

There are many ways to place layout items in a layout. Similar to a typesetting system, an important goal is to create layouts that are aesthetically pleasing for the user [68]. In this section the different solving strategies with regard to aesthetic perception are evaluated.

The differences in the perceived aesthetics of layouts generated by different solving strategies are small and difficult to measure. For some users it may not be obvious what
are the differences between the same layout rendered with different solving strategies. Furthermore, the criteria of aesthetics are subjective and vary between users.

An important aspect of this evaluation is the analysis of the resize behavior of a layout. Layouts should look pleasing at different sizes, not just for a particular initial size. Therefore, the evaluation will consider different sizes of the same GUI, a small size close to the layout minimum size and a large size approximately twice as large as the preferred layout size. Here, three solving strategies that place items in a one-dimensional and two-dimensional layout are analyzed. All layouts and solving strategies described in the following were implemented using ALM’s constraint system.

4.3.1 Single-Row Layouts

A very simple layout is a layout consisting of just a single row, e.g., a group of buttons arranged besides each other. Three different solving strategies to distribute the size discrepancy to the buttons in the row are evaluated.

Firstly, equal distribution gives each item the exact same amount of space in a line. Here the preferred size of an item is not taken into account. Note that generally the theoretical minimum size of a layout cannot be reached, i.e., when one of the layout items reached its minimum size then the other ones cannot be made smaller. In practice this can be solved by violating the equality constraint once an item reaches its minimum size. A disadvantage of this is that the items do not all reach their minimum size at the same time, but in a “staggered” fashion.

Secondly, weighted distribution keeps the size ratio between items in a line constant. This means a weight is assigned to each item. The layout item sizes are given by

\[
\text{Size}_{\text{item}} = \frac{\text{Size}_{\text{layout}} \cdot w_{\text{item}}}{\sum_{i \in \text{items}} w_i}.
\]

For this evaluation the item weight is chosen as the relative item size at a small initial layout size, where the items are close to their preferred size.

Thirdly, the item sizes are determined by calculating the minimal deviation from the preferred size for each layout item. This can be achieved with a solving strategy that uses a quadratic objective function (see Section 2.3.2). For very large layouts the minimal deviation approach converges to the equal distribution approach because the preferred size becomes small compared to the actual item size. For layout sizes close to the layout’s preferred size the result is close to the weighted distribution because the weights are chosen to match the preferred size.

An example for a simple two-button row at small layout size is shown in Figure 4.3. For simplicity no item maximum size is taken into account. Maximum sizes result in more complex layouts and thus make the analysis of the evaluation results more complicated.
For example, when the maximum of one layout item in an equal distribution layout is reached the layout cannot be resized any further.

![Diagram](image)

Figure 4.3: Two different solving strategies for a simple two-button layout: (a) minimal deviation and (b) equal distribution.

### 4.3.2 Multi-Row Layouts

Another interesting question is how the three different solving strategies from the previous section perform in a multi-row layout. In this evaluation a two-row and a three-row layout is evaluated with regard to its perceived aesthetics. The two-row layout has three buttons in the first row and two buttons in the second row. In the three-row layout, another row with two buttons is appended.

The first case that was considered in this study is a minimal deviation approach, where the sizes of the items are chosen as close to their preferred sizes as possible. However, as shown in Figure 4.4 (a), this leads to an irregular, misaligned appearance, which is unusual for multi-row layouts. In multi-row layouts, the items are usually aligned in a grid, which comes naturally when using a common grid-bag layout model. Therefore, some layouts are evaluated where the items are aligned in a grid, with each item taking up as many cells as seems natural for their given preferred size. More specifically, two different solving strategies for the grid-aligned layouts are compared: either the space of the items in the first row uses an equal distribution, as in Figure 4.4 (b), or: it uses a weighted distribution, as in Figure 4.4 (c). The sizes of the items in the second and third row follow from their alignment with the first row. If the minimal deviation approach were used for the first row, with the items in the second and third row being aligned to the first row, the resulting layout would look very similar to the cases (b) and (c), depending on the layout size. Therefore, this case is not examined.

### 4.3.3 Methodology

Participants were asked to compare various single- and multi-row layouts shown on paper. Each layout was rendered at two different widths: a small width close to the minimal width, and a larger width about twice as wide as the small width. The participants were
4.4 Results and Discussion

The study had 15 participants. All of them had a Computer Science background and most of them had experience in designing graphical user interfaces but only casual experience in graphics design. While it was easy for them to see the differences between the given layouts, it was not so easy for them to judge them (see Figure 4.5). To determine the significance of the differences in preference a one-sided Welch t-test is used.

4.4.1 Significant Preference Differences

For all the single-row layouts there is no significant difference between the minimal deviation and the equal distribution layouts. For large layouts this is expected because minimal deviation and equal distribution layouts look almost identical. For the three-
button layout in its large size the minimal deviation and the equal distribution layouts are significantly ($p < 0.05$) better than the weighted distribution layout.

For the multi-row layouts the grid-aligned layout is clearly preferred over the unaligned minimal deviation layout. For the two-row layout, the deviation layout gets only 10% for the small size, and 15% for the large size. The scores were even worse for the three-row layout, where only two participants liked the minimal deviation layout.

This is an interesting finding and means a symmetrical layout where the layout items borders are aligned to each other is more pleasant than a layout where each individual item gets the space closest to its preferred space. This can be explained with Gestalt psychology: objects that are aligned to each other are perceived as a group, thus it is easier for us to understand the layout, which makes it preferable [47]. For the usage of a constraint-based layout model such as ALM, this indicates it is better to reuse existing tabstops to create more alignment in layouts.

### 4.4.2 Preference Trends

Apart from the clear findings of the previous section, some other interesting observations can be made from the taken data. Firstly, a similar tendency as in the multi-row layouts can be seen for the single-row layout at large size: for large layout sizes, the equal distribution layout is liked more than the weighted distribution layout. When combining the results from the two- and the three-rows layouts, this tendency is significant at a $p < 0.1$ level.

A contrary tendency can be seen with small layout sizes. For the multi-row layouts with small layout sizes, the weighted distribution layout is preferred over the equal distribution layout. When the two- and three-row layout results are combined, this is significant at a $p < 0.1$ level. In the small single-row scenario, where minimal deviation layout and

---

Figure 4.5: Results from Likert-scale questionnaire.
weighted layout look identical, this observation cannot be made. However, at least it could be said that the equal distribution layout does not lead to better results than the weighted distribution layout.

To sum up, there is a tendency that at small layout sizes weighted distribution layouts are preferred more than equal distribution layouts. At large sizes the equal distribution and minimal deviation layouts are preferred above the weighted distribution approach. This means that the minimal deviation approach, which is equal to the weighted distribution at small layout sizes and very similar to the equal distribution at large layout sizes, is well-suited for small and large layout sizes. Furthermore, the minimal deviation solution scales smoothly down to small layout sizes where it seems to be important that each item gets a fair amount of the size discrepancy. For large layouts the minimal deviation approach roughly distributes all layout items equally, which has been found to be the most preferred solution for large layouts.

4.4.3 Qualitative Responses

When asked about the criteria for preferring one layout over another, the most frequent answer from the participants was that alignment of the buttons is an important factor (6 participants). For others participants, enough space for the button labels and the button margins was important (3 participants). Furthermore, three participants stated that they prefer layouts with equal button size.

These qualitative statements are consistent with the findings from the quantitative layout evaluation. Firstly, aligned multi-row layouts are preferable over unaligned layouts. Secondly, for small layout sizes, there is the trend that minimal deviation layouts, where the size discrepancy is uniformly distributed on the button margin, are preferred. Thirdly, for large layout sizes, each layout item should get the same amount of space, which is the case in equal distribution and minimal deviation layouts.

4.5 Conclusion

Layout managers are a convenient way to arrange items in a layout, independent from the actual layout size. The used layout model needs to define a strategy to distribute additional or lack of space, i.e., the discrepancy between the preferred and the actual layout size. Looking at principles such as the Gestalt laws, it is clear that the distribution strategy is likely to affect the aesthetics of a layout. However, the review of existing layout models shows that there is no agreement on how this is best done.

Using constraints is the most powerful approach for layout management, and most other approaches can be reduced to it. To deal with conflicting constraints such as pre-
ferred sizes, constraint solvers have to optimize an objective function. Two issues that affect the constraint-based layout model have been identified: a linear objective function can lead to nondeterministic layouts, and spring effects can lead to layout distortions when using a quadratic objective function. For the latter one, a solution is to make sure that the preferred size constraints are specified for rows and columns rather than for individual items.

In an empirical evaluation, the effects of three layout solving strategies – equal distribution, weighted distribution and minimal deviation – on aesthetics has been investigated. The evaluation shows that while a weighted distribution tends to be preferred at small layout sizes, an equal distribution is preferred at large layout sizes. As a good tradeoff, the minimal deviation approach yields aesthetically pleasing results at small and large layout sizes. Another finding is that users prefer GUI layouts in which the items are aligned over layouts with less alignment – a finding that is consistent with the Gestalt principles.
In this chapter Stack & Tile is presented. Stack & Tile is an extension of the traditional desktop metaphor and allows users to group windows by stacking and tiling them into groups. This can, for example, be used to group windows of different applications by task. Because windows within a Stack & Tile group behave like a single window, window management can become much faster, e.g., moving one window moves all windows in a group. Furthermore, windows in a Stack & Tile group are always aligned so that available screen space can be used more efficiently, e.g., resizing one window in a Stack & Tile group aligns other windows. With the exception of stacked windows, that are either completely visible or completely occluded, all windows in Stack & Tile group are non-overlapping. Stack & Tile uses the constrain-based Auckland Layout Model (ALM) to describe the window layout. In a user evaluation it has been found that Stack & Tile performs significantly better for multi window tasks and task switching. Moreover, Stack & Tile is already available and has been used for over two years. In a web survey it has been investigated how and if users actually use Stack & Tile. In this chapter presented evaluation and web survey have been published at the INTERACT’13 conference [115].

The following Section 5.1 describes the Stack & Tile system. Section 5.2 gives a detailed overview of the edit operations used in Stack & Tile. Related work is discussed in Section 5.3. The Stack & Tile user evaluation is presented in Section 5.4 and the web
survey in Section \[5.5\]

\section*{5.1 Window Management with Stack & Tile}

There are various approaches that try to help users to manage windows on the desktop. Some automatically tile windows beside each other to make windows visible all the time, other approaches help to group windows by tasks. Automatic tiled windows have been found to be superior compared to overlapping windows in certain use-cases \[15, 64\]. Grouping windows by task can also increase productivity \[64, 88\]. This project investigates the development of a window manager that helps users to manage their windows more effectively. In Stack & Tile, users stack and tile windows together to create window groups suitable for their own needs. At the same time Stack & Tile combines the advantages of tiled windows and the freedom of overlapping windows. Furthermore, arranging windows on the screen requires less effort in Stack & Tile. Multiple windows can be resized, activated or moved with only one operation.

Currently there is a fully working Stack & Tile implementation available. It is integrated into the window manager of the open source operating system Haiku\(^1\). Stack & Tile is well known and appreciated in the Haiku community. It has been showcased at Haiku community meetings, and many users appreciated the integration into the Haiku window manager. The source is available in the git repository \url{git://git.haiku-os.org/haiku} in the directory \texttt{src/servers/app/stackandtile}.

Stack & Tile offers two operations that can be used to connect windows. Firstly, tiling of windows, that means that windows are arranged beside each other. Frequent read-write tasks are a good example where tiled windows can give a user an outstanding facility to increase their productivity. Secondly, stacking, that makes use of the tab-like appearance of the Haiku window title bars. Windows can be stacked on top of one another, and the title tabs are automatically arranged beside each other (see Figure \[5.1\]).

The stacking behavior is comparable with a tab bar, e.g., in a tabbed web browser. Window groups created with Stack & Tile behave like a single window; e.g., activating one window in a Stack & Tile group brings all the windows of the group to the front, and moving one window moves all the other windows by the same offset.

A Stack & Tile operation can be triggered by holding the Stack & Tile key, that is by default the Windows key, and dragging a window near to another window (see Figure \[5.2\]). The dragged window is called the candidate window, and the window that it is dragged to is called the parent window. In this manner, stacking and tiling groups can be created. A window can be removed from a Stack & Tile group by holding the Stack & Tile key and dragging the window away from the group.

\(^1\)The Haiku operating system, \url{www.haiku-os.org}.
5.1 Window Management with Stack & Tile

Figure 5.1: Example of a Stack & Tile group. On the right side, three windows are stacked into a stacking group. Tiled to this group on the left side are a text editor and at the bottom a terminal.

Figure 5.2: Moving a window while holding the Stack & Tile key initiates stacking or tiling. Window tabs and borders are highlighted in gray, respectively. Left: The editor window is going to be stacked on top of the Terminal. Right: MediaPlayer is going to be tiled to the Media folder.

5.1.1 Stack & Tile Window Groups

Windows that have been connected using a tiling or a stacking operation are organized in window groups. For example, three windows tiled in a row belong to the same window group. When stacking another window to one of the windows in the row, this new window becomes a member of the same group. All existing groups have to be disjunctive, which
means that a window can only be in one group at a time.

A window manager manages a set of windows which are ordered according to their depth layer:

\[ w_0 < w_1 < \ldots < w_n. \]

Windows are drawn in this order, i.e., the backmost window \( w_0 \) is drawn first and all following windows afterwards. Like windows, groups are ordered in different layers:

\[ g_0 < g_1 < \ldots < g_n. \]

Each group \( g \) consists of a list of ordered windows. A complete window set might look like:

\[ w_0 < w_1 < w_2 < w_3 < w_4 < w_5 < w_6 < w_7 \]

When moving a window that belongs to one group to a different layer position, all the other windows in the group have to be moved accordingly. For example, when activating \( w_1 \) from (5.1), \( w_1 \) becomes the front-most window and the whole group \( g_0 \) moves to the front.

\[ w_0 < w_4 < w_5 < w_6 < w_7 < w_2 < w_3 < w_1 \]

### 5.1.2 Stack & Tile Layout

The layout of a Stack & Tile groups is described using ALM as described in Section 2.2. Windows are placed into layout areas and the layout areas are connected to tabstops. Solving the resulting constraint system leads to the actual window position on the screen.

For example, when adding a window to the right side of an existing window, the left border of the new window is connected to the right tabstop of the existing window; the top and bottom borders are connected to the top and bottom tabstop. On the right side of the new window, a new vertical tabstop is created and connected to the right border. Figure 5.3 shows three windows connected in this way.

The rectangular spaces between the tabstops are called areas. All areas not covered by a window but adjacent to at least one window border, and all rectangular unions of such areas, are called free areas. There are four corners located at each intersection of two tabs. The corners in a free area that are located near a window are called free corners (see Figure 5.3).

In contrast to GUI layout items, windows do not have a preferred intrinsic size in the first place. However, a natural value for a preferred windows size is the current window
size. When inserting a window into a Stack & Tile group or resizing a window of a Stack & Tile group, the preferred size value of the modified window’s layout area is updated to the new window size. Furthermore, when solving the layout the updated window temporarily gets a very high weight for its preferred size constraints; one can say the soft constraints get hard to violate. For example, this ensures that a resized window really gets resized as the user intended.

5.2 Stack & Tile Operations

Dragging a window while holding the Stack & Tile key at the same time can either trigger a stacking, a tiling or a remove operation. The last operation removes a window from a Stack & Tile group. To detect a possible stacking or tiling operation all window groups, as well as windows that are not in a group, are searched in reverse layer order, i.e., from front to back. For example, for the following window set

\[
\begin{align*}
  w_0 < & w_1 < w_2 < w_3 < w_4 < w_5 < w_6 \\
  & g_0 \quad g_1
\end{align*}
\]  

a matching operation is searched first in group \( g_1 \), then for window \( w_4 \), then in group \( g_0 \), and finally for window \( w_0 \).

In some cases a user interaction could be interpreted either as a stacking or as a tiling operation. To solve this problem the stacking operation is prioritized over the tiling operation. This means that the search for a possible stacking parent window is performed first, and only if no stacking parent window is found, a free area for tiling is sought. This ensures that a window can only be stacked or tiled to windows in one window group, e.g., a trigger for a tiling operation could not be valid for two different Stack & Tile groups.
5.2.1 Stacking

Briefly, a stacking operation is triggered by moving the title tab of the candidate window onto the tile tab of a parent window while holding the Stack & Tile key. To be more precise, the candidate window has to be dragged by the title tab. Furthermore, the upper edge of the candidate title tab must be on the parent title tab and the $x$-position of the mouse cursor must be in the $x$-range of the parent title tab. When two valid stacking candidates are found, the window title tabs are highlighted (see Figure 5.2).

By releasing the Stack & Tile key or dropping the candidate window, the window borders of the candidate window are connected to the tabstops of the parent window. As a consequence, if a stack of windows is part of larger Stack & Tile group, every window in the stack is connected to the same tabstops in the group. Hence, windows can easily be removed from a stack, leaving the remaining group intact.

When stacking a candidate window, its title tab is moved to the right side of the parent window’s title tab. If necessary the window title tabs of the other windows in the stack are moved to make the new tab fit into the row of tabs. In this way, new window title tabs can be inserted at each position in the tab bar except the first one. To move a title tab to the first position one can re-stack the first window in the group or move the tab manually using Haiku’s window management operation for moving tabs.

5.2.2 Tiling

When searching for a tiling operation, first a free corner to connect a corresponding corner of the candidate window has to be found. The top-left candidate corner can only be tiled to a top-left free corner, and so on. A tiling trigger area is defined around each free corner, as shown for the top-left corner in Figure 5.4. By dropping a corner of a candidate window into a suitable trigger area while holding the Stack & Tile key, the tiling operation is applied to the corresponding free corner.
Consider the following case illustrated in Figure 5.5. When dropped, the blue candidate window activates the tiling trigger for the bottom-right and the top-right free corner. The problem is that these two free corners are in two different free areas, and tiling to both of them would cause overlap. To prevent this, the interaction mechanism has been changed to ensure that only one free corner is chosen.

![Figure 5.5: The trigger criteria are satisfied by the blue candidate window for two free corners in two different free areas: the upper and the lower free area.](image)

To choose only one free corner in such a case, the free corners are prioritized. Following the left-to-right, top-to-bottom reading order of the English language, the top-left corner gets the highest priority, followed by the top-right, the bottom-left and the bottom-right corners. In this manner, there is at most one free corner where a candidate window can be tiled to. In the above example the blue window would be tiled to the upper right free corner.

Starting from the free corner in the trigger area, the optimal free area for the candidate window has to be found to tile the candidate window. A good choice is the free area with the smallest deviation in size from the candidate window. The candidate window is then connected to the outer tabstop of this optimal free area. Tiling windows in this way ensures that tiled windows within a group never overlap each other because by definition a window can only be inserted at a free area in a group.

### 5.2.3 Removing Windows from a Group

A window can be removed from a Stack & Tile group by holding down the Stack & Tile key and dragging the window away from the group. After removing the window from the group, the window behaves just like an ordinary window in the desktop metaphor. In case
the removed window was the only connection between other windows in the group, the

The removed window was the only connection between other windows in the group, the
group is split into several independent groups. For example, Figure 5.6 shows a Stack &
Tile group that needs to be split when window C is removed. After the removal of C
there are two new groups: A + B and D + E.

![Figure 5.6: Removing window C makes it necessary to split the Stack & Tile group.](image)

To be more formal, a Stack & Tile group has to be split if there are windows in the
groups that are not connected by a path. Two windows are on the same path if they
share at least one vertical and one horizontal tabstop and are not positioned diagonal.
This means that, one of these tabstops must be connected to the same border of both
windows, e.g., to the top of the first and to the top of the second window.

If windows are not connected by a path, all windows that belong to the same path
are split into a new group while keeping their path connections to their neighbors. This
ensures that the split groups are setup correctly, preserving their old structure.

5.2.4 Traditional Window Management Operations

When interacting with a Stack & Tile group the semantics of the traditional window
management operations change slightly. Stack & Tile applies window management op-
erations to multiple windows, which has already been considered to be helpful in Elastic
Windows [64].

**Activating** one window in a Stack & Tile group raises all windows in the Stack &
Tile group. Only the window that triggered the group operation gets the input focus.

**Moving** a window by a certain offset also moves all other group windows by the same
offset. This means windows in a Stack & Tile group keep their position relative to each
other.

**Resizing** one window in a Stack & Tile group leaves all windows in the group aligned
to each other. For example, windows that are tiled to a resized window are moved
or resized accordingly. This is done by temporarily setting a high priority for the size
constraint of the resized window. In this way the resized window gets its new size and
the other windows adapt according to the solution of the constraint system.
Hiding or showing a window also hides or shows all other windows in the group. Thus, all windows in a Stack & Tile group are either hidden or shown.

Move a window to a different workspace also moves all other windows in the group to this workspace.

5.2.5 Magnetic Border of a Stack & Tile Group

The standard Haiku window manager provides a magnetic window border. This means when the user moves a window close to a screen edge the window border snaps to the actual screen edge. The window manager only takes care of the magnetic border of the currently moved window. In a Stack & Tile group windows can be indirectly moved to the screen edges by moving another window of the group. For this indirectly moved window the magnetic border is not calculated by the normal window manager implementation. This is quite unintuitive for the user as a Stack & Tile group feels like a single window.

To avoid this problem a minimal rectangular outer frame containing all window of the group is calculated. When moving a Stack & Tile group the magnetic border is also applied to this outer frame. The magnetic border for the window of the group actually moved is still applied.

5.2.6 Offscreen Windows

When windows of a group are moved completely out of the screen (see Figure 5.7), and the last window of that group that is still remaining visible is removed, then the offscreen windows may get stranded. The user is no longer able to access the windows or move them back to the screen. To avoid such a situation, stranded offscreen windows are automatically moved back onto the screen. To be more precise, the window nearest to the visible screen area of the stranded group is moved back onto the screen. The window is only moved to the extent that a relatively small part becomes visible again. In this way the user is not distracted by the sudden appearance of a Stack & Tile group but is still able to drag the complete group back onto the screen.

5.3 Related Work

Novel techniques for overlapping windows such as tiling windows to a master window or organizing them in tabs have already been proposed in 2001 [12]. However, these new techniques have not been evaluated for traditional window managers.

In a comparison between overlapping, stacked and “piled” windows it has been found that tabbed interfaces are doing well when it comes to finding a document [62]. The study only looked at windows of the same application, while our study also looked at
Figure 5.7: Windows in a Stack & Tile group can be offscreen.

tasks involving windows of different applications (in contrast to windows of the same application, they are not grouped in the taskbar). Furthermore, they did not consider the presence of windows that were not part of the task at hand.

The Google Chrome browser can stack web pages running in different processes and extends the usage of the tab interface to stack different instances of the same application. The Stack & Tile approach is more general and is working not only with windows of a particular application, but with arbitrary windows of arbitrary applications.

Roughly one year after the first release of Stack & Tile a similar stacking feature was implemented in the KDE desktop environment. Although KDE has an optional Notion-like tiling window manager (see below), a combination of stacking and tiling in the standard KDE desktop is not possible.

Tiling window managers allow users to tile windows beside each other. Windows are arranged so that they do not overlap each other and are aligned without gaps and fragmentation. In general, windows are arranged automatically by the window manager, using the whole screen, but the user has the ability to rearrange the window tiling using different layouts. For example, the tiling window manager Notion allows the user to rearrange the tile layout manually and multiple windows can be stacked into one tile. By comparison, Stack & Tile integrates seamlessly with the traditional desktop metaphor. In Windows 7, two windows can be tiled horizontally using the whole screen space. However, more complex tiling layouts with more than two windows, like in Stack & Tile, are not possible.

Tiling windows can help the user to organize their windows better. Already in old studies it has been shown that tiled windows can be superior to overlapping windows in certain use-cases. For example, if all window content fits into the allocated tiles, the tiling approach leads to shorter task completion times. Another study found that the completion time is lower for tiled windows when comparing information sources in

---

2Google Chrome. [www.google.com/chrome](http://www.google.com/chrome)
3KDE Desktop Environment. [www.kde.org/](http://www.kde.org/)
4Notion Window Manager. [http://notion.sourceforge.net](http://notion.sourceforge.net)
multiple windows and scanning through windows of a certain window group [64]. Stack &
Tile leverages the advantages of tiled windows but integrates tiling with the traditional
overlapping window management.

It is quite common that users work on different task in parallel and switch back
and forth between different windows [102]. Typically, windows from different tasks are
overlapping each other, and task switching involves the manipulation of many windows.
One approach to address this is to let the user group windows by tasks [64]. Another
approach is to analyze the user activities to assign windows to tasks automatically [59]
[88]. To reduce the probability of overlapping existing windows on the screen and to
group windows by task, the concept of multiple virtual workspaces can be used [49].
Another option is to increase the physical workspace by attaching multiple monitors to a
computer [57].

However, often windows overlap other windows, and some approaches use alternative
methods to make overlapped windows temporary available without losing the focus on
the active window [21] [22]. For example, an occluded window can be made visible by
folding the overlapping window back like a piece of paper [22].

The Scwm window manager [6] gives the user the opportunity to specify relations and
constraints for each window. For example, it is possible to set a relative distance between
two windows or set the minimal or maximal size of a window. With this approach it is
also possible to tile windows beside each other. However, because constraints in Scwm are
on a lower level of abstraction, tiling is not as easily accessible as in other tiling window
managers and Stack & Tile.

Various approaches try to automatically group windows by tasks [14] [103] [109]. For
example, WindowScape creates a timeline of previous window configurations [103]. Win-
dowScape also allows assigning windows manually to a certain task. This is similar to
an activity-centered desktop [2] where applications can be assigned to activities. Also
Stack & Tile can be used to group windows by tasks. Additional to that Stack & Tile
helps the user to create optimized group layouts.

5.4 Experimental Evaluation

In this section Stack & Tile is compared to a traditional window manager in a controlled
experiment, using tasks that are representative of certain use-cases that involve multiple
windows (multi-window tasks). Stack & Tile makes it possible to group windows with
direct manipulation operations, and offers window management operations that affect all
the windows in a group. This can be used to manage occlusion and quickly switch between
groups of windows related to different tasks. An interesting question is how much time
can be saved using Stack & Tile, and if the time taken to set up a Stack & Tile group
is reasonable compared to the time saved. The following hypotheses about Stack & Tile are formulated for this experiment:

**H1** The task completion time for multi-window tasks is lower.

**H2** Switching between tasks that each involves multiple windows is faster.

**H3** The time for setting up Stack & Tile groups is acceptable (in context of the task).

**H4** Stack & Tile is perceived as more pleasant to use.

### 5.4.1 Methodology

A within-subject study is conducted, i.e., each participant performed two runs of tasks, one run with Stack & Tile and one run without Stack & Tile. To avoid order bias, the order of the runs was alternated between participants, i.e., the study was counterbalanced. Furthermore, there were two different sets of similar tasks for each run. Also the order of the sets of tasks was permuted. Each run included a window setup phase followed by three question tasks and a group-switching task.

At the beginning of the user evaluation, each participant got an introduction into the Haiku window manager, its taskbar and Stack & Tile. After that the participant had time to get familiar with the system. Then each participant performed a training run, which was similar to the main runs but with shorter tasks. Participants were allowed to use different methods to switch between windows, e.g., Alt+tab or the taskbar.

Each run was guided by instructions shown in a small instruction window that was placed at the lower right screen corner on top of all other windows (see Figure 5.8). The size of the instruction window was chosen to be as small as possible so as not to interfere with other windows on the screen. At the beginning of each task the instruction window was in fullscreen mode, so that the participant was not distracted by other windows on the screen. After the participant had internalized the instruction, she had to press the start button and a timer was started. This resized the instruction window back to the smaller version in the lower right screen corner. For the question tasks, the instruction window also contained a three-point multiple choice question. When the participant had completed the task, she had to press a finish button to stop the timer. Pressing the finish button adds a small and fairly constant offset to the task completion time, which does not affect the outcome of the comparison. Afterwards the instruction window was shown in fullscreen mode again with the next instructions.

Each task involved multiple windows, and all windows of all tasks (in total 12 windows) were open all the time. The author considers this as a realistic number of windows but it is not unusual that users have even more windows open at a time [106]. Here the efficiency of managing multiple windows and tasks is interesting. For that reason the time spent
on activities unrelated to window management should be minimized. Thus the question tasks were designed to be very easy but answers for a question were not guessable. The three question tasks were targeting H1, and each task was addressing a different use-case.

Setup Phase

The first step of a run was to open all windows for all tasks. For each task participants got a description in the instruction window about what windows were involved in the task. Then the windows were opened automatically so that the participant could get familiar with them before starting the tasks. When using Stack & Tile, there was an extra step where the participant had to stack or tile the windows, depending on the tasks. Here the setup time to create a certain Stack & Tile group was measured.

Task I

The first question task was about finding a picture in a set of five pictures and answering an easy question about it. Each picture was opened in a single window and there was one question for each picture (e.g., Figure 5.8). This targeted the use-case of working with multiple windows of the same application, where no data needs to be exchanged between the windows. It simulates typical lookup tasks where a user needs to activate the window that contains a particular piece of information.

When it comes to finding a certain window, it can be helpful if all windows of interest for the task have been stacked previously. In this case, activating one window of the stack brings the whole group to the front, and the correct window can be chosen through the tab interface. When using Stack & Tile the participant was asked to stack all five pictures on top of one another, to take advantage of this behavior (Figure 5.9). Without Stack & Tile the participant had to search for the right window on the desktop or use the taskbar that lists all windows ordered by application. Because the taskbar groups windows by
applications the taskbar provides a similar grouping facility as a window stack and it is unclear if Stack & Tile is advantageous.

Task II

The second question task was similar to Task I. Here, three windows were involved: a web page, a PDF document and a text document (Figure 5.10). This targeted the use-case of working with windows of different applications where no data needs to be exchanged between the windows. As before, there was one question for each document, and under the Stack & Tile condition all windows involved in the task had to be stacked first. For each question the participant had to find the right document and answer one question about it, e.g., “What is the first word of the second paragraph in the web page?”

There are two reasons why it is expected that navigation in this more heterogeneous group is more difficult than in Task I. Firstly, because windows are opened with different applications they are not grouped together in the taskbar anymore. Secondly, because documents of different types also differ visually users have to find other ways to associate a window with a certain group; one cannot play on visual similarities anymore.

Task III

The third question task targeted the use-case of working with windows of the same application where data needs to be exchanged between the windows. To simulate this use-case
a simple coordinate treasure map game was chosen. A treasure map is a 7x5 table with each cell containing either a coordinate pointing to a cell in another map or a treasure. For example, the coordinate M2(D,5) points to map 2, cell (D,5). Starting from an initial coordinate the participant had to visually follow the path through four different maps to find a “treasure” (see Figure 5.11). The treasure was always reached after three steps and the path crossed all four maps in random order. When the treasure was found the kind of treasure had to be selected in the instruction window. This had to be repeated three times, each time starting at a different start point in the first treasure map.

While this task may seem artificial, it does simulate real tasks where related information has to be collated from multiple sources. For example, a real task of that kind would be looking up the location of an appointment mentioned in an email from a calendar, and then checking a booking for that location. Many such tasks involve tabular information, as simulated by the treasure maps.

There were four treasure maps, each opened in a text editor window. When using Stack & Tile, the participant had to tile the four editor windows beside each other in a 2x2 layout. In this way it was possible to display all four maps without occlusion on the screen, i.e., all maps were completely visible.
Figure 5.11: Task III: Participants have to find their way through four treasure maps.

Task IV: Group switching

This task evaluated the efficiency of switching between different tasks (H2). All windows from the previous tasks were used, and the user was asked to bring all windows of one task to the front at the same time. Without Stack & Tile this can be done by directly clicking the windows on the desktop, or by activating them from the taskbar. When using Stack & Tile only one window had to be activated to bring the whole group to the front. It is expected that Stack & Tile is clearly faster here, and this task was included to shed light on how much faster it actually is. Because the groups had three, four and five windows, a correlation to the activation time was expected for non-Stack & Tile condition.

Questionnaire

After finishing both runs the participant was asked to fill in a Likert-scale questionnaire and give some general comments about what they liked or disliked about Stack & Tile. Furthermore, the participant had to estimate their usage in percentages of the following window management techniques: taskbar, short cuts, direct window access, virtual desktops and others.
5.4 Experimental Evaluation

5.4.2 Results and Discussion

There were 30 participants with an average age of 31 (σ = 6). 6 of them were female. Most of them (∼ 70%) were software developers or students of Computer Science or Software Engineering. Of the participants 15 were from the Haiku community and 7 of them had used Stack & Tile before. For the users who had used Stack & Tile before, it has been found that they performed only slightly better, thus there is no great differentiation between them.

For the analysis of the task completion times the difference between the task completion time with Stack & Tile $t_{S&T}$ and the task completion time without Stack & Tile $t_{noS&T}$ was calculated: $\Delta t = t_{noS&T} - t_{S&T}$. A pairwise, one-sided Wilcoxon rank-sum test was used to calculate the probability $p_{wrs}$ that $t_{S&T}$ is generally smaller than $t_{noS&T}$. The Wilcoxon rank-sum test was chosen because the task completion times are only roughly Gaussian-distributed.

For each question task three answer options were given, with only one correct option. In only ∼ 3% of all cases a question was answered wrongly. Furthermore, there was no difference between the error rates with and without Stack & Tile. Thus it can be said that it was easy for the participants to answer the questions. It has been observed that participants followed the instructions carefully, and even when they chose the wrong answer they applied the necessary window operations. Because the primary focus was not on how accurately a task was performed but how Stack & Tile affects window management, wrong answers were not removed from the result set.

Task I Results (5 Pictures)

Figure 5.12 depicts the task completion times for the first task. Table 5.1 shows the average task completion time $\overline{t}_{S&T}$ under the Stack & Tile condition, the average time difference $\Delta t$, their standard deviation $\sigma$ and the Wilcoxon rank-sum test p-value. For all five questions in Task I, Stack & Tile had significantly shorter task completion times. Thus H1 can be accepted for Task I.

Answering the first question had the longest completion time. This can be explained by an additional step: first the Stack & Tile picture group had to be found and activated. After that participants were able to just select the next picture from the tab bar and the task completion time stayed roughly constant. The time difference $\Delta t$ decreases from Picture 1 to Picture 5. A possible explanation is that for the non-Stack & Tile conditions after some time all pictures were moved to the front and it became easier to find the right window. Another explanation is that the participants got into the routine of selecting the right picture from the taskbar.
UI Customization in the Large: Stack & Tile

Figure 5.12: Boxplot for the task completion time for Task I.

Table 5.1: Task completion times in [s] for Task I.

<table>
<thead>
<tr>
<th>S&amp;T</th>
<th>σ</th>
<th>Δt</th>
<th>σ</th>
<th>p_wrs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Picture 1</td>
<td>12</td>
<td>5</td>
<td>11.8</td>
<td>0.01**</td>
</tr>
<tr>
<td>Picture 2</td>
<td>8</td>
<td>5</td>
<td>5.7</td>
<td>&lt; 0.01**</td>
</tr>
<tr>
<td>Picture 3</td>
<td>8</td>
<td>4.3</td>
<td>3</td>
<td>&lt; 0.01**</td>
</tr>
<tr>
<td>Picture 4</td>
<td>8</td>
<td>6.5</td>
<td>3</td>
<td>&lt; 0.01**</td>
</tr>
<tr>
<td>Picture 5</td>
<td>7</td>
<td>3.2</td>
<td>2</td>
<td>&lt; 0.01**</td>
</tr>
</tbody>
</table>

Task II Results (3 Documents in 3 Apps)

The results are shown in Figure 5.13 and Table 5.2. For the web page question there was no significant completion time difference. However, for the two questions following, Stack & Tile yielded a significantly better performance.

So why did participants have problems with the web page question while they were fine with the first picture question in Task I? A possible explanation is that the window with the web page was harder to activate than the windows with the pictures. When using direct activation by clicking on a window, all a participant had to do to select the Stack & Tile group of picture windows was to click any window with a picture on the screen, since there was only one group with pictures in it. Once the group was activated, the tab interface could be used to raise the right picture. However, there were two window groups with textual documents in it, so activating the window group containing the web page was not as simple as clicking any window containing text. When using the taskbar to activate a window, for the window group with the pictures there was exactly one group of windows listed in the taskbar, as all the pictures were shown using the same application. However, the documents for Task II were all opened with different applications and hence
there was a different entry in the taskbar for each of the windows. The taskbar gave no hint that the windows were grouped together, as it was for the five pictures. In fact, the taskbar obfuscated the Stack & Tile grouping of the windows, by grouping them by application together with other windows belonging to different Stack & Tile groups.

Hypothesis H1 cannot be accepted for the initial question of Task II. It is reasonable to assume that once a desired window group is activated, the advantages of Stack & Tile show more clearly. To facilitate this, as a future work, the taskbar could be changed to group windows by their Stack & Tile groups.

![Figure 5.13: Boxplot for the task completion time for Task II.](image)

**Table 5.2: Task completion times in [s] for Task II.**

<table>
<thead>
<tr>
<th></th>
<th>$\bar{t}_{S&amp;T}$</th>
<th>$\sigma$</th>
<th>$\Delta t$</th>
<th>$\sigma$</th>
<th>$p_{wrs}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Web Page</td>
<td>17</td>
<td>7.7</td>
<td>-1</td>
<td>11.5</td>
<td>0.65</td>
</tr>
<tr>
<td>PDF</td>
<td>10</td>
<td>13.9</td>
<td>5</td>
<td>18.2</td>
<td>&lt; 0.01**</td>
</tr>
<tr>
<td>Text file</td>
<td>9</td>
<td>4.85</td>
<td>8</td>
<td>10.9</td>
<td>&lt; 0.01**</td>
</tr>
</tbody>
</table>

**Task III Results (Treasure Maps)**

The results of Task III are shown in Figure 5.14 and Table 5.3. When using Stack & Tile, finding the first treasure required a lot less time (21s). This time difference decreased to 7s and 5s for the second and third treasure. Hypothesis H1 can clearly be accepted for Task III.

From observing the participants during the task this can easily be explained. Without Stack & Tile, many participants first tried to position all treasure maps in a 2X2 grid to make them visible at the same time. Thus, finding the following treasures became easier for them.

However, finding the second and third treasure is still significantly faster using Stack & Tile. A reason for that is that many participants did not manage to align the treasure
maps precisely without occlusion and thus had to rearrange windows for the later tasks. Moreover, some participants accidentally activated a non-task related window that made more window operations necessary. This is consistent with the findings from Elastic Windows [64] where they found that setting up and working with overlapping windows becomes more difficult for an increasing number of involved windows.

Figure 5.14: Boxplot for the task completion time for Task III.

| Table 5.3: Task completion times in [s] for Task III. |
|----------------------------------|-----------------|-----------------|-----------------|-----------------|
|                                | $t_{S&T}$ | $\sigma$ | $\Delta t$ | $\sigma$ | $p_{ars}$ |
| Treasure 1                     | 23        | 15.6     | 21           | 22.7           | < 0.01** |
| Treasure 2                     | 19        | 8.6      | 7            | 14.5           | 0.01**   |
| Treasure 3                     | 15        | 5.1      | 5            | 8.1            | < 0.01** |

Task IV Results (Activate Groups)

As expected, activating all windows of a task is much faster using Stack & Tile (Figure 5.15 and Table 5.4). This clearly supports hypothesis H2. However, no significant correlation between the number of windows of a task and the activation time could be detected. Here, the heterogeneous group of Task II resulted into the largest $t_{S&T}$ and $\Delta t$. Activating the groups containing only windows of the same application seems to be easier than selecting a group containing windows of different applications. This is consistent with the results of Task II.

Stack & Tile Setup Time Results

To assess whether the time for setting up Stack & Tile groups is acceptably low (H3), the average setup time $\bar{t}_{\text{setup}}$ is compared with the average saved time $\bar{t}_{\text{saved}}$ for each task (Figure 5.16 and Table 5.5). Here $\bar{t}_{\text{saved}}$ is the sum of $\Delta t$ for all questions of a
5.4 Experimental Evaluation

Figure 5.15: Boxplot for the task completion time for Task IV.

Table 5.4: Task IV: Time in [s] to activate the window groups of Task I-III.

<table>
<thead>
<tr>
<th></th>
<th>t_{S&amp;T}</th>
<th>σ</th>
<th>Δt</th>
<th>σ</th>
<th>p_{wrs}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group I</td>
<td>6</td>
<td>2.5</td>
<td>12</td>
<td>5.5</td>
<td>&lt; 0.01**</td>
</tr>
<tr>
<td>Group II</td>
<td>8</td>
<td>9.0</td>
<td>9</td>
<td>12</td>
<td>&lt; 0.01**</td>
</tr>
<tr>
<td>Group III</td>
<td>4</td>
<td>1.3</td>
<td>7</td>
<td>4.7</td>
<td>&lt; 0.01**</td>
</tr>
</tbody>
</table>

The experimental tasks were artificial hence one could argue that such a comparison is not meaningful. However, all experimental tasks were quite short compared to real world tasks, so these numbers serve to indicate that \( t_{saved} \) is reasonably likely to outweigh \( t_{setup} \) when working with Stack & Tile groups a bit longer. We observed that participants still had problems in setting up Stack & Tile groups in an optimal manner after the training tasks. It is reasonable to assume that once users get more practice with Stack & Tile the setup times will drop. A non-significant indication (\( p < 0.25 \)) for that is that users who had used Stack & Tile before had a slightly shorter setup time (on average 2-3 seconds for each of the three tasks). Note that the setup time does not include the time users needed to decide what a Stack & Tile group should look like because the layout was given in the experiment.

Figure 5.16: Boxplot for the setup time for the three task groups.
<table>
<thead>
<tr>
<th></th>
<th>Task I</th>
<th>Task II</th>
<th>Task III</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_{\text{setup}}$ ($\sigma$)</td>
<td>29 (15.7)</td>
<td>14 (11.1)</td>
<td>22 (10.9)</td>
</tr>
<tr>
<td>$T_{\text{saved}}$</td>
<td>18</td>
<td>12</td>
<td>33</td>
</tr>
</tbody>
</table>

Table 5.5: Setup times in [s] for window groups of Task I-III.

**Questionnaire Results**

Figure 5.17 shows the results from the Likert-scale questions. All participants agreed that they often use tabbed user interfaces (Q4), which shows that the stacking feature is not a fundamentally new concept to them.

There was an overall agreement that they often have multiple windows open (Q3), and over 2/3 of them agreed that window management often becomes frustrating when working with overlapping windows (Q6). This indicates that there is a need for better window management.

Only 3 of the 30 participants agreed that the initial overhead to setup a Stack & Tile group would prevent them from using Stack & Tile (Q7), and 24 of them agreed that they often adjust position and size of a window before using it (Q5). Both support H3 (see also Section 5.4.2).

Lastly, there was a general agreement that Stack & Tile makes window management easier and more enjoyable (Q8). This is also supported by many comments. As an example, one participant said: “Much easier to use. Better grouping, while still having more freedom than MDI apps”. This supports H4.

The results for the question about the techniques participants use to manage windows are depicted in Figure 5.18. The results are consistent with the empirical findings in [102].
Participants said that they use direct window access only in 17% of the time. This is another indication that direct window activation is not perceived as optimal by them.

### 5.5 Web-based Survey

Stack & Tile has been available in Haiku for over two years now. An interesting question is if and how people are actually using Stack & Tile. To answer this question a web-based survey has been conducted in the Haiku community. The questions were mainly a mix of Likert-scale questions and open questions.

The survey began with some demographic and general questions:

**S1:** I often use computers in my daily life.

**S2:** I heard about S&T before.

**S3:** I think S&T can be useful.

**S4:** I don’t think there is any need for S&T.

**S5:** Have you ever tried S&T. (Negative answer ended the survey.)

**S6:** How often do you use S&T?

Afterwards the participants were asked to upload screenshots of how they use Stack & Tile most frequently and provide details, followed by questions about Stack & Tile:

**S7:** I think the stacking feature is more useful than the tiling feature.

**S8:** S&T helps with resizing window groups.

**S9:** Estimate the percentage of how often you use stacking and how often you use tiling.

**S10:** When you use S&T, how many S&T groups do you use on average at the same time?

**S11:** I exchange information between single windows (not in a S&T group).

**S12:** I exchange information within a single S&T group.

**S13:** I exchange information between multiple S&T groups.

**S14:** I exchange information between S&T groups and other single windows.

**S15:** Are there certain tasks where you use S&T? For example, programming or browsing.

**S16:** In what other situations are you using S&T?

Finally, participants had the opportunity to make suggestions and comments.
5.5.1 Result

During a period of two months there were 146 participants. The average age of all subjects was 32 (σ = 10). There was only one female participant. The majority of the participants was working in or had a degree in an IT related field.

There were two groups of participants, distinguished by the general questions at the beginning. This was the group that not had tried Stack & Tile before (left of Figure 5.19) and the group that had tried Stack & Tile (right of Figure 5.19). Generally both groups are heavy computer users. For the group that had not tried Stack & Tile before, almost half of them had heard about Stack & Tile. Most participants agreed that Stack & Tile is useful and disagreed that there is no need for Stack & Tile. This general opinion was much stronger for the group that had tried Stack & Tile before, indicating that Stack & Tile had made a positive impression.

![Figure 5.19: Answers for S1 - S4. Left: users who have not tried Stack & Tile. Right: users who have tried Stack & Tile.](image)

Figure 5.19: Answers for S1 - S4. Left: users who have not tried Stack & Tile. Right: users who have tried Stack & Tile.

Figure 5.20 shows the results from question S6. Here it can be seen that most people who tried Stack & Tile before are still using it.

![Figure 5.20: Answers for S6.](image)

Figure 5.20: Answers for S6.

Question S7 asked for preferences for the stacking or the tiling feature. There is a trend that the participants feel stacking is more useful than tiling (see Figure 5.21). S9 asked for the usage of the stacking and tiling features. The average estimated percentage of tiling is 45% (σ = 29%) and for stacking 55% (σ = 29%). Stacking is used slightly more than tiling with $p_{wrs} = 0.06$. This indicates that participants think that stacking is more useful (S7) and consequently use it more (S9). However, keeping in mind that
5.5 Web-based Survey

stacking is much more common in today’s applications (e.g., tabbed browsing), it is still interesting that the tiling feature is reportedly used as much as it is.

S8 targeted the feature that windows in a group stay aligned when resizing one window in the group. Most of the participants judged this to be helpful (see Figure 5.21).

Figure 5.22 shows the average number of groups when Stack & Tile is used. Most participants use between one and four groups while the majority uses more than one Stack & Tile group. This is quite interesting because it means they not only use it sporadically for a single task but seem to use it for different purposes in parallel.

The advantages of Stack & Tile when exchanging data between windows or Stack & Tile groups have already been analyzed in the controlled experiment. S11 - S14 targeted the question in how far participants encounter such use-cases in practice (Figure 5.23). The results are quite similar for S11, S12 and S14. Most people are exchanging data between groups/windows, which shows that the results from the controlled experiment...
are relevant for real users. The least frequent data exchange was between Stack & Tile groups (S13), which indicates that Stack & Tile groups are used to group windows of different tasks.

5.5.2 Use-Cases for Stack & Tile

The participants have been asked to upload screenshots of how they use Stack & Tile most frequently and 23 screenshots have been submitted in response. Furthermore, one user sent a link to a video with a demonstration of how he uses Stack & Tile. S15 and S16, targeting the questions for what tasks and in what situations Stack & Tile is used, received 47 and 23 responses, respectively.

The results indicate that there are three main use-cases where the participants used Stack & Tile: programming (28 participants), web browsing (17 participants) and file management (11 participants). For example, C++ source and header files were tiled beside each other or all source files were stacked to one stack and header files to another stack. Many users reported that they use Stack & Tile to group windows by task, e.g., grouping a web browser and a chat window together. Other examples were grouping a music directory and a media player, a picture directory and an image viewer, or creating an ad-hoc development environment by grouping source files, source directory and terminal together. One user tiled a web browser and a text editor together to copy information from the browser to the editor. An unexpected use-case was the creation of a Stack & Tile group to move multiple windows across different virtual desktops more easily.

An observation made from the screenshots is that the Stack & Tile groups had mostly moderate complexity. There were only a few use-cases where more than two windows were tiled together. However, the stacked window groups usually had multiple windows in them.

There were 34 responses for the open-ended comment/suggestion field. Here participants had some ideas for improvements and better integration into the desktop. For example, there was the request to show Stack & Tile groups in the taskbar, and that it should be possible to store and restore Stack & Tile groups, especially on reboot. Applications that are using a tabbed interface should use the stacking feature instead. These are points planned to be targeted in future work. Another participant stated that she had already started to replace the tabbed interface of a browser with the Stack & Tile stacking feature. 11 participants explicitly said that they like Stack & Tile, while one participant does not think that Stack & Tile can alleviate manual window management.
5.6 Conclusion

In this chapter Stack & Tile, a window manager that integrates stacking and tiling seamlessly with traditional window management operations has been presented. Stack & Tile uses the constraint-based Auckland Layout Model to manage group layouts. The edit operations in Stack & Tile manipulate the constraint-based group layout in a sound way, i.e., it stays solvable and non-overlapping. Furthermore, the advantages of integrating stacking and tiling features in the traditional desktop metaphor were investigated. Stacking and tiling can help to manage windows more effectively, for example, by grouping windows by task.

In a controlled experiment, it has been found that a stacking and tiling features can significantly improve completion times for tasks involving several windows (of the same application as well as of different applications). Furthermore, switching between different tasks is found to be much faster when windows are grouped by task. Setting up a Stack & Tile group is an initial overhead which may prevent users from using these features. However, the potential time savings as well as questionnaire answers indicate that the advantages outweigh the overheads.

In a web survey it is investigated how often and how Stack & Tile is used in practice. There was a wide agreement that Stack & Tile can be useful, especially by participants who had used Stack & Tile. The stacking feature was perceived as being slightly more useful and also estimated to be used more than the tiling feature. It has been found that people are using the stacking and tiling features for a multitude of different use-cases. In the field for general comments many people wrote that they like Stack & Tile and suggested further ideas to integrate it more into the desktop. These ideas include future works such as grouping of windows by their Stack & Tile group in the taskbar and Stack & Tile group persistence.

These results show a use-case were the constraint-based layout model is successfully used for user interface customization. Simple and sound operations ensure that the Stack & Tile system stays in a usable state. The complexity of constraint-based layout is hidden from the end user. An interesting further question is how these achievements can be augmented for more complex edit operations and generalized for different applications.
This chapter we target layout editing of graphical user interfaces (GUIs). This enables the user to customize the spacial position of widgets, remove or insert widgets. The here presented layout customization framework for GUI end user customization provides a set of powerful edit operations. These are moving, swapping, resizing, inserting and removing of widgets. All edit operations maintain alignment and establish appropriate constraints automatically. This small set of operations is enough to edit complex constraint based layouts.

When we came to a point to evaluate these edit operations we faced the problem that no other comparable customization framework is available. To overcome this problem the edit operations have been integrated into a layout GUI builder, the Auckland Layout Editor (ALE). Conceptually a GUI builder and the customization framework are quite similar. Both systems require to edit existing layouts and both systems must support adding and removing widgets. ALE makes it possible to create and edit constraint-based layouts rapidly with simple mouse operations. Technically, ALE uses the same library that is used for a layout customization framework. The usability of ALE can then be transferred to the usability of its edit operations. Hence, the following work on ALE can directly be transferred to the layout customization framework.

Modern GUI builders offer good support for simple layout models, such as grid and
gridbag. While constraint-based layouts are more powerful, their creation may be more complex and poses challenges. General constraints are difficult to visualize and even harder to manipulate directly. Specifying individual constraints can be tedious and error-prone, as they are situated at a low level of abstraction. Thus widgets may overlap each other (Figure 6.2) or become over-constrained, i.e., there may be no solution for a layout specification. There is currently no method to automatically generate a reasonable set of sound constraints for a given layout. Thus, support for constraint-based layouts in GUI builders is less developed than for other layout models. This makes it harder for GUI designers to leverage the advantages of constraints. Finally, GUI design often involves iterations on a given layout. This makes it important to support easy modification of previously created layouts.

Similar to other GUI builders, ALE uses a component palette and an editing canvas (Figure 6.1). Widgets are dragged from the palette into the editing canvas, where the designer can change the layout using a rich set of edit operations. ALE allows making a designed layout specification persistent. A saved layout specification can then be loaded into an application and widgets of the loaded specification can be accessed via a simple programming API. The sources for ALE and the customization framework are available in the git repository https://github.com/czeidler/haiku.git in the ALE branch. The evaluation presented in this chapter has already been published at the INTERACT’13 and the UIST’13 conferences [111, 112].

In the next Section 6.1 a list of requirements for layout editing is compiled. Related work is presented in Section 6.2. Section 6.3 describes the layout edit operations in detail. How non-overlapping layout specifications can be achieved is described in Section 6.4. The edit operations are then evaluated in Section 6.5.

### 6.1 Requirements for Layout Editing

In this section we specify our design goals for ALE. These are derived from related work, existing usability guidelines, and our own experience with layout creation. We also define requirements for all layout specifications that can be created with a GUI builder. We differentiate between concrete layouts, as they are rendered on a screen, and layout specifications, which can be rendered at many different sizes and hence lead to many concrete layouts. Our design goals can be subdivided into usability, completeness, and soundness requirements. Common usability guidelines [58, 83] naturally apply to GUI builders. In particular, a builder should be intuitive, easy to use, and edit operations should lead to predictable results. Common GUI design guidelines, such as [40], emphasize that it is not desirable to place widgets completely “freely”, i.e., at an arbitrary position in a layout. Aligned layouts are more compact and easier to understand [47]. Consequently, widgets
6.1 Requirements for Layout Editing

Figure 6.1: Screenshot of the ALE GUI builder. New widgets can be inserted into the left editing canvas, via drag and drop from the component palette on the right.

should be easy to align and the system should automatically maintain such alignments.

Completeness guarantees that every correct layout specification that is both possible in the underlying layout model and appropriate in practice can actually be created.

Soundness ensures that every layout specification that can be created is correct and adequate, i.e., no erroneous layouts can be created. Here we use three soundness requirements.

Unique Solution A layout specification must have exactly one solution. When computing a layout, the underlying specification needs to be solved, i.e., a concrete visual layout needs to be generated. If the specification contains conflicts, then there is no solution. If there is more than one solution, then the layout may behave non-deterministically. For example, if the position of a widget is underspecified it may “randomly” change its position during resizing, which is undesirable. However, the solution of a linear solver is usually “stable” and a widget is placed consistently at a certain position.

Non-Overlap A layout specification must be non-overlapping, i.e., in all layouts derived from the specification, widgets may not intersect each other or the boundaries. This property ensures that all widgets are completely visible and accessible at all GUI sizes. A widget that is overlapped by another may be inaccessible and thus useless (Figure 6.2).

Well-Defined Layout Sizes Each layout specification must have a single minimum, preferred, and maximum layout size. It is theoretically possible for a layout specification
to have multiple valid extreme sizes (Figure 6.3). Yet, such situations are not desirable. Firstly, window managers support only a single minimum and maximum window size. Secondly, in order to keep the layout non-overlapping, “or” constraints must be used, which are difficult to solve. Lastly, a layout with multiple extreme sizes potentially has a very different appearance at different sizes, which may confuse users. Note that we still permit discrete layout specification changes at runtime. For example, when the screen is rotated one can switch between different layout specifications with different extreme sizes.

### 6.2 Related Work

Myers [85] summarized various non-constraint-based techniques for creating GUIs, including graphical tools for placing interface objects on screens. IBuild [105] enabled the creation of complex GUIs in a WYSIWYG manner. It already supported nested layouts, spring-like layout elements, and interactive testing of resizing. Druid [96] predicted intended alignment and spacing of a widget during editing, facilitating placement. FormsVBT [4] supported simultaneous editing of a textual and a graphical representation of a layout.

Some GUI builders permit manual constraint editing. OPUS used tabstops for speci-
ifying constraints [53]. Lapidary [110] provided rich constraint editing functions. In the Gilt system [46], widgets could be aligned relative to user-specified tabstops or other widgets. In the Intui GUI builder [95], constraints can be toggled between struts and springs, between and inside widgets. Yet, only one constraint per widget side is supported, which can make it impossible to create non-overlapping layout specifications. Peridot [86] analyzes objects drawn by the user and guesses constraints to be added. The user then has to confirm the proposed constraints. Rockit [66] automatically proposes constraints based on the “gravity field” of other objects. The user can select the desired constraints from a set. This is similar to previous work where constraints are inferred by snapping graphical objects relative to other objects [54]. In ALE widgets can be snapped to others in order to set up the corresponding constraints. However, ALE adds more powerful edit operations, such as inserting a widget between others and swapping widgets. Bramble [41] connects objects using a set of interactors, which establish non-linear constraints. While this can be used to prevent overlap, the user has to add interactors manually. ALE adds such constraints automatically.

Today, there are many open-source GUI builders, such as WindowBuilder Pro [51], Matisse/Swing [2] and Qt Designer [3]. There are also commercial GUI builders, such as MS Expression Blend [4] and Visual Studio [5]. Most of them support aligning widgets via snapping, but do not maintain alignment when objects are moved or resized. Also, layouts must be resized manually in order to evaluate their resize behavior. In contrast, ALE’s edit operations keep widgets aligned by default. ALE also provides an automatic preview of resize behavior.

Apple added support for constraints to the Xcode Interface Builder [6] in 2012. It aligns objects via snapping, but also permits free placement of widgets. Constraints between widgets can be added manually but are removed upon conflict. All the abovementioned constraint-based approaches require that the user knows how to specify constraints manually. In contrast, ALE enables the creation of many interesting layouts without exposing a single constraint to the user. Moreover, the edit operations automatically maintain all constraints for snapped items and for non-overlap. ALE disables conflicting manual constraints to support error diagnosis. Finally, a recent comparison between Xcode and ALE suggests that our new approach may result in better usability [111].

Adaptive document layout [55] in the print and web domains is somewhat similar to GUI layout. However, the flow of a document constrains placement differently than a GUI. Documents typically arrange text and images in a sequential manner, and support more
flexibility in the layout using algorithms for figure placement, line-break, and pagination. Grid-based [61] as well as constraint-based [17,56] methods have been used for document layout.

Firefox’s responsive design view[7] can simulate a GUI at different screen sizes. Also Android Studio[8] shows layouts at different device sizes. However, this does not take an optimally enlarged layout size into account. FormsVBT[4] supports a graphical and a textual edit view as well as a result view. However, the result and graphical edit views are displayed at a fixed size and give no indication of how a layout would look at a different size. In contrast, ALE shows a live preview of the layout at the minimum and an optimally enlarged layout size. This reduces the potential for user errors, as the resize behavior is visualized immediately and directly. ALE’s layout previews are targeted particularly at situations where enough constraints exist, but do not lead to the desired resize behavior. In these situations, displaying all constraints clutters the design view, which does little to help the designer quickly identify the problem.

6.3 ALE Layout Edit Operations

Ideally, a GUI builder should make the creation of a constraint-based layout easy and intuitive. If a GUI builder exposes each constraint, margin, tabstop, strut or spring directly to the user, this significantly increases the complexity of layout editing. In other words, the user has to create, correctly configure, and maintain a substantially larger number of entities to achieve a given result. To avoid this, ALE’s layout edit operations were designed to automate the construction and maintenance of the layout specification as much as possible, while still ensuring that the operations are sufficient to create and edit complex GUIs.

As in most GUI builders, a GUI can be created and edited by drag and drop operations. The edit operations provided are moving, swapping, resizing, inserting, and removing of a layout area and the widget contained therein. All operations try to connect layout areas to existing tabstops through snapping, and create as few new tabstops as possible. Reusing tabstops leads to good alignment and simpler layout specifications that are easier to understand. Only the intrinsic size constraints are updated when a layout area is connected to new tabstops, and new constraints are added only in a few cases. Furthermore, all operations are designed to leave an initially sound layout in a sound state. Thus no overlap between widgets can be generated. In the following sections and for brevity, we describe only the horizontal case for each operation. Vertical cases are handled analogously.

Edit operations are started by dragging a widget or a border of a selected widget. Then, ALE checks at each mouse position whether an operation can be applied and gives corresponding visual feedback (Figure 6.4). This is done by applying the operation tentatively in the background and checking the result. If an operation is applicable, the user can commit it by “dropping” the widget, and the result becomes visible. In general, edit operations can affect multiple widgets and a layout may differ in several places after an operation. To help the user understand the changes, a short animation visualizes how affected widgets are altered. All edit operations can be reverted using undo. Inserting, removing, moving and swapping of widgets are done via dragging the widget; dragging a border is used for resizing.

6.3.1 Inserting a Widget

New widgets can be inserted into the layout by drag and drop from the widget palette into the edit canvas. When dragging a widget, it is visualized as a dotted rectangle, the *dragged widget outline*, which has the preferred size of the dragged widget. We limit the size of the dragged widget outline to a reasonable one to avoid problems with very large widgets. Two cases need to be considered when inserting a new widget into the layout. A widget can be inserted into an empty area (Figure 6.5) or it can be placed between an existing tabstop and a widget adjacent to that tabstop (Figure 6.6).

**Inserting into an Empty Area**

A widget can be inserted into an empty area if the mouse position is in that empty area. Dropping the dragged widget into an empty area must constrain that widget to at least two tabstops, one in each direction, because of the requirement for a unique solution. If an edge of the widget is close to a tabstop, the widget is snapped to that tabstop. If the widget can only be snapped to a single tabstop in one direction, a new tabstop is created at the opposite widget edge.

![Figure 6.4](image)

**Figure 6.4**: ALE automatically selects the largest empty area, depending on user input. The area where the widget will be inserted when dropped is highlighted in green. Here, the widget is entirely within the lower empty area (left) or within the right one (right). In the lower-right quadrant, the larger (right) area would be used.
If the widget cannot be snapped to any existing tabstop in a given direction, we solve the problem in a novel way. In this case, the largest rectangular empty area that completely contains the dragged widget outline is identified (Figure 6.4). If the outline overlaps another non-empty layout area, i.e., no rectangular empty area completely contains the outline, the largest empty rectangular area that contains the mouse position is used. For the direction in question, the widget is then connected to the outer tabstops of this largest rectangular empty area. This is also an easy way to fill a whole rectangular empty area with a resizable widget.

These rules enable the user to select the desired area in an intuitive way. The combination of being able to use the full extent of an empty area or simply to snap a widget to one corner or border makes this operation quite versatile (Figure 6.5), see also the Tk pack model.

![Figure 6.5: A widget can be inserted at various positions within an empty rectangular area. Left: Far from the tabstops, the dragged widget fills the whole empty area. Middle: When close to only the left tabstop, the widget snaps to that tabstop and resizes vertically. The right edge creates a new tabstop. Right: When close to the left and bottom tabstops, the widget is constrained to the corner.](tcl.tk/man/tcl8.6/TkCmd/pack.htm)

**Inserting between Tabstop and Area**

A widget can be inserted at any existing tabstop. To trigger this operation, the mouse position must be over an existing widget and close to a tabstop, i.e., an edge, of that widget. One edge of the new widget then aligns with that existing tabstop, while a new tabstop is created for the opposite edge. The existing widget is connected to the new tabstop (Figure 6.6). Initially, the size of the new widget is *temporarily* set to zero, as the final size is determined later by the solver.

![Figure 6.6: When inserting a new widget $A_2$ between area $A_1$ and tabstop $x_1$, a new tabstop $x_2$ is added.](tcl.tk/man/tcl8.6/TkCmd/pack.htm)
Once the new widget has been inserted at an existing tabstop in one direction (say horizontal), it must also be connected in the other (vertical) direction. If its preferred height is smaller than the height of the existing widget, the new widget is either connected to the top or the bottom tabstop of the existing widget, whichever is closer. In this case, a new tabstop is added at the opposite vertical side of the new widget. Figure 6.7 shows such an example where a small button is inserted beside a larger text view. If the preferred height of the new widget is larger or close to the height of the existing widget, the new widget is connected to both the top and the bottom tabstop of the existing widget. Because the maximum size constraints of the existing widget are soft, this never generates a conflict.

Figure 6.7: Inserting a small button at an existing tabstop between a list and a text view. The existing tabstop and widget are highlighted in green. As the button is dropped close to the top of the text view, it is connected to its top tabstop.

6.3.2 Removing Widgets

A widget is removed from the layout by dragging it outside the edit canvas. Any created gaps are filled, see below.

6.3.3 Moving Widgets

When moving a widget, a valid position for the insertion is determined and then the widget is moved from its original place. Here the same logic is used as for insertion. However, when looking for an insertion position the area occupied by the moving widget is ignored. This makes it impossible to snap a widget to itself, for example.

6.3.4 Swapping Two Widgets

Dropping one widget onto another swaps the positions of the two widgets. Here it is sufficient to connect the moved widget to the tabstops of the other one and vice versa.
6.3.5 Resizing a Widget

When a widget is connected to wrong/undesired tabstops, the user can adjust this with a resize operation. Resizing is done by dragging one of the borders or corners of a widget. During resizing, all relevant tabstops are visualized as light blue lines to aid alignment. There are two cases to consider. Firstly, a widget may be resized to an existing tabstop, via snapping (Figure 6.8). Here, the system ensures that the resized widget does not overlap with any other one; otherwise, the resize operation is not permitted. Secondly, a widget can be resized to match its preferred size. This occurs when the dragged border is released in an empty area, i.e., without snapping. In this case, a new tabstop is created for the dragged border (two for a corner). This can also be interpreted as detaching a widget from a tabstop. Preferred sizes can be adjusted via the properties window (Figure 6.1 bottom right).

![Figure 6.8: Resizing the top of the list widget to the bottom of the string.](image)

6.3.6 Filling Gaps

A move, resize, or remove operation can detach adjacent widgets as one or more widgets may lose their connection to a tabstop. This is usually visible as a “gap”. Such “gaps” and the associated “floating” widgets violate the unique solution requirement (Figure 6.9). ALE avoids this by checking for unconnected widgets after remove, move and resize operations. If the layout contains parts that are unconnected, all “floating” widget groups are moved one after another into the direction where the removed widget has been located. For example, if a group was connected to the right side of the removed or resized widget, the group is moved to the left. When the foremost widget of the floating group hits the border of another widget, it is connected with the corresponding tabstop of the other widget (Figure 6.9). During this process the moved group may become connected to another floating group. Groups are moved until they are connected directly or indirectly to at least one horizontal and one vertical layout border.
6.4 Non-Overlapping Layouts

We start with a formal definition of non-overlap to explain how ALE guarantees non-overlapping layouts. A widget is completely left of another if the right side of the first widget is left of the left side of the other widget. Two widgets are called horizontally non-overlapping if either one is completely left of the other. Vertically non-overlapping is defined analogously. Two widgets are non-overlapping if they are horizontally or vertically non-overlapping. A layout is non-overlapping if all possible pairs of widgets are non-overlapping.

While a given layout may be non-overlapping, this does not imply that the underlying specification produces only non-overlapping layouts. This is a central problem in GUI layout. For example, the layout on the left of Figure 6.2 is non-overlapping. Yet, as the size is reduced, the check box starts to overlap the button due to a missing constraint. Any specification that produces only non-overlapping layouts is a non-overlapping specification.

The main idea of the non-overlap algorithm presented here is that for a given non-overlapping layout, the underlying specification can be made non-overlapping by adding additional non-overlap constraints. These are simple hard linear constraints that ensure a non-negative distance between two widgets. In the following, we show that all layouts created using ALE’s operations are non-overlapping. Moreover, we explain where non-overlap constraints are added.

6.4.1 Non-Overlap of Created Layouts

All edit operations transform a non-overlapping layout so that it stays non-overlapping. Starting from the naturally non-overlapping empty layout we prove via structural induction that all layouts evolving from there are non-overlapping. Inserting or moving a widget into an empty area keeps the layout non-overlapping by definition. When inserting or moving a widget horizontally between a tabstop and another widget, it does not overlap the involved widget as a new “column” is created between the tabstop and the
existing widget, and the new widget is temporarily assigned zero size. Furthermore, the
new widget does not overlap any existing widgets above or below in a vertical direction
because per definition of the operation it has at most the height of the existing widget,
and this widget did not overlap any other widgets before. Swapping two widgets does
not change the topology of the tabstops and thus works as well. When increasing the
size of a widget, the system always verifies that the resized widget does not intersect with
another; otherwise the operation fails. Decreasing a widget’s size also keeps the layout
non-overlapping because the resulting widget is always smaller than the old one.

The last case that has to be considered is gap filling. When filling a gap, floating
groups are moved one after another into the direction of the removed widget. Because
a floating group is only moved until it hits another widget, this operation also keeps the
layout non-overlapping. Given that each edit operation leaves the layout non-overlapping,
the system can never get into a state where the layout has overlaps. Thus ALE is suitable
for creating and editing a concrete layout under the non-overlap requirement. In the next
section we discuss how we expand this to keep all layout specifications non-overlapping.

6.4.2 Non-Overlapping Layout Specifications

To maintain the non-overlap requirement of a layout specification during all resize opera-
tions, one needs to ensure that all pairs of widgets are non-overlapping. Yet, checking all
pairs would be inefficient, so we propose a more sophisticated approach. We first recall
that in full layouts, i.e., layouts without empty space, all widgets share tabstops with all
neighbors. Thus the layout is non-overlapping.

Tiling of Empty Areas

It remains to argue what to do if a layout has one or more empty area(s). If two visually
adjacent widgets do not share a tabstop, the gap between the two widgets is treated as
an empty area with no extent. Otherwise, the empty area(s) are orthogonal polygons.
Our solution is to tile these orthogonal polygons by introducing new, virtual, empty
rectangular widgets (Figure 6.10). Each such virtual rectangular widget is connected to
existing tabstops. These virtual widgets only need a minimum size of zero to guarantee
non-overlapping layout specifications.

We now discuss the algorithm for tiling all empty areas. All empty areas are orthogonal
polygons. In general, such polygons can be tiled in \(O(n^{3/2} \log n)\) time \[98\] (\(n\) being
the number of polygon edges), producing \(O(n)\) tiles. As discussed later, the algorithm
presented here permits a choice of the shape of the inserted tiles, which affects the resizing
behavior of the layout. In a first step, all U-shaped segments in the orthogonal polygons
are identified. A U-shape has a horizontal edge and two upwards-pointing vertical legs,
6.4 Non-Overlapping Layouts

The empty area is an orthogonal polygon (black outline) that is filled with tiles (red). Each tile has a minimum size of zero, which guarantees a non-overlapping layout specification.

![Figure 6.10: The empty area is an orthogonal polygon (black outline) that is filled with tiles (red). Each tile has a minimum size of zero, which guarantees a non-overlapping layout specification.](image)

The two neighboring edges. In general, the length of the legs can differ (Figure 6.11). In a second step, the identified U-shapes are tiled. Each U-shape is tiled up to its shorter leg (Figure 6.11 left). As depicted on the right of Figure 6.11 it is possible that a widget may extend into a U-shape. In this case the U-shape is only filled up to this widget. When inserting a new tile, new U-shapes may be created and these U-shapes must be added to the list of U-shapes.

After tiling all U-shapes, all orthogonal polygons are completely tiled and the layout becomes non-overlapping. Moreover, the requirement that a layout has well-defined layout sizes is satisfied, as the inserted tiles conserve the topology of the widgets relative to each other. When tiling U-shapes, the inserted tiles become “row-like” (Figures 6.10 and 6.11). By choosing C-shapes instead, tiles become “column-like”. This may affect the resize behavior, i.e., if widgets move horizontally or vertically relative to each other. By default, ALE produces “row” tiles, but the user can toggle this.

If there are \( n \) widgets in the layout, then the empty areas cannot have more than \( 4n + 4 \) corners. This gives us a small linear bound on the number of necessary constraints, as we need only two minimum size constraints per tile. In general, the tiling will have to be recomputed if a layout is modified.

After adding all non-overlap constraints identified by this algorithm the layout becomes non-overlapping. Moreover, the requirement that a layout can only have a single minimum and preferred size is satisfied, as the inserted tiles keep the orientation of the widgets towards each other constant.
6.4.3 When to Add Non-Overlap Constraints

For all edit operations discussed above and to test an operation for applicability, the operation is temporarily applied, the resulting specification solved, and then it is verified for soundness. As a first step, all existing non-overlap constraints are removed from the current layout so as not to interfere with the change. After the layout operation is temporarily applied, non-overlap constraints are inserted for the new specification. If the resulting specification is solvable, the operation can be applied. Otherwise the previous layout and non-overlap constraints are restored. As a last step of a successful operation, the solver recalculates the minimum, maximum and preferred layout sizes. If necessary, the parent window size is updated to fit the modified layout.

6.5 Evaluation

ALE automatically aligns widgets to one another when placing a widget. It also moves other widgets aside if necessary. Additionally, it automatically keeps layouts overlap free. Moreover, swapping of widgets does not require manual resizing or moving of the adjacent widgets. Consequently, two hypotheses relative to other available GUI builders are formulated:

H1 ALE makes it easier to create layouts from scratch.

H2 ALE makes it easier to edit existing layouts.

The aim of the first experiment is to investigate how ALE performs in comparison to one of the main GUI builders for the popular gridbag layout model. Here the focus is only on layout creation. In a second experiment ALE is compared to a modern GUI builder that uses a constraint-based layout model. Here also layout editing is evaluated.

6.5.1 Experiment 1: Comparison with a Gridbag GUI Builder

The first experiment targets H1. For this comparison the GUI builder in MS Visual Studio 2010 (VS) is chosen as a reference, which was a very popular tool at the time of the study. Thus it can be deemed as a representative for the state of the art. Furthermore, VS supports the very common grid-bag layout model and it is an interesting question if a constraint-based GUI builder can compete against it directly. 16 participants, mostly software engineering students with experience in GUI development, were asked to perform four GUI creation tasks, each either with ALE or with VS. In each task, they were asked to rebuild a realistic GUI layout from a sample screenshot. Figures 6.12, 6.13 and 6.14 show the four tasks. The task completion time is measured as an indicator of efficiency and a post-questionnaire is used to determine participants’ preferences. With the VS GUI
6.5 Evaluation  

builder it is not easy to modify the row- and column-span in a gridbag layout, since this cannot be done visually. It can only be achieved by opening a properties dialog. Thus, participants were instructed to nest multiple gridbag layouts, permitting users to recreate the target layouts more easily.

For both ALE and VS a training task was given before the respective main tasks to ensure a reasonable amount of training with both tools. To counteract potential learning effects, half the participants were allocated to a group which first performed the training and tasks I and II with ALE, and then the training and tasks III and IV with VS. The second half used the tools in the opposite order.

If there were errors in the layout after the participants had finished a task, e.g., a widget was placed erroneously, the experimenter indicated the errors to them. Afterwards, timing continued and the participants had to fix these errors. In this way all participants were able to succeed in all tasks.

Results and Discussion

![Figure 6.12: Experiment 1, Task I: VS on the left and ALE on the right side.](image)

![Figure 6.13: Experiment 1: Task II for VS on the left and Task III for ALE on the right side.](image)

The measurement dataset was not normally distributed. The medians of ALE and VS were 74 and 188 seconds, respectively. A Wilcoxon signed-rank test identified a
significant effect of the GUI builder \((Z = -5.31, p < 0.01)\), which supports \(H1\). A pair-wise Wilcoxon tests show that ALE was significantly faster than VS on every task, with \(p < 0.01\) or better. Figure 6.15 shows the individual times broken down by task. According to the post-questionnaire, 11 of the 16 participants preferred ALE over VS.

One potential threat to validity is the fact that in VS participants did not use a single gridbag layout, but a nested gridbag layout with a column- and row-span of one. According to observations during the experiment, many participants had difficulties when nesting multiple layouts to create the desired outcome with VS, even though this was simpler in the VS GUI builder compared to creating a single gridbag layout. A possible explanation is that a gridbag layout specification has to be understood more thoroughly upfront and cannot be developed easily on the fly during the design process, as with a constraint-based layout approach.

Another potential issue is that for this evaluation a slightly older ALE version was used [116]. Although this version was less polished and had some small usability problems,
all operations discussed here were available. Thus, it can be expected that ALE would perform even better in its current version.

6.5.2 Experiment 2: Comparison with a Constraint-Based GUI Builder

In a second experiment both H1 and H2 are tested. Apple’s Xcode currently offers the only easily available GUI builder for constraint-based layouts. It builds on the AutoLayout model, which supports simple linear hard and soft constraints. Also, Xcode permits free placement and resizing of items on the canvas. Consequently, in this experiment ALE is compared to Xcode.

The evaluation involved two main tasks (Task V and VI), preceded by a training task that was similar to the main tasks. Each participant performed all tasks once with ALE and once with Xcode. To eliminate order effects the study was counterbalanced; half of the participants started with ALE, the other half with Xcode. Each task was divided into a layout creation subtask and three editing subtasks.

The first editing subtask (a) required swapping two widgets, while the second subtask (b) required moving a widget to a position between two other widgets (see Figure 6.6). For these two subtasks it is expected that ALE will perform better because users need only a single operation, while in Xcode multiple operations are necessary. With the free placement approach in Xcode, moving a widget between two other widgets requires the user to first move at least one of the other widgets aside to make room for said widget. Furthermore, it is necessary to manually fill the empty space that the moved widget has left. The last editing subtask (c) was more complex and required a combination of multiple edit operations. Figure 6.16 shows the layout for the creation subtask for Task V, and Figure 6.17 shows the editing subtask (c) for Task VI. In Xcode, participants were asked to align widgets as good as possible via the snapping functionality provided by the builder. As for the previous experiment, after each task the experimenter pointed out layout errors to the participants and participants had to fix them.

After finishing the tasks using Xcode and ALE, the participants were given a Likert-scale questionnaire to gather general information and to analyze their preferences. Furthermore, they were asked for comments in an open-ended question.

Results and Discussion

For this study, 14 participants were recruited, mostly graduate Computer Science students. All of them stated that they had understood the tasks. Most participants had experience with user interface design and were familiar with creating GUIs using a GUI builder. Four participants had used Xcode, while only one participant had used ALE
Overall, the measured task completion time was not normally distributed. The results for the creation tasks are shown in Figure 6.19.

The medians of all creation times for ALE and Xcode were 66 and 80 seconds, and for all editing times 9 and 38 seconds respectively. Figure 6.20 depicts the results for the editing subtask for Task V. The results for the editing subtasks of Task VI are shown in Figure 6.21. A Wilcoxon signed-rank test identifies a significant effect of the GUI builder for creation ($Z = -2.05$, $p < 0.05$), which supports $H_1$. There is also a significant effect for editing tasks ($Z = -9.19$, $p < 0.01$), which supports $H_2$. For layout creation and layout editing ALE was clearly faster for Task V and Task VI. Pair-wise Wilcoxon tests show that ALE was significantly faster than Xcode on every creation and editing subtask, with $p < 0.01$ or better. The swapping (a) and the moving (b) subtasks were much faster using ALE.
6.5 Evaluation

I often use computer in my everyday life. (Q1)
I understood the task and was able to perform it. (Q2)
I have experience with designing user interfaces. (Q3)
I have experience in creating GUIs using a graphical GUI builder. (Q4)
I have used Xcode before. (Q5)
I have used ALE before. (Q6)

Figure 6.18: Results for the general Likert-scale questions of Experiment 2.

The results from the post-experiment questionnaire show a consistently positive response (Figure 6.22). Most participants preferred ALE and found it easier for creating and editing layouts. Furthermore, participants enjoyed ALE more than the Xcode builder.

Several participants commented in the open-ended question that they liked the swap operation and that layout editing was easier. Other comments pointed out that “one first had to get used to the different concepts of ALE, e.g., that [a widget] cannot be placed freely.” Another stated: “I would imagine that ALE may perhaps be very efficient when acquainted with.” This is consistent with the observations where participants needed more time to get used to ALE. However, after the training phase most participants were able to perform all tasks without problems.

One noteworthy observation is that with the free placement approach of Xcode it was more difficult to align items precisely. Participants made many erroneous alignments or seemed to be unaware of misalignments in the layout. Another observation is that when using Xcode, participants first aligned a newly inserted or modified widget precisely with some other widgets, and realized only later that they had to align said widget again to
achieve the target layout. ALE avoids this problem by automatically keeping widgets aligned, which is one contributor to the much shorter completion times. A threat to validity is that in Xcode, resizing a text or list widget was sometimes difficult, as users had difficulties clicking resize handles. However, users also experienced similar minor usability problems in ALE.

Another threat is that layouts in both experiments were relatively small. Since the evaluation already took about an hour the layouts are kept small. However, we believe that the layouts have a reasonable size because for larger layouts one would naturally start to use nested layouts.

### 6.6 Conclusion

In this chapter, the GUI builder ALE has been presented. The same layout editing library that has been developed for ALE can also be used for layout customization at application
runtime. ALE makes it possible to create and edit constraint-based layouts quickly with simple mouse operations. ALE defines a set of layout edit operations that hide much of the complexity of constraint-based layouts from the designer. These edit operations are, moving, swapping, resizing, inserting and removing of widgets and all operations maintain alignment and establish appropriate constraints automatically. ALE’s edit operations make it easy to place and connect widgets in the layout and a large set of layouts can be created. Furthermore, we discussed how ALE’s edit operations can be combined with general constraint editing, and how conflicts are resolved. This keeps common layout editing tasks easy, while making arbitrary constraint-based layouts possible.

ALE’s new edit operations automatically keep the layout sound and solvable. They also keep widgets aligned relative to each other, which leads to well-structured layouts. By automatically adding non-overlap constraints, layouts created with ALE are guaranteed to be non-overlapping for all possible layout sizes.

In two comparative evaluations, we found that ALE permitted participants to construct several realistic layouts significantly faster than with current commercial solutions, both for a gridbag and a constraint-based layout model. Furthermore, editing existing constraint-based layouts is also significantly faster with ALE. Participants also enjoyed using ALE, and once familiar with the new edit operations, found it easier to use.

This work demonstrates that it is feasible to utilize the power of constraint-based layouts in graphical GUI builders and thus for layout customization. The encouraging results from the evaluations illustrate that operations that automatically keep widgets aligned can result in a substantial boost in productivity. Overall this can be seen as an indication that there is ample potential for improvements in today’s GUI builders.
Now all pieces are in place to describe these powerful and sound, constraint-based edit operations in a more formal way.
Algebra for UI Customization

Working on Stack & Tile as well as on layout editing (ALE) revealed similar problems when editing layouts. Both systems use rectangular areas that are constrained adjacent to each other, and need to keep layouts in a sound state. That is, all these rectangular areas are not permitted to overlap. Furthermore, both systems use a constraint-based model and all edit operations need to leave the constraint system in a valid and solvable state.

In this chapter a new algebraic approach is presented, which is able to describe layouts of rectangular areas that are connected relative to each other. This approach can be used to derive constraint-based layout specifications. Furthermore, a set of algebraic operations is defined that can be used to modify layouts in a way that keeps the layout sound, i.e., solvable and non-overlapping. These operations can be used to describe all the edit operations in Stack & Tile and ALE.

Section 7.1 deals with related work. Section 7.2 describes the algebra and how it can be used to describe static layouts. In Section 7.3 a set of sound algebraic edit operations is presented. Section 7.4 covers how these operations can be used for layout editing. Section 7.5 shows how the algebra can be used to formalize applications such as Stack & Tile and ALE.
7.1 Related Work

An algebraic user interface description, as it is presented in this chapter, is a special type of description language. A user interface description language gives an abstract definition of the content of a UI. Depending on the problem there can be different abstraction layers to describe a UI [74]. This allows creating UIs for multiple platforms [1, 90]. However, these approaches do not consider actual layouts or layout models much, focusing mostly on more abstract UI specifications [76].

A formal constraint description language has already been used to specify layouts [42]. In this approach arbitrary constraints can be used and combined using logical operators. Constraints can be established relative to other layout items as well as to absolute coordinates. In our work only linear constraints are considered that are typically considered sufficient to describe GUI layout problems [5, 75]. Moreover, layout items are only constrained relative to each other, which makes it possible to define simple algebraic edit operations. Finally, our approach ensures soundness.

There are also domain specific UI description languages. A description language for the graph editor EDGE allows describing a graphical representation of a graph [87]. For example, the user is able to specify layout and appearance of the graph. However, the graph data is not included in this description language and no real layout editing is possible.

A formal language to describe relations of intervals has been introduced by Allen [2]. In this interval language the relative position of two intervals can be specified, e.g., if an interval is equal to, is before, meets or overlaps a second interval. While in the original language temporal intervals were targeted, the language has been augmented to describe spatial multi-dimensional intervals algebraically [7, 44]. Specifically, it has been applied to describe the block layouts of elements in a document [72].

The algebra presented here is comparable to the interval algebra described in [72]. However, we consider only adjacent intervals. Moreover, we consider how the soundness of a layout can be ensured when using algebraic operations. In our approach, only relations between layout items that hold for all layout sizes are described. Temporary position orders are not considered. For example, that a widget is on the left of another widget may be true for a large layout size but may become false when decreasing the layout size.

7.2 Tiling Algebra

The constraint-based Auckland Layout Model (ALM) [75] can be used to describe layouts for Stack & Tile as well as for GUI layouts. As described in Section 2.2, every rectangular layout area in ALM is surrounded by four tabstops. A layout area can be specified by the
five-tuple $A = (l, t, r, b, \text{content})$. This specifies the left, top, right and bottom tabstops the area is connected to. For GUI layouts the content of an area is usually a widget like a button or a text view, while for Stack & Tile the content is a window. Note that the five-tuple does not describe the intrinsic sizes of the content item but only the connections to tabstops.

**Definition 1.** The set of layout specifications that can be described using sets of five-tuples $(l, t, r, b, \text{content})$ is called $\text{LayoutSpec}$.

The layout specifications in Definition 1 represent layout topologies, i.e., arrangements of widgets regardless of concrete widget positions and sizes. For each layout specification, many different concrete layouts, i.e., layouts with concrete widget positions and sizes, are possible. Each layout specification can also be described by a set of equality constraints that specifies which widget borders are aligned with each other. In the following, we will choose the constraint notation (i.e., logical conjunctions of linear constraints) in some places to clarify the semantics of the operators of the proposed algebra.

One problem that arises when writing a complete layout specification using five-tuples is that the layout specification becomes difficult to read. For example, it becomes hard to see how different layout items are related. Furthermore, for layout edit problems such as layout item insertion, removal or resizing it is difficult to define simple and intuitive operations on a set of five-tuples.

For this reason, we introduce a new formalism that helps to make layout specifications more readable and more amenable to edit operations. In the following we use algebraic expressions, called terms, to describe connections of layout items. Most layouts can be described in a single term which results in an intuitive representation of how the actual layout would look. Only interlocked layouts such as the pinwheel (depicted in Figure 7.4) need more terms. One requirement for this formalism is that layout specifications are sound, i.e., that they describe only solvable and non-overlapping layouts. Furthermore, layout edit operations are defined so that they keep the layout specifications sound.

### 7.2.1 Tiling Operators

In ALE tabstops are used to specify the connections between layout items, i.e., the borders of a layout item can be connected to tabstops. However, tabstops are a somewhat virtual construct in the sense that they are not visible. Tabstops are not permanently fixed to a certain layout item. A tabstop can be disconnected from a layout item and connected to another layout item. In the following, a description language for layout specifications is presented that directly represents the connections between layout items, instead of just representing the connections of items to tabstops. The language tries to avoid the management of tabstops wherever possible.
To describe a relation between adjacent layout items a horizontal slash `/` and a vertical pipe `|` tiling operator are introduced. For example, the expression `A/B` means that a layout item `A` is on top of the layout item `B`, and analogously `C|D` means `C` is on the left of `D`. The tiling operators operate on *groups*, that are either single widgets or groups of widgets that are created by tiling widgets together. The set of all groups Group is a subset of all possible layout specifications LayoutSpec.

**Definition 2.** The horizontal and vertical tiling operators `/` and `|` operate on two groups and a tabstop. The tiling operator maps two groups `A, B` and a tabstop `t` to a new group. Tabstop\(x\) is the set of all vertical tabstops and Tabstop\(y\) is the set of all horizontal tabstops.

Type:

```
| : Group × Group × Tabstop\(x\) → Group

/ : Group × Group × Tabstop\(y\) → Group
```

Notation:

```
|(A, B, x) = A\|B

/(A, B, y) = A\)/B
```

Semantics:

```
[A\|B]_{x} =_{def} (A_{right} = x = B_{left} \land [A\|B]_{right} = B_{right} \land [A\|B]_{left} = A_{left})

[A\)/B]_{y} =_{def} (A_{bottom} = y = B_{top} \land [A\)/B]_{bottom} = B_{bottom} \land [A\)/B]_{top} = A_{top})
```

The infix notation for tiling allows us to form terms that have a visual appearance similar to that of the group that is specified. The `[]` operator above denotes the semantics of a tiling term, which is given as a logical conjunction of equality constraints on the right side of the definitions. Note that these sets also contain definitions for the borders of a group as appropriate, e.g., `[A\|B]_{right}`. Not all borders are defined for a group if it is not simply a single widget, e.g., the horizontal tiling group `[A\|B]_{x}` defines only left and right borders but no top and bottom. This means that without further definitions, it is not yet possible to describe groups by combining tiling operators such as `(A\|B)/C`. This will be resolved in the following.

In general a layout specification comprises a set of groups. To describe such a complete layout specification we introduce the asterisk operator `*`:

Type:

```
* : LayoutSpec × LayoutSpec → LayoutSpec
```

Notation:

```
*(A, B) = A * B
```
7.2 Tiling Algebra

The semantics of $*$ is the logical conjunction:

$$[A \ast B] =_{def} [A] \land [B].$$

For the $*$ operator we can specify the following axioms.

**Axiom 1** (Associativity). The $*$ operator is associative:

$$(A \ast B) \ast C = A \ast (B \ast C).$$

**Axiom 2** (Commutativity). The $*$ operator is commutative:

$$A \ast B = B \ast A.$$

**Definition 3** (Zero Group). There is a zero group with the semantic:

$$[0] =_{def} false$$

A zero group can be interpreted as an invalid group, i.e., a constraint specification that has no solution. From the semantics of the $*$ operator the following axioms can be motivated.

**Axiom 3** (Infeasible Specification). Every specification containing a zero group becomes zero

$$s \ast 0 = 0. \quad (7.1)$$

**Axiom 4** (Idempotence). Duplicated groups in a specification can be simplified:

$$A \ast A = A.$$

![Diagram](https://via.placeholder.com/150)

**Figure 7.1:** Grid layout.

A tabstop only has to be named if a second group uses this tabstop. Otherwise, the tabstop index in the tiling operator is simply omitted. The tabstops without indices are all distinct. Such shorthands for distinct arguments are well known. It is, for example, comparable to an underscore variable _ in the Prolog programming language.
Example 1. The layout connection-specification of the grid layout in Figure 7.1 can be written as:

\[ A | B * C | D = A * C | B / D \]

\[ x \quad x \quad y \quad y \]

However, a layout consisting solely of the first row can be written without a tabstop index:

\[ A | B \]

**Axiom 5** (Parallelism). Two connected groups containing the same tiling operator can be rewritten as follows. For the horizontal case:

\[ A | B * C | D = A | D * C | B \]

\[ x \quad x \quad x \quad x \]

and for the vertical case:

\[ A / B * C / D = A / D * C / B \].

**Axiom 6** (Concatenation). Two groups of the form \( A | B * B | C \) or \( A / B * B / C \) can be concatenated:

\[ A | B * B | C = A | B | C \]

and

\[ A / B * B / C = A / B / C \].

**Axiom 7** (Associativity). The tiling operators are associative:

\[ (A | B) | C = A | (B | C) \]

and

\[ A / (B / C) = A / (B / C) \].

In recollection of common distributivity laws, the following group of axioms can be postulated:

**Axiom 8** (Tiling Distributivity).

\[ (A | B) / C = A | B * A / C * B / C \]

\[ A | (B / C) = A | B * A | C * B / C \]

and

\[ (A / B) / C = A / B * A / C * B / C \]

\[ A / (B | C) = A / B * A / C * B | C \].
Both associativity and tiling distributivity can be intuitively derived from the semantics of the tiling operators. For example, the group \((A|B)/C\) means that \((A|B)_{\text{bottom}} = C_{\text{top}}\), that intuitively implies \(A/C * B/C\). Moreover, the group \(A|B\) is still valid, that leads overall to \(A|B * A/C * B/C\).

**Example 2.** The grid layout specification from Example 1 (Figure 7.1) can be specified as a single group:

\[
(A|B)/(C|D).
\]

**Example 3.** This above specification (for Figure 7.1) can be transformed to:

\[
(A|B)/(C|D) = (A/C)|(B/D).
\]

**Proof.** Using the tiling distributivity law the specification can be written as

\[
(A|B)/(C|D) = A|B * A/(C|D) * B /(C|D) \tag{7.2}
\]

\[
= A|B * A/C * A/D * B/C * B/D * C/D * C/D \tag{7.3}
\]

\[
= A/C * A/D * B/C * B/D * C/D * A/B. \tag{7.4}
\]

Using Axiom 3 for parallel groups

\[
A/D * B/C = A/C * B/D \tag{7.5}
\]

and Axiom 4 for duplicated groups

\[
C/D * A/B = C/D * A/C * B/C * A/B \tag{7.6}
\]

\[
= C/D * A/C * B/C * B,A \tag{7.7}
\]

\[
= C/D * A/C * B/C * B,A \tag{7.8}
\]

that becomes

\[
= A/C * A/C * B/D * B/D * C/D * A/B * C/B * A/B \tag{7.9}
\]

\[
= A/C * B/D * B/D * C/D * A/B * C/B * A/B \tag{7.10}
\]

\[
= A/C * C/(B/D) * A/(B/D) \tag{7.11}
\]

\[
= (A/C)|(B/D) \tag{7.12}
\]
**Definition 4** (Chain). A group $c$ with just horizontal or vertical tiling operators is called a chain.

For example:

$$c = A_0|A_1|...|A_{n-1}|A_n$$

**Axiom 9** (Zero Chain). A chain that contains a tabstop more than once is zero.

For example:

$$c_{\text{zero}} = A|x|B|x|C|x|D = 0$$

This is an insolvable specification as tabstop $x$ cannot be at two different places at the same time.

**Lemma 1.** If there is a chain that contains a group twice, the specification is zero.

*Proof.* This proof only describes the horizontal case, the vertical case is similar. The only chain that contains the same group twice and does not have a duplicated tabstop is of the form

$$A|B|A.$$  

Every chain that is longer would contain the left or the right tabstop of $A$ twice. However, $A|B|A$ is zero because, for example, the group $A|B$ in the chain can be duplicated:

$$A|x|B|x|A|B$$

which becomes

$$A|x|B|x|A|x|B = 0.$$

However, groups can be in multiple chains, for example:

$$c_0 = A_0|x|A_1|x|A_{i+1}|x|...|A_m$$

$$c_1 = B_0|x|B_j|x|B_{j+1}|x|...|B_n$$

$$c_2 = A_0|x|A_{i}|B_{j+1}|x|...|B_n.$$  

### 7.2.2 Rectangular Layouts

In the previous layout examples some connections were omitted. These are the connections to the outer tabstops of the layout. To describe these connections four outer layout items
l, t, r, b as depicted in Figure 7.2 can be used. These four layout items are “virtual” layout items to describe the outer tabstops; they do not represent real objects in a GUI layout. For example, the complete specification for a layout of two layout items A and B in a row is:

\[ l/A/B/r \ast t/(A/B)/b. \]

However, this specification is quite verbose and in the following a convention is used to simplify the notation significantly. This convention omits the outer layout items l, t, r and b from the specification. Instead, layout items that have no further connection in a certain direction are implicitly connected to the outer layout item in this direction. For example, the simplest layout specification s contains only a single layout item A: \( s = A \). Since there are no groups connected to any side of A, A is connected to all four outer layout items. Another advantage of this convention is that a layout specification is very easy to extend. For example, when extending the layout A by tiling an item B to the right, the layout simply becomes \( A \mid B \) and it becomes unnecessary to insert B between A and the outer layout item r. Using this convention, only rectangular layouts can be described.
Example 4. The rectangular layout shown in Figure 7.3 can be completely specified as:

\[(A|B)/C.\]

Example 5. A more complex example that cannot be described using a single group is the pinwheel (Figure 7.4). This layout can be written as:

\[(A|(B/E))/D*(E/D)|C*B/C.\]  \hspace{1cm} (7.13)

Figure 7.4: This entangled pinwheel layout cannot be specified as a single group.

7.2.3 $\lambda$ - Elements

In general a layout contains areas that are not occupied by layout items (see also Chapter 6). In the following, an empty rectangular area is associated with a so-called $\lambda$-element. These $\lambda$-elements are introduced for three reasons. Firstly, $\lambda$-elements are needed to describe layouts containing empty space. Secondly, as described in Section 7.2.2 layouts should always have a rectangular shape. $\lambda$-elements make it possible to describe non-rectangular layouts by filling the surplus space (see Section 6.4). Thirdly, a non-overlapping specification can be ensured by tiling all empty areas with $\lambda$-elements. By giving a $\lambda$-element a minimum size that is greater or equal zero, the layout becomes non-overlapping. In the following, we differentiate between $\lambda$-elements and layout items such as widgets. However, they are treated similarly and in most cases it does not matter whether an element is a layout item or a $\lambda$-element.

Definition 5 ($\lambda$-Element). A $\lambda$-element has the same properties as a regular group and can be connected to other $\lambda$-elements or groups using the horizontal and vertical tiling operators. A $\lambda$-element is connected to tabstops that are connected to at least one layout item.
The property that \( \lambda \)-elements are always connected to tabstops that are connected to at least one layout item ensures that a \( \lambda \)-element is always adjacent to layout items, as opposed to just other \( \lambda \)-elements. This means there are no “virtual” tabstops that are not connected to a layout item.

### 7.2.4 Symmetry Properties

It is quite intuitive that a layout is invariant against rotations or reflections. This symmetry can be described using a dihedral group \( D_4 \). The presentation of a dihedral group \( D_4 \) is:

\[
D_4 = \langle s, r \mid s^2 = 1, \ r^4 = 1, \ (sr)^2 = 1 \rangle.
\]

On the left side of the vertical line are the group generators \( s \) and \( r \) and on the right side are the relators of the group \( [63] \).

For our algebra we define \( s \) to be the horizontal reflection operator with the properties:

\[
(A|B)^s = B|A
\]
\[
(A/B)^s = A/B.
\]

Furthermore, \( r \) is the counter-clockwise rotation:

\[
(A|B)^r = B/A
\]
\[
(A/B)^r = A/B.
\]

It is quite easy to see that \( s \) and \( r \) are correct generators. For that we have to show that \( s^2 = 1 \)

\[
(A|B)^{ss} = (B|A)^s = A|B
\]
\[
(A/B)^{ss} = (A/B)^s = A/B
\]

and \( r^4 = 1 \)

\[
(A|B)^{rrrr} = (B/A)^{rrr} = (B|A)^{rr} = (A/B)^r = A|B
\]
\[
(A/B)^{rrrr} = (A|B)^{rrr} = (B/A)^{rr} = (B|A)^r = A/B.
\]

Furthermore, \( s \) and \( r \) must fulfill \( (sr)^2 = 1 \)

\[
(A|B)^{srsr} = (B|A)^{rs} = (A/B)^{sr} = (A/B)^r = A|B
\]
\[
(A/B)^{srsr} = (A/B)^{rs} = (A/B)^{sr} = (B|A)^r = A/B.
\]
An interesting transformation for our application is the flip operator

\[ f =_{def} sr. \]

This operator transforms a group as follows:

\[
(A|B)^f = (A|B)^{sr} = (B|A)^r = A/B
\]

\[
(A/B)^f = (A/B)^{sr} = (A/B)^r = A/B.
\]

This means the pipe operator is replaced by a slash operator and vice versa. As a consequence, all properties valid for the horizontal case are also valid for the vertical case. We will use this property to only define an operation for the horizontal case; the vertical case follows by applying the flip operator.

### 7.3 Layout Edit Operations

So far the algebra only describes static layout specifications. In this section a small set of edit operations is introduced that allows modifying a specification in a sound way. This means the edit operations leave the layout in a solvable and non-overlapping state.

The overall idea for editing an algebraic layout specification is to have a dense layout that is either filled with layout items or with \( \lambda \)-elements. A new layout item can then be inserted into a layout by replacing an existing \( \lambda \)-element with the new item. Removing items works analogically. However, because in general the right \( \lambda \)-element for an insertion is not available, operations are needed to transform \( \lambda \)-elements into the desired shape and size. This can be achieved by splitting and merging \( \lambda \)-elements. Furthermore, operations are needed to extend or shrink a layout by adding or removing \( \lambda \)-elements.

After presenting the edit operations, the layout soundness requirements are defined and the soundness of the edit operations is shown.

#### 7.3.1 \( \lambda \)-Operations

In the following we describe how \( \lambda \)-elements can be split and merged in a sound way. Furthermore, the extending and shrinking of a layout by a \( \lambda \)-element is discussed. For brevity only the horizontal case is described, the vertical case works analogously (see Section 7.2.4).

**\( \lambda \)-Operation 1 (Splitting).** A \( \lambda \)-element \( \lambda_0 \) can be split at a connection \( A|x|B \) if there is
no chain \( c = \lambda_0 \cdots |_x B \) or \( c = A \cdots |_x \lambda_0 \). The \( \lambda \)-element is then split:

\[
\lambda_0 = \lambda_1 |_x \lambda_2.
\]

Note, if the tabstop name \( x \) in \( A|B \) is not already given, then it is added, i.e., \( A|B \) becomes \( A|\lambda_0 x B \). By making sure that \( \lambda_0 \) is not directly or indirectly connected to the involved layout items \( A \) and \( B \), we avoid conflicting specifications where \( x \) appears in two different locations, e.g., \( \lambda_1 |_x \lambda_2 \cdots |_x B \).

\textbf{\( \lambda \)-Operation 2} (Merging). A group of the form \( \lambda_1 |_x \lambda_2 \) can be merged if there \( \exists \) \( A \) and \( B \) with \( A/(\lambda_1 |_x \lambda_2)/B \), and \( C \) and \( D \) with \( C/(\lambda_1 |_x \lambda_2)/D \). Furthermore, there are no other groups containing \( \lambda_1 \) or \( \lambda_2 \) in the specification. Then both \( \lambda \)-elements can be merged:

\[
\lambda_1 |_x \lambda_2 = \lambda_0
\]

and the two groups \( A/(\lambda_1 |_x \lambda_2)/B \) and \( C/(\lambda_1 |_x \lambda_2)/D \) become:

\[
A/\lambda_0/\lambda_0 /B
\]
\[
C/\lambda_0/\lambda_0 /D
\]

Note that any other groups containing \( \lambda_1 \) or \( \lambda_2 \) can usually be eliminated by using Axiom\([5]\) for parallel groups.

\textbf{\( \lambda \)-Operation 3} (Extending a Layout). A \( \lambda \)-element \( \lambda_0 \) can be added at one layout border (i.e., left or right in the horizontal direction) to the layout specification by extending all chains that end at the layout border by the new \( \lambda_0 \).

\textbf{\( \lambda \)-Operation 4} (Shrinking a Layout). A \( \lambda \)-element \( \lambda_0 \) can be omitted from the layout if all chains containing \( \lambda_0 \), \( \lambda_0 \) is either at the beginning of the chain (shrinking on the left side) or at the end of the chain (shrinking on the right side).

For example, \( \lambda/(A/B) \) becomes \( A/B \).

\section*{7.3.2 Removing and Inserting Layout Items}

When removing a layout item from the layout, the freed spaced is occupied by a new \( \lambda \)-element. This ensures that a layout is always completely filled, either with layout items or with \( \lambda \)-elements, and is thus overlap-free.

A layout item can be inserted into a layout by replacing an existing \( \lambda \)-element with the new layout item. To specify where the layout item should be inserted, all four surrounding
tabstops have to be named. Alternatively, four surrounding layout items can be given to specify the insertion position. In case a layout item should be inserted adjacent to a layout border, no layout item needs to be specified in this direction. The insert position of a new layout item \( X \) can be written using the insertion operator

\[
\begin{array}{c}
L \bar{X} R \\
T \bar{B}
\end{array}
\]

Here \( L \) is the layout item to the left, \( T \) the one to the top, \( R \) the one to the right and \( B \) the one to the bottom. The insertion can then be applied if there is a \( \lambda \)-element with the same adjacent layout items as specified in the insertion operator.

Sometimes only part of the free area occupied by a \( \lambda \)-element should be used for the new layout item. For example, it should be possible to insert a new layout item just into a corner of an empty area. It is important that at least one horizontal and one vertical tabstop of the new layout item is connected to a border of the empty area, so that its position is well-defined. To connect a layout item to a side or a corner of an empty area, the empty area can be filled with a group that has the desired properties. For example, to insert a new layout item \( A \) into the left top corner of an empty area, the group that has to be inserted could be

\[
A \rightarrow (A|\lambda_0)/\lambda_1.
\]

Here, \( \lambda_0 \) and \( \lambda_1 \) are two new \( \lambda \)-elements.

To specify which borders are not directly connected to the borders of the empty area, the corresponding indices are marked by an asterisk \( * \). For example,

\[
\begin{array}{c}
L \bar{X} R \\
T \bar{B}
\end{array}
\]

specifies that a group \( \lambda_0/X \) should be inserted into the area surrounded by \( L, T, R \) and \( B \). Indices with a * are called *loose insertions*, and normal indices are called *tight insertions*.

### 7.3.3 Soundness Requirements

There are two soundness requirements for the algebraic edit operations. Firstly, a layout specification must be solvable. Secondly, a layout specification must be non-overlapping.

A layout becomes unsolvable if there are contradicting constraints. For example, there is a conflict if one constraint ascertains a layout item to be on the left of a tabstop but a second constraint requires it to be on the right of the same tabstop. A conflicting layout specification for this problem is for example:

\[
s = A|B|C|D \ast C|A.
\]
This can be rewritten to
\[ s = A|B|C|A \star C|D \]
which is zero because \( A \) occurs twice in the same chain (Lemma 1)

\[ s = A|B|C|A \star C|D = 0 \star C|D = 0. \]

For that reason the first soundness requirement is equivalent to requiring a non-zero layout specification.

A non-overlapping layout specification can be defined as followed.

**Definition 6.** A layout specification \( s \) is called non-overlapping if for any two groups \( A, B \in s \) a horizontal or a vertical chain \( c \) exist that contains \( A \) and \( B \).

For example, the layout group \( s = A|x|\lambda_0|x|B \) is non-overlapping because for all pairs of groups a horizontal chain exists that contains both groups. These are \( c_0 = A|x|\lambda_0 \) for \( A \) and \( \lambda_0 \), \( c_1 = \lambda_0|x|B \) for \( B \) and \( \lambda_0 \), and \( c_2 = A|x|\lambda_0|B \) for \( A \) and \( B \).

### 7.3.4 Soundness

Assuming a set of sound layout operations, the soundness of all layout specifications can be shown by induction. A single layout item is naturally sound. Furthermore, editing an initial sound layout specification using sound layout operations results in a new sound layout specification. The proof of soundness for all edit operations is presented in the following.

**Proof: Splitting.** Non-zero specification: When splitting \( \lambda_0 \) at the tabstop \( x \), the only new connection that is introduced is \( \lambda_1|x|\lambda_2 \). This can cause a conflict if there are chains of the form \( c = \lambda_1|x|\lambda_2|...|B \) or \( c = A|x|...|\lambda_1|x|\lambda_2 \). However, this is not possible because from the splitting condition there are no chains \( c = \lambda_0|x|...|B \) or \( c = A|x|...|\lambda_0 \). Thus the specification is non-zero.

Non-overlapping specifications: Because \( \lambda_0 \) did not overlap any other layout item, also the group \( \lambda_1|x|\lambda_2 \) does not, as it fills the same area. Furthermore, the new group \( \lambda_1|x|\lambda_2 \) is in itself non-overlapping. This means for all new group combinations a connecting chain can be found, which is the definition for a non-overlapping specification.

**Proof: Merging.** Non-zero specification: Replacing the group \( \lambda_1|x|\lambda_2 \) by \( \lambda_0 \) does not add any existing tabstops, and therefore cannot create any chains that contain the same tabstop twice.
Non-overlapping specifications: All chains that show non-overlap for $\lambda_1|\lambda_2$ are also valid for $\lambda_0$. No other chains are needed to prove that the specification is non-overlapping.

Proof: Extending a Layout. A layout can always be written in the form $s = L * ... * R$. Here $L$ contains all groups that are connected to the left border and $R$ contains all groups that are connected to the right border. Extending the layout specification yields either $s = L * ... * R|\lambda_0$ or $s = \lambda_0|L * ... * R$. Both specifications are non-zero because only a new tabstop is added and the chains remain non-zero. Furthermore, $\lambda_0$ will be either to the left or to the right of all existing groups, which means the specification is non-overlapping.

Proof: Shrinking a Layout. The proof is similar to the proof for extending. After the operation, all chains are still non-zero as no tabstops were added. Furthermore, if the groups on the left (or right) of $\lambda_0$ were non-overlapping, then they are also non-overlapping after removing $\lambda_0$ as the corresponding chains for each pair of groups are still intact.

Proof: Removing and Inserting. The operations for removing and inserting layout items are trivially sound because they do not modify the topology of the layout specification. Also the loose insertion operation keeps the specification sound as it makes sure that gaps are correctly filled with $\lambda$-elements.

7.4 Layout Editing Process

After specifying the layout edit operations, all necessary tools for complex layout editing are at hand. However, there are various challenges when mapping a user operation to one or more algebra operations. $\lambda$-elements can be transformed in multiple ways in order to find a suitable configuration for an operation. Choosing the right configuration is not easy and depends on how the user actually sees the layout.

7.4.1 Equivalence Classes

In general there is no fitting $\lambda$-element available for an insertion operation and there are multiple ways the empty space in a layout can be tiled. For example, Figure 7.5 shows a layout specification with various options to tile the $\lambda$-elements.

The set of $\lambda$-elements that tiles the empty space of a layout specification is called $\lambda$-specification. As there is generally a set of different possible $\lambda$-specifications to describe the same empty space in a layout, these possibilities define an equivalence class of possible specifications for a layout. This reflects that the connections of layout items to other layout items stay constant for all $\lambda$-specifications of the equivalence class.
7.4 Layout Editing Process

To transform between different $\lambda$-specifications within an equivalence class, split and merge operations can be used. For example, the L-shaped configuration of $\lambda$-elements in Figure 7.6 can be transformed by splitting the longer $\lambda$-element and merging the resulting corner piece with the shorter $\lambda$-element.

\begin{figure}[h]
\centering
\includegraphics[width=0.7\textwidth]{figure7_5.png}
\caption{In general there are multiple ways to tile empty space with $\lambda$-elements.}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=0.2\textwidth]{figure7_6.png}
\caption{There are different ways to tile an L-shaped empty area.}
\end{figure}

7.4.2 Concrete and Abstract layouts

To describe further how the user can manipulate a layout, one has to differentiate between abstract and concrete layouts. An abstract layout describes the connection of groups relative to other groups (Section 7.2). To calculate an actual layout as it is rendered on the screen, the connections specified in the abstract layout together with the intrinsic sizes of the layout items and size constraints for the overall layout form a complete constraint system for the layout. By solving this constraint system the actual size of the layout items can be calculated. The solved layout, as it is rendered on a screen, is called a concrete
An abstract layout can have an unlimited number of concrete layouts. This is because it can be displayed at different layout sizes. Moreover, the intrinsic layout sizes can change the appearance of the layout dramatically. For example, Figure 7.7 shows a part of two different concrete layouts that have the same abstract layout. The only difference between the two concrete layouts is that the preferred intrinsic item sizes differ.

![Figure 7.7: Two concrete layouts with the same abstract layout \((A|\lambda_0)/(\lambda_1|B)\). On the left, layout items have a relatively large intrinsic preferred width. On the right, the preferred widths are relatively small.](image)

The difference between concrete and abstract layouts also has consequences for the operations the user is able to perform. In the example on the right of Figure 7.7, the user is able to insert a new item that separates the items \(A\) and \(B\) horizontally, e.g., after changing the \(\lambda\)-specifications:

\[(A/\lambda_3)/C|(\lambda_2/B).\]

However, in the concrete layout on the left of Figure 7.7, where the preferred sizes of the layout items are larger, this is not possible. While the operation is still possible in the abstract layout, the user would not be able to trigger this operation in the concrete layout.

A similar problem occurs when looking for the right member of the equivalence class for a desired insertion. Figure 7.5 depicts such a case. Assuming the initial concrete layout is the one in the left-top corner and the user wants to insert a new item into the center of the layout, there are multiple possible \(\lambda\)-specifications, e.g., the other three \(\lambda\)-specifications in Figure 7.5. While for all three \(\lambda\)-specifications \(\lambda_0\) is at the right position, the resize behavior or even the topology varies. For example, a change to the \(\lambda\)-specification on the bottom-right is usually not intended, as the right of \(D\) is then forced to be on the right side of \(\lambda_0\).

### 7.4.3 User Interaction on Abstract Layouts

As discussed above, changing an abstract layout may not have the desired effect in the concrete layout. A way to handle this problem is to take the concrete layout into account when modifying the abstract layout specification. The abstract layout specification is
used by the edit operations to ensure soundness for all possible concrete layouts. To summarize:

- A concrete layout is needed to decide what operations should be applied to the abstract layout.
- The abstract layout is needed to decide if an operation leads to sound concrete layouts.

### 7.4.4 Completeness

Using the algebraic description as described above allows one to specify a large set of layouts. To be more specific, all layouts where layout items are connected relative to each other can be described. Moreover, using $\lambda$-elements makes it possible to describe gaps between layout items.

While the algebraic description is able to describe every possible connection between layout items, it is unclear if the edit operations described here are powerful enough to generate all these possible layout specifications. It has to be shown that for the insertion of a new layout item into an empty area of a concrete layout, a suitable $\lambda$-element $\lambda_s$ in the abstract layout can be found.

To show this the following argument can be made. The concrete layout can be put into a grid structure. To do so, all the layout item borders are associated with grid lines. All $\lambda$-elements are then split so that they never occupy more than one cell in the grid. Because the concrete layout satisfies the connections from the abstract layout, this splitting of $\lambda$-elements is always possible in the abstract specification. From this grid, a suitable $\lambda_s$ can then be found by merging $\lambda$-elements as required.

### 7.5 Applications

In this section we discuss how the algebra can be used in real applications. Therefore, two applications, Stack & Tile and ALE, are analyzed and we describe how the application-specific edit operations can be mapped to the corresponding algebra operations.

#### 7.5.1 Algebra for Stack & Tile

There are only two operations in Stack & Tile that need to be considered. These are inserting and removing of a window to or from a Stack & Tile group. The Stack & Tile tiling operation can directly be mapped to the algebra insert operation and the Stack & Tile removing operation to the algebra removing operation described in Section 7.3.2. The stacking operation is not described by the algebra. However, stacking windows means that
all windows in a stack have exactly the same position and size. For that reason a simple solution is to allow multiple windows to occupy the same layout item. In other words, in Stack & Tile a layout item is a container that can hold one or more windows.

Splitting Stack & Tile groups

As described earlier in Section 5.2.3, removing a window from a Stack & Tile group can make it necessary to split the group into multiple smaller groups. Unfortunately the algebra does not directly help to discover such cases and a manual check for unconnected groups is needed. For example, the diagonal layout of two windows

$$\frac{(A|\lambda_0)}{x} \frac{1}{\lambda_1} \frac{B}{x}$$

needs to be split, although the algebra gives no indication for that. Note that looking for such diagonal connections is not enough to find the splits needed, since there could be other windows in the group that connect these diagonal windows.

In the following, we specify algebraically how Stack & Tile groups can be identified after removing a window, i.e., how to make sure that Stack & Tile groups are split appropriately. For this, we identify the windows that belong to a group by considering layout item paths. All windows on the same layout item path belong to the same Stack & Tile group. If there is more than one path, the Stack & Tile group needs to be split accordingly.

**Definition 7 (Layout Item Path).** A layout item path is a set of layout items that fulfills the following properties. Two layout items $A$ and $B$ belong to the same path $p$ if there is one of the following groups in the specification:

$$(A|B)/C \quad \text{or} \quad C/(A|B) \quad \text{or} \quad (A/B)|C \quad \text{or} \quad C|(A/B).$$

Note that $C$ could also be one of the “virtual” outer layout items. It is used to make sure that $A$ and $B$ are aligned to at least one common horizontal and one common vertical tabstop, in a way that $A$ and $B$ are clearly touching on one side (i.e., not just corner to corner). A layout item path can be found by using the transitive relation between items on the same path: if $A, B$ are on a path $p$ and $B, C$ are on the same path $p$, then also $A, C$ are on path $p$.

Once all independent paths are found, a new Stack & Tile group for each path has to be created. A simple approach to create a non-overlapping layout specification for a path is to copy the original specification of the overall layout and replace all layout items not belonging to the path with $\lambda$-elements. In general this leads to a layout that is surrounded
by unnecessary $\lambda$-elements. However, these superfluous $\lambda$-elements can be transformed as needed and then removed by shrinking the layout (see Section 7.3).

### 7.5.2 Algebra for ALE

Compared to Stack & Tile, ALE defines a few more layout edit operations. While inserting and removing of widgets can be handled similarly, four other operations have to be discussed, i.e., moving, swapping, resizing and inserting between an existing tabstop and an existing widget.

**Moving** a widget in ALE can be described by an algebraic remove and an insert operation. First the widget in question has to be removed. Then the resulting layout can be used to perform the insert operation. If the operation is interrupted in the GUI builder, the original layout specification is restored.

**Swapping** a widget is a fairly trivial operation: the two elements in the algebraic specification have to be swapped.

**Resizing** a widget can be described as well using an algebraic remove and an insert operation. When inserting the widget, only the connections at the resized borders of the widget (either one or two) change.

Resizing a widget to its preferred size (by detaching a widget from a tabstop) can be done by first removing the widget and then performing a loose insertion of the same widget (see Section 7.3.2). In the following a detach operation on the right layout item side is described; the other sides can be described analogously. The loose insertion operator for the right detach is

$$L^T_XR BS.$$  

Here, $L$, $T$ and $B$ are the widgets the detached widget was originally connected to. Now a suitable widget $R$ has to be found. To do so we search for the widget $R$ that gives the largest empty area encompassed by $L$, $T$, $R$ and $B$ in the concrete layout (note that this cannot be done in the abstract layout). Applying this loose insertion results in a new $\lambda$-element $\lambda_{new}$ between the detached widget and the widget $R$.

**Inserting a widget between an existing tabstop and an existing widget** can be achieved by first detaching the existing widget from the tabstop, as described for the resize operation. This results in a new $\lambda$-element $\lambda_{new}$. Then, the new widget is inserted into $\lambda_{new}$.

### Filling Gaps

As discussed in Section 6.3.6, remove, move and resize operations can cause gaps between widgets, so that a widget is not directly or indirectly connected to a horizontal or vertical layout border anymore. For example, a group of widgets can become “stranded” in the
middle of the layout, only surrounded by \(\lambda\)-elements. To fill the gap, the unconnected floating widget group is moved into the direction of the edited widget.

To describe this problem algebraically, first a definition of what a direct or indirect connection to a layout border means is needed. In the following, an indirect or direct connection is just called a \textit{connection} to a layout border, and defined recursively as follows.

\textbf{Definition 8} (Connection to a Layout Border). A layout item is connected to a layout border if it is connected to a tabstop that is connected to another layout item (not a \(\lambda\)-element) that is already connected to that layout border.

If a group of layout items is unconnected to a layout border in a certain direction, a \(\lambda\)-specification has to be found that has a \(\lambda\)-element \(\lambda_c\) that is the only separating item either to another connected widget or to the layout border in this direction. By eliminating this \(\lambda_c\), the gap can be filled. Such an elimination operation can be defined as follows.

\textbf{Definition 9} (\(\lambda\)-Elimination). Given a group:

\[
A | \lambda_c | B
\]

\[\begin{array}{c}\downarrow k \\
\end{array}\]

\[\begin{array}{c}\downarrow l \\
\end{array}\]

for which there is no other chain that connects \(A\) and \(B\): \(c = A | ... | B\). Then the elimination operation removes \(\lambda_c\) from the specification and merges the tabstops \(\downarrow k\) and \(\downarrow l\) to a new tabstop \(\downarrow m\)

\[
A | B
\]

\[\begin{array}{c}\downarrow m \\
\end{array}\]

with

\[\downarrow m = \downarrow k = \downarrow l\].

Note that by using the outermost tabstops of the groups \(A\) and \(B\) for \(\lambda_c\), we can ensure that the \(\lambda\)-elimination operation is applicable.

\textit{Proof of Soundness.} Non-zero specification: Because there is no chain \(c = A | ... | B\) other than \(A | \lambda_c | B\) before the elimination, there also is no group \(c = A | ... | B\) after the elimination. Non-overlapping specifications: All groups that were previously connected by the chain \(A | \lambda_c | B\) are still connected by \(A | B\) after the annihilation. This means the specification stays non-overlapping. \(\square\)

One might think a resize operation that increases the size of a widget could also be used to reconnect an unconnected group, similar to the elimination operation. However, in contrast to the elimination operation other connections of the resized widget are lost when performing a resize on that widget only. For example, if a widget \(A\) is connected to
\(\lambda_c\) and to another widget \(B\), e.g., \(A|\lambda_c \ast A|B\), and \(A\) is resized to take all space occupied by \(\lambda_c\), then the connection to \(B\) is lost. By contrast, this connection would be preserved when eliminating \(\lambda_c\).

### 7.6 Conclusion

In this chapter an algebra for tiled layout items has been presented. The algebra can describe arbitrary connections between adjacent layout items. By introducing vertical and horizontal tiling operators and defining rules to combine these operators, it is possible to describe layouts using the fairly straightforward notation of groups. This makes it easier to associate a concrete layout with the abstract layout specification, as opposed to a notation based on linear constraints.

The algebra was not only designed to describe static layouts but also to provide edit operations for modifying existing layout specifications. A requirement for these edit operations is that they are sound, which means that they keep a layout solvable and non-overlapping.

To fulfill the requirement of non-overlapping layouts, \(\lambda\)-elements were introduced. A \(\lambda\)-element is a rectangular element that has a minimum size greater or equal to zero. By tiling the whole empty space of a layout with \(\lambda\)-elements, the layout specification becomes non-overlapping. A new layout item can be inserted into the layout by replacing a \(\lambda\)-element with the new layout item. Similarly, a layout item can be removed by replacing it with a \(\lambda\)-element. Inserting and removing of layout items are sound edit operations.

In general there are multiple ways to tile an empty area with \(\lambda\)-elements. To change between these different tilings, \(\lambda\)-operations have been defined. These are splitting and merging of \(\lambda\)-elements, and extending and shrinking of a layout. We showed that these operations leave the layout in a sound state.

Furthermore, we analyzed the differences between abstract and concrete layouts. This differentiation is necessary to map edit operations triggered by the user to edit operations of the algebra.

Finally, we showed how the algebra can be mapped to operations of concrete applications. This proves that the algebra is powerful enough to describe the editing of Stack & Tile groups as well as the editing of GUI layouts in ALE.
Outlook: An Evaluation of Advanced User Interface Customization

In the previously presented Stack & Tile web survey (Section 5.5) it has already been shown that Stack & Tile is actually used and liked by real users. An interesting question is if these results can be transferred to more advanced customization systems. The goal of this chapter is to get an insight into the user’s opinion towards layout and functional customization. We analyzed functional customization to give an outlook on a further field of UI customization.

Previous work mostly addressed customization of menus and toolbars (see Section 8.1). However, more advanced customization covering all aspects of layout and also addressing functionality, are largely unexplored. Here, we focus on two advanced customization systems that allow users to change applications at runtime. The first system allows users to do complex layout editing by rearranging, adding and removing arbitrary widgets. The second system enables users to change the behavior of an application by adding, removing and rewiring functional components of the application. For both systems, the intended target population is that of power users, i.e., experienced users that spend a considerable time using certain applications. The customization systems were implemented for the Haiku open-source operating system [1].
In particular, we are addressing the following two research questions, focusing on technical users:

R1 Are users able to use customization approaches for layout and functionality?
R2 Would users use customization approaches for layout and functionality in practice?

Answering these questions helps us to understand whether such approaches would be useful.

To target these questions, a user evaluation was conducted using the two customization systems for layout and functionality. Both customization systems were demonstrated to 18 technical users, who were asked to perform customization tasks for three layout and two functional customization scenarios. The scenarios were designed to represent real-world customization use cases. The participants were observed during the tasks and given questionnaires at the end.

Most participants said user interface customization is useful and they would use it if available. In general it was easy for them to customize an application, and they easily understood how a problem could be solved using the customization systems. Participants stated that they encountered non-optimal GUI layouts during their daily work, and that they would like to be able to remove widgets, add non-standard functionality or optimize a layout using layout customization. Similar for functional customization, participants encountered applications where they would have liked to alter the behavior. Some participants feared that a customized application would not work properly, which illustrates the importance of a customization system to keep an application in a sound state. Furthermore, participants came up with interesting suggestions, e.g., that the system could be used to create GUI mockups or to enable non-programmers to create new applications. The evaluation presented in this chapter has already been published at the OzCHI’13 conference [114].

Section 8.1 discusses related work. The used customization framework is described in Section 8.2. Then the prototype for a functional customization framework is presented in Section 8.3. These customization frameworks are then used in a user study in order to answer the research questions (Section 8.4).

### 8.1 Related Work

A previous evaluation explored how users customized a text processor (Word Perfect) [89]. They found that users actually use macros, customize toolbars and to some extent change the appearance of a GUI. It was also shown that the performance of an application can be improved by customizing a GUI [32, 81]. Customization can also lead to a higher sense of control and identity with an application [80].
Adaptive approaches automatically optimize GUIs [24, 34, 39, 43]. For example, an adaptive system can add or remove buttons depending on the previous usage of the buttons in a toolbar [30]. However, it is not clear if adaptive GUIs are superior to customizable (adaptable) GUIs. For this reason, mixed-initiative approaches have been proposed [19, 20, 32] that combine the two. Here, the system merely proposes a GUI customization; the user is still in control and may apply or modify it. While previous work mostly looked at how menus or toolbars can be customized, our work enables the user to do more complex layout customization. More comparable are User Interface Façades, which allows users to compose a new layout by cloning and integrating visual areas from arbitrary windows in a single layout [100].

Mashup tools allow users to combine existing web resources and widgets in a new application [52], using a web component framework such as Google Gadgets [4]. The web resources or widgets do not need to be compatible or meant to work together in the first place. For example, mashups can display information from different web pages in a single view. To make heterogeneous resources work together, adapters are needed that achieve compatibility with the mashup tool. In a mashup editor, end-users can manipulate and wire different mashup components. Notable here is that there is often no difference between the design and the runtime phase [52]. This approach makes it quite similar to our functional customization prototype where users are allowed to modify applications at runtime. Setting up the communication between components is sometimes considered too hard for end-users. However, this can be addressed with approaches for auto-wiring of components [104].

The idea to wire predefined components at runtime is already quite old [65]. Block-based visual programming tools such as Scratch [92] and StarLogo [25] have been used for teaching programming to children. They allow users to assemble procedural programs from visual instruction blocks that behave similarly to puzzle pieces. While easier for novices than textual programming languages, these approaches are still too hard to use for average end-users. Node-based programming tools such as Quartz Composer [3] or LabVIEW [4] are frequently used for domain-specific applications, e.g., in multimedia and engineering.

### 8.2 Layout Customization

Layout customization gives the user the ability to rearrange, add and remove widgets in a layout. The underlying layout model used for our customization prototype is the
constraint-based layout model. This layout model is very powerful and can describe layouts that cannot be described with most other layout models [75]. The customization prototype uses the edit operations of the Auckland Layout Editor (ALE) to edit constraint-based layouts. These edit operations leave a constraint-based layout in a sound state; the layout stays solvable and non-overlapping. Furthermore, all edit operations keep widgets automatically aligned to each other, which increases the productivity when specifying a new layout or editing an existing layout (see Chapter 6).

To customize a layout, the following drag and drop edit operations can be used (see also Section 6.3). Existing widgets can be moved to an empty area in the layout. Here, a moved widget is placed aligned to existing widgets. A widget can also be moved between two other widgets or between a widget and a layout border. A widget can be resized by dragging a border of that widget to the border of another widget. Dragging a widget onto another widget swaps the positions of the two widgets. Furthermore, widgets can be removed from the layout by dragging them out of the layout, and added back in again if desired. Similarly to the move operation, a widget can be inserted into an empty area, between two widgets or between a widget and a layout border. All operations can be undone if necessary.

The prototype lets users switch an application into an editing mode at runtime, using a special keyboard shortcut. This is shown in Figure 8.2 for a media player application. In edit mode, the GUI becomes editable, using the operations outlined above. A properties window assists in the editing process, letting the user change the layout properties of the widgets. This window also contains a list of widgets that have been removed from the layout, or were not part of the initial layout.

8.3 Functional Customization

Our customization prototype is built on top of a component framework, i.e., the application to be customized is composed of components that are connected to each other [48, 73]. Functional customization lets users change the functionality of an application, by adding, removing and rewiring components.

In a component framework, e.g., COM or CORBA components usually have one or more interfaces that define the properties, methods and events of the components. We implemented a simple component framework that is widely inspired by OpenBinder. The component framework provides a unified way to access properties and to call methods. Events and methods both have parameter signatures, and an event can be connected to a

---

method with the same signature. Whenever an event is emitted from within a component, all connected methods are called. This can be done in a synchronous or an asynchronous manner; by default we use asynchronous events so that concurrent events can be handled in parallel. This keeps the user interface responsive and the user cannot create GUI deadlocks by creating circular connections.

Our component framework implementation is heavily inspired by the OpenBinder project. To aid the developer to create a new interface an interface description language (IDL) is used to define interfaces. From the interfaces specified in the IDL, code for the component framework is generated. For example, to add introspection abilities and make it easy to connect events and methods.

Sometimes the connections between two components can be complex, i.e., if the communication between the components involves a complex protocol. The simple mechanism of event-method connections would be too low-level and tedious in this case. For that reason we introduce the concept of socket-interface connections that allows to connect a whole set of functionality to a component. A component can have multiple sockets and each socket is associated with a certain interface. If a component implements the interface of a socket, the component can be wired to that socket. A component can then use all functionality of the connected component by using its interface. However, how the component uses the functionality cannot directly be controlled by the user. This is similar to a plug-in that adds functionality to an application.

Similar to layout customization, the user can switch an application into an editing mode using a special keyboard shortcut. In this mode, there is an additional component layer window that shows all customizable components of the running application (see Figure 8.1). To add a new component, the user can select from a list of available components that can be instantiated. The available components are usually part of the application but could also be provided by third party libraries. A new component can be integrated into an application by dragging it into the layer window and then wiring it to existing components.

In the component layer window the user can wire components using simple drag and drop operations. By dragging at an event slot and dropping at a compatible method slot (or vice versa), a connection between compatible events and methods can be established. Analogously, a socket can be connected to an interface. Components can be positioned freely in the component layer window.

Figure 8.1 shows an example of event and socket connections. Events are listed in the right bottom and methods are listed on the left bottom corner of a component. For example, the component labeled “Button” has an event “Invoke” that is connected to a “ComposeText” method in the “TextComposer” component. Thus, when clicking the but-
ton the event “Invoke” is fired and the ComposeText method is called. The interfaces of a component are listed on the left top and sockets are listed at the right top. For example, the TextComposer has a socket called “TextHook” that can be connected to an “IDemoTextModifier” interface. The components “HeaderAdder” and “SignatureAdder” implement this IDemoTextModifier interface and thus can be connected to the “TextHook” socket. The TextComposer then uses the HeaderAdder and the SignatureAdder to manipulate text, i.e., this is done internally by calling the “ProcessText” method.

![Diagram showing functional customization](image)

**Figure 8.1**: Functional customization: Methods are listed on the bottom-left and events on the bottom-right of a component. Interfaces are shown on the top-left and sockets on the top-right.

### 8.3.1 Challenges and Future Work

There are various challenges when using a functional customization approach as described above. While the current prototype allows us to investigate the research questions, a productive functional customization system has to satisfy more requirements. For users it is very important that an application stays in a sound state, i.e., it must stay usable at all times. In the following, challenges that have been identified during our work are summarized. Also ideas for future work are mentioned.

It is important for the integrity of an application, that the user cannot remove crucial components or crucial components connections. In general, the developer has to define what parts of the application is customizable. However, the customization system should assist the developer as much as possible and warn if conflicts are detected. Furthermore, the system must detect or avoid misbehaving application states. For example, the user could create circular event and method loops, i.e., an event calls a method that then
8.4 Evaluation

We investigated the two research questions in a user study. Each participant went through two evaluation parts, one for layout customization and one for functional customization. The participants had never used the customization systems before. Each part consisted of a guided walkthrough as a structured training and tasks participants had to solve, followed by a questionnaire. The detailed schedule of each part was as follows. In a short introduction participants were made familiar with layout and functional customization in
a number of customization scenarios. These scenarios covered the removal and insertion of widgets and functionality, as well as the optimization and simplification of UI layouts. For each scenario a customizable example application was given. The applications were chosen from a wide range of different domains to convey a representative impression of the many possibilities of UI customization. Firstly, participants were given time to explore the customization features of the application themselves, until they were comfortable using them. Then they were asked to perform some simple customization tasks. If a participant got stuck during a task, the participant was helped by the experimenter. Participants were encouraged to ask questions anytime. After finishing all tasks, they were asked to fill in a questionnaire containing Likert-scale and open-ended questions.

8.4.1 Methodology: Layout Customization

The first part of the evaluation covered layout customization. In a short introduction, participants got an overview of the use-cases for layout customization covered by the evaluation. These use-cases are that layout customization can be used to:

a) remove unneeded widgets,
b) add desired widgets that are hidden,
c) optimize a layout, e.g., by simplifying it.

The first demo application was a media player with a layout that was completely editable (see Figure 8.2). Furthermore, there were two widgets available, a record button and an MP3 tag view, which were not part of the initial layout. After letting the participants play with the editing functionality, they were asked to adjust the interface to their personal needs. In a second step, they had to add the hidden widgets, the record button and the MP3 tag view, to a position in the layout that they thought appropriate. Figure 8.3 shows examples of customized media players. These tasks covered the scenarios a), b) and c).
The second application was a mail application with an editable mail header interface (Figure 8.4). The initial layout of this application was not optimal, i.e., it used a lot of space and there was one very uncommon text encoding field. The participants were asked to simplify and improve this interface, which covered the scenarios a) and c).

While the first two scenarios used customizable standard applications of the underlying operating system, the last scenario used a mockup for a typical enterprise application, Employee Manager. This application shows a list of employees and an editable view of their attributes (left of Figure 8.5). There were two ways to select an employee: by directly selecting an employee in the employee list, or by using the up and down buttons below the list.

The following user story was presented to the participants: The chef of a company wants to know the exact number of employees who are vegetarian. So far this information was only stored in a general text comment field, and thus it cannot be easily queried from the database. In an upgrade of the Employee Manager, a new checkbox for vegetarian choice was added, which should now be used for the existing data records. In order to do so, somebody has to go through the whole database manually, read the comment field and tick the vegetarian checkbox if applicable. This job only has to be done once and the GUI is not optimized for this use-case, i.e., the mouse path between the vegetarian checkbox
and the employee list is quite long, which has a significant impact on performance. However, the GUI layout can be optimized for the task, e.g., by moving the checkbox next to the employee list or the up and down buttons, making the job much quicker and easier.

The layout was designed so that a clear performance improvement can be achieved by optimizing the layout. The question is not how much faster the participants perform the task after the customization, but if the participants are able to see the usability problem and customize the layout accordingly, targeting use-case c). The participants were first asked to do the checkbox-ticking job once without customization, and in a second time they were allowed to customize the layout. However, the participants were not told how an optimized layout could look like.

Finally, the participants were asked to fill in a questionnaire with the following 5-point Likert scale questions.

General questions:

Q1 I often use computers in my everyday life.
Q2 I would like to theme a GUI or change its look and feel.
Q3 If there are alternative widgets to control an application, I would like to choose between them.

Questions about the layout customization tasks:

Q4 I understood the Media Player example.
Q5 I understood the Mail example.
Q6 I understood the Employee Manager example.
Q7 It was easy to customize the GUI layouts.
Q8 I think I would be able to do my own GUI layout customizations using the customization system.
Q9 I understood what layout customization is.

Opinions about layout customization:
Q10 I would use layout customization.
Q11 Layout customization is useful.
Q12 I would use layout customization for applications I am using frequently.
Q13 I would use layout customization for applications I am using rarely.
Q14 I see no need for layout customization.

Need for layout customization:
Q15 I have encountered GUI layouts which are not optimal for my purposes.
Q16 I have encountered layouts that did not contain all the functionality I needed.
Q17 I would use layout customization to add non-standard functionality to a layout, i.e., add “hidden” widgets (similar to adding a record button to Media Player).
Q18 Applications have more functionality than I normally use.
Q19 I would use layout customization to remove widgets from a layout that I don’t use (similar to the peak view in Media Player or the encoding field in Mail).
Q20 I would use layout customization to optimize certain tasks (similar to the Employee Manager example).

Reusing layouts and expected customization problems:
Q21 I would like to share my customized layouts with other users.
Q22 I see the problem that my customized layouts could not work properly.
Q23 Layout customization is complicated.
Q24 Depending on the task I would like to have different layouts for the same application.

There were also some general open-ended questions:

- Can you give examples where you wanted to change the layout of a GUI?
- Can you give examples where you wanted to add additional widgets to an application?
- Can you give examples where you wanted to change the appearance of a layout, e.g., by rearranging widgets or removing unused widgets?
- Can you give examples where you wanted to change a layout to improve productivity for your purposes?
8.4.2 Methodology: Functional Customization

One of the goals of the functional customization evaluation was to find out whether participants are able to understand how to manipulate components to achieve a certain functionality (see R1). There were two demo applications, a simple “Bitmap Viewer” and a more complex “Message Composer” example.

Bitmap Viewer was a very simple bitmap viewer that displayed an image and a reload button situated below that image. When switching into the editing mode, the component layer window already contained a “bitmap input” component that loads a bitmap from disk, and a “bitmap view” component to display a bitmap. The widget palette contained a bitmap view and a “gray-scale filter” component, which could be connected to the bitmap view.

The participants were asked to connect the bitmap input with the bitmap view component in order to display the bitmap. Then they had to insert a new bitmap view component into the layout and connect it to the existing bitmap input component, so that the bitmap was shown twice. In the next step, a gray-scale filter had to be added to the component layer and then connected to one of the bitmap views. The resulting application displayed the same bitmap twice: once as a color picture and once as a gray-scale picture (Figure 8.6).

Message Composer was a simple chat application: it had a text view for entering a chat message, a “New Text” button for clearing the text view, and a “Send” button for sending the message entered into the view. When switching into editing mode, the component layer window contained components for the aforementioned widgets, as well as a “text composer” component, which was used to format and embellish the text entered in the text view. After pressing the “Send” button, the entered text got modified by the
text composer and was redisplayed in the text view.

The widget palette contained a new “timer view” widget, which implemented timer functionality. This widget was to be used to generate message timestamps, and show the timestamp of the last message. Furthermore, there was a set of “adder” components that could be connected to the text composer component, to add text to messages: a header adder to automatically add a message header, a signature adder to add a signature at the end, and a timer adder to add a message timestamp obtained by connecting the adder to the timer view (Figures 8.7 and 8.1).

![Figure 8.7: Message Composer: On the left side of the layer window the timer view component can be seen. This component is connected to the “New Text” and the “Send” button to start and stop the timer view (button components are not visible). On the right side a header adder is connected to the text composer component.](image)

The participants were asked to add and connect the header and signature adders to the text composer. Afterwards, they had to add the timer view to the layout, and connect the “New Text” button with a restart timer method and the “Send” button with a stop timer method of the timer view. In order to add the composing time to the message text, they had to add the timer adder component and connect it to the text composer and timer view. After each step, they were able to test if their customization worked as intended.

Finally, the participants were asked the following 5-point Likert-scale questions:

Q25 I understood what functional customization is.
Q26 I understood the difference between functional and layout customization.
Q27 I understood the functional customization examples.
Q28 There is need for functional customization.
Q29 I have used applications that did not have the behavior that I expected.
Q30 I would use functional customization.
Q31 I would have liked to change the behavior of some applications I have encountered (similar to adding a gray-scale filter to the Bitmap Viewer).
Q32 I would have liked to add more functionality to some of the applications I have encountered (similar to adding a timer view to the Message Composer).

Q33 Functional customization is not necessary.

There were also some general open-ended questions:

- Can you give examples where you wanted to change the behavior of an application, similar to the example of adding a gray-scale filter?
- Can you give examples where you wanted to add functionality to an application, e.g., by adding new components?
- Comments? Suggestions?

8.5 Results and Discussion

The evaluation was conducted during a user meeting of the Haiku community (BeGeistert 026, November 2012). During this meeting, 18 participants were recruited. They were all male, between 19 and 43 years old (average age 34). All of them were technical users, and at least 11 of them had programming experience. The media player and the mail application used in the study are standard applications of Haiku. Hence, we could assume that participants had seen or even used these applications before.

While only roughly half of the participants were interested in changing the look and feel of an application, over two-thirds stated they would like to choose between different alternative widgets if available (Figure 8.8). One interpretation for that is that people are less interested in purely visual changes, but see the possible benefits of alternative widgets, which could potentially offer new functionality.

8.5.1 Observations: Layout Customization

It seemed to be easy for the participants to change the layout of the media player. They rearranged, removed and added new widgets to the layout without problems. This is consistent with the findings of an earlier evaluation of the used layout edit operations [111].
For the mail application, participants came up with a variety of interesting layouts. For example, two participants recreated a layout they knew from other mail applications, e.g., Outlook. Other participants removed widgets they were not using frequently, i.e., the CC, BCC or the encoding widget. One user made the layout very space efficient by just using two rows.

For the Employee Manager application there were two main layouts that participants used to optimize. The first most common solution was to move the vegetarian checkbox directly beside the list box (9 participants). The second most common solution was to move the vegetarian checkbox beside the up and down buttons (4 participants). One participant updated his initial customization by moving the vegetarian checkbox from beside the list view next to the up and down buttons after he realized that this might be an even better solution. The customizations show that participants understood the UI problem, and were able to solve it using layout customization.

### 8.5.2 Questionnaire: Layout Customization

All participants understood the layout customization tasks (Figure 8.9, Q4 - Q6). Almost all of them stated that it was easy to customize the layouts (Q7). All participants said they think they would be able to do their own layout customization (Q8). Moreover, they understood what layout customization is (Q9).

These results indicate that the tasks were easy to understand and that the participants got a good understanding of layout customization. This is in agreement with the observations made. The participants had no problems using the layout customization system, and would apparently also be able to use the system without the help of the experimenter.

There was a wide agreement (89%) that layout customization is useful (Figure 8.10, Q11). Consistently, they disagreed (89%) that there is no need for layout customization (Q14). Over 75% of the participants stated they would use layout customization (Q10).
While 66% of the participants stated they would use it for frequently used applications (Q12), 44% would also use it for rarely used applications (Q13).

While it was not surprising that people found layout customization useful, it is interesting that most users also found they would use layout customization in practice. Many participants stated they would not only use it for frequently used applications, but also for rarely used applications. This indicates that there is an actual demand for layout customization.

There was wide agreement (100%) that the participants had encountered applications with non-optimal layouts (Figure 8.11, Q15). All participants had encountered layouts with missing functionality (Q16), and 72% would use layout customization to add non-standard functionality to a layout (Q17). Over 80% of the participants said they used applications which had more functionality than necessary for them (Q18), and almost all agreed that they would remove such functionality using layout customization (Q19). Furthermore, over 85% stated they would use layout customization to optimize a layout for a certain task (Q20).

These results are a strong indication that layout customization is considered valuable.
Almost all participants said they had encountered applications which would benefit from layout customization, and agreed that they would use of it. All three suggested use-cases, i.e., adding widgets, removing unused widgets and optimizing a layout, were considered relevant in practice.

There was no clear agreement whether users would like to share their customized layouts with other users (Figure 8.12, Q21). While 33% of the participants did not see non-functional layouts as a result of customization as a problem, 28% had concerns about them (Q22). Less than 20% agreed that layout customization is complicated (Q23). Only three participants disagreed that they would like to use different layouts for the same application (Q24).

According to Q21, many participants think that their customized layouts can also be useful for other users. However, it is not clear if users would like to use customized layouts from others. The results from (Q24) suggest that many participants can imagine managing multiple layouts for an application, e.g., to optimize it for different tasks. Most participants disagreed that layout customization is complicated (Q23), which is compatible with the inverse question (Q7).

As seen in Q22, some users are concerned that problems occur as a result of layout customization. For example, users fear that modified layouts are non-functional or reduce their productivity. The participants of this study were mostly advanced users, so one might expect stronger concerns for less experienced users. This indicates the importance for a customization system to leave the layout sound and the application in a functional state.

8.5.3 Observations: Functional Customization

The observations for the functional customization scenarios were similar to those of layout customization. Most participants had a clear idea of how to approach the customization tasks. A common problem was that participants forgot to connect some of the components. However, the participants had no big difficulties finding the problem when trying
out the application, at most requiring a short comment from the experimenter to point
them into the right direction. Some participants did not even need the full task explana-
tion from the experimenter, e.g., one participant asked if certain components need to be
connected even before the task was explained.

8.5.4 Questionnaire: Functional Customization

Almost all participants understood what functional customization is; only one par-
ticipant was neutral on this point (Figure 8.13, Q25). Only one participant did not
understand the difference between layout and functional customization (Q26). All partic-
ipsants understood the two functional customization examples (Q27). Only one participant
disagreed and 72% agreed that there is need for functional customization (Q28). This is
consistent with question Q33, were 66% disagreed that functional customization is not
necessary. 78% of the participants had used an application that did not have the behavior
they expected (Q29). Roughly 60% would use functional customization (Q30). Questions
Q31 and Q32 had quite similar responses: only 1-2 participants disagreed, while 66%
agreed that they would have liked to change or add functionality to an application.

From the responses we can say that the introduction to functional customization was
sufficient to familiarize participants with it and make them understand what the difference
to layout customization is. Furthermore, participants seem inclined to use functional cus-
томization in practice. Most participants would like to change or extend the functionality
of some applications. Although functional customization is typically more complex than
layout customization, participants apparently still want to use it. However, participants
see a lesser need for functional customization than for layout customization.
8.5 Results and Discussion

8.5.5 Feedback

There were many comments from the open-ended questions in the questionnaire. Participants listed many applications where they would have liked to change the layout or the functionality. In the following, only the most frequent and interesting comments are summarized.

About layout customization, one participant said he “would like to use the layout customization system to create mockups for applications.” Another participant mentioned he already changed the GUI of an application by modifying the source code, since no customization options were available. There were many comments that said that they would like to add widgets for certain functionality, e.g., menu items, to applications. As examples of where unused widgets should be removed, two participants mentioned complex application such as Photoshop\textsuperscript{9}, Gimp\textsuperscript{10} and Blender\textsuperscript{11}.

About functional customization, one participant stated that it could work well as a simple way for non-programmers to build basic applications, given that a sufficiently large component library is provided. Moreover, there were many examples where participants wanted to enhance the functionality of an application. Two participants wanted to pass data between different applications. Another interesting example was to add consistency checks before sending a mail to specific recipients. One participant wrote that he would like to be able to execute a macro after a certain event occurred.

8.5.6 Threats to Validity

There are some internal threats to validity. At the beginning of the user meeting where the evaluation was conducted, the experimenter gave a talk about layout and functional customization, and also showed some customization examples. Some participants did not attend this talk, so they may have had a disadvantage compared to the participants who attended. However, during the experiment no difference between the two groups was observed and all participants understood the tasks. Hence, we can assume that the talk had no significant impact on the results of the evaluation.

The functional customization prototype was in a very basic state, e.g., the layer window showed a lot of irrelevant information and connecting components was implemented in an unintuitive way. Despite these limitations, participants had no major difficulties in performing the functional customization tasks. Thus, one might expect even more positive results for a more mature functional customization prototype.

Another potential issue is that most participants were part of the Haiku community. The experimenter, who had implemented the prototypes for Haiku, is also part of that

\textsuperscript{9}\url{photoshop.com}
\textsuperscript{10}\url{gimp.org}
\textsuperscript{11}\url{blender.org}
community, which could lead to a social desirability bias. This means the participants could have been influenced in favor of the customization prototypes. Although some questions were asked twice with different formulations to counterbalance this effect (Q11 and Q14, and Q28 and Q33), one must be aware of this fact when interpreting the results.

An external threat to validity is that most participants were power users or programmers. It is difficult to say if the results would be the same for less experienced users. However, layout and functional customization targets mainly experienced users who are interested in adapting their applications. Thus the results are valid for that target group.

In the evaluation, there were in total five examples for layout and functional customization. This limited set of examples may have conveyed an incomplete picture to the participants of what layout and functional customization is. To avoid this problem, the examples had been designed to cover a broad spectrum of customization use-cases. Thus, we believe it is possible to generalize the results to other use-cases of layout and functional customization.

8.6 Conclusion

Previous work about customization by end-users focused on simple customization such as for toolbars and menus, leaving more advanced customization approaches largely unexplored. We presented prototypical systems for layout and functional customization, and a user study with these systems, exploring if users are able to use such advanced customization approaches (R1), and whether they would use them in practice (R2). The study was performed with 18 participants, using customization tasks for five different plausible scenarios.

The results suggest that technical users with an advanced level of experience are able to use advanced customization approaches for layout and functionality. From our observations, it was easy for the participants to perform the given layout and functional customization tasks. Several participants had concerns that their customized layouts might not work properly. Customization can indeed potentially render an application unusable, so it would be important to develop safeguards against this. This could be done by making sure that important widgets cannot be removed, or that a sound application state can always be restored.

The results also indicate that technical users with an advanced level of experience would indeed customize their applications. Most participants stated that they do encounter use-cases for both layout and functional customization in the applications they are using, that they would customize layout and functionality, and that they consider such customization useful. They stated that they would, for example, like to add functionality to UIs that did not contain all the functionality they needed, remove unused widgets, or
optimize layouts for their tasks.

Several participants had concerns that customized layouts might not work properly. Customization can indeed potentially render an application unusable, so it would be important to develop safeguards against this. This could be done by making sure that important widgets cannot be removed, or that a sound application state can always be restored. Customization may also lead to user documentation getting out of sync with the current application UI, and it may interfere with keyboard-based interaction. These problems are future work.

While this study provides some initial insight into the research questions, a long-term field study with typical users could give more accurate answers and also provide an insight into the long-term benefits of customization. In many comments participants expressed that they would like to use layout and functional customization in many of the applications they were using. Some of the suggested use-cases would lead to entirely new applications, such as using layout customization to create UI mockups, or using functional customization as a tool for non-programmers to create basic applications. Exploring the long-term uses and benefits of customization is future work.
This thesis covers many different parts of UI customization and various research questions have been addressed. The main achievements of our research are summarized in the next Section 9.1. Section 9.2 lists ideas and future work we were not able to address in the scope of this thesis.

9.1 Achievements

In this thesis we investigated how constraint-based layout can be used for UI customization. The following list highlights the major achievements of our work:

- **Usability study comparing the grid-bag layout model and the constraint-based layout model.** The grid-bag layout model is probably the most used layout model, e.g., it is widely used in many GUI toolkits and for HTML tables. While the constraint-based layout model is more powerful it can also be more complex. In this thesis we compared the usability of layout creation and editing tasks for both layout models. While the grid-bag layout model performs better when creating new layouts, the constraint-based layout model is faster for layout editing tasks. Furthermore, users found the constraint-based layout model easier to use. These results have been published at the OzCHI’12 conference [117]. Consequently, the constraint-based layout model is an excellent choice to describe layouts that need
to be resizable, e.g., to adapt to different screen sizes of desktops, tablets or phones. This confirms the direction of our research.

- **Effects of soft constraint solving strategies on layout aesthetics.** A common problem in a GUI is that in general a widget cannot assume its preferred size, i.e., usually there is too much or too little space available in the layout. In a constraint-based layout the preferred size of a widget is described using a soft-constraint. How these soft-constraints are solved depends on the objective function used. When using a linear objective function the widget sizes are in general underspecified, and it is undetermined how space is distributed to the widgets. We solved this problem by distributing available space in a well-determined way using a quadratic objective function. In a comparison of this approach to a weighted and an equal distributed space approach, we found that a quadratic objective function yields good aesthetics for small as well as for large layout sizes. These results have been published at the CHINZ’12 conference [113]. Current implementations of the constraint-based layout model usually use a linear objective function. We showed how these implementations can be improved by using a quadratic objective function.

- **Constraint-based window customization with Stack & Tile.** The Stack & Tile system allows users to stack windows on top of one another or tile them beside each other. This can, for example, be used to group windows of multiple applications by task. Stack & Tile uses the constraint-based layout model to lay out windows in a Stack & Tile group. While holding the Stack & Tile key, simple drag and drop operations can be used to manipulate the constraint-based layout of a group. A challenge here was to ensure that only sound layouts that are solvable and non-overlapping can be created.

In a user study, we compared the usability of Stack & Tile with the traditional desktop metaphor. We found that multi window tasks can be performed much quicker using Stack & Tile. Moreover, switching between multi window tasks is faster as well. Stack & Tile is already integrated and used in Haiku. In a web survey actual users of Stack & Tile were interviewed. The results showed that Stack & Tile is used and liked by many users and that they are using Stack & Tile for a variety of tasks. These results have been published at the INTERACT’13 conference [115]. Our research showed that such an approach can also be advantageous for other desktop window managers.

- **Complex GUI editing of constraint-based layouts.** Layout editing in Stack & Tile is quite simple compared to the requirements for GUI layout editing, i.e., only insertion and removal of windows is supported. We developed novel edit operations for specifying and editing complex constraint-based GUI layouts. These edit
operations were designed to keep a layout solvable and non-overlapping. The developed library can not only be used for layout editing, as demonstrated in the Auckland Layout Editor (ALE), but also to customize a GUI at application runtime. A detailed description of ALE’s edit operations has been published at the UIST’13 conference [112].

In a user evaluation, the performance of ALE and its edit operations was compared to a GUI builder that supports the grid-bag layout model and another GUI builder that supports the constraint-based layout model. ALE was found to perform significantly faster in both cases, and users liked layout editing with ALE more. This shows that complex constraint-based layout editing can be done efficiently with good usability. These results have been published at the INTERACT’13 conference [111].

Integrating our new approaches for graphical constraint-based layout editing into existing layout builders could significantly improve the productivity of GUI designers. Our work makes the constraint-based layout model accessible for GUI designers of different skill levels, including those who have no knowledge of the underlying constraint system. ALE makes it possible to create layouts that are not possible with most other layout models, and the full power of the constraint-based layout model can be utilized.

- **Algebra for layout specification and editing.** We developed an algebra that can describe all layouts consisting of rectangular, adjacent layout items. Moreover, we defined algebraic layout edit operations. The algebraic specifications are easy to read and most layouts can be written in a single term. We proved that the algebra operations generate sound layouts that are solvable and non-overlapping. Furthermore, we discussed the differences between abstract and concrete layouts and how user interactions can be mapped to the algebra operations. Finally, we showed that all edit operations available in Stack & Tile and in ALE can be mapped to a set of algebra operations. The developed algebra makes it possible to describe constraint-based layouts in a formal and consistent way.

- **Evaluation of advanced user interface customization.** The last chapter of this thesis targeted the questions if users are able to use layout and functional customization, and if users would use such systems in practice. First, prototypical layout and functional customization systems were described. While the layout customization system uses the edit operations described in Chapter 6, the functional customization system utilizes a component framework. The layout customization framework allows users to modify a GUI layout and to remove and add widgets from the layout at runtime. In the functional customization system users are able to add, remove and rewire components at runtime using a simple node-based graphical user interface.
In a user study, experienced users were able to try and use these systems in several customization scenarios. Afterwards they were asked for their experiences and opinions. We found that it was easy for them to use these customization systems, and that they would like to use them in practice if available. These results show the relevance of integrating layout and functional customization abilities into future applications. These results have been published at the OzCHI’13 conference [114].

Considering these achievements, we believe that the question whether the constraint-based layout model can be leveraged for UI customization can be answered positively. While the constraint-based layout model is more powerful and complex than, for example, the grid-bag layout model, we showed that it can be used in complex systems for layout customization. Furthermore, we showed how constraint-based layout customization systems can generate solvable and non-overlapping layouts. Hopefully, this work will encourage more developers to utilize constraint-based layout editing and customization in their projects.

## 9.2 Future Directions

During our work we identified several points that we were not able to address. This was either because of lack of time or the issues were out of the scope of this thesis. The major points are listed in the following.

To strengthen the results from the layout aesthetics evaluation (Section 4) a similar experiment should be conducted with a larger number of participants. Moreover, participants with formal training and experience in graphic design should be considered.

There are many ideas how Stack & Tile can be further improved and integrated into Haiku (see also Section 5.3). One idea is to show Stack & Tile groups in the taskbar and group windows in the taskbar by their group affiliation. A smaller improvement is to make the Stack & Tile key customizable, e.g., to use the middle mouse button as it is used for stacking in KDE. Another important point is to make Stack & Tile groups persistent. This would allow restoring a group at a later point, e.g., after a reboot or by explicitly restarting a previously closed group. However, this requires a way to store the state of an application, which is currently not supported by the Haiku API.

An interesting feature that has been requested by users is merging of Stack & Tile groups, i.e., stacking or tiling multiple windows at the same time. This is challenging since it is unclear how arbitrarily shaped groups should be merged. The simplest solution would be to handle a Stack & Tile group as a single rectangular area that can be inserted like a regular window. To combine two Stack & Tile groups new interaction methods must be developed. At the moment, pressing the Stack & Tile key while moving a group removes the dragged window from the group, hence multi window operations are not possible.
Currently, there is an API available that allows stacking windows programmatically. Hence, Stack & Tile can be used by an application as an alternative for a tab-bar widget. For example, instead of using a tab-bar widget in a browser Stack & Tile can be used to provide similar functionality. However, no application currently makes use of the stacking API. Also missing is a tiling API. Such an API could implement the algebraic insert operator as described in Section 7.3.2.

There are also various improvements that can be made to ALE. For example, currently there is no interface to edit widget properties, such as labels or the appearance. Furthermore, there is no interface to use custom, non-standard widgets in the builder. Another important feature is to support nested layouts. While the current set of layout edit operations can be used to edit complex layouts, additional edit operations could be considered. For example, an interesting operation would be the insertion of a widget between a tabstop and several other widgets aligned at that tabstop. In this case, the tabstop could be split into two tabstops and the widget could be inserted between these two tabstops.

Currently there is no way to edit multiple widgets in a single operation, e.g., it is not possible to move a group of widgets. Describing such multi-widget operations algebraically can be challenging and needs to be investigated. A simple solution is to handle a group of widgets as a rectangular area that is defined by its bounding box. However, it becomes very difficult when the exact shape of a group is taken into account. For example, when combining two comb-like groups there are in general many possible ways how they could be connected, and it is not clear what the most intuitive connection is.

As already discussed in Section 8.3.1 there are many problems that have to be addressed for functional customization. However, functional customization is targeting a different field that is only discussed briefly in this thesis.


