Widget set independent layout management for UIML

Jo Vermeulen

Promotor: Prof. dr. Karin Coninx
Supervisors: dr. Kris Luyten, Tim Clerckx

Academic year 2004-2005

Eindverhandeling voorgedragen tot het behalen van de graad van
Licentiaat in de Informatica afstudeervariant Multimedia

School voor Informatie Technologie
Transnationale Universiteit Limburg

Diepenbeek, June 6, 2005

Preface

The most prevalent reason for choosing this thesis was that it seemed an interesting challenge to me. I found the prospect of learning about different techniques for realizing multi-device layout management appealing.

The thesis also provided an opportunity for me to find out more about the problems surrounding multi-device user interface development, a topic I became acquainted with through my internship at the EDM [1], in July 2004. During this internship I worked on Uiml.net, an open source UIML [2] renderer by dr. Kris Luyten. Uiml.net makes it possible to render a high-level user interface description in UIML with different widget set backends, thereby supporting multiple computing platforms.

Another decisive reason for me was the fact that it is a relevant and important problem. Considering the growing diversity of computing platforms, more efficient techniques for multi-device user interface development are needed. I experienced first hand that a generic layout management technique for UIML was needed to easily support multi-device user interface development using Uiml.net. While the larger part of a UIML document could be shared across widget sets, the layout was specified differently for each individual widget set. Manual intervention by the interface designer was thus still required when changing the backend.

I learned a lot about these topics during the course of this thesis. Additionally, while studying constraint-based layout management techniques, I gained knowledge about the problem of constraint solving, which I found very interesting.

[1]Expertise Centre for Digital Media
[2]User Interface Markup Language

Acknowledgements

I would like to express my gratitude to a number of people who helped realize this thesis.

First of all, I wish to thank my supervisor dr. *Kris Luyten*. His enthusiasm, encouragement and inspiration along with countless suggestions, critique and helpful feedback proved invaluable. I am very grateful for the time he spent guiding me.

I also want to thank my promotor Prof. dr. *Karin Coninx* and my second supervisor *Tim Clerckx*, for reviewing my text and providing valuable suggestions.

Furthermore, I would like to thank the people of the *Constraints Research Group at the University of Washington* under the lead of *Alan Borning*, for their original Cassowary constraint solver. Special thanks go to *Greg J. Badros* for taking the time to answer my questions and providing useful insights in the behavior of the solver.

Finally, thanks to my girlfriend *Evelien* for always being there for me and helping me through stressful times, and to *my parents* for supporting me in my studies.

Abstract

The growing diversity of computing environments requires a new, more efficient methodology for developing user interfaces. With traditional techniques one develops a user interface per *computing platform*. There is a plethora of existing research available on the development of multi-device user interfaces. We can conclude that it is mostly clear how to deduce a set of *generic widgets*. It is however difficult to generalize the different layout mechanisms widget sets use.

We present a generic, *constraint-based* layout mechanism. It relies only on the least common denominator of layout management: absolute positioning. A constraint solver is used to find values for the position and size of each widget, while adhering to the imposed constraints. These positions and sizes are then set using absolute positioning.

Automated layout management, although greatly improving on speed compared to designing a layout by hand, is not guaranteed to produce satisfying results. Human interference is usually still needed. There are many recurring tasks in user interfaces (e.g. providing a username and password). For each task, a suitable layout must be determined. Our method supports *layout patterns* which enable the designer to reuse existing layouts which have proven to be usable and aesthetically pleasing.

Our method will be based on the UIML language. While succesfully achieves abstraction in many ways, there is no generic way to specify the layout. This makes it more difficult to use one UIML document for several widget sets, since each widget set requires its own specific layout description. We do not seek to complement the UIML specification with a generic, multi-modal layout mechanism though: it is limited to 2D graphical user interfaces.

For the practical part of this thesis, we integrated the layout specification in *Uiml.net*, an open source renderer for UIML [LC04]. We also developed *Cassowary.net*, a port of the Cassowary constraint solving toolkit to the .NET platform, which is used to solve the layout constraints.

Contents

List of Figures

List of Listings

Part I

Preliminaries

Chapter 1

Introduction

Contents

1.1 Problem description

The growing diversity of computing environments requires a new, more efficient methodology for developing user interfaces. With traditional techniques one develops a user interface per *computing platform*. A computing platform (or environment) is an architecture in hardware and/or software that allows software to run. It typically includes the device's hardware, operating system, programming language, runtime libraries and widget set. With the rise of mobile and embedded devices, the set of different computing platforms software will need to support grows significantly. It is therefore beneficial to develop a user interface in a high-level, device-independent way, in order to minimize the effort in supporting a new platform. Every year, five times more software is realized for embedded systems than for desktop systems. The number of companies developing software for embedded systems is expected to grow to ten million by 2010 [vUvAB^{+}98].

There is a plethora of existing research available on the development of multi-device user interfaces. We can conclude that it is mostly clear how to deduce a set of *generic widgets*. It is however difficult to generalize the different layout mechanisms widget sets use.

A *layout manager* is a software component which handles the positioning and scaling of individual widgets in a user interface. The most simple layout manager requires the designer to specify an absolute position for each widget. We call this *absolute positioning*. It is clear that this method is not very flexible and requires a lot of input from the interface designer. It is the only layout mechanism available for the MFC [1] widget set for Microsoft Windows, a toolkit generally known to be complex and difficult for developing user interfaces. Other techniques can adapt the layout to a certain degree when the available screen area shrinks or grows. Most noticeable are the layout managers used by Gtk, Qt, WxWidgets or the Java GUI toolkits (Swing and AWT). Although these are certainly more flexible than absolute positioning, they cannot stretch as far to handle both desktop systems and embedded devices [LCC03]. It would be interesting if the user interface could dynamically adjust itself, in order to better satisfy the constraints of the new environment. Often we can accomplish the same type of interaction through different concrete widgets. Selecting a value in a certain range can for instance be achieved either by a slider widget or by a spinbox. While a slider widget is usually fairly large in one dimension (be it horizontal or vertical), the spinbox is very compact. When our interface (or a part of it) doesn't fit in the available space, we could substitute every slider widget by a spinbox widget, thereby freeing up precious screen space. One could also regroup parts of the interface when there is less space available. A desktop interface consisting of three columns could by this idiom be regrouped in a tab view widget, containing the three columns in separate tabpages.

The ideal situation would be to completely automate the layout process. Unfortunately this seems not yet feasible. Most succesful methods require some form of designer input. We call this *semi-automatic* layout management. The designer could for example logically group parts of the user interface. Coming back to our tabpage example, we would put unrelated groups in different tabpages.

If only the common characteristics of layout mechanisms are extracted, we end up with a very weak, banal layout mechanism. A lot of possible solutions, such as spatial constraints, originate from a high-level, abstract view on the problem. A widget set independent layout mechanism should ideally support every layout that is possible by each of the specific widget sets.

1.2 Scope of this thesis

We are searching for possible techniques to make the layout specification independent of the widget set that is used. As well in the literature as in existing implementations, possible solutions have been explored. One possibility is to use spatial constraints and a contraint solving algorithm, but there are also a number of other solutions that can be used.

Our method will be based on the UIML language. We do not seek to complement the UIML specification with a generic, multi-modal layout mechanism though: it is limited to 2D graphical user interfaces. If one wants

[1]Microsoft Foundation Classes

to support a speech interface for example, a completely different notion of *layout* is needed. That is however, beyond the reach of this thesis.

The practical part of this thesis consists of integrating the layout specification in Uiml.net, an open source renderer for UIML[LC04].

1.3 Outline of this document

In the next chapters of the preliminaries, we will introduce some general terminology and provide a detailed overview of UIML, model-based user interface development and automated layout management.

The next part covers the research that was conducted in the scope of this thesis. We discuss constraint solvers and flexible presentations.

The last part gives details about the implementation part of this thesis. In order, we cover Uiml.net, the implementation of the UIML template element, and the implementation of the constraint solver.

Finally we draw the conclusions and provide a summary of the thesis in Dutch.

Chapter 2

UIML

Contents

The *User Interface Markup Language* (*UIML*) is an XML language that permits a declarative description of a user interface in a highly device-independent manner [AP99]. It is a fairly mature language, which is currently in the process of being standardized by the OASIS Standards Committee. There is interest in submitting it as a World Wide Web Consortium (W3C) specification.

UIML is a meta- or extensible language, analogous to XML. It does not contain widget set specific information. It only contains generic elements such as *part*, *property* or *event*. UIML allows you to define your own user interface specification, which is similar to XML in the sense that XML allows you to define your own data format. Of course, eventually the abstract concepts must map to their specific counterparts. This is handled by a so-called *vocabulary*.

2.1 Structure of a UIML document

We based this text on the latest version of the specification [AH04] currently available: version 3.1.

In UIML, a user interface is really just a set of *interface elements*. An interface element is called a *part*. Parts may be organized differently, for different end-users or devices. Each interface element has *content* (being text, sounds, images, etc.). Of course, an end-user must be able to interact with the user interface. This is accomplished by the *behavior* element, which is built on rule-based languages. Each rule contains a condition and a series of actions. When the condition is true, the corresponding actions are executed. A condition is associated with an *event*. And as we said earlier, the eventual mapping of the abstract concepts to their specific counterparts happens in the *presentation* element (which is also known as the vocabulary).

Summarizing, this is the outline of a UIML document:

- **Interface**: defines a virtual tree of parts with their associated content, behavior and style
  - **Structure**: the initial virtual tree organization of parts
  - **Style**: a set of style properties for the interface
  - **Content**: a set of constant values
  - **Behavior**: rules for runtime behavior
- **Peers**:
  - **Presentation**: mappings of part and event classes, property names and event names
  - **Logic**: the underlying application logic, the glue between the user interface described in the UIML document and other code

2.2 The template element

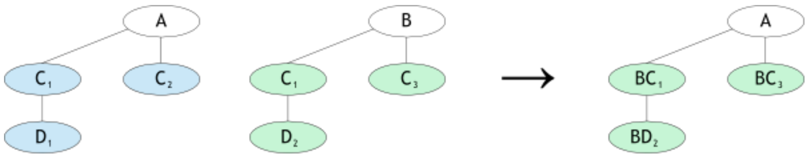
The UIML template element enables user interface designers to create reusable interface components. A detailed description of the possible applications of reusable interface components can be found in Section 6.2. Chapter 8 describes how templates fit in with our approach for layout management.

A template element can be viewed as a separate branch on the UIML tree [AP99]. A template branch can be joined with the main UIML tree anywhere there is a similar branch. This means the first and only child of the template must have the same tag name as the element on the UIML tree where the join is made. There are three different ways of *sourcing* a template element with another UIML element:

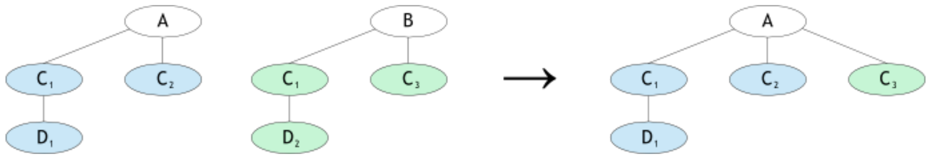
- *replace*: all the children of the element on the main tree that sources the template are deleted and replaced by the child nodes of the template element.
- *append*: all the children of the element of the main tree that sources the template are kept, and additionally all the child nodes of the template element are added to the list too. Name conflicts are handled by appending the template's name to the names of its children.
- *cascade*: the children of the template are added to the element on the main, but if there is a name conflict, the child of the element on

the main tree is retained. This is similar to what happens in CSS (Cascading Style Sheets).

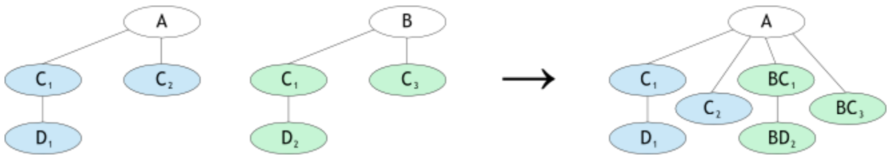
Figure 2.2 gives an overview of the three different sourcing methods.



(a) Part A sources part B using “replace”



(b) Part A sources part B using “append”



(c) Part A sources part B using “cascade”

Figure 2.1: The three sourcing methods

2.3 Examples

Let us look at a few concrete UIML interfaces. The first interface we are going to cover consists of a combo box packed inside a frame container. A combo box is a widget supporting the selection of a unique value in a list. Listing 2.1 shows the UIML document for this interface. Rendering this document with Uiml.net would result in the interface shown in Figure 2.2.

Listing 2.1: Combo.uiml

```
<?xml version="1.0"?>
<uiml>
  <interface>
    <structure>
      <part id="Top" class="Frame">
        <part class="VBox">
          <part class="Label" id="thelabel"/>
          <part id="combo1" class="Combo"/>
          <part class="Label" id="thelabel2"/>
        </part>
      </part>
    </structure>
    <style>
      <property part-name="Top" name="label">
        UIML Combo Example
      </property>
      <property part-name="thelabel" name="text">
        Belgian Alternative Music
      </property>
      <property part-name="thelabel2" name="text">
        Pick one!
      </property>
      <property part-name="combo1" name="content">
        <constant model="list">
          <constant value="dEUS"/>
          <constant value="Nemo"/>
          <constant value="The Evil Superstars"/>
          <constant value="Channel Zero"/>
          <constant value="Star Industry"/>
        </constant>
      </property>
    </style>
  </interface>
  <peers>
    <presentation base="gtk-sharp-1.0.uiml" />
  </peers>
</uiml>
```

The structure section is a hierarchy of parts, with the `Top` frame as the root element. `Top` contains a `VBox` container, which we will not discuss further (they are used for layout management in the GTK# widget set). It contains three parts, two text labels and the combo box.

Every part has a `class` and an optional `id`. The `class` indicates the type of interactor. This can be a high level interactor (such as `Label`), or a widget set specific interactor (such as `VBox`). The `id` attribute is used to uniquely identify each part when they are referred to in other parts of the UIML document.

Next is the style section, which consists of a list of properties, describing the style of the interface. In this case, it sets the labels of `thelabel` and `thelabel2` and initializes the content of `combo1`. The `constant` element is used to populate trees or lists with data.

Finally the peers sections specifies which vocabulary to use. This example uses the GTK# vocabulary.

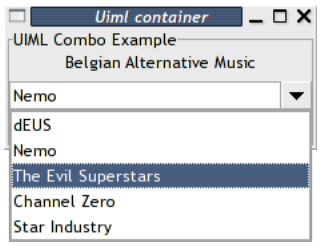


Figure 2.2: Combo.uiml rendered with Uiml.net

2.4 Inter-vocabulary distances

Of course the goal of an high-level user interface description language is to minimize the designer's effort in supporting a different platform, or more specific a different widget set. The ideal case would be to only change the vocabulary of the UIML document. Unfortunately different widget sets have similar widgets in common, but still have other properties [Luy04]. A commonly used idiom is to provide a *generic vocabulary* [AnAS02]. However, interfaces supported by a generic vocabulary tend to be rather trivial.

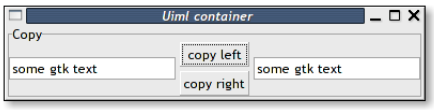
Uiml.net (see Chapter 7) utilizes a different approach: *inter-vocabulary distance* is minimized, not reduced to zero, by following a few simple rules. The vocabularies use the same naming scheme for the widget mappings. The semantics of a specific widget determine its name in the vocabulary. Furthermore, event handling is solely handled by UIML's behavior section, and thus completely independent of the specific widget set that is used. This approach is not yet complete though. Descriptions of widget set specific layout management in UIML documents widen the gap between different vocabularies, and makes it hard to switch from one backend to another. It is necessary to specify layout management in a generic way across different vocabularies in order for this approach to work.

Figure 2.3 shows a UIML interface rendered with two different rendering backends: GTK# and System.Windows.Forms. This interface consists of two entries (for entering text) and two buttons. The upper button copies the text from the left entry into the right entry, while the lower button does the opposite (copy from right to left). These actions are specified in a generic manner in the behavior section of the document. The behavior section is thus the same for both backends. Figure 2.4 is a graphical view of the differences between the two UIML documents.

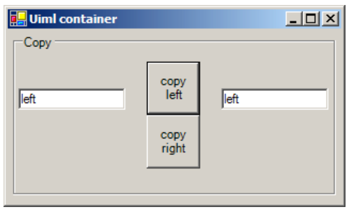
Listing 2.2 and Listing 2.3 show the UIML documents behind these interfaces.

Listing 2.2: GTK# version of the Copy interface

```
<?xml version="1.0"?>
<uiml>
```



(a) GTK# backend



(b) System.Windows.Forms backend

Figure 2.3: SWF and GTK

```
<interface>
  <structure>
    <part class="Frame" id="Frame">
      <part class="HBox">
        <part class="Entry" id="leftentry"/>
        <part class="VBox">
          <part class="Button" id="copyleft"/>
          <part class="Button" id="copyright"/>
        </part>
        <part class="Entry" id="rightentry"/>
      </part>
    </part>
  </structure>
  <style>
    <property part-name="Frame" name="label">
      Copy
    </property>
    <property part-name="copyleft" name="label">
      copy left
    </property>
    <property part-name="copyright" name="label">
      copy right
    </property>
    <property part-name="leftentry" name="text">
    </property>
    <property part-name="rightentry" name="text">
    </property>
  </style>
  <behavior>
    <rule>
      <condition>
        <event class="ButtonPressed"
          part-name="copyleft"/>
      </condition>
      <action>
        <property part-name="rightentry" name="text">
          <property part-name="leftentry"
```

```
              name="text"/>
          </property>
        </action>
      </rule>
      <rule>
        <condition>
          <event class="ButtonPressed"
            part-name="copyright"/>
        </condition>
        <action>
          <property part-name="leftentry" name="text">
            <property part-name="rightentry"
              name="text"/>
          </property>
    </action>
    </rule>
  </behavior>
  </interface>
  <peers>
    <presentation
      base="gtk-sharp-1.0.uiml" />
  </peers>
</uiml>
```

Listing 2.3: System.Windows.Forms version of the Copy interface

```
<?xml version="1.0"?>
<uiml>
  <interface>
    <structure>
      <part class="Frame" id="Frame">
        <part class="Entry" id="leftentry"/>
        <part class="Button" id="copyleft"/>
        <part class="Button" id="copyright"/>
        <part class="Entry" id="rightentry"/>
      </part>
    </structure>
    <style>
      <!-- Absolute positioning  -->
      <property part-name="Frame" name="position">
        5,5
      </property>
      <property part-name="Frame" name="size">
        302,150
      </property>

      <property part-name="leftentry" name="position">
        5,50
      </property>
      <property part-name="leftentry" name="size">
      100,25
    </property>
    <property part-name="rightentry" name="position">
      195,50
    </property>
    <property part-name="rightentry" name="size">
      100,25
    </property>

    <property part-name="copyleft" name="position">
      125,25
    </property>
    <property part-name="copyleft" name="size">
      50,50
    </property>
    <property part-name="copyright" name="position">
      125,75
    </property>
    <property part-name="copyright" name="size">
      50,50
    </property>
    <!-- /Absolute positioning -->

    <property part-name="Frame" name="label">
      Copy
    </property>
    <property part-name="copyleft" name="label">
      copy left
    </property>
    <property part-name="copyright" name="label">
      copy right
    </property>
    <property part-name="leftentry" name="text">
    </property>
    <property part-name="rightentry" name="text">
    </property>
  </style>
  <behavior>
    <rule>
      <condition>
        <event class="ButtonPressed"
          part-name="copyleft"/>
      </condition>
      <action>
        <property part-name="rightentry" name="text">
          <property part-name="leftentry"
            name="text"/>
        </property>
      </action>
      </rule>
      <rule>
        <condition>
          <event class="ButtonPressed"
            part-name="copyright"/>
        </condition>
        <action>
          <property part-name="leftentry" name="text">
            <property part-name="rightentry"
              name="text"/>
          </property>
    </action>
    </rule>
  </behavior>
  </interface>
  <peers>
    <presentation
      base="swf-1.1.uiml" />
  </peers>
</uiml>
```

The documents are for the greater part alike. There are two main differences though:

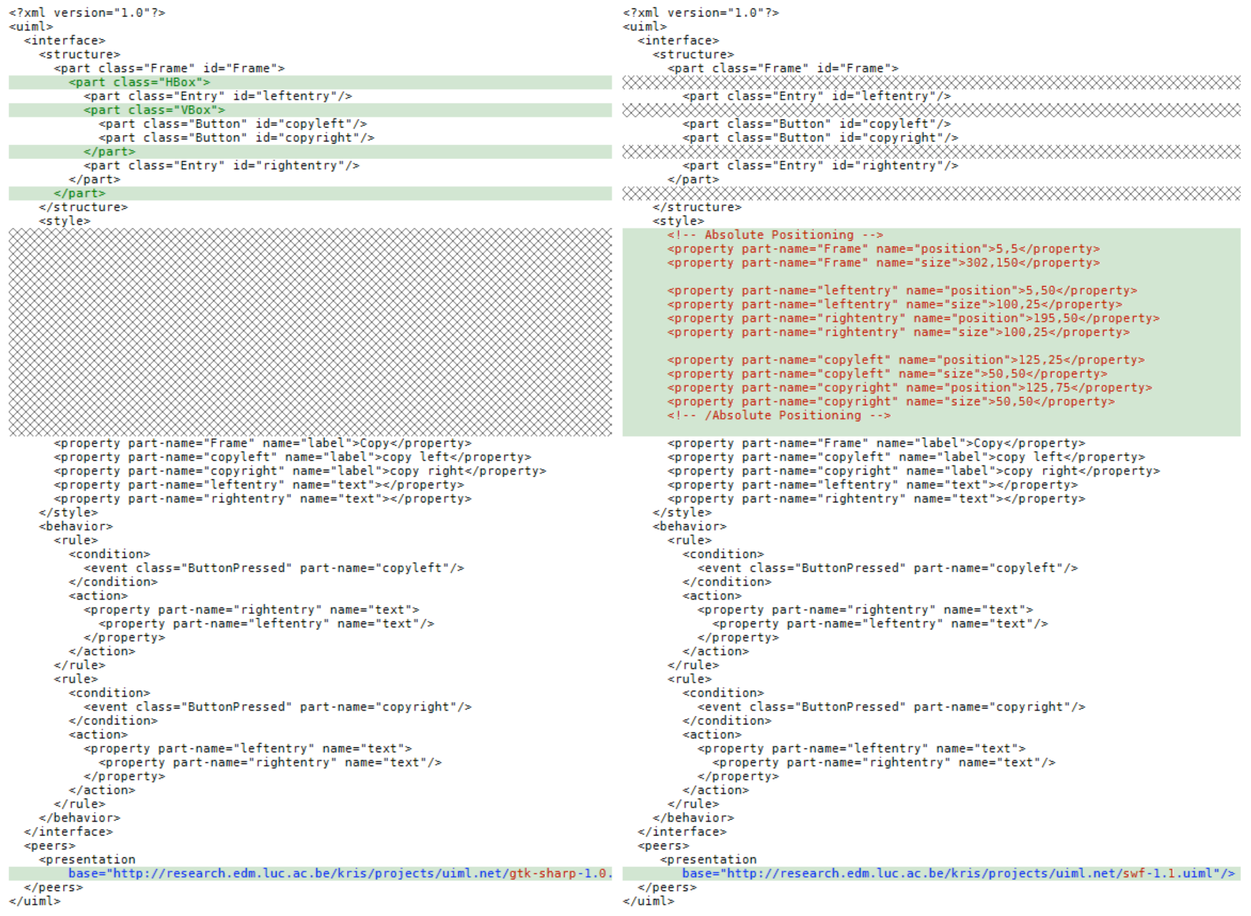
- layout management specific components in the *structure* section
- layout management specific properties in the *style* section

The GTK# version includes special *layout management containers* in the structure section, namely parts of the `HBox` and `VBox` classes. These define the layout of their child widgets. `HBox` lays out its children horizontally, while `VBox` does so vertically. These containers can be nested inside each other to create a more complex layout.

The System.Windows.Forms version on the other hand uses special *layout properties* in the style section. These specify the absolute position and size of each widget, in order to realize a similar layout as the GTK# version. This method also known as absolute positioning which we briefly discussed in Section 1.1.

Although this is a fairly simple example, more complex *form-based* interfaces are also mostly similar between different backends except for these two differences. In other words, these two differences occur independent of the complexity of the user interface [Luy04].

Layout management should be expressed in a generic manner, if we want to easily switch from one widget set to another (let alone switch from one platform to another). The style and structure sections are not sufficient for generalizing layout management across different widget sets. Another representation of graphical layout is needed for UIML.



```
<?xml version="1.0"?>
<uiml>
  <interface>
    <structure>
      <part class="Frame" id="Frame">
        <part class="HBox">
          <part class="Entry" id="leftentry"/>
          <part class="VBox">
            <part class="Button" id="copyleft"/>
            <part class="Button" id="copyright"/>
          </part>
          <part class="Entry" id="rightentry"/>
        </part>
      </part>
    </structure>
    <style>
      <property part-name="Frame" name="label">Copy</property>
      <property part-name="copyleft" name="label">copy left</property>
      <property part-name="copyright" name="label">copy right</property>
      <property part-name="leftentry" name="text"></property>
      <property part-name="rightentry" name="text"></property>
    </style>
    <behavior>
      <rule>
        <condition>
          <event class="ButtonPressed" part-name="copyleft"/>
        </condition>
        <action>
          <property part-name="rightentry" name="text">
            <property part-name="leftentry" name="text"/>
          </property>
        </action>
      </rule>
      <rule>
        <condition>
          <event class="ButtonPressed" part-name="copyright"/>
        </condition>
        <action>
          <property part-name="leftentry" name="text">
            <property part-name="rightentry" name="text"/>
          </property>
        </action>
      </rule>
    </behavior>
  </interface>
  <peers>
    <presentation
        base="http://research.edm.luc.ac.be/kris/projects/uiml.net/gtk-sharp-1.0.
  </peers>
</uiml>
```

```
<?xml version="1.0"?>
<uiml>
  <interface>
    <structure>
      <part class="Frame" id="Frame">
        <part class="Entry" id="leftentry"/>
        <part class="Button" id="copyleft"/>
        <part class="Button" id="copyright"/>
        <part class="Entry" id="rightentry"/>
      </part>
    </structure>
    <style>
      <!-- Absolute Positioning -->
      <property part-name="Frame" name="position">5,5</property>
      <property part-name="Frame" name="size">302,150</property>

      <property part-name="leftentry" name="position">5,50</property>
      <property part-name="leftentry" name="size">100,25</property>
      <property part-name="rightentry" name="position">195,50</property>
      <property part-name="rightentry" name="size">100,25</property>

      <property part-name="copyleft" name="position">125,25</property>
      <property part-name="copyleft" name="size">50,50</property>
      <property part-name="copyright" name="position">125,75</property>
      <property part-name="copyright" name="size">50,50</property>
      <!-- /Absolute Positioning -->
      <property part-name="Frame" name="label">Copy</property>
      <property part-name="copyleft" name="label">copy left</property>
      <property part-name="copyright" name="label">copy right</property>
      <property part-name="leftentry" name="text"></property>
      <property part-name="rightentry" name="text"></property>
    </style>
    <behavior>
      <rule>
        <condition>
          <event class="ButtonPressed" part-name="copyleft"/>
        </condition>
        <action>
          <property part-name="rightentry" name="text">
            <property part-name="leftentry" name="text"/>
          </property>
        </action>
      </rule>
      <rule>
        <condition>
          <event class="ButtonPressed" part-name="copyright"/>
        </condition>
        <action>
          <property part-name="leftentry" name="text">
            <property part-name="rightentry" name="text"/>
          </property>
        </action>
      </rule>
    </behavior>
  </interface>
  <peers>
    <presentation
       base="http://research.edm.luc.ac.be/kris/projects/uiml.net/swf-1.1.uiml"/>
  </peers>
</uiml>
```

Figure 2.4: Differences between the GTK# and System.Windows.Forms versions of the same interface

Chapter 3

Model-Based User Interface Development

Contents

As we discussed earlier in the problem description, the rise of embedded and mobile devices poses a series of unique challenges for user interface design and development. User interfaces must run on a broad range of computing platforms, each having its own constraints.

To meet these challenges, the most frequently adopted practice consists in developing unique interfaces for each case. Clearly, this is not an efficient approach. First of all, there is an unnecessary repetition involved in implementing a UI for each platform and usage case. One must guarantee a consistent design across these platforms, although these different interfaces are likely implemented by many different designers. Revisions to the design, must be separately implemented on each specific interface. Finally, the introduction of a new device requires a complete re-implementation of the UI.

Current practices for multi-device user interface development are thus in need of significant improvement. *User interface modelling* is proposed as a possible solution [EVP01]. User interface modelling consists of the creation of knowledge bases, describing various components of the user interface, such as the presentation, the platform, the task structure, the context, etc. We can exploit these knowledge bases to *automatically* produce a usable UI matching the requirements of each context of use. Model-based user interface development is supported by tools such as Mobi-D [Pue97] or Dygimes [CLV+03].

3.1 The user interface model

A set of model-based techniques can be used to greatly improve the design and development of multi-device user interfaces. They all depend on the development of a *user-interface model*, which is a formal, declarative, implementation-neutral description of the UI. We will overview three relevant model components: the *platform model*, the *presentation model* and the *task model*.

A platform model describes the various computer systems that may run a UI, including their features and constraints. This model can be exploited at design-time only, or be dynamically exploited at run-time (which is preferred). When used at run-time, it can be sensitive to changes, such as a reduction in bandwidth. The user interface could then respond appropriately.

A presentation model describes the visual appearance of the UI. It includes information about the hierarchy of windows and their widgets, style attributes, and the selection and placement of these widgets. Each widget is modeled abstractly as an *AIO*: an *abstract interaction object*. AIOs are platform-neutral. Furthermore, each AIO is associated with several *CIO*s: *conrete interaction objects* which are specific to a certain platform. This allows our UI to run on any computing platform, as long as the appropriate CIOs are present. Figure 3.1 describes the relationship between an AIO (the Push Button), several related CIOs, and the platforms on which they are instantiated.

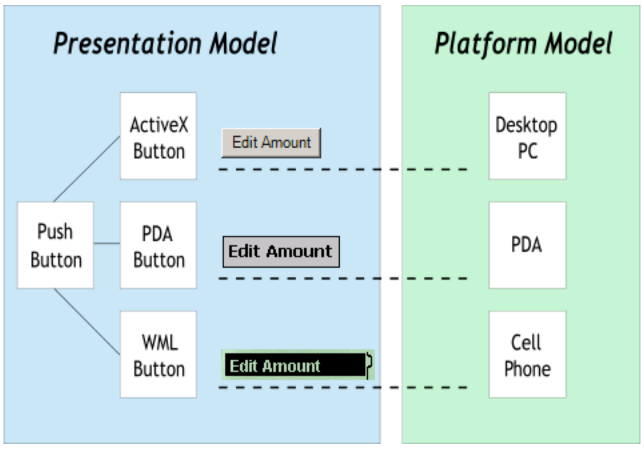


Figure 3.1: CIOs are subclassed from an AIO, and are then mapped on to specific platforms

A task model is a structured, hierarchical representation of the tasks that the user of the software may want to perform. It supports a clear abstraction of the interactive system without omitting the significant details that matter in UI design [LCA05]. The task model is decomposed into subtasks. Additional information may be supplied (goals, preconditions, postconditions, etc.). The model also includes whether a task is optional, whether it may be repeated and whether it enables another subtask (meaning the subtask is available through this task).

Of course these are not the only relevant models. However, the model-based techniques described in [EVP01] depend only on these three models. The connections between the various model components are stressed to be very important. They define the interactive behavior of a UI. For instance, the connection between the platform model and the presentation model describes how the constraints posed by various platforms will influence the visual UI.

3.2 Model-based techniques

Every technique involves creating mappings between the various model components. These mapping can be interpreted to produce a customized UI for the relevant device and context of use.

3.2.1 Handling platform constraints

We will use the *display resolution* as an example of a platform constraint. It is also very useful for our purpose (layout management), and often perceived as the most difficult constraint to deal with. Other constraints such as bandwidth and interaction capabilities can be dealed with in a similar manner.

The screen resolution is expressed in pixels (e.g. 1024x768). It should not be confused with the screen surface area: two displays having different sized surfaces can share the same resolution. Many mobile systems have small-size, low-resolution displays. An optimal layout for a desktop display may be simply impossible to render on a PDA. There are three main influences on the amount of display size required by a certain user interface [EVP01]: *size of each interactor*, *layout of interactors within a window* and *allocation of interactors among several windows.*

Interactor size

Individual interactors (or interaction objects) can be *shrunk* or *replaced.* When shrinking an interactor, one must of course take usability into account. For example, experiments have shown that an icon may not be smaller than 8 by 6 pixels, otherwise it becomes illegible. Not every interactor can be significantly shrunk. It is then more suitable to replace that interactor by a smaller alternative.

Although reducing the size of interactors provides an immediate improvement, often a more global solution is necessary: we must also examine window layout and allocation of interactors among windows.

Selecting the appropriate presentation structure

The layout of interactors within a windows and allocation of interactors among several windows together form the *presentation structure*. We want to select the appropriate presentation structure, given the amount of available screen space.

Therefore we would need a set of alternative presentation structures, either manually created by an interface designer, or generated by the system. Then we would have to create mappings between each platform and its appropriate presentation structure.

However, this can also be done automatically. The platform model already includes the screen resolution of each platform. We could also represent the amount of screen space required by each presentation structure in the presentation model. This knowledge can then be used to allow a *mediator agent* to select the appropriate presentation structure for each platform. The mediator would choose a presentation structure whose required amount of screen space falls just under the maximum screen resolution.

The dynamic solution is clearly preferable. The screen resolution may change at runtime. Only the amount of required screen space has to be specfied. The mediator will then automatically select the appropriate presentation structure.

Generating the appropriate presentation structure

It is still up to the designer to specify the alternative presentation structures. It would be better though if the correct structure could be generated by the system, given a set of user-defined constraints.

Therefore we need additional abstractions in our presentation model [EVP01]:

- *Logical Windows* (*LW*): any grouping of AIOs—a physical window, a dialog box, a panel, etc.
- *Presentation Unit* (*PU*): a complete presentation environment required for carrying out a certain, interactive task. Each PU consists of several LWs, which may be displayed simultaneously, alternatively or a combination thereof. Each PU has at least one window called the *main window*, from which the user can navigate to other windows.

These abstractions can be structurized in a hierarchy as depicted in Figure 3.2. We can use this hierarchy to generate specific, platform-optimized presentation structures from a original, platform-neutral presentation structure. Two strategies are available to do so:

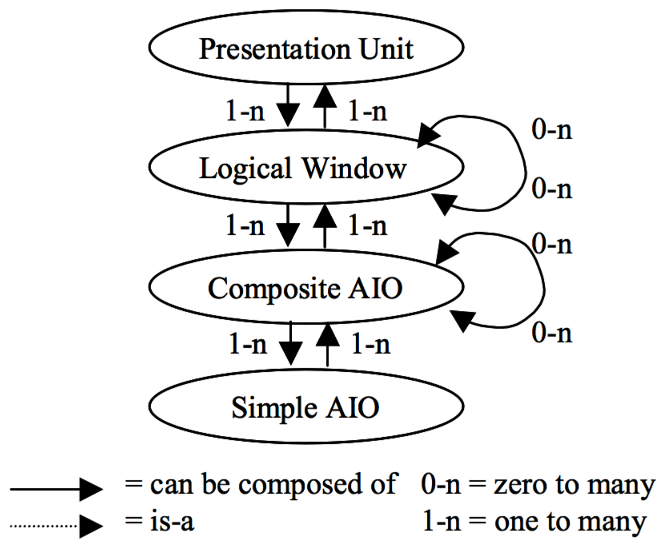


Figure 3.2: Hierarchy of presentation concepts (extracted from [EVP01])

- *Rearranging LWs within a PU*: the contents of the initial LWs are redistributed into new LWs within a single PU. An LW can be ungrouped into several smaller LWs; a number of LWs can be grouped into a single LW; and AIOs can be moved from one LW to another.

- *Rearranging AIOs within a LW*: the contents of an initial LW are redistributed into new AIOs, composite or simple. AIOs are thus replaced by other AIOs that offer the same functionality.

Spatial constraints can assist in the rearrange process, ensuring the hierarchy respects the platform constraints. For more information on spatial constraints we refer to Section 4.2.1.

3.2.2 Focusing on contexts of use

Often the user will not want to accomplish the same set of tasks on each device. Some devices are especially suited for a specific subset of the overall task model. For example, tracking the user's current location on a map would not be suited for a desktop workstation, because a desktop pc mostly stays at the same location, but this task would be very suited for a PDA.

It is possible to optimize the UI for each device by creating mappings between platforms (or classes of platforms) and tasks (or sets of tasks). There are several ways in which a presentation model can be optimized for the performance of a specific subset of tasks. A common principle is: *important tasks should be represented by AIOs that are easily accessible.*

3.3 Conclusion

In this chapter, we discussed model-based user interface development, and a couple of model-based techniques which simplify the design of multi-device user interfaces. Many approaches concerning automatic layout management center around a model-based methodology. This chapter will thus prove useful for better understanding the other material presented in this thesis.

Chapter 4

Automated Layout Management

Contents

Layout refers both to the process of determining the sizes and positions of the visual objects that are part of an information presentation, and to the result of that process. A presentation is material that is intended to be viewed and manipulated by people (e.g. UIs, webpages, newspapers, etc.). *Automated layout* refers to the use of a computer program to automate either all or part of the layout process. This field lies on the crossroads between artificial intelligence and human computer interaction [LF01].

A presentation's layout can have a significant impact on how well it communicates information to and obtains information from those who interact with it [LF01]. For example, the relations between individual objects can be made clear by the layout. A well laid out presentation can visually guide the viewer to infer correct relationships about its objects and help the user to accomplish tasks quickly and correctly, increasing the efficiency of the presentation. Thus, layout determines in part the time, efort and accurarcy with which tasks can be accomplished.

Most layouts created today are done manually: they are almost completely created from start to finish by a human graphic designer. These people spend years learning the craft of designing effective layouts, and may need hours or even days to create a single screen of a presentation. Clearly, designing presentations by hand is too expensive and too slow, as we also discussed in Section 1.1.

4.1 Simple techniques

Most modern user interface toolkits support *layout managers*, mainly to assist the interface designer in creating a hierarchy of nested objects and containers, without having to specify the absolute position and size of each object [LF01]. They allow the programmer to specify that a button should be added, or added to a part of the window. Optionally, additional numeric constraints can be specified. A layout manager makes it possible to create layouts that can adapt to a certain degree when their container changes in size.

A layout manager has a set of constraints provided by a simple, built-in *layout policy* and parameters specified by the programmer. These are used to choose positions and sizes at runtime for the objects it controls. Typical policies include strict *row* (horizontal) or *column* (vertical) layout; *row-major* or *column-major* layout in which objects wrap to the next row or column to avoid exceeding the managed container's bounds; *border layout* where objects can be specified to reside in one of the managed container's borders (north, south, east, west) or in the center of the container; and *grid layout* in which objects reside at one position in a programmer-specified 2D grid. Programmer-specified parameters include preferred, minimum and maximum *widths* and *heights*; and *spacing*, both between objects and between the objects and the managed container. Managed containers can be nested inside other managed containers in order to create more complex layouts. They are treated just like other objects by their encompassing container.

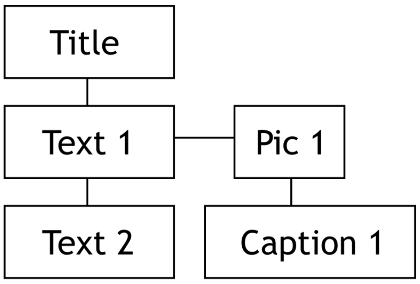
The programmer thus designs a layout as a hierarchy of managed containers, further constrained by programmer-specified paramters. A layout manager *does not actually design a layout*, but rather *instantiates* a layout at run-time from the structure and parameters specified by the programmer. Creating a simple layout is very easy for the programmer. Complex layouts are possible, but tedious and difficult, especially if they need to behave robustly when resized. The popularity of layout managers stems mostly from the ease of implementing the managers themselves and the relative ease with which programmers can specify simple layouts. Additionally, because a final layout is only determined at run-time, the system works fairly well under changing conditions.

Word-processing and presentation systems intended primarly for sequential presentations (e.g. Microsoft Word, Powerpoint), provide a set of predefined *style templates* (and the ability to create new ones). These systems are usually simpler than layout managers [LF01].

The TeX typesetting system uses a more sophisticated layout algorithm [Knu84]. It decomposes a page into a hierarchy of cells (also called boxes). At the smallest level we have character glyphs, which can be combined into words. Words can be grouped in lines, which can again be combined in paragraphs and so on. So in fact, a TeX document is just a combination of vertical and horizontal layouts (also called *vboxes* and *hboxes*), similar to the row and column layout commonly used by layout managers. A set of lines has a vertical layout, while each line has a horizontal layout of words. In order to create correct spacing between boxes (e.g. spaces between words), TeX uses a concept called *glue*. Glue has three attributes: its *natural space*, its *ability to grow* and its *ability to shrink*. Glue is placed between boxes and grows or shrinks in order to comply to the preferred size of its parent container. This is similar to so-called *springs* in layout managers, which fulfill the same task. TeX's line-breaking algorithm is very advanced and takes a lot of details into account, such as general typesetting rules.

4.2 Constraint satisfaction

Since the very beginning of graphical user interfaces, researchers explored constraint-based methods for addressing the layout problem. The vast majority of research in automated layout to date, is still centered around this approach. Representing a layout as a set of constraints is very intuitive, as depicted in Figure 4.1. A constraint-based automated layout system takes



Title ABOVE Text 1	Title DESCRIBES Text 1
Title FULL PAGE WIDTH	Title IS IMPORTANT
Text 1 LEFT-OF Pic 1	Text 1 REFERENCES Pic 1
Caption 1 BELOW Pic 1	Caption 1 DESCRIBES Pic 1
Text 2 BELOW Text 1	Text 2 FOLLOWS Text 1

Figure 4.1: Spatial and abstract constraints describing the relations between a number of components

as input a constraint network and generates a set of positions and sizes for each of the components in the network.

Equation 4.1 is an example of a constraint system. Figure 4.2 depicts the corresponding one-way constraint graph. One-way constraints have only one output variable, all other variables are read-only. The algorithm for solving such a system is very simple: all it has to do is decide which constraints to solve and in what order. Once this topology is constructed, it can easily be maintained, allowing for incremental solving. In Figure 4.2, when x_1 changes, the topology is traversed and new values for each variable are calculated. New values are found (in order) for $x2$, m and r. Constructing the topology is also known as the *planning phase*, while calculating new values for each variable is called the *execution phase* [Bad00]. For more information about the terminology of constraint solving we refer to Section 5.2.

$$\begin{array}{lrcl}
C_1: & m & = & \frac{(x_1 + x_2)}{2} \\
C_2: & x_1 & = & \texttt{pointer position} \\
C_3: & x_2 & = & x_1 + 6 \\
C_4: & r & = & m^2
\end{array} \tag{4.1}$$

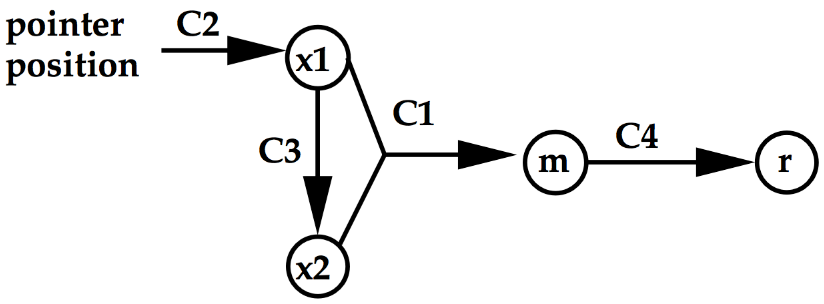
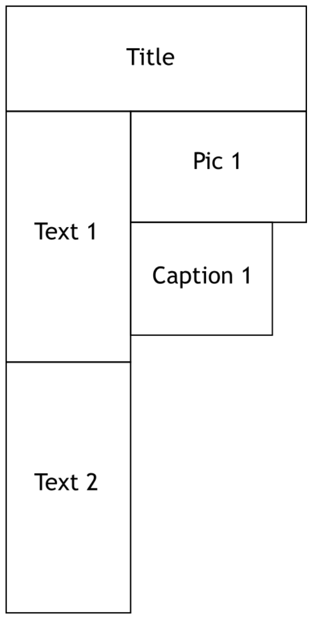


Figure 4.2: One way (directed) constraint graph for Equation 4.1 (extracted from [Bad00])

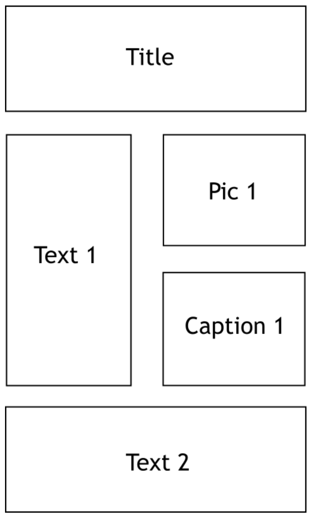
4.2.1 Types of constraints

Most constraints can be classified as either *abstract* or *spatial.* Abstract constraints describe a high-level relationship between two components (e.g. *caption DESCRIBES picture*), while spatial constraints enforce position or size restrictions on the components (e.g. *caption BELOW picture*). Spatial constraints can be fed directly into a constraint solver, while abstract constraints must first be reduced to spatial constraints. This reduction can often be quite challenging.

Most systems use a combination of both abstract and spatial constraints. Spatial constraints alone are known to generate efficient layouts. On the contrary, abstract constraints alone are not enough to create visually pleasing, esthetic layouts. An example can be seen in Figure 4.3.



(a) A simple layout that can be generated by a system that only considers abstract relationships between components



(b) A layout of the same components where additional spatial constraints are specified so that each component completely fills a regular grid and leave margins between the elements

Figure 4.3: Abstract versus spatial constraints

Abstract constraints

As we said earlier, abstract constraints express high-level relationships between the different components, to be placed in a layout. They are sufficiently high-level that content authors can easily specify them. Futhermore, abstract constraints can be specified without much technical or artistic knowledge.

The conversion from abstract to spatial constraints is done by a *translation component*, before the constraints are passed to the numeric constraint solver. This component can choose *any* set of spatial constraints to represent the abstract constraint. Although one might conclude that the abstract constraint *caption DESCRIBES picture* implicates that *caption* is placed below to *picture*, that is not always the case. Multiple solutions are possible.

Spatial constraints

Spatial constraints directly represent the geometric structure of the presentation. For example, one could require that each component would occupy a space equal to or an integral multiple of a certain size.

There are a number of situations where spatial constraints are useful. They can improve upon the visual quality and esthetics of the presentation, by expressing simple legibility rules (e.g. text cannot be shrunk to a height smaller than 6 pixels) and style guidelines (e.g. captions are placed beneath their associated figures). Earlier, we discussed Figure 4.3(b), which shows the effect of adding spatial constraints on the esthetics of the presentation.

It is possible to use concepts from graphic design to create a legible and pleasing output. These systems often enforce the presentation to conform to a *grid system*. In a grid system, every screen or page of the presentation is divided into upright rectangles. Each object must occupy one or more complete rectangles. A complication with this system is that a component may need to be *cropped*, *padded with a border* or *non-uniformly scaled* (which means its aspect ratio will change). This is because uniform scaling may not be sufficient for the object to occupy an integral number of rectangles.

Automated layout systems for well-defined environments, such as network diagrams, often employ spatial constraints exclusively. Abstract constraints can be used for formatting (e.g. important cities use a bigger icon and a bigger font for their name), but are generally not utilized for layout directly.

Some systems allow the specification of *abstract data constraints*, separately from spatial constraints. This is mainly useful for maintaining a separation between content authors and layout experts.

4.2.2 Expressing constraints

One could define a formal grammar for representing constraints. Using this approach, we would be able to leverage a rich body of existing research for manipulating and parsing constraints. Such grammar may also be difficult to use, if they try to be overly general. Additionally, a very complex solver would be needed to solve the constraint systems expressed with such a system.

Another powerful approach is the use of *Boolean predicates*. This method is quite contrary to the previous one. It tries to avoid the problems caused by a very expressive grammar, by limiting the space of what can be expressed. It also eases the process of solving the set of constraints, because the input now requires almost no translation before being fed to the solver.

4.2.3 Obtaining constraints

For our purpose, there are three possible approaches of obtaining constraints. The most obvious one is where the user declaratively specifies the constraints. While this is doable for simple examples, it becomes quite tedious when dealing with complex layouts. Another method is a tool that allows the user to interactively specify the constraints and preview the results. A similar tool has been created for Dygimes [CLV$^+$03]. Finally, it is also possible to use predefined sets of constraints, similar to the approach taken by layout managers described in Section 4.1.

4.2.4 Constraint solvers

A constraint solver is a component which calculates a solution to a constraint system. For a more thorough overview of constraint solvers and the associated difficulties in developing one, we refer to Chapter 5.

4.3 Evaluation of a layout

In the field of layout, a *good* layout, can refer to the *usability* (e.g. whether the user's tasks can be accomplished efficiently), and the *esthetics of the presentation*. Most techniques focus around *heuristic inferences* or *quantitative metrics*. Heuristics are more flexibile, while metrics can be more easily incorporated into computer systems. It is for example possible to measure the amount of mouse movement a user needs to accomplish a task associated with the presentation.

SUPPLE treats automated layout management as an optimization problem [GW04]. The system tries to minimize the estimated effort for the user's expected interface actions while still adhering to the device's constraints. It is innovative in that it takes into account a user- and device-specific cost function. An essential component of the cost function are user traces. SUPPLE uses these metrics to dynamically adapt the layout at runtime. Suppose a user has to click two distant buttons very often due to some external factor, SUPPLE could recalculate the layout, and place these buttons closer together. It is however uncertain which exact changes will be incorporated by the system, and whether these will always improve the usability. It is possible SUPPLE's rearrangements and remappings break the user's mental model of the interface. Furthermore, it is not clear how the designer can influence SUPPLE's behavior.

4.4 Conclusion

We gave a broad overview of existing work on automated layout management, and further deepened our knowledge in this field. We introduced simple techniques used in modern UI toolkits, along with constraint-based techniques. Finally, we briefly discussed computer-based mechanisms for evaluating a layout.

Part II

Research

Chapter 5

Constraint solvers

Contents

5.1 Introduction

5.1.1 Constraints

A constraint restricts possibilities. We use constraints daily when we describe things. For example, one could describe a basketball as being round and orange. When someone thinks about this description, he or she automatically rules out other geometric forms and other colours. We effectively restrict the set of objects that correspond to the description, by requiring them to be round and orange.

Although this is a intuitive description of a basketball, it is not complete. In the winter (when there's snow on the field), soccer footballs are orange too. This kind of football also corresponds to our description of a basketball. We could limit the possibilities even more by adding additional constraints (such as requiring the ball to be approved by the FIBA [1]).

Eventually we would have created a list of constraints, that describe our object sufficiently. A list of constraints is called a *constraint system.*

We can conclude that constraints are a very natural way for describing things. It is therefore also a good choice for describing a graphical layout. One could for example specify that the layout consists of three columns, thereby limiting the position of widgets or widget groups to these three columns. One would then further refine the constraint system, until the generated layout represents what the designer had in mind.

In essence, a constraint is a relation we would like to maintain. The key advantage of constraints is the separation of *what* relation we specify from *how* we actually maintain it [Bad00]. The user can declaratively state a relation that they want to be satisfied, instead of writing a procedure to maintain the relation himself [FBMB90]. Constraints therefore fit in seamlessly with a declarative description of a user interface.

In a more mathematical sense, a constraint is a mathematical relation: a *Cartesian product* over a number of domains [Bad00]. A relation over the domains $X_1, X_2, ..., X_n$ can be formally described as follows:

$$R = (X_1, ..., X_n, G(R))$$

where:

$$G(R) \subseteq X \times Y$$

$G(R)$ is thus a subset of the Cartesian product of X and Y.

Constraints don't have to be spatial. The domain can be anything. However, in layout management, abstract constraints are generally converted to spatial constraints at a later stage [LCC03]. Therefore we only consider spatial constraints from now on.

5.1.2 Applicability

There is a constant trade-off between *expressibility* and *performance* [Bad00]. We must be able to express interesting, non-trivial relationships, while still supporting interactive applications.

Maximum expressiveness can be obtained from arbitrary constraints, but unfortunately solving such systems is undecidable. On the other hand, there are many systems that provide excellent performance, but are limited in expressiveness.

The interface designer should be able to *understand* and *predict* the resulting solution to a certain system of constraints. By doing so, he can figure out which constraints to add, edit or remove in order to acquire the desired result.

These problems should be taken into consideration when choosing a constraint solving algorithm, and a notation for constraints.

[1] International Basketball Federation

5.1.3 Constraint solvers

A constraint solver is an application that takes a constraint system as input and produces a set of values for each of the featured variables as output.

5.2 Basic principles and techniques

5.2.1 Local versus global propagation

The main limitation of local propagation solvers is that they only examine individual constraints in isolation. When cycles occur in the constraint graph, more advanced algorithms must be called upon to handle the more complex relations [Bad00].

5.2.2 Read-only annotations

Variables can be marked as read-only in a certain constraint, which means the solver cannot change their value in order to satisfy the constraint. This annotation is specific to a particular constraint though. Other constraints may change the values of these variables.

5.2.3 One- versus multi-way constraints

One-way constraints

A one-way constraint has a single method for maintaining it. This method calculates a new value for a single output variable. Easy to satisfy when there are no circularities.

Multi-way constraints

A multi-way constraint has several methods for maintaining the constraint. In general a method for calculating a value for each of the variables it constrains, in terms of the values of the other variables.

Advantages over one-way constraints

- more general
- every one-way constraint can be represented by a multi-way constraint with all but one of its variables marked as read-only
- more powerful
- the choice of method to use can solve problems that one-way constraints cannot

- clearer and more uniform way to specify relationships. It's more intuitive to represent a multi-directional relationship with multi-way constraints instead of multiple one-way constraints

Disadvantages

- complexity. It's more difficult to predict and control the behavior of a network of multi-way constraints.

Conclusion

The greater expressive power of multi-way constraints justifies the additional complexity for many applications.

5.2.4 Constraint hierarchies

If multi-way constraints are used or if there are cycles in the constraint graph, there may be many ways to satisfy the constraints. We don't want to specify declaratively what to change when perturbing a constraint system however.

A constraint hierarchy consists of required and preferential constraints. The required constraints must hold. The system should try to satisfy the preferential constraints if possible. Preferential constraints may be defined in multiple levels of strength, each successive level more weaker than the previous one. A stay constraint is an example of a preferential constraint, indicating that a variable's value should not be changed, if possible.

A constraint hierarchy is a set of labeled constraints. The label is the level of the constraint. Level 0 is a required constraint. Levels $1, ..., n$ are the preferential constraints.

5.2.5 Comparators

A solution to a constraint hierarchy is a valuation for all the free variables in it. We have to define a comparator for choosing the best solution: the solution which satisfies the required constraints, and additionally satifies the preferential constraints the best according to their relative strengths.

The remainder of this subsection will discuss two commonly used comparators.

Locally-predicate-better

A possible comparator is the locally-predicate-better comparator, which is used in DeltaBlue (see Section 9.1.1). It only concerns itself with whether or not a certain constraint is satisfied, rather than how nearly satisfied it is. By the locally-predicate-better comparator, it is more desirable to have a solution that satisfies all required constraints and a single *strong* constraint, rather than one that satisfies all the required constraints and twenty (or even millions of) *weak* constraints [Bad00]. The definition is as follows:

> Solution x is *locally-predicate-better* than solution y for constraint hierarchy C if there exists some level k such that for eveyr constraint c in levels C_1 through C_{k-1}, x satisfies c if and only if y satisfies c, and at level C_k, x satisfies every constraint that y does, and at least one more.

By definition, S will not contain any solutions that are worse than some other solution. However, S may contain multiple solutions, none of which is better than the others [FBMB90].

Locally-error-better

This comparator is often used for satisfying inequality constraints. It takes into account the error in satisfying a constraint. This error is 0 if and only if the constraint is satisfied, and becomes larger the further away the solution is from a satisfying one. The definition is:

> A solution x is *locally-error-better* if there is no other solution y that is better than x. Informally, y is better than x if there is some level k in the hierarchy such that the errors for all the constraints in levels 0 through $k - 1$ are exactly the same for y and x, and at level k the errors in satisfying each constraint using y are less than or equal to the errors using x, and strictly less for at least one constraint.

In general, there may be more than one locally-error-better solution to a given hierarchy [BAFB96].

5.2.6 Perturbation versus refinement model

Perturbation model

At the beginning variables have specific values associated with them that satisfy the constraints. This is useful for layout management, since widgets have a specific position. The constraint solver adjusts the values of the variables when one of the variables is changed by an outside influence, so that the constraints are again satisfied. This outside influence could be a change in screen size for example. Adding and removing constraints can be done in any order (which is again useful for our problem).

Refinement model

Variables are initially unconstrained. As constraints are added, the permissible values of the variables get refined. Variables have no unique value, which is not useful for graphical user interfaces.

5.2.7 Cycle avoidance

Redundant constraints may introduce cycles in the constraint graph. Constraint solvers that cannot handle cycles, need to call a specialized cycle solver to eliminate the cycle.

5.2.8 Conflict resolving

Another common cause for the failure of a constraint solver are conflicting constraints (e.g. $x \geq 5$ and $x \leq 3$). Constraint hierarchies already provide a good solution to resolve conflicts when the conflicting constraints have different strengths. Weaker constraints can be dropped in favor of the stronger ones. However, a system with two conflicting required constraints for example, cannot be satisfied. An error should be given in this case.

Chapter 6

Flexible Presentations

Contents

6.1 Self-adaptable widget mappings

A more intelligent layout management approach could use *remappings* to dynamically switch between several CIOs starting from a certain AIO. An abstract *range* widget could for example be represented as a slider when there is enough screen size available, and switch to a spinbox when the area becomes too small.

In the remainder of this section we discuss a couple of techniques for realizing remappings. Additionally, we provide a detailed description of how remappings could be implemented in UIML (using UIML's template element).

6.1.1 Graceful degradation

Graceful degradation is a technique to guarantee maximum continuity between platform-specific versions of a high-level user interface [FV04].

Users expect to be able to employ their existing knowledge when using a service on another platform. This implies that the transition between different platform versions has to be very smooth. In other words, we have to guarantee *continuity* between these different versions.

Graceful degradation centers the design effort on a single *source interface* which is designed for the least constrained platform. Then a set of transformation rules are applied to this source interface in order to produce specific interfaces for the other platforms. For example, if we want to provide interfaces for a cellphone, a PDA and a desktop PC, we would use the desktop interface to generate versions for the PDA and cellphone.

We call the process a *degradation* because we produce more constrained interfaces. The degradation is *graceful* because we strive to guarantee continuity.

Graceful degradation is tailored towards *model-based user interface development* (*MBUID*). There are rules for several models in the development process. We refer to Chapter 3 for more information about MBUID.

Rules at the CUI level

There are two important kinds of rules for the *Concrete User Interface level*: rules that transform the layout relationships between graphical objects, and rules that modify the number and nature of the graphical objects.

Transformations of layout relationships There are three different types of rules for layout relationships: *resizing rules*, that modify the dimensions of a graphical object; *reorientation rules*, that modify the orientation of an object without changing its size or position; and *moving rules* that modify the localization of a graphical object. Resizing rules must keep in mind the minimum width and height to which a graphical object can be shrunk and if we want to preserve the aspect ratio. Of course, we have to take into account the limits of human perception (the user must be able to read and distinguish each graphical object). Reorientation rules are useful when switching from landscape to portrait mode or vice versa. They can only be applied to a small set of objects (e.g. labels). Moving rules can be used when components don't fit in a one dimension and there is space left in the other dimension; when we wan to avoid scrolling in one dimension (horizontal scrolling for example); and when some ergonomic rule or convention on the target platform has to be respected.

Transformations of graphical objects Object transformations can take three different forms: *modification*, *substitution* and *removal.* Modification rules act upon the appearance of the graphical object (e.g. represent an emergency by a red background on a workstation and by a flickering screen on a cellphone). They modify the physical rendering of a semantic feature. Substitution rules replace an interactor (i.e. an interactive graphical object) by an alternate interactor that enables the same type of functionaltities.

Substitution rules are mainly used when a certain interactor is not available on the target platform, or when the interactor requires a too large screen area. There are three types of substitutions:

- *simple substitution* ($1 \rightarrow 1$): interactor X on the source platform is replaced by interactor Y on the target platform.
- *regrouping* ($N \rightarrow 1$): a set of interactors on the source platform is replaced by a single interactor on the target platform.
- *splitting* ($1 \rightarrow N$): a single interactor on the source platform is is replaced by a set of interactors on the target platform.

The last type of graceful degradation rule for graphical objects is the removal rule. One could for example delete pictures on a cellphone, because of space constraints.

Rules at the AUI level

The *Abstract User Interface level* defines the distribution of interactive tasks among the presentation units. A *presentation unit* groups logically linked low level tasks that are to be achieved in the same presentation (window, panel, etc.).

There are two useful graceful degradation rules here: *splitting rules* and *reorganization of tasks within the same presentation unit.* Splitting rules split a presentation unit in the source interface into two or more presentation units for the specific interface. Reorganization of tasks occurs when the frequency of certain tasks on the target platform changes compared to the frequency on the source platform. For example, the consultation of an address book would be more frequent on a cellphone than on a desktop PC.

Rules at the Task and Concepts level

At the Tasks and Concepts level, a broad range of graceful degradation rules can be applied. We can apply rules to general functionalities (high level tasks), to the procedures the user must follow in order to complete the general functionalities (low level tasks), to the temporal ordering between tasks, and finally to the concepts.

Transformations of general functionalities Two types of rules apply here: *high level task deletion* and *high level task insertion.* Tasks may be removed when they are inappropriate for or impossible on the target platform (e.g. manipulating complex graphics on a cellphone). Tasks may be added for the same reason (changes of interaction capabilities or changes of the typical context of use on the target platform).

Transformations of procedures As with high level tasks, there are again two types of transformations here: *subtask deletion* and *subtask insertion*. Subtasks can be deleted because they are unnecessary on the new platform (e.g. entering the user's location on a platform with a GPS system), or because they require too many resources (e.g. one can still book theatre tickets but not the subtask of viewing the free seats in a picture of the hall). Insertion of a subtask can occur because the target platform does not permit several tasks to be executed at the same time (e.g. it is not possible to edit several information items simultaneously on a cellphone), or because additional navigation tasks are needed when a task needs to be split.

Transformations of temporal ordening Tasks can be *sequentialized* when the style of interaction changes (e.g. from a graphical interface to a speech-based interface). Conversely, sequential tasks can of course also *become concurrent* when the style of interaction changes.

Transformations of concept level Graceful degradation can change the way some concepts are viewed: information can be summarized or cut, some attributes can be masked, alternative shorter labels can be chosen, . . .

Graceful degradation rules and continuity

Of course not all rules we apply on the source interface have the same impact on the continuity within the multiplatform system. Intuitively, low level rules are expected to generate less discontinuity than rules applied at a higher level.

A priority ordering between different rules is proposed. Rules with a high priority generate less discontinuity than rules with a lower priority, and should thus be tried first when adapting the source interface to the target platform. Let's have a look at a list of rules, starting with the rules with the highest priority to the rules with the lowest priority:

- Layout transformation
  - resizing objects
  - reorienting objects
  - moving objects
- Graphical object transformation
  - modification of appearance
  - simple substition
    - with shape preserved

    * without change of shape
  - regrouping or splitting
  - interactor removal

- Task reorganization
  - task reorganization within the same presentation unit
  - task reorganization across several presentation units (splitting rules)
- Transformations at the tasks and concepts level
  - temporal ordering transformations
  - concept level transformations and precedure transformations
  - general functionality transformations

For transformations at the tasks and concepts level, a higher priority is given to temporal ordering transformation rules that preserve the displayed information and the available tasks. Concept level transformations and procedure level transformations introduce more discontinuity. General functionality transformations significantly modify a system, and thus are given the lowest priority.

6.1.2 Comets

Comets (*COntext of use Mouldable widgETs*) are a new generation of widgets, that can adapt to the context of use. A comet is an *introspective* widget that is able to self-adapt (or can be adapted by a tier-component) to some context of use, or can be dynamically discarded (versus recruited) when it is unable (versus able) to cover the current context of use. To do so, a comet publishes the quality in use it guarantees, the user tasks and the domain concepts that it is able to support, as well as the extent to which it supports adaptation [CCODD04]. Let us start with some definitions.

By *context of use* we mean a triple <user, platform, environment>. The user denotes the person who is intended to use the interactive system. The platform corresponds with the same definition of computing platform we supplied in Section 1.1: a computing platform is an architecture in hardware and/or software that allows software to run. Finally, the environment refers to the physical and social conditions in which the interaction takes place.

To master the diversity of contexts of use, the plasticity property has been introduced. *Plasticity* is defined as the ability of an interactive system to withstand variations of context of use while preserving quality in use. Basically, plasticity refers to the adaptation to context of use that preserves the user's needs and abilities. *Quality in use* is defined by the ISO standards committee [703]. It refers not only to usability but is based on both internal and external properties, including usability, which is depicted in Figure 6.1. Plasticity is thus not limited to the user interface alone, but may also impact

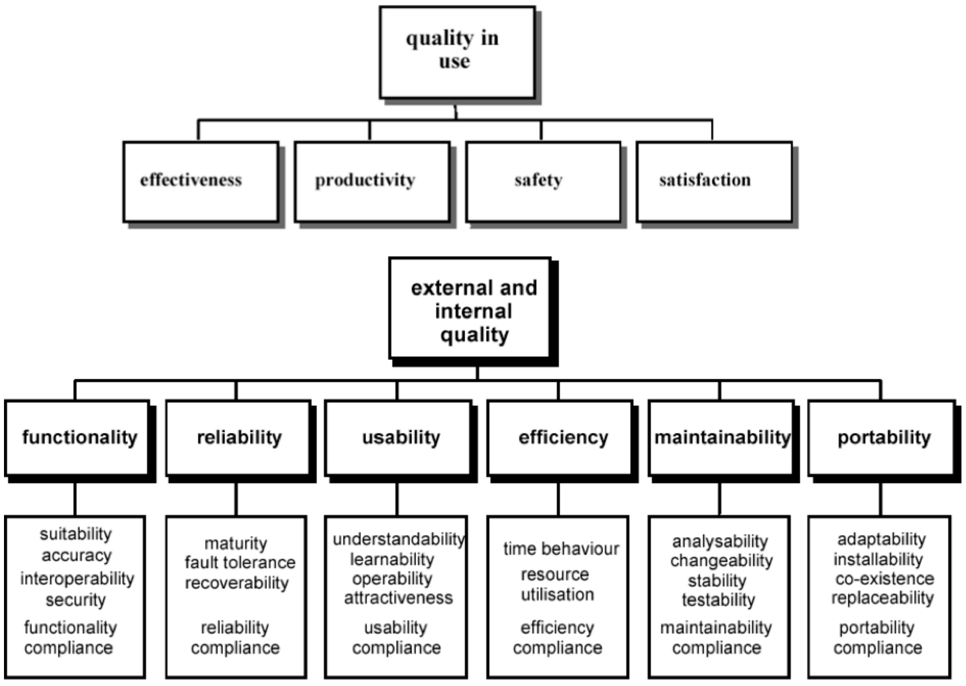

quality in use
effectiveness
productivity
safety
satisfaction
external and internal quality
functionality
reliability
usability
efficiency
maintainability
portability
suitability accuracy interoperability security functionality compliance
maturity fault tolerance recoverability reliability compliance
understandability learnability operability attractiveness usability compliance
time behaviour resource utilisation efficiency compliance
analysability changeability stability testability maintainability compliance
adaptability installability co-existence replaceability portability compliance

Figure 6.1: Quality models for quality in use and internal and external qualities (extracted from [CCODD04])

the functional core. A typical example is services discovery. Because Bob is now in a place that makes a new service available, the service now appears on his PDA. Thus, plasticity refers to the capacity of *interactive system*, not only a UI, to adapt to the context of use. An interactive system is said to be "plastic for a set of properties and a set of contexts of use" if it is able to guarantee these properties whilst adapting to another context of use. Plasticity is thus not an absolute property, it is specified and evaluated against a set of relevant properties.

6.1.3 Remappings in UIML

As we discussed earlier we would like to use *remapping* to dynamically switch between different CIOs. We would like an abstract class (*d-class*) to have a couple of alternative mappings. This can be realized by using the *template* element (for more information, see Section 2.2).

In fact, all we have to do is change the *source* attribute of the abstract d-class in the vocabulary in order to change it from one specific CIO to another. Of course we have to make sure that both CIOs support the same set of properties. We can thus state that UIML has fairly good support for remapping widgets.

When do we switch between CIOs? An easy solution is to let the layout manager decide. Unfortunately, this requires metadata about the interaction objects. Each CIO should be given a weight, including amongst others its relative dimensions compared to other CIOs deriving from the same AIO. This weight could then be used to decide which CIO to choose in particular situation.

For graceful degradation and comets it would be beneficial to provide a set of rules that specify when a switch should occur. This could then be passed to the UIML renderer or layout manager.

6.1.4 Visual techniques

[VG94] describes *visual techniques* exported from the area of visual design, and a set of guidelines for effectively applying these visual techniques. A visual technique relies on a commonly accepted visual principle to suggest the arrangement of the layout components. Examples of these techniques are *symmetry*, *realism*, *grouping*, etc.

However, it seems rather difficult to integrate these techniques in an automated layout management system. Physical visual techniques are easily computed, but there are also techniques that are difficult to evaluate using a machine (e.g. photographic visual techniques). There have been some systems (e.g. GRIDS) that incorporated a subset of the visual techniques though [VG94].

6.1.5 Conclusion

Graceful degradation and comets are complementary techniques to our notion of remappings. We can use the graceful degradation rules for defining when and which remapping to apply. Of course, only the rules at the concrete and abstract user interface level are suitable for our purpose. Comets are more general than our remappings. They react to different changes in context of use, while remappings only take into account the layout.

Dynamically remapping and rearranging widgets would certainly increase the degree of *plasticity*. However, a difficult problem is how to decide when remappings or rearrangements should occur. It would be possible to let the layout manager handle the remappings. This would require metadata about each CIO though.

Visual techniques are again a set of rules we could take into account when generating a layout. It seems difficult to effectively integrate these techniques into the automated layout management system though.

6.2 Patterns

Simply stated, a *pattern* is a proven solution to a recurring design problem [Bor01]. It pays special attention to the context in which it is applicable and to the positive and negative consequences of its application.

The architect Christopher Alexander introduced the idea of patterns in the early 1970s [Ale79]. His original definition stated:

> Each pattern is a three-part rule, which expresses a relation between a certain context, a problem and a solution.

Alexander presented patterns as a way to quickly solve architectural problems that occurred over and over again in a particular environment, by providing commonly accepted solutions. *Design patterns* solve recurring problems in object-oriented development [GHJV95]. They became very popular in the software engineering community and are probably the most well-known example of patterns in software development.

Over the last years, there has been a lot of momentum around *HCI patterns*. A HCI pattern is a named, reusable solution to a recurrent problem in a particular context of use. At the moment there exist a number of HCI pattern catalogues, carrying a welth of reusable design knowledge. HCI patterns are applicable to different levels of abstraction, and can in fact be used to drive the entire UI design process [Bor01]. Because patterns exist on different levels of abstraction, they can be used to migrate a user interface from one platform to another. The same patterns can still be applied, but are converted to a more suitable, platform-specific implementation. Pattern descriptions should include advice for selecting the most suitable implementation for a given context.

6.2.1 Assisting in user interface migration

Two approaches can be used when migrating UIs from one platform to another [JSES04]:

- *Reengineering* reuses the original system with the goal of maintaining it and adapting it to required changes
- *Redesign* is a simplified version of reengineering, and can be more practical in certain contexts

Redesign takes advantage of HCI patterns by incorporating *pattern mapping*. The patterns of the existing UI are transformed or replaced in order to redesign and reimplement the interface. Patterns include information about design solutions and context of use, causing platform capabilities and constraints to be automatically addressed in the transformed patterns.

Reengineering is more complex. It consists of three phases:

1. *Reverse engineering phase*: uses a pattern extraction and abstraction phase to create a platform-independent UI model

2. *Transformation phase*: analyzes patterns and designs in the platform-independent UI model and transforms them if appropriate.

3. *Forward engineering phase*: first instantiates the UI to different platforms based on constraints and capabilities, and later implements the patterns on different platforms

The reverse engineering phase tries to extract patterns, and abstract these in high level design strategies (e.g. query-based navigation). In the end a *platform-independent UI model* is constructed. The transformation phase transforms the abstract model in a restructured abstract model, taking new design requirements (e.g. new user requirements or task-based changes) into account. The forward engineering phase first of all instantiates the platform-independent UI model to target devices based on their constraints and abilities. It is only here that platform-specific decisions are made. Different presentation components may apply for each pattern and design strategy. Other patterns may be simplified according to the limitations of the target device. The second step of forward engineering is pattern implementation, which actually applies and combines the instantiated patterns.

The main disadvantage of redesign is that we have to repeat the same exercise for each platform [JSES04]. Reengineering can easily support multiple target devices due to its creation of a platform-independent UI model. Furthermore it increases coherence between each platform. The platform-independent UI model facilitates future maintenance, since we only have to change the model, instead of each platform-specific version. The reverse engineering phase encourages re-evaluation of the requirements for the system, resulting in improved usability. Reengineering is time-consuming and complex though, and requires a lot of information about the patterns (e.g. a pattern taxonomy, implementation strategies for each pattern, depending on the platform used, etc.).

6.2.2 Layout Patterns

Chapter 4 discussed that it is difficult and tedious to design an efficient and visually pleasing user interface by hand. Automated layout management, although greatly improving on speed compared to designing a layout by hand, is not guaranteed to produce satisfying results. Human interference is usually still needed. There are many recurring tasks in user interfaces (e.g. providing a username and password). For each task, a suitable layout must be determined. It would thus be beneficial to provide *layout patterns* for these tasks. Layout patterns could also be more general (e.g. a two-column layout). A pattern should always include a detailed description of its applicability, allowing the designer to pick a suitable pattern for his problem.

[LCA05] presents an approach for multi-device user interface development, based on task-centered design. Tasks are related to user interface building blocks. *Building blocks* correspond to the *presentation units*, which we discussed in Section 3.2.1. They specify the interface in an abstract way. This method uses UIML to describe a building block, because of UIML's good support for mapping AIOs to CIOs through its vocabulary. From the task specification, a dialog model is generated.

Layout patterns are used as guidelines for relating different building blocks to each other when merged in an integrated presentation. Places in the layout pattern are filled according to the interaction or output type of the task that is related to the building block to be placed in the layout. Possible interaction and output types are *selection*, *control*, *editing*, *text*, *feedback*, etc. A lot of flexibility is achieved by combining layout patterns with the other design artifacts suitable for model-based user interface development [LCA05].

6.2.3 Conclusion

We can conclude that HCI patterns, and more specificly layout patterns, are an interesting tool for multi-platform user interface development. They must be supported in the presentation model however. It is thus certainly useful to include support for layout patterns in our implementation.

Part III

Development

Chapter 7

Uiml.net

Contents

Our layout management specification for UIML was implemented in Uiml.net. This chapter gives a bit more information about this framework, in order to better place the following chapters into context.

Uiml.net is an open source UIML renderer for the .NET framework, developed at the Expertise Centre for Digital Media (EDM) [LC04]. We gave a quick overview of UIML in Section 2.

The .NET framework offers interesting capabilities for implementing a UIML renderer compared to Java, including but not limited too on-the-fly code generation, better integration with web services and support for multiple *native* widget sets, such as System.Windows.Forms, Gtk and WxWidgets. .NET is also available for mobile devices through the .NET Compact version.

Uiml.net is a portable, cross-platform application. It has been tested on the Microsoft .NET framework as well as on the Mono implementation of .NET and the .NET Compact version for mobile devices.

7.1 Architecture of the renderer

A high-level user interface description (HLUID) like UIML can either be *rendered* or *compiled.* When compiling a HLUID, program code is generated. When rendering it, there is a rendering engine that interprets the interface, and renders it appropriately. In fact this is comparable to the approach taken by a HTML rendering engine, used in web browsers.

Uiml.net uses the rendering approach, which is more complex but also more flexibile [LC04]. It supports rapid prototyping because a UIML document can be tested immediately. Again, this is comparable to the development of a HTML webpage. Furthermore, it can offer dynamic changes in the user interface and a transparent mechanism for connecting the rendered UI with the application logic.

Figure 7.1 gives a schematic overview of Uiml.net's overall design.

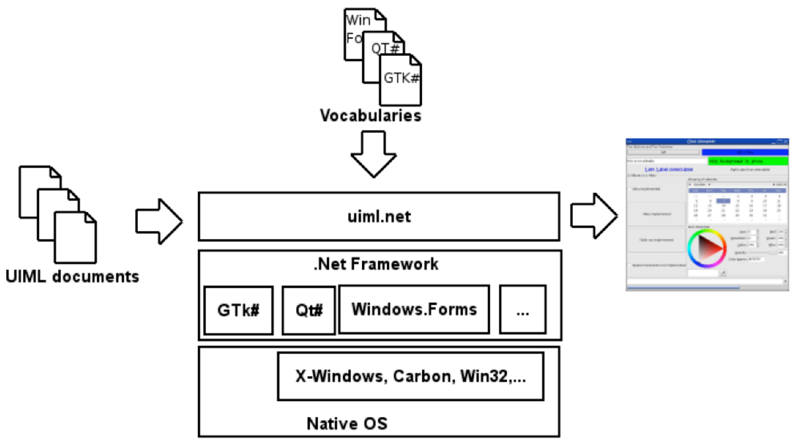


Figure 7.1: Overview of Uiml.net's architecture (extracted from [LC04])

7.1.1 Main components

Uiml.net consists of a few distinct parts:

Interface reader Before we can do anything useful, we must of course parse the UIML document. The Interface reader processes the document and represents it in appropriate datastructures. These are kept in memory during the lifespan of the user interface, allowing for dynamic changes in the style and structure of the interface. The corresponding vocabulary is stored in a searchable structure in order to easily find the concrete widget corresponding to an abstract interaction object.

Rendering Backends Uiml.net supports several *rendering backends*. A rendering backend implements a layer which knows how to generate a specific interface, starting from a UIML document and its vocabulary. Each widget set (System.Windows.Forms, Gtk#, . . .) has his own rendering backend.

System Glue The system glue connects the user interface with the application logic. It allows for subscribing to certain *events* from the UIML interface, and provide appropriate actions when these events occur. Moreover, we can describe mappings to library routines in the vocabulary, which can then be used in the UIML document itself (using the `<call>` element).

7.1.2 Dynamic design

The renderer is highly dynamic: there is a seperate layer which has specific knowledge of the widget set. The layer corresponding to the vocabulary of the UIML document, will be loaded *at runtime*. This approach is very flexible. It relies heavily on reflective programming. The rendering engine has no notion of the concrete widgets. It uses solely the vocabulary to dynamically load widgets by examining the available libraries for the appropriate classes. Reflection then allows us to create new instances of these classes, without knowing their name or structure. There is only a small part of the rendering engine that is widget set-specific: the rendering backend.

This design has several advantages

- The base rendering engine is *reusable* for every widget set (it does not explicitly create the concrete widgets)
- The vocabulary is *independent* of the renderer. When a widget set is updated, mostly only its vocabulary has to be updated.
- The renderer is more *portable*, since there is almost no widget set- or platform-specific code.

Chapter 8

UIML template element

Contents

As we described in Section 6.2, a *pattern* is a proven solution to a recurring design problem [Bor01]. Chapter 4 discussed that it is difficult and tedious to design an efficient and visually pleasing user interface by hand. Automated layout management, although greatly improving on speed compared to designing a layout by hand, is not guaranteed to produce satisfying results. There are many recurring tasks in user interfaces (e.g. providing a username and password). For each task, a suitable layout must be determined. It would thus be beneficial to provide *layout patterns* for these tasks. Layout patterns could also be more general (e.g. a two-column layout). A pattern should always include a detailed description of its applicability, allowing the designer to pick a suitable pattern for his problem.

8.1 Layout Patterns in UIML

It is possible to support layout patterns in UIML. The UIML *template* element is perfectly suited for this purpose [AP99]. We refer to Section 2 for more details. A layout pattern is built out of several parts or *UI building blocks*, and has an associated layout between these parts.

Once we define a general layout pattern, we can use it to built several specific user interfaces. It enables user interface designers to reuse existing layouts which have proven to be usable and esthetically pleasing. Each building block of the pattern is mapped to a specific hierarchy of user interface elements.

We use a constraint solver to determine the layout between building blocks, and inside each building block. This corresponds to the technique of a *local constraint solver*, discussed in Section 5.2.1.

Let us look at an example. Listing 8.1 shows a simple two-column layout, consisting of two parts: `left` and `right`.

Listing 8.1: Two-column layout pattern

```
<uiml>
  <interface>
    <structure>
      <part id="left" source="prefs.uiml#menu" />
      <part id="right" source="prefs.uiml#content" />
    </structure>
    ...
  </interface>
  ...
</uiml>
```

Listing 8.2 shows a template structure used to fill in the parts. It describes a preferences interface for a internet browser.

Listing 8.2: Preferences layout (prefs.uiml)

```
<uiml>
  <template id="menu">
    <part id="menuFrame">
      <part id="prefList" class="ListBox">
        <style>
          <property part-name="prefList" name="content">
            <constant model="list">
              <constant value="Appearance"/>
              <constant value="Privacy"/>
              <constant value="Proxy"/>
              <constant value="Navigation"/>
            </constant>
          </property>
        </style>
      </part>
    </part>
  </template>
  <template id="content">
    <part id="contentFrame">
      ...
    </part>
  </template>
</uiml>
```

The `content` will change according to which value is selected in `prefList`. We did not go into detail on the `content` part and layout constraints because that would lead us too far. It's important to note that it is very simple to reuse existing patterns this way.

The children of the parts in the layout pattern are replaced by the template element's children corresponding to the template referenced in the `source` attribute of the part. The generic `left` and `right` parts can be replaced by any part of UIML code, defined in the template.

8.2 Implementation

Recall that templates are resolved when they are *sourced* by an element. Whenever this happens, a new instance of the `Template` class is created from the `source` attribute. It is then resolved by a object implementing the `ITemplateResolver` interface, according to the `how` attribute.

Each element that can be sourced is derived from the `Sourceable` abstract class. This class implements the same `Process()` method as other elements. Derived classes call this `Process()` method before doing their own processing. Listing 8.3 shows the implementation of `Process()`.

Listing 8.3: The Sourceable#Process() method

```
public void Process()
{
    ReadAttributes();

    if (Source != null)
        ApplyTemplate(Source, How)
}
```

If a template is sourced, the `ApplyTemplate()` method gets called. Its implementation is depicted in Listing 8.4.

Listing 8.4: The Sourceable#ApplyTemplate() method

```
public void ApplyTemplate()
{
    Template t = new Template(new Uri(Source));
    ITemplateResolver tr = Template.GetResolver(How);
    tr.Resolve(t, this);
}
```

Summarizing, a new `Template` class is created from `source`. Then a new template resolver is created according to the `how` attribute, which finally resolves the template.

There are three types of template resolvers, corresponding with each possible value of the `how` attribute:

- `UnionTemplateResolver`
- `ReplaceTemplateResolver`
- `CascadeTemplateResolver`

Each of these classes is implements the interface `ITemplateResolver`, which in fact just consists of the `Resolve` method. For further details concerning the three possible resolving methods, we refer to Section 2.2.

Chapter 9

Constraint solver

Contents

In order to generate a layout for the constraints specified inside a building block, we had to integrate a constraint solver in the Uiml.net renderer. There were a few requirements, it had to:

- be suitable for interactive applications
- support required and preferential constraints
- handle cycles gracefully
- support (linear) equality and inequality constraints

Clearly the constraint solver has to be suitable for real-time, interactive applications. The solver must be able to find a new solution without disturbing the interactivity of the system. This is commonly realized by using an incremental algorithm.

Additionally, support for required as well as preferential constraints is necessary. Required or hard constraints have to be enforced, while the system will try to satisfy preferential or soft contraints if possible. The ability to specify both types, and thus form a constraint hierarchy is useful for handling conflicts. Furthermore, we would like to be able to express a preference for stability. Objects should stay where they were, unless there is a reason for them to move (this particular relation is also known as a stay constraint).

Cycles will most probably occur in our constraint network. They could have been introduced by redundant relations, or the problem could just be intrinsically cyclic. To illustrate how easily cycles are introduced, let's have a look at a simple constraint system containing a cycle [Bad00]:

$$C_1 : x + y = 6 \qquad C_2 : x - y = 2$$

The solution to this constraint system is obviously:

$$x = \{4\} \qquad y = \{2\}$$

Solvers that cannot deal with cycles would not be able to find this solution. The constraint graph is cyclic because constraint C_1 as well as C_2 relate both variables. It would be an added burden on the interface designer if he had to analyze the constraint system himself to detect cycles. A constraint solver that can handle cycles gracefully is thus preferred.

Finally, linear equality and inequality constraints, such as *above*, *left-of*, *inside*, etc. are needed for geometric relations [Bad00]. Geometric relations are of course very common in the specification of a graphical layout.

In the remainder of this chapter, we'll discuss a couple of possible candidates, explain our choice, and conclude with the actual implementation and integration in Uiml.net.

9.1 Candidate solvers

9.1.1 DeltaBlue

DeltaBlue is an incremental version of the *Blue* algorithm [FBMB90]. Blue is a multi-way local propagation solver, which respects constraint hierarchies.

At a first glimpse, DeltaBlue seems a good algorithm to use. It is also fairly easy to implement and efficient. Let's have a closer look.

Algorithm

DeltaBlue constructs an incremental algorithm out of Blue by maintaining an evolving solution to the constraint hierarchy as constraints are added and removed. It takes advantage of the fact that the external factors often only have local effects on the constraint hierarchy. Changes mostly occur gradually: for example when resizing a window. It is therefore beneficial to reuse the previous solution instead of solving the system from scratch everytime.

DeltaBlue uses a constraint hierarchy: there are a number of constraints with labeled strengths. The strongest level represents *required* constraints. The algorithm tries to find a solution which satisfies all required constraints, and satisfies the *preferential* constraints as well as possible according to their relative strength. This is accomplished by using the *locally-predicate-better* comparator which we discussed in Section 5.2.5.

A high-level description of the algorithm would be:

- decide which constraints should be satisfied
- decide which method should be used to satisfy each constraint
- decide in which order to satisfy the constraints

DeltaBlue's data consists of a set of constraints C, a set of variables V and the current solution or *plan* P. The initial configuration is $C = \emptyset$ and $V = \emptyset$. These sets will be modified when the client program adds or removes variables and constraints to the system. Every time a constraint is added or removed, the current solution P is incrementally updated.

DeltaBlue's key technique is the annotation of variables with their *walkabout strength*. The walkabout strength of a variable is the strength of the weakest constraint that could be revoked to allow another, stronger constraint to be satisfied. It covers the global information the algorithm needs in order to incrementally update the solution whenever a constraint is added to or removed from the system. Using this information, the algorithm can predict the effect of the addition or removal of the constraints by examining only the immediate variables of that constraint [FBMB90].

Let's examine a concrete example from [FBMB90]: Figure 9.1 shows a constraint graph consisting of four variables and two constraints. Variables

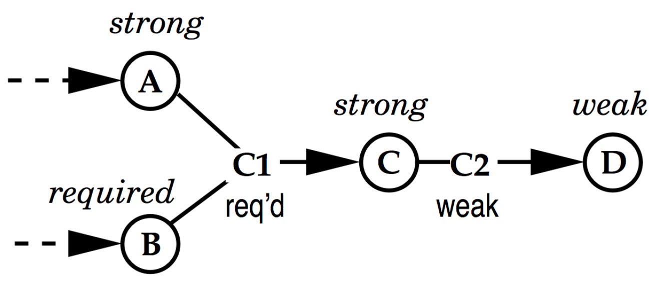


Figure 9.1: Example of DeltaBlue's walkabout strength assignments to variables (extracted from [Bad00])

are represented by circles, while constraints are depicted as arcs. Variable D's walkabout strength is `weak` because the only constraint with D as output variable, namely constraint $C2$ is `weak`. This means that to allow another constraint to change D's value, only a `weak` constraint (namely $C2$) has to be revoked. Variable C's walkabout strength on the other hand is `strong`, because its input variable A is `strong`. Note that although B is `required`, C's walkabout strength is only `strong`: the walkabout strength is the strength of the *weakest* constraint that could be revoked! In the case of C, that's A's strength. Weak walkabout strengths propagate through stronger constraints in the constraint graph [Bad00].

DeltaBlue always finds a locally-predicate-better solution to the constraint hierarchy, given that there are no cycles or conflicting constraints. If these do occur, DeltaBlue aborts. The paper by Freeman and Benson [FBMB90] gives a proof of DeltaBlue's correctness, which we will not cover in detail here. We will just give an intuitive explanation of the inner workings of DeltaBlue.

It comes down to the fact that no solution ever generated by DeltaBlue has *blocked constraints.* A blocked constraint is an unsatisfied constraint whose strength is stronger than the walkabout strength of one of its potential output variables. The *blocked constraint lemma* states:

> If there are no blocked constraints, the set of satisfied constraints represents a locally-predicate-better solution to the constraint hierarchy.

Intuitively this means that if the current solution contains blocked constraints, we can do some more work to find a better solution. After all, there is still a weaker constraint which can be revoked in order for the blocked constraint to be satisfied (remember the blocked constraint is stronger than the walkabout strength of one of its potential output variables). The absence of blocked constraints thus means we have found an optimal solution (by to the locally-predicate better comparator).

Application

DeltaBlue has been used extensively in user interface applications, including but not limited to ThingLab II [Bor79], Voyeur and Multi-Garnet.

Evaluation

The algorithm's complexity is $O(C + V)$, just like *simple* local propagation [Bad00]. Assigning new values given the same configuration (e.g. recalculating the layout when resizing a window) is very fast: $O(C)$ as only one method per constraint gets executed. This makes DeltaBlue suitable for our purpose on terms of efficiency.

There are some limitations of the DeltaBlue solver though [Bad00]:

- it can handle only functional constraints which compute a *single value* for a variable

- it cannot solve cyclic constraint graphs
- all methods must have exactly one output variable

The first restriction comes down to not being able to manage inequalities. It is for instance not possible to specify that one widget is to the left of another widget: $widget1.right_side \leq widget2.left_side$. This is a feature we would like to support in our implementation though.

As we said in the beginning of this chapter, it would be beneficial if the solver could deal with cycles. Unfortunately Blue, and consequently DeltaBlue too, cannot handle cycles.

Finally, multi-output constraints are not supported. These are useful in many situations, but mainly just improve the expressiveness. The classic example is to keep the representation of Cartesian coordinates of a point and the representation of polar coordinates of the same point consistent [San93]. Let x and y be the Cartesian coordinates, while θ and ρ are the polar coordinates. We could do this by defining a constraint with a method to calculate (x, y) from (ρ, θ) and another method to calculate (ρ, θ) from (x, y).

The Constraints Research Group at the University of Washington developed a number of improved versions of the DeltaBlue algorithm. For example, SkyBlue [San93] is a multi-way multi-output local propagation solver, featuring *walkbounds* which is a generalisation of DeltaBlue's walkabout strengths. It thus improves on DeltaBlue by allowing multiple outputs. Other researchers also introduced more capable solvers. We will discuss and evaluate the ones relevant to our problem in the remainder of this section.

9.1.2 Ultraviolet

Ultraviolet is a hybrid constraint solver, consisting of a number of different components.

Algorithm

It partitions the constraint graph into different regions, and uses a *subsolver* appropriate for the kind of constraints in that particular region [BAFB96]. The subsolvers consist of two local propagation solvers (Blue and Indigo) and two cycle solvers (Purple and Deep Purple). They communicate using shared variables.

After partitioning the constraint graph, we have to call on the subsolvers. Invoking each subsolver one at a time and letting each subsolver finish its algorithm before invoking the next would not be consistent with the constraint hierarchy theory. A `weak` constraint in region A might attempt to set a shared variable before a stronger constraint the same variable had been considered [BFB98].

Blue is used for functional constraints over an arbitrary domain, Indigo for inequality and other numeric constraints, Purple for cycles of linear equality constraints and Deep Purple for cycles of linear inequality constraints. In the following paragraphs, we will discuss Blue and Indigo, which are most relevant to the thesis.

Blue Blue is a batch version of the DeltaBlue algorithm we discussed earlier in Section 9.1.1. The main difference is that Blue doesn't use walkabout strengths. Instead Blue keeps a list of active constraints. If enough information is available to deduce the value of one of the constraint's variables, this is done and we also check if there are other constraints whose value can be deduced after this change. Otherwise the constraint is put into the list of active constraints, and we continue with the following constraint [BFB98].

Indigo Indigo is an efficient local propagation algorithm for satisfying acyclic constraint hierarchies, including inequality constraints [BAFB96]. As we discussed earlier in section 9.1.1, inequality constraints are very useful for specifying a graphical layout. One could for example declare that one widget is to the left of another. DeltaBlue cannot handle linear inequalities.

Traditional local propagation algorithms require constraints to be *functional*: they must map to a unique value. Each constraint has several *methods* which can be used to satisfy the constraint. A method fills in the value of the constraint's variables, so that the constraint is satisfied. Each constraint for this kind of algorithm must be functional because otherwise it would not be possible to supply the methods for it [BAFB96].

Indigo's algorithm uses lower and upper bounds instead of a unique value. Indigo propgates bounds instead of values. These bounds get tightened as Indigo moves along the constraints (of course considering their individual strengths). In the end, a variable's upper bound will be equal to its lower bound: the variable will have a unique value.

Indigo uses the locally-error-better comparator which we covered in Section 5.2.5. When dealing with inequality constraints, the locally-predicate-better comparator is not good enough. It is necessary to take the error in satisfying a constraint into account: we need to know *how nearly satisfied* a certain constraint is.

When the algorithm starts, each variable has a stay constraint with the weakest strength. This ensures that in the end each variable will have a specific value. Each variable is annotated with a certain interval. This interval gets tightened as the algorithm executes and will eventually provide us with a locally-error-better solution. A big difference with Blue is that Indigo can use more than one method to satisfy a constraint. It can tighten bounds on any of the variables, if deemed necessary. More details on Indigo's inner workings (and an intuitive description of its correctness) can be found in [BAFB96].

Application

Ultraviolet has used in the Constraint Drawing Framework: a commercial graphics library written in Smalltalk [BFB98]

Evaluation

The solver is definitely very capable. Composite solvers increase their capability by allowing for different sorts of constraints. Unfortunately their implementation is rather tedious. From a software engineering point of view, Ultraviolet is not a good option. First of all, there must be a working implementation of each subsolver. Additionaly, the communication between the solver must be thoroughly tested.

Moreover, Ultraviolet is not an incremental solver, which makes it less suited for our purpose.

9.1.3 Cassowary

Cassowary is an incremental constraint solving toolkit which was created with the problems of earlier solvers and the requirements of user interface applications in mind. It has a few distinctive features. First of all it simultaneously supports *linear equality and inequality constraints*, which arise naturally in specifying many aspect of user interfaces (as we discussed earlier). Secondly, It can gracefully *handle cycles*. And finally, Cassowary finds a *locally-error-better* or *weighted-sum-better* solution [BB98].

Algorithm

Cassowary is based on the *simplex algorithm* from the domain of linear programming, which solves optimization problems. Consider a collection of n real-valued variables $x_1, ..., x_n$, each of which is constrained to be non-negative: $x_i 0$ for $1 \leq i \leq n$. There are m linear equality or inequality constraints over the x, each of the form:

$$\begin{aligned} a_1x_1 + ... + a_nx_n &= b, \\ a_1x_1 + ... + a_nx_n &\leq b, \text{ or} \\ a_1x_1 + ... + a_nx_n &\geq b. \end{aligned}$$

Given these constraints, we wish to find values for the x_i that minimizes (or maximizes) the value of the *objective function*

$$c + d_1x_1 + ... + d_nx_n.$$

Unfortunately the simplex algorithm is not really suitable for interactive applications. We need to solve similar problems repeatedly rather than solving a single problem once [BB98]. In other words, the simplex algorithm is not incremental. On the hand we must be able to quickly resolve thesystem when there for example is a one-way constraint relating the mouse position to the desired x and y coordinates of a figure. On the other hand, when adding or removing constraints and other parts, we would like to reuse as much of the previous solution as possible.

Another issue is to define a suitable objective function. This must be a linear expression, but the objective functions for the comparators used by Cassowary are non-linear. They are technique to handle this though.

Finally, variables may take on both negative and positive values, while the standard simplex algorithm requires all variables to be non-negative [BB98].

Augmented Simplex Form For dealing with negative variables, Cassowary uses the *augmented simplex form.* An optimization problem is in augmented simplex form if constraint C has the form $C_U \wedge C_S \wedge C_I$ where C_U and C_S are conjunctions of linear arithmetic equations and C_I is $\bigwedge\{x \geq 0 \mid x \in vars(C_S)\}$, and the objective function f is a linear expression over variables in C_S. There are *two tableaux* instead of one. All *unrestricted variables* (variables that may take on negative values) are placed in C_U, the unrestricted tableau. C_S, the simplex tableau contains only variables constrained to be non-negative. The simplex algorithm ignores the unrestricted tableau, and determines an optimal solution for the equations in the simplex tableau. The equations in the unrestricted variable tableau are then used to determine values for its variables. An augmented simplex form optimization problem is in *basic feasible solved form* if the equations are of the form

$$x_0 = c + a_1x_1 + ... + a_nx_n$$

where the variable x_0 does not occur in any other equation or in the objective function. If in C_S, c must be non-negative. The variable x_0 is said to be *basic.* Other variables in the equation are called *parameters.* A problem in basic feasible solved form defines a *basic feasible solution*, which can be obtained by setting each parametric variable to 0 and each basic variable to the value of the constant in the right-hand side. [BB98].

Simplex Optimization The method used to find an optimum solution to a constraint in basic feasible solved form, is in fact just phase II of the standard two-phase simplex algorithm. We repeatedly look for an "adjacent" basic feasible solved form whose basic feasible solution decreases the value of the objective function. When no such form can be found, the optimum has been found. We call this *pivoting.* It involves exchanging a basic and a parametric variable using matrix operations. An adjacent form is a new basic feasible form that can be reached by performing a single pivot [BB98].

Other adjustments to support incremental solving We will discuss the other adjustments briefly, because they are rather technical. For more details we refer to [BB98]. In order to incrementally add a new constraint, it is first converted to augmented simplex form. Next the current tableau is used to substitute out all basic variables. An artificial variable is created that represents the constraint. Finally, the system is solved again (or an error is given when the system is unsatisfiable). Removing a constraint is handled by using *marker variables* to keep track of its influence in the tableaux. Non-required constraints are handled by adding the errors of each constraint to form an objective function. A special technique called *quasi-linear optimization* is used to handle the non-linear objective functions represented by the comparators. One-way input constraints are handled by the *dual simplex algorithm*, which starts from an infeasible optimal tableau, and finds a feasible optimal solution. Summarizing, it comes down to updating the constants in the tableaux to reflect the updated stay constraints, then updating the constants to reflect the updated edit constraints, and finally re-optimizing if necessary.

9.1.4 Evaluation

Cassowary is known to be very efficient [BB98]. It also fulfills the requirements we stated in the beginning of this section, so it is definitely a good choice.

9.1.5 Conclusion

We can conclude Cassowary is best-suited for our purpose. It is tailored towards interactive user interfaces. Its very efficient incremental solving will be useful if we want to automatically re-evaluate the layout when the presentation changes (e.g. when a window is resized). The solver can simultaneously handle linear equations and inequalities. Cycles are gracefully handled, which enables the interface designer to fully concentrate on his work. Cassowary has proven to be efficient and expressive enough to be used in many applications such as CCCS/CSVG [1] and Scwm [Bad00].

9.2 Porting Cassowary to the .NET platform

Cassowary [2] is available on many platforms and in many programming languages, such as Smalltalk, C++ and Java.

[1]Constraint Cascading Style Sheets/Constraints Scalable Vector Graphics

[2]`http://www.cs.washington.edu/research/constraints/cassowary/`

Uiml.net is available on the Microsoft .NET platform, the free Mono implementation of .NET and the .NET Compact platform (a specialized version of .NET for mobile devices) [LC04]. In order to use automated layout management on all platforms Uiml.net supported (especially on the .NET Compact platform), a native .NET version of Cassowary was needed. In early development, as a temporary solution we used a tool (IKVM.NET [3]) to generate .NET code from the Java sources. After the port was finished, we could switch to the native .NET version.

Cassowary.net [4] is a full port of the Cassowary toolkit to the .NET platform. It is free software and is available under the same license terms (the GNU LGPL [5]) as the Java version by Greg J. Badros.

9.3 Integration in Uiml.net

9.3.1 Changes to the rendering process

Of course, eventually we had to integrate the constraint solver into Uiml.net's rendering process. Before, Uiml.net would in order:

- parse the UIML document
- create a renderer, with the appropriate backend dynamically loaded according to the document's vocabulary
- render the document
- show the resulting interface

It is beneficial to use the original position and size of each widget as initial values. In order to obtain these, we need to render the interface before actually solving the layout constraints. We would like this because when no specific size is specified, most widget sets render their widgets in the correct dimensions. For instance, a button containing a fairly long caption, would be given a size such that the caption would fit into the button's area. Moreover, the designer could specify absolute positions and sizes, which would be used as initial values. We place stay constraints on the initial values, allowing the designer to provide hints to the constraint solver. Using this technique, manual layouts that adhere to the constraints will not be altered by the constraint solver!

Afterwards we can set the position and size of each widget according to the solution found by the solver, and render the interface to again to reflect these changes. Then the interface would be ready to be shown on the screen. The rendering process changed as follows:

[3]`http://www.ikvm.net/`

[4]`http://lumumba.uhasselt.be/jo/projects/cassowary.net/`

[5]`http://www.gnu.org/copyleft/lesser.html`

- parse the UIML document
- create a renderer, with the appropriate backend dynamically loaded according to the document's vocabulary
- render the document
- *solve the layout constraints and set the resulting position and size of each widget*
- *render the document again to reflect these changes*
- show the resulting interface

9.3.2 Specifying the layout in UIML

Since UIML has no direct support for layout management, we added a *layout* element to the specification. This element is associated with a certain container part in the structure section, and describes the constraints for that specific building block. A *building block* corresponds to a *presentation unit*, which we discussed in Section 3.2.1.

Let us look at an example. Listing 9.1 describes a simple interface with two buttons and two entries.

Listing 9.1: A simple layout

```
<?xml version="1.0">
<uiml>
  <interface>
    <structure>
      <part class="Frame" id="frame">
        <part class="Entry" id="leftentry"/>
          <part class="Button" id="copyleft"/>
          <part class="Button" id="copyright"/>
          <part class="Entry" id="rightentry"/>
      </part>
    </structure>
  <style>
    ...
  </style>
  <layout part-name="frame">
    <!-- Left entry left of buttons -->
    <constraint type="leq">
      <property part-name="leftentry" name="right"/>
      <property part-name="copyleft" name="left"/>
    </constraint
    <constraint type="leq">
      <property part-name="leftentry" name="right"/>
    <property part-name="copyright" name="left"/>
    </constraint>
    <!-- Buttons left of right entry -->
    <constraint type="leq">
      <property part-name="copyleft" name="right"/>
      <property part-name="rightentry" name="left"/>
    </constraint>
    <constraint type="leq">
      <property part-name="copyright" name="right"/>
      <property part-name="rightentry" name="left"/>
    </constraint>
    <!-- Left copy button above right copy button -->
    <constraint type="leq">
      <property part-name="copyleft" name="bottom"/>
    <property part-name="copyright" name="top"/>
    </constraint>
  </layout>
  </interface>
</uiml>
```

There is a `<layout>` element which is associated with the the surrounding container (`frame`). The layout consists of a list of `<constraint>` elements. Constraints have a type, and contain `<property>` elements. In this case, each constraint is of the type `leq`, meaning that it is a linear inequality constraint, specifying that the first property is *less or equal* than the second property. These properties can be any of `width`, `height` `left`, `right`, `top` or `bottom`. Figure 9.2 describes the geometrical interpretation of these properties. The first constraint represents `leftentry.right` $\leq$

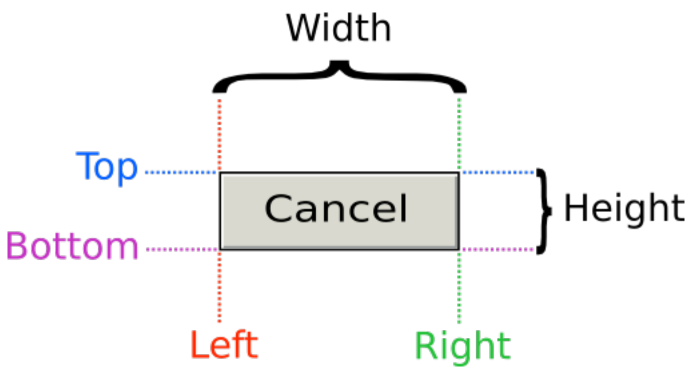


Figure 9.2: The six layout properties

`copyleft.left`. Intuitively this means that `leftentry` should be left of `copyleft`. The next constraint analogously describes that `leftentry` should be left of `copyright`. Summarizing, the two first constraints enforce that `leftentry` is left of the buttons. The following two constraints relate the buttons and `rightentry`, specifying that the buttons should be to the left of `rightentry`. Finally, we want `copyleft` to be above `copyright`. Remember the upper left corner of the interface has coordinates $(0, 0)$, while the lower right corner's coordinates are $(width, height)$: the Y axis increases from top to bottom. Since `copyleft` is to be placed above `copyright`, the correct constraint is `copyleft.bottom` ≤ `copyright.top`. The resulting interface, rendered with System.Windows.Forms backend is shown in Figure 9.3.

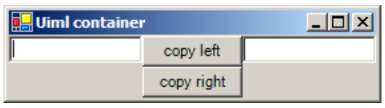


Figure 9.3: The resulting interface, rendered with the System.Windows.Forms backend

We can compare this with the UIML documents in Section 2.4, which describe the same interface. These use widget set specific layout management, which increases the inter-vocabulary distance. The UIML document specified in Listing 9.1 on the other hand would be the same for all backends! The layout properties (see Figure 9.2) have to be mapped on each widget set's specific properties in the vocabulary. At the moment only the System.Windows.Forms backend is fully supported though. The Gtk# backend still needs more work, as well as the Compact System.Windows.Forms (SWF for the Compact .NET platform) backend.

9.3.3 Software architecture

A couple of changes have been made to Uiml.net's architecture. There is a new namespace `Uiml.Layout`, grouping most of the layout code. First of all, we had to parse `<layout>` elements in a UIML document. This is reflected in the `Layout`, `Constraint` and `LayoutProperty` classes, mapping on the `<layout>`, `<constraint>` and `<property>` tags respectively.

Furthermore, each UIML document has an instance of the `ConstraintSystem` class. This class keeps a list of all constraints and properties for a certain UIML interface. The properties are in fact the variables that will be given a specific value by solving the system. Each `ConstraintSystem` instance has a `ClSimplexSolver`, which represents the Cassowary solver and is eventually passed an appropriate representation of the constraints and variables described in the UIML document.

The `LayoutProperty` class is derived from the general `Property` class, and is treated the same way by the renderer. The difference lies in the fact that `LayoutProperty` has a `ClVariable` member, which is a specific representation of the variable to be passed on to the constraint solver. The behavior of the `Value` property getter was also altered: it now directly returns the variable's value. Intuitively, one could see `LayoutProperty` instances are smart versions of absolute positioning setter properties. They both set a property to an absolute, fixed value. A `LayoutProperty` is however linked to a variable of the constraint solver which determines its value, while a general `Property` setter's value is directly specified in the UIML document.

`Constraint` instances keep a list of the properties they refer to, and a specific constraint representation, ready to be fed to the solver. The constraint representation is an instance of one of `ClConstraint`'s derived classes.

9.4 Conclusion

We enlightened our choice of the Cassowary constraint solver. Furthermore we discussed the development of Cassowary.net, a port of Cassowary to the .NET platform. And finally, we had a look at an example UIML interface, using spatial constraints to specify the layout.

Our approach relies only on the least common denominator of layout management, namely absolute positioning. Therefore Uiml.net in combination with our constraint-based layout management can prove useful for widget sets that only support fixed widget positions, such as System.Windows.Forms.

It is clear from Section 9.3.2 that a generic layout management specification can significantly reduce the inter-vocabulary distance, and thus facilitate switching from one widget set to another.

There is still room for improvements however. For one, it would be interesting to support a number of different widget sets, more specifically Gtk# and Compact SWF (which Uiml.net supports best along with SWF). Compact SWF should not pose too many problems, since it is very related to SWF. Cassowary.net only requires a few changes to be ported to the .NET Compact platform. Gtk# will be a bigger challenge though. The toolkit makes it rather difficult to use absolute positioning. The WxWidgets widget set supports Gtk as a backend, and also relies on absolute positioning in combination with spatial constraints, meaning it is certainly possible to use absolute positioning with Gtk#.

Additionally, specifying a layout should be made as easy as possible for the interface designer. A possible solution is to use layout managers similar to the ones used in traditional widget sets. Layout managers are widely known for their ease-of-use when creating simple layouts (see Section 4.1).

Part IV

Conclusion

This thesis introduced a widget set independent layout mechanism for UIML documents, which can be used for 2D graphical user interfaces. A specific syntax for automated layout management had to be introduced. We believe our technique is useful for 2D GUI's, but it is by no means a general layout solution. Different modalities require a completely new notion of layout.

Another contribution of this thesis is the development of the Cassowary.net contraint solver, a port of the original Cassowary toolkit to the .NET platform.

Our approach relies only on the least common denominator of layout management, namely absolute positioning. Therefore Uiml.net in combination with our constraint-based layout management can prove useful for widget sets that only support fixed widget positions, such as System.Windows.Forms.

It is clear from Section 9.3.2 that a generic layout management specification can significantly reduce the inter-vocabulary distance, and thus facilitate switching from one widget set to another.

There is still room for improvements however. For one, it would be interesting to support a number of different widget sets, more specifically Gtk# and Compact SWF. Additionally, specifying a layout should be made as easy as possible for the interface designer. A possible solution is to use layout managers similar to the ones used in traditional widget sets. Another useful improvement would be to increase the degree of *plasticity* by remapping and rearranging widgets dynamically (see Section 6.1). Section 6.1.3 explained how remappings could be realized in UIML. However, a difficult problem is how to decide when remappings or rearrangements should occur.

Part V

Appendices

Appendix A

Dutch summary

Deze bijlage geeft een Nederlandstalige samenvatting van de thesis. De indeling komt overeen met de verschillende hoofdstukken.

A.1 Inleiding

De groeiende diversiteit aan computeromgevingen vereist een nieuwe, meer efficiënte methodologie om gebruikersinterfaces te ontwikkelen. Traditioneel ontwikkelt men één gebruikersinterface per computerplatform. De opgang van mobiele en ingebedde systemen vergroot het probleem alleen maar. Men moet een gebruikersinterface dus in een abstracte, apparaatonafhankelijke manier gaan ontwikkelen, om zo de kost om een nieuw platform te ondersteunen te beperken.

Bestaand onderzoek heeft al uitgewezen dat het redelijk duidelijk is hoe we een uit de specifieke interface elementen voor elk platform, een verzameling *generieke widgets* (of generieke interface elementen) kunnen abstraheren. Het is echter moeilijk de verschillende layout mechanismen die widget sets gebruiken te generaliseren.

Een layout manager is een software component die de positionering en schalering van individuele interface elementen in een gebruikersinterface afhandelt. De eenvoudigste layout manager gebruikt *absolute positionering*, waarbij elk widget een absolute positie en grootte toegewezen krijgt. Traditionele toolkits zoals Gtk, Qt en Java Swing gebruiken flexibelere technieken. Jammer genoeg kunnen deze technieken niet zowel mobiele systemen als desktop systemen aan. Het zou interessant zijn als de gebruikersinterface zich automatisch zou kunnen aanpassen, om beter aan de beperkingen van de nieuwe omgeving te voldoen. Er zijn immers meerdere concrete implementaties van generieke widgets beschikbaar, elk met hun eigen specifieke eigenschappen. Onderdelen van de interface zouden ook gehergroepeerd kunnen worden wanneer er te weinig schermruimte beschikbaar is.

Ideaal gezien zouden we het layout proces volledig willen automatiseren.

Dit is echter nog niet mogelijk. Meestal is er nog inbreng van de gebruikersinterface ontwerper nodig.

Een widget set onafhankelijke methode zou elke layout die mogelijk is met de specifieke methoden moeten ondersteunen. De meeste technieken bekijken het probleem op een abstracte manier. Als we enkel de gemeenschappelijke kenmerken behouden, eindigen we immers met een banaal layout mechanisme.

Onze methode is gebaseerd op de UIML taal. We gaan op zoek naar een layout specificatie voor een grafische, 2D interface. Het praktische gedeelte van de thesis bestaat er in de specificatie te integreren in Uiml.net.

A.2 UIML

UIML is een taal waarmee een gebruikersinterface gespecifieerd kan worden in een declaratieve, abstracte manier. UIML is een meta-taal, net als XML. Een UIML gebruikersinterface is een verzameling interface elementen, of *parts*. Elk element bevat inhoud of *content*. Er is een *behavior* onderdeel dat de interactie van de interface bepaalt. Hier wordt gewerkt met regels, die een conditie en een reeks van acties bevatten. Een conditie komt overeen met een gebeurtenis of *event* in de gebruikersinterface.

UIML biedt ondersteuning voor herbruikbare interface componenten. Dit wordt gerealiseerd door het UIML *template element*. Een template kan gezien worden als een aparte tak op de UIML boom. Die tak kan samengevoegd worden met de hoofdboom waar er een gelijkaardige vertakking is, namelijk: het enige kind van het template element moet hetzelfde zijn als het kind dat het template element aanroept. Er zijn drie mogelijke methoden om een template aan te roepen: *vervangen* (replace), *toevoegen* (append) en *trapsgewijs vervangen* (cascade).

De *vocabulary* van een UIML document bevat de associaties tussen abstracte interactoren en specifieke interactoren. Het is de bedoeling zoveel mogelijk van de UIML beschrijving te kunnen hergebruiken wanneer er gewisseld wordt van *backend*. De backend is de concrete widget set die gebruikt wordt om de gebruikersinterface op het scherm weer te geven. Specifieke layout technieken in het UIML document zorgen ervoor dat dit moeilijk wordt. Deze specifieke technieken moeten immers voor iedere backend veranderen. Een generieke layout specificatie zou ervoor zorgen dat dezelfde UIML beschrijving voor verschillende backends bruikbaar zou zijn.

A.3 Model-gebaseerde gebruikersinterface ontwikkeling

Gebruikersinterface modellering wordt gezien als een oplossing voor het probleem van multi-apparaat gebruikersinterface ontwikkeling. Hierbij wordt er gewerkt met verschillende *modellen*, die elk een andere component van de gebruikersinterface beschrijven. Zo zijn er o.a. het taakmodel en het presentatiemodel. Gebruik makend van die modellen kunnen we automatisch een gepaste gebruikersinterface genereren. Het gebruikersinterfacemodel bestaat uit het platformmodel, het presentatiemodel en het taakmodel.

Model-gebaseerde technieken construeren *associaties* tussen de verschillende modelcomponenten. Deze associaties kunnen geïnterpreteerd worden om zo een aangepaste gebruikersinterface voor het relevante apparaat en context te genereren.

Het presentatiemodel associeert abstracte interface elementen, ook *abstracte interactieobjecten (AIO's)* genoemd, met concrete interface elementen, ook *concrete interactieobjecten (CIO's)* genoemd.

A.4 Geautomatiseerd layoutbeheer

Eenvoudige technieken in het domein van geautomatiseerd layoutbeheer zijn o.a. de *layout managers* die gebruikt worden in moderne gebruikersinterface toolkits. Ze steunen op een hiërarchie van containers en interface elementen. Deze containers bevatten andere containers of interface elementen, en zorgen voor de layout van hun kindelementen. De layout kan zich niet aanpassen aan extreme omstandigheden, maar het is wel erg gemakkelijk in gebruik voor eenvoudige layouts. Andere eenvoudige technieken zijn de layoutalgoritmen van tekstverwerkers. Verder is er nog het $\TeX$ typesetting systeem dat op vergelijkbare principes als de layout managers in gebruikersinterface toolkits steunt.

Een veelgebruikte methode is het gebruik van *constraints* en een *constraint solver*. Er zijn abstracte en spatiale constraints. Meestal worden abstracte constraints in een later stadium omgezet naar spatiale constraints.

Het is mogelijk een layout automatisch te *evalueren* gebruikmakende van heuristieken of metrieken. Zo kan men bijvoorbeeld meten hoeveel de gebruiker met de muis beweegt, om te bepalen of de layout al dan niet gebruiksvriendelijk is. *SUPPLE* is een systeem dat layout management als een optimalisatieprobleem beschouwt. Het probeert de inspanning die de gebruiker moet doen te minimaliseren, en tegelijkertijd ook aan de beperkingen van het apparaat te voldoen. De layout kan dynamisch veranderen. Als de gebruiker bepaalde interface elementen vaak na elkaar gebruikt bijvoorbeeld, kunnen ze dichter bij elkaar gezet worden. Het is echter onzeker welke veranderingen er precies zullen gebeuren, en of die altijd het gebruiksgemak ten goede zullen komen. Het is ook onduidelijk hoe het gedrag van SUPPLE beïnvloed kan worden.

A.5 Constraint solvers

Een constraint beperkt mogelijkheden. Het is een relatie die we willen onderhouden. Een lijst van constraints noemen we een *constraint systeem.* Constraints zijn erg natuurlijk om iets te *beschrijven*, daarom zijn ze ook erg geschikt om een grafische layout te beschrijven.

Er is een constante afweging tussen *expressiviteit* en *performantie.* We moeten interessante, niet-triviale relaties kunnen beschrijven, terwijl we toch interactieve applicaties willen ondersteunen.

Een *constraint hiërarchie* bestaat uit verplichte en voorkeursconstraints. De verplichte constraints moeten bevredigd worden. Het systeem zou de voorkeursconstraints moeten bevredigen in zoverre dat mogelijk is. Voorkeursconstraints kunnen verschillende niveau's van sterkte hebben. Om een constraint hiërarchie op te lossen, moeten we kunnen beslissen wat de beste oplossing is. Hiervoor wordt er een *comparator* gebruikt.

Het kan voorkomen dat er een *cycle* in de constraintgraaf zit. Sommige constraint solvers kunnen met cycles omweg, terwijl anderen daarvoor een gespecialiseerde cycle solver nodig hebben. Cycles duiden op redundante informatie in het constraintsysteem. Verder kunnen er ook conflicterende constraints in het constraint systeem zitten. Als ze een verschillende sterkte hebben, kan de zwakste genegeerd worden, maar wanneer het constraintsysteem niet oplosbaar is, dient er een foutmelding gegeven te worden.

A.6 Flexibele interfaces

Een intelligente layout management methode zou *remappings* kunnen gebruiken om dynamisch te switchen tussen verschillende CIOs, beginnend van een bepaald AIO.

Graceful degradation is een methode om zoveel mogelijk continuïteit te garanderen tussen de verschillende platformspecifieke versies. Men vertrekt van een platformspecifieke gebruikersinterface voor het minst beperkte platform. Deze broninterface wordt dan omgezet voor de andere platformen. Graceful degradation voorziet verscheidene regels doorheen de hele cyclus van modelgebaseerde ontwikkeling.

Comets zijn een nieuwe generatie van interface elementen, die zich kunnen aanpassen aan de specifieke context. Met context bedoelen we het platform, de gebruiker en de omgeving. Plasticiteit is de mate waarin een interactief systeem variaties in de context kan weerstaan, zodat tegelijkertijd toch de gebruikskwaliteit behouden blijft.

We kunnen remappings in UIML realiseren door het template element. Het is echter moeilijk te beslissen wanneer een remapping plaats moet vinden, en welk specifiek CIO we kiezen. Hiervoor is metadata over elk CIO nodig.

Een *HCI patroon* is een algemeen aanvaarde oplossing voor een steeds terugkerend probleem in de ontwikkeling van gebruikersinterfaces. Zo zijn er ook *layout patronen*, die een visueel aantrekkelijke en efficiënte layout beschrijven voor een bepaalde, steeds terugkerende taak in een gebruikersinterface. Via layout patronen kunnen we de expertise van visuele ontwerpers hergebruiken. Layout patronen in combinatie met het taakmodel bieden erg veel flexibiliteit. Het is dus zeker nuttig ondersteuning voor layout patronen in onze implementatie te voorzien.

A.7 Uiml.net

Onze implementatie werd geïntegreerd in *Uiml.net*, een open source UIML renderer voor het Microsoft .NET platform. Een renderer is complexer dan een compiler maar biedt ook meer flexibiliteit (o.a. rapid prototyping), omdat het de interface rendert. Deze aanpak is vergelijkbaar met de aanpak van web browsers, die webpagina's renderen.

Uiml.net bestaat voornamelijk uit een interface reader, de rendering backends en de connectie tussen de gebruikersinterface en de applicatielogica. Het ontwerp is erg dynamisch, het grootste deel van de code is onafhankelijk van de gebruikte backend. Veranderingen aan een widget set, moeten meestal enkel aangepast worden in de vocabulary, aangezien Uiml.net nergens expliciet specifieke widgets aanmaakt. Er wordt veel gebruik gemaakt van *reflectie*.

A.8 Het UIML template element

We hebben het UIML template element geïmplementeerd in Uiml.net, o.a. om layout patronen te ondersteunen. Er is een abstracte `Sourceable` klasse, waarvan UIML elementen die een template element kunnen aanroepen afleiden. Deze klasse delegeert het effectief aanroepen van het template element door naar een *template resolver*. Voor elke aanroepmethode is er een klasse afgeleid van `TemplateResolver`.

A.9 De constraint solver

We hebben een aantal kandidaat constraint solvers onderzocht. Onze vereisten waren dat de solver geschikt was voor interactieve applicaties, dat hij zowel verplichte als voorkeursconstraints ondersteunde, dat cycles moeiteloos afgehandeld werden en dat lineaire gelijkheden en ongelijkheden ondersteund werden.

DeltaBlue leek op het eerste zicht een goede kandidaat. Het kan echter enkel functionele constraints aan die een enkele waarde voor elke variabele berekenen, cycles worden niet toegelaten, en elke oplossingsmethode voor een constraints heeft precies één enkele output variabele.

UltraViolet is een hybride constraint solver, die o.a. Blue, Indigo, Purple en Deep Purple gebruikt. Het is echter geen incrementeel algoritme (het kan eventueel wel gecompileerd worden). De implementatie ervan is ook zeker niet vanzelfsprekend.

Cassowary is een incrementele constraint solving toolkit die tegelijkertijd lineaire gelijkheden en ongelijkheden ondersteunt. Cassowary is erg efficiënt en volledig toegespitst op gebruikersinterface applicaties. Het kan ook moeiteloos omgaan met cycles. Uiteindelijk was Cassowary de beste keuze

We hebben Cassowary geport naar het .NET platform, om het zo beter te kunnen integreren met Uiml.net, en het ook mogelijk te maken de solver te gebruiken op het .NET Compact platform. *Cassowary.net* is een volledige port van Cassowary en is vrij beschikbaar.

We hebben enkele veranderingen aangebracht aan het *renderproces van Uiml.net*. We renderen het document nu twee keer. Een keer om de initiële waarden van de posities en groottes van de widgets te bepalen, en de tweede keer om de oplossing van de constraint solver te gebruiken. We hebben een `<layout>` element toegevoegd aan de UIMl specificatie. Een layout is geassocieerd met een bepaald containerelement. Binnen een layout worden de constraints gespecifieerd. We maken gebruik van *zes layouteigenschappen*, die de absolute positie en grootte van een widget bepalen.

De architectuur van Uiml.net is uitgebreid met de `Uiml.LayoutManagement` namespace.

A.10 Slotbeschouwing

Deze thesis introduceerde een widget set onafhankelijk layoutmechanisme voor UIML. Dit mechanisme kan gebruikt worden voor 2D grafische gebruikersinterfaces. We hebben een specifieke syntax voor layout management moeten introduceren. Onze oplossing is zeker geen generieke, multi-modale layout oplossing.

Verder hebben we de Cassowary.net constraint solver ontwikkeld, een port van de originele Cassowary toolkit naar het .NET platform.

Onze aanpak vereist enkel absolute positionering, waar elke widget set ondersteuning voor biedt. Uiml.net in combinatie met ons layout mechanisme kan op zichzelf al nuttig zijn voor widget sets die enkel absolute positionering aanbieden. Een generieke methode voor layout management biedt een significante verbetering voor het hergebruik van UIML documenten.

Er zijn echter nog verbeteringen mogelijk. Zo zou het onder andere interessant zijn meerdere widget sets te ondersteunen, zoals Gtk# en Compact SWF. Verder zouden we het zo makkelijk mogelijk moeten maken om een layout te specifiëren. Zo zouden we bijvoorbeeld traditionele layout managers kunnen inbouwen. Een andere verbetering zou zijn het verhogen van de plasticiteit door dynamisch widgets te remappen en te herschikken. Het is echter niet eenvoudig te beslissen wanneer en hoe we die remappings gaan realiseren.

Bibliography

[703] JTC 1/SC 7. Software product quality requirements and evaluation (square) guide to square. Technical report, ISO/IEC, 2003. 46

[AH04] Marc Abrams and Jim Helms. User interface markup language (uiml) specification version 3.1. Technical report, OASIS Open, Inc., 2004. `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=uiml`. 10

[Ale79] Christopher Alexander. *The Timeless Way of Building.* Oxford University Press, 1979. 48

[AnAS02] Mir Farooq Ali, Manuel A. Pérez-Qui nones, Marc Abrams, and Eric Shell. Building multi-platform user interfaces with uiml. In *CADUI'2002*, pages 225–236, Université de Valenciennes, France, 2002. 14

[AP99] Marc Abrams and Constantinos Phanouriou. Uiml: An xml language for building device-independent user interfaces. In *XML '99*, Philadelphia, USA, 1999. 10, 11, 55

[Bad00] Gregory Joseph Badros. *Extending interactive graphical applications with constraints (user interface).* PhD thesis, University of Washington: Department of Computer Science, 2000. Chair-Alan Borning. 4, 29, 36, 37, 39, 60, 61, 62, 67

[BAFB96] Alan Borning, Richard Anderson, and Bjorn Freeman-Benson. Indigo: a local propagation algorithm for inequality constraints. In *UIST '96: Proceedings of the 9th annual ACM symposium on User interface software and technology*, pages 129–136, New York, NY, USA, 1996. ACM Press. 39, 63, 64, 65

[BB98] Greg J. Badros and Alan Borning. The cassowary linear arithmetic constraint solving algorithm: Interface and implementation. Technical Report UW-CSE-98-06-04, University of Washington, Seattle, Washington, USA, 1998. 65, 66, 67

[BFB98] Alan Borning and Bjorn Freeman-Benson. Ultraviolet: A constraint satisfaction algorithm for interactive graphics. *Constraints*, 3(1):9–32, 1998. 63, 64, 65

[Bor79] Alan Hamilton Borning. *Thinglab–a constraint-oriented simulation laboratory.* PhD thesis, Stanford University: Department of Computer Science, 1979. 62

[Bor01] Jan Borchers. *A Pattern Approach to Interaction Design.* John Wiley & Sons, Inc., New York, NY, USA, 2001. Foreword by Frank Buschmann. 48, 55

[CCODD04] Gaëlle Calvary, Joëlle Coutaz, Lionel Balme Olfa Dâassi, and Alexandre Demeure. Towards a new generation of widgets for supporting software plasticity: the “comet”. In *Preproceedings of EHCI/DSV-IS'04*, pages 41–60, 2004. 4, 45, 46

[CLV$^+$03] Karin Coninx, Kris Luyten, Chris Vandervelpen, Jan Van den Bergh, and Bert Creemers. Dygimes: Dynamically generating interfaces for mobile computing devices and embedded systems. 2795:256–270, sept 2003. 20, 32

[EVP01] Jacob Eisenstein, Jean Vanderdonckt, and Angel Puerta. Applying model-based techniques to the development of uis for mobile computers. In *IUI '01: Proceedings of the 6th international conference on Intelligent user interfaces*, pages 69–76, New York, NY, USA, 2001. ACM Press. 4, 20, 22, 23, 24

[FBMB90] Bjorn N. Freeman-Benson, John Maloney, and Alan Borning. An incremental constraint solver. *Commun. ACM*, 33(1):54–63, 1990. 36, 39, 60, 61, 62

[FV04] Murielle Florins and Jean Vanderdonckt. Graceful degradation of user interfaces as a design method for multiplatform systems. In *International Conference on Intelligent User Interfaces IUI 2004*, Funchal, Madeira Island (Portugal), 2004. 41

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995. 48

[GW04] Krzysztof Gajos and Daniel S. Weld. Supple: Automatically generating user interfaces. In *IUI'04*, Funchal, Portugal, 2004. 32

[JSES04] Homa Javahery, Ahmed Seffah, Daniel Engelberg, and Daniel Sinnig. Migrating user interfaces across platforms using hci patterns. *Multiple User Interfaces: Multi-Devices, Cross-Platform and Context-Awareness*, pages 241–259, 2004. 48, 49

[Knu84] Donald E. Knuth. *The TEX book*. Addison-Wesley, 1984. Describes TEX in detail. 28

[LC04] Kris Luyten and Karin Coninx. Uiml.net: an open uiml renderer for the .net framework. In *CADUI'2004*, Funchal, Madeira Island (Portugal), 2004. iii, 4, 9, 52, 53, 67

[LCA05] Kris Luyten, Karin Coninx, and Marc Abrams. Integrating uiml, task and dialogs with layout. In *HCI International 2005*, Las Vegas, Nevada, USA, 2005. 21, 50

[LCC03] Kris Luyten, Bert Creemers, and Karin Coninx. Multi-device layout management for mobile computing devices. Technical Report TR-LUC-EDM-0301, EDM/LUC, Diepenbeek, Belgium, 2003. 8, 36

[LF01] Simon Lok and Steven Feiner. A survey of automated layout techniques for information presentations. In *SmartGraphics Symposium (SG2001)*, New York, USA, 2001. 26, 27, 28

[Luy04] Kris Luyten. *Dynamic User Interface Generation for Mobile and Embedded Systems with Model-Based User Interface Development*. PhD thesis, transnational University Limburg: School of Information Technology, 2004. 14, 18

[Pue97] Angel R. Puerta. A model-based interface development environment. *IEEE Softw.*, 14(4):40–47, 1997. 20

[San93] Michael Sannella. The SkyBlue constraint solver and its applications. In Paris Kanellakis, Jean-Louis Lassez, and Vijay Saraswat, editors, *PPCP'93: First Workshop on Principles and Practice of Constraint Programming*, Providence RI, 1993. 63

[VG94] Jean Vanderdonckt and Xavier Gillo. Visual techniques for traditional and multimedia layouts. In *2nd ACM Workshop on Advanced Visual Interfaces AVI'94*, 1994. 47

[vUvAB$^+$98] E.P.C. van Utteren, J. van Amstel, M. Boasson, L.O. HertzBerger, J. Baeten, and J.P. Veen. Program for research on embedded software and systems, 1998. `http://www.stw.nl/progress/programma/rapport.pdf`. 7