# ui2go White Paper

Dirk Struve
phylofriend at projectory.de
https://github.com/yogischogi/ui2go/

March 16, 2015

# Contents

**1 About ui2go** **3**
1.1 What is ui2go? . . . 3
1.2 Why was ui2go created? . . . 3
1.3 Why not use some Web technology? . . . 3

**2 Software Design Philosophies** **6**
2.1 Possibility Approach . . . 6
2.1.1 Metrics . . . 6
2.1.2 Frequently attributed as . . . 6
2.2 Complexity or Big Boots Approach . . . 7
2.2.1 Metrics . . . 7
2.2.2 Frequently attributed as . . . 7
2.3 Usability Approach . . . 7
2.3.1 Metrics . . . 7
2.3.2 Frequently attributed as . . . 8

**3 Program Control** **9**
3.1 Control Flow . . . 9
3.2 Events . . . 10
3.2.1 General Idea . . . 10
3.2.2 Traditional Event Handling . . . 13
3.2.3 Callback Disaster . . . 13
3.2.4 ui2go Solution . . . 15

**4 Layout Management** **23**
4.1 Border Layout . . . 25
4.2 Grid Layout . . . 26
4.3 Classical Input Mask . . . 27

**5 Font Size** **28**
5.1 Distance Dependency . . . 28
5.2 Font Metrics . . . 28
5.3 Viewing Distances . . . 30
5.4 Recommended Font Sizes . . . 31
5.5 Cap and Rem Units . . . 31

In software industry innovations are like alcohol. They make you euphoric at first, but leave you with a big headache.

— Anonymous

# 1 About ui2go

## 1.1 What is ui2go?

ui2go is a GUI library for the Go programming language. It should enable the programmer to create graphical user interfaces in an easy way. At the moment ui2go is in a very early stage of development and practically nothing more than a showcase.

The project is hosted at https://github.com/yogischogi/ui2go/.

## 1.2 Why was ui2go created?

I started this project out of frustration. Although there is an ongoing fuzz about new user interfaces, programming technologies themselves doesn't seem to make much progress. Even worse, I can't help the feeling, that creating graphical user interfaces has become more complicated and time consuming than ever before.

So I asked myself, if there was a better way and started to explore different alternatives. ui2go is a first prototype of my thoughts.

## 1.3 Why not use some Web technology?

Well, the answer to that question is a bit complicated. I went to an expert to ask him for advice and this is what happened :-)

| | |
|---|---|
| Me: | I want to write a program with a graphical user interface. |
| Expert: | Sure, nothing easier than that. Just use some Web technology. |
| Me: | Isn't it a bit too complicated? |
| Expert: | No, not at all! You just install a web server, write some HTML code and point your browser to it. |
| Me: | Are you telling me, that all I need is to install and configure a web server, write some HTML and use my web browser to view the result? |
| Expert: | Yeah, easy, isn't it? |
| Me: | But isn't it difficult to organize all these HTML pages? |
| Expert: | No, not at all. There is this thing called Big Typo. It makes things really easy. |
| Me: | Typo? Isn't that another word for mistake? |
| Expert: | Yeah, but that's the way people write on the web these days. |
| Me: | To be honest. I tried HTML before and found it quite hard to get the layout right. |

| | |
|---|---|
| Expert: | No need to worry about that. Use the jWeary library! It makes things really simple and reduces the amount of HTML code a lot. |
| Me: | So I only need to learn HTML and this j-scripting stuff? |
| Expert: | Yeah, that's really easy. |
| Me: | Actually I have tried this scripting stuff before and it just doesn't seem to work out right. |
| Expert: | Oh, you probably used the wrong browser. What are you using? |
| Me: | I like Excalibur a lot. |
| Expert: | See? I guessed right. You don't want to use Excalibur. These guys have microscopic brains and are a bit soft in the head. You surely don't want to be like them. |
| Me: | Well, what about using Burning Tails instead? |
| Expert: | Yeah, guys with Burning Tails are fast, but it's painful sometimes. You better use Nice n' Shine to stay out of trouble. |
| Me: | I have got another problem. Whenever I use some kind of scripting language my programs tend to get a bit tangled when they get bigger. |
| Expert: | This scripting stuff might not be for everyone. But don't worry. There is an extremely simple solution for that. Use Smart! I's a brand new programming language and even compiles to scripts. You surely hit your target with this one! Won't get any easier, I can tell you that. |
| Me: | So I only need to install and configure a web server, use this Big Mistake and the right web browser, install the Smart stuff and this jWeary, learn some HTML, scripting and Smart. Is this really all there is to it? |
| Expert: | Yeah, I told you it's easy, right? |
| Me: | But what if I need access to some operating systems library? |
| Expert: | I tell you a secret. There is this awesome Holy Grails framework. It makes things even simpler. |
| Me: | I'm not really sure about that. |
| Expert: | I show you a program, I've written in just a couple of hours. Look! |
| Me: | Well,... that's *Hello World!* |
| Expert: | Yeah, but it's kind of cool, isn't it? |

Apart from the fun I believe that Web solutions are not well suited for a lot of serious programs. Here are some arguments:

1. Web programs require a complex multilayer architecture (operating

system, web browser, scripting libraries) to work.

The resulting system is overly complex, hard to maintain, error prone, slow and creates lots of potential security risks.

2. Web solutions are slow and energy consuming. This is certainly not be the best contribution to green computing.

3. Web programs are a security risk. There are many software components involved and the browser is the one program, that is exposed most to the dangerous world.

   People, who take Web security seriously, often end up with some kind of multi browser multi configuration environment, that is costly and hard to maintain.

I do not doubt the tremendous benefits of the Web, but one should always keep in mind, that it might not be the best platform for a lot of applications it is frequently used for.

# 2 Software Design Philosophies

When creating a software system there are always a lot of design decisions to make. Usually a specific solution is chosen by using a set of metrics based upon some kind of philosophy. Because different people value different things there are different design philosophies.

People with different philosophies usually don't get along very well and there are a lot of misunderstandings, just because they use different metrics.

So it is important to take a look at some design approaches to software development. There are many, but these are some common ones:

## 2.1 Possibility Approach

In many cases people don't consider advantages or disadvantages of a design. In fact, they don't design at all. They just code, trying to achieve some goal somehow on the path of least resistance.

Such people often point out that their design is straightforward, simple and easy. Their biggest argument is that something is possible (*Look! It is possible to do this and that. So it must be good!*). If something is possible somehow, many people stop thinking and don't ask themselves if there is a better way to do it.

The possibility approach often works quite well for smaller programs, but when programs get bigger design problems start to emerge.

### 2.1.1 Metrics

1. Is it possible to create a solution in this way?

### 2.1.2 Frequently attributed as

1. hands on
2. straight forward
3. simple, easy
4. pragmatic
5. quick and dirty

## 2.2 Complexity or Big Boots Approach

Most people tackle complexity by adding even more complexity. They add software layers, libraries and tools to make things easier, but in the end it turns out, that the overall complexity got even bigger.

So why is this called the *Big Boots Approach*? Imagine you are hiking and get lost in a swamp. Now you could either admit, that you took the wrong way, turn around and walk some steps back or you could put on your big boots and continue walking, highly praising your boots. Due to some mysterious reasons in software industry wearing big boots is actually considered as being smart. So it might be a good idea wearing them :-) (There is also a nice SpongeBob episode called *Squeaky Boots*.)

### 2.2.1 Metrics

1. Lots of commands
2. Shortcuts for every task anyone could imagine.

### 2.2.2 Frequently attributed as

1. powerful
2. professional
3. works like magic (no one knows why magic works and if it works at all)

## 2.3 Usability Approach

I personally believe that legibility is one of the most important aspects in writing code because usually most of the time is spend maintaining the code and not writing it. In bigger projects writing a piece of code and maintaining it is often done by different people. So legibility comes to be even more important.

One way to write legible software is using the usability approach. Unfortunately when it comes to usability everybody has their own opinion. Many are surprised, that usability is actually quite well defined and based upon psychological research. Usable code satisfies the following criteria:

### 2.3.1 Metrics

1. simple
2. efficient

3. predictable
4. fault tolerant
5. transparent

### 2.3.2 Frequently attributed as

1. easy
2. user/programmer friendly
3. less complex
4. oversimplified

ui2go tries to implement usable code, because I believe that usable code is more important than simple, elegant or powerful code. However, often it just turns out to be simple, elegant and powerful.

# 3 Program Control

## 3.1 Control Flow

Programs for graphical user interfaces typically utilize some concept of control flow between the user interface and the actual program logic, often some kind of model view controller pattern.

However, when programs grow above a certain size the big picture usually looks like a plate full of spaghetti containing some classes as decoration.

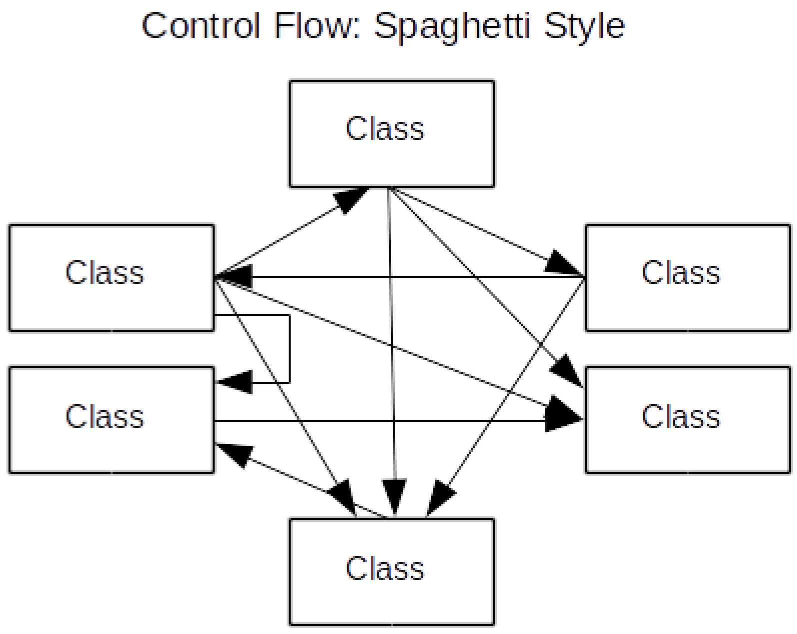

Figure 1: Control Flow in a common program. Even well designed programs have a tendency to end up looking like a plate full of spaghetti.

To help the programmer writing maintainable applications ui2go uses the metaphor of a circular control flow. In this style of programming user input is first directed to the graphical user interface (for example clicking a button) and than forwarded to the program logic. The program logic evaluates user input and updates the user interface, which is observed by the user.

ui2go supports this style of coding by implementing a predefined flow of events. The next section will go further into detail.

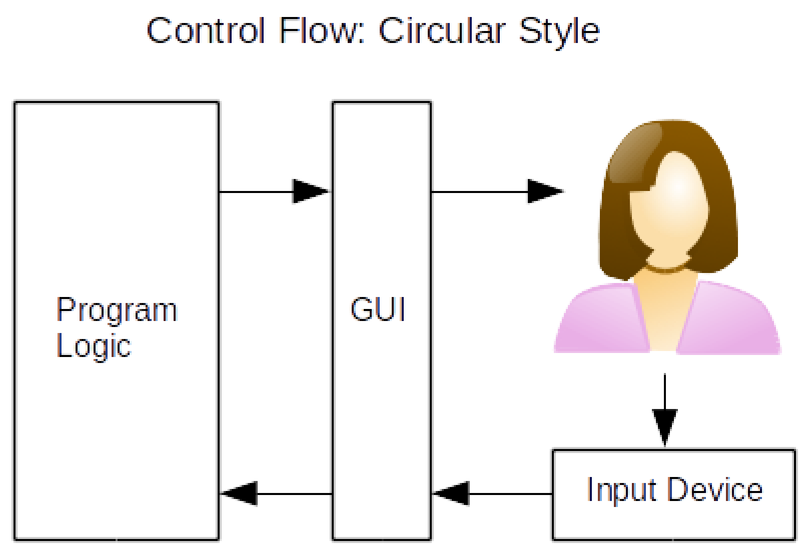

Figure 2: ui2go supports a circular control flow, where user input is transferred to the user interface and than to the program logic. The program logic updates the user interface, which is observed by the user.

## 3.2 Events

### 3.2.1 General Idea

In this world all complex organisms and societies have a hierarchical structure. Larger systems are composed of smaller units (usually serving a specific purpose) that are arranged in a hierarchical order. This seems very natural to us and so we created technical systems according to the same design. Here the units are often called modules.

The fact that hierarchical design is the result of millions of years of evolution leads us to the conclusion that this design is somewhat trustworthy. It is probably the best structure to reduce management complexity in larger systems.

It is no surprise that computer programs were modelled in a hierarchical way too. In the golden age of procedural programming a command was resembled by a procedure that called other procedures further down the hierarchy. This was called the top down approach in software development. The top down approach worked very well for a while, but programs had a tendency to grow in complexity and then graphical user interfaces entered the scene.

One particular annoying problem with GUI programs is the non linear control flow. The old style of programming worked very well for programs that had a well defined entry point and processed data in a linear way but in GUI programs user input can happen anytime anywhere and interrupt the linear program execution.

This gave rise to a new style of programming: object orientation and the massive use of events. This style is even closer to reality than top down procedural programming. In reality anything can happen at anytime (and it often does).

In real world hierarchical structures (biological organism, society, organisation, military,...) commands and events have distinctive meanings:

1. A command is given by someone who knows what to do. It travels downwards the hierarchy. Characteristics:

   - Receiver (object)
   - What to do (method name)
   - Additional information (method parameters)

   In object oriented programming a command is well resembled by a method.

2. An event is signalled by someone who does not know what to do. It travels upwards the hierarchy until it reaches some control instance who knows. An event often has multiple receivers. Characteristics:

   - Sender (object raising the event)
   - What has happened (kind of event)
   - Additional information (event parameters)
   - Time of the event. (This is only important is some cases, for example in distributed or multitasking environments where the original order of events might get lost)

   In object oriented programming events are often modelled by function calls, which gives a totally wrong idea from a semantically point of view.

Imagine a town in medieval times. There were watchtowers at frequent intervals to scan for enemies. The biggest enemy during these times was

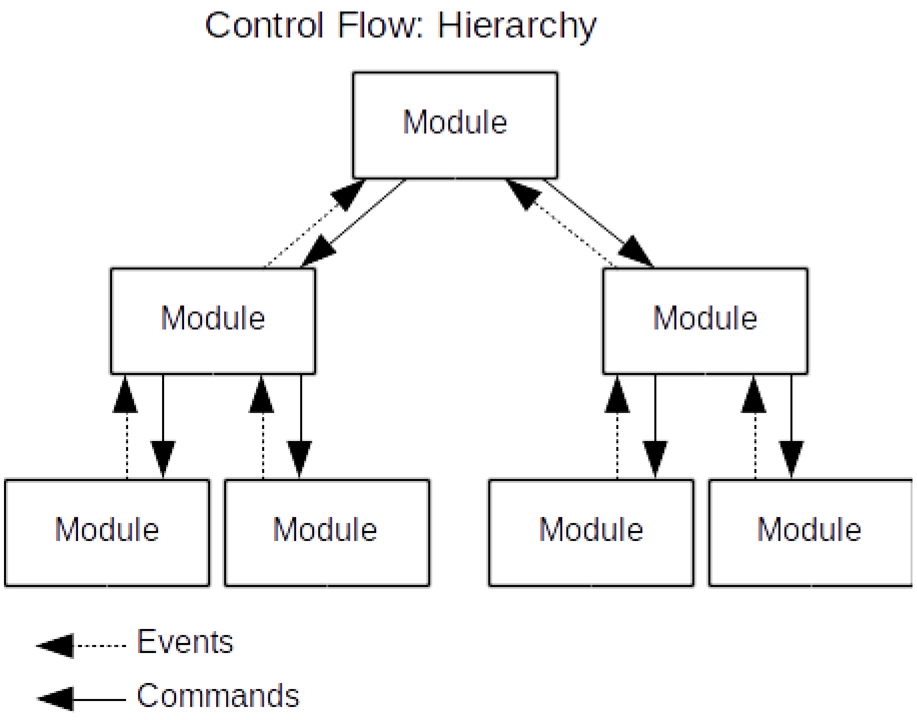

Figure 3: Control flow in a hierarchical system. Events usually move upwards the hierarchy and commands move downwards. If you don't like the term module you can easily substitute it with department, military unit or even class.

probably fire. Fires broke out frequently and often destroyed large parts of the town. The outbreak of a fire is a perfect example for an event.

When the fire was spotted by a watchman (event dispatcher), he usually shouted out loud *Fire!* (event propagation) to warn the people and to contact the fire department. At the fire department there was someone who know what to do and he ordered some people to put out the fire.

Now lets turn to a more modern time example. An aircraft enters an airspace. This event is monitored by several observers, a civilian air traffic controller and a military air traffic control operator.

The civilian observer might feel a strong desire to guide the incoming aircraft to safety, but if the pilot does not respond, he will assign control to some military facility.

On the other hand the military air traffic control operator might have observed the incident without the knowledge of the civilian controller and already has taken actions. Intercepting planes are on their way. In this case the message (event) from the civilian operator has no effect.

This example demonstrates that an event is no command and that it may be fatal to handle an event like a command. In many programming environments events are modelled as function calls and this suggests to start the control flow at the source of the event.

But building a chain of command starting from the incoming aircraft is like putting an enemy aircraft in control and that's clearly a bad idea.

I find it noteworthy that many textbooks about object oriented programming totally neglect the notion of a proper hierarchy or modules. They just deal with data structures (objects) and some design patterns that are suited for programming in the small often leaving the impression that objects happily communicate together on the same level.

But as programs grow larger, this approach no longer works and leads to a mess of object calls that are hard to follow. I guess this is one of the reasons why object oriented programs often gain a reputation of being hard to maintain.

### 3.2.2 Traditional Event Handling

When events were introduced for graphical user interfaces event handling was rather complicated. I showed some old style event handling code to a bunch of people and they all agreed it looked surprisingly similar to something like this:

```
ddkl?k$jj@d{alda
aldue)%%$kd§%hjg
df$!/kzhr)%§%@/}
```

Some even insisted that the above code actually is traditional event handling code.

It is easy to understand that nobody wanted to write code like this and programmers hated to write programs with a graphical user interface. They were in desperate need for a better solution and willing to take a bet on everything that was different from the old style. In this situation callback functions entered the scene (or at least the massive use of callbacks).

### 3.2.3 Callback Disaster

Because event handling was rather painful some smart people looked for a more convenient solution and came up with something like this (pseudo code):

```
// Handle click event from button
function onClick(event){
    // Do something useful with event
}

// Connect click event to function
button.onClick = onClick(event)
```

This looks easy, doesn't it? Every other programmer thought the same and event handling by using callback functions quickly became the standard pattern in GUI programming.

Unfortunately what looks good in the small often is not as good in the large and as programs got bigger some people started to realize that there are a lot of problems associated with callback functions:

1. GUI and program code get mixed.

   This makes it hard to separate the concerns of user interface design and programming. It is also very hard to try out another interface design, because design and program are hardwired together.

   A maintainer who tries to grasp the program's control flow has to scan tons of user interface code, which has nothing to to with the actual program logic.

2. Strong coupling between components.

   This is not only the case for the user interface and the rest of the program. There are usually many callback functions weaving many connections between different components of the program. Thus it is hard to maintain or to implement changes in a certain module.

3. Control flow is hard to follow.

   The introduction of many callback methods makes the program's control flow hard to follow. Events often belong logically together, but callback methods rip this apart, creating a huge semantic gap between human understanding and the code.

   Some of these problems are usually being tackled by using some kind of model view controller pattern, but bigger programs still look like a big mess.

4. Inverted control flow.

   When it comes to big programs the program maintainer often has nothing to do with user interface design. This makes his job especially hard, because he has to take care of classes with no well defined entry points. So there is no concept of a control flow, that is easy to understand and could be followed.

5. Interdependencies between callback methods.

   Callback methods often introduce interdependencies, that make it necessary to introduce additional class variables. This makes the code hard to understand and hard to maintain.

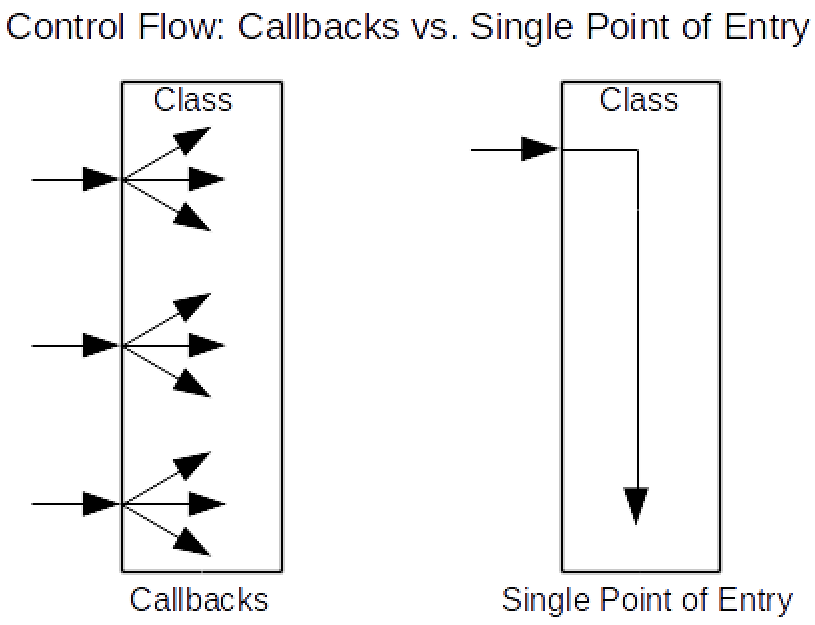

Figure 4: Control flow in a single class. The callback style is confusing and hard to maintain. A single point of entry enables a more traditional and easier linear control flow.

Now some people are looking for better solutions and try to get rid of callback methods by whatever means possible, sometimes even introducing new keywords to programming languages.

### 3.2.4 ui2go Solution

ui2go tries to solve the problems of callback functions by actually taking one step back (Remember the big boots approach?). It fits somewhere in between traditional event handling and callback methods, but in a more modern form. The key features of ui2go event handling are:

1. A well defined flow of events.
2. Entry points into classes for groups of events.
3. Use of channels to handle associated events in a linear style.
4. I wish, I could write *well structured events* here, but at the moment ui2go events are just a showcase.

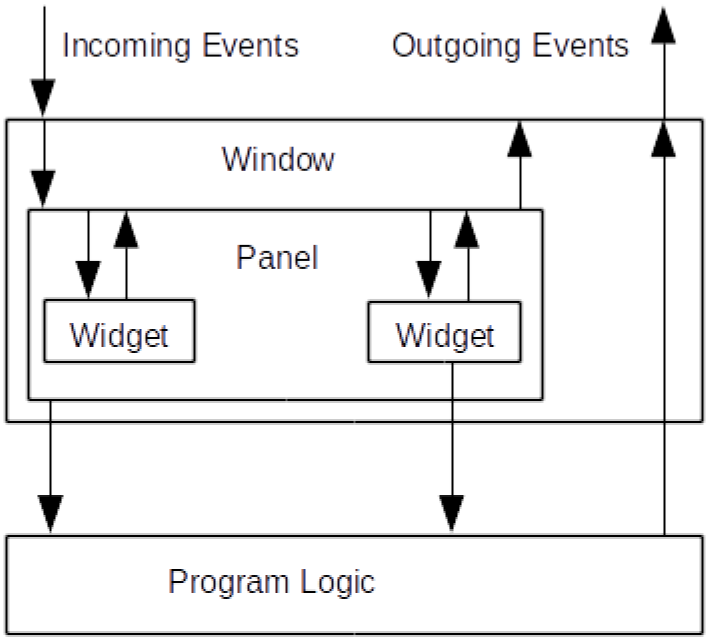

Figure 5: Event Flow in a ui2go program. The events flow from the window down to the child widgets and after processing again back up to the main window. At any given point the program logic may be hooked into the event flow.

**Example: Window Event Handling using a Single Method**

This example presents a simple program for painting onto a canvas. It shows, how mouse events are handled by a single method.

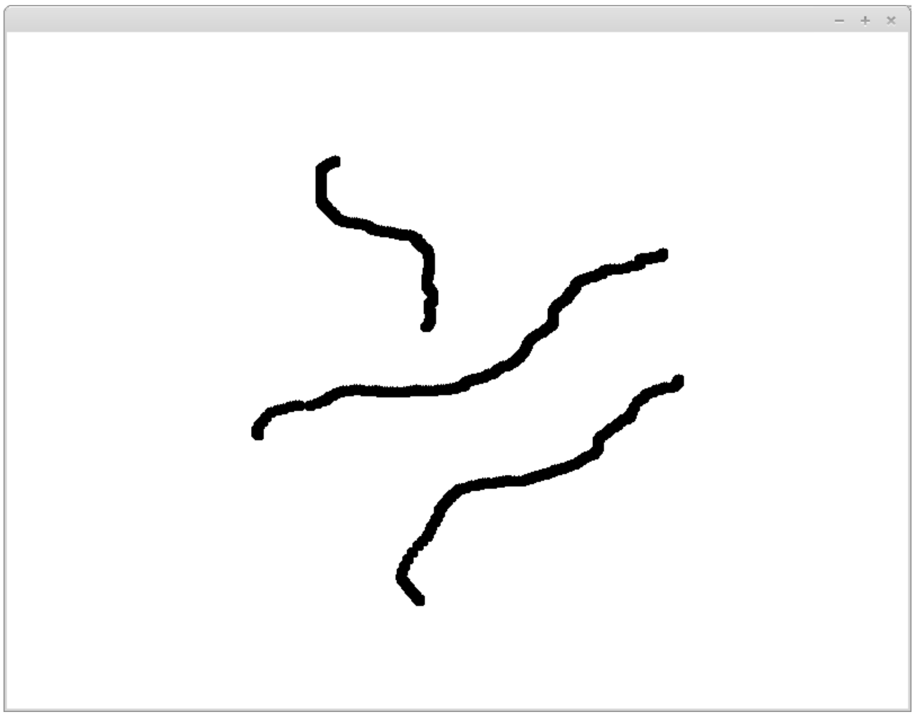

Figure 6: An extremely simple program for painting.

Here is the complete source code:

```
// Package 02-paint implements a simple example on how to use
// the mouse for painting on a canvas.
package main

import (
    "code.google.com/p/ui2go/event"
    "code.google.com/p/ui2go/widget"
    "image"
)

// onEvent handles all events, that are sent from components
// embedded into the main window.
func onEvent(canvas *widget.Canvas, evt interface{}) {
    switch evt := evt.(type) {
    case event.PointerEvt:
        switch evt.Type {
        case event.PointerTouchEvt:
            canvas.MoveTo(image.Point{X: evt.X, Y: evt.Y})
        case event.PointerMoveEvt:
            if evt.State == event.PointerStateTouch {
                canvas.LineTo(image.Point{X: evt.X, Y: evt.Y})
            }
        }
    }
}

func main() {
    win := widget.NewWindow()
    canvas := widget.NewCanvas()
    // Layout one component in the window.
    win.Addf("%c growxy", canvas)
    // Redirect all events from the main window
    // to the onEvent method.
    event.NewReceiverFor(win).SetEvtHandler(
        func(evt interface{}) { onEvent(canvas, evt) })
    win.Show()
    win.Run()
}
```

The line

```
    event.NewReceiverFor(win).SetEvtHandler(
        func(evt interface{}){ onEvent(canvas, evt) })
```

creates a new event receiver for all window events and sets the `OnEvent` method as it's event handler. The event handler is a function, that processes the received events.

This coding style works fairly well, when there are only a few events to manage, but when programs get bigger, it is more sensible to bundle associated events together and to put them into a channel.

When events belong together, it is often useful to push them into a channel. Retrieving events from a channel enables a linear coding style and eliminates the need for class variables to keep track of event interdependencies.

**Example: Event Handling using a Channel**

This is a more advanced painting program as an example:

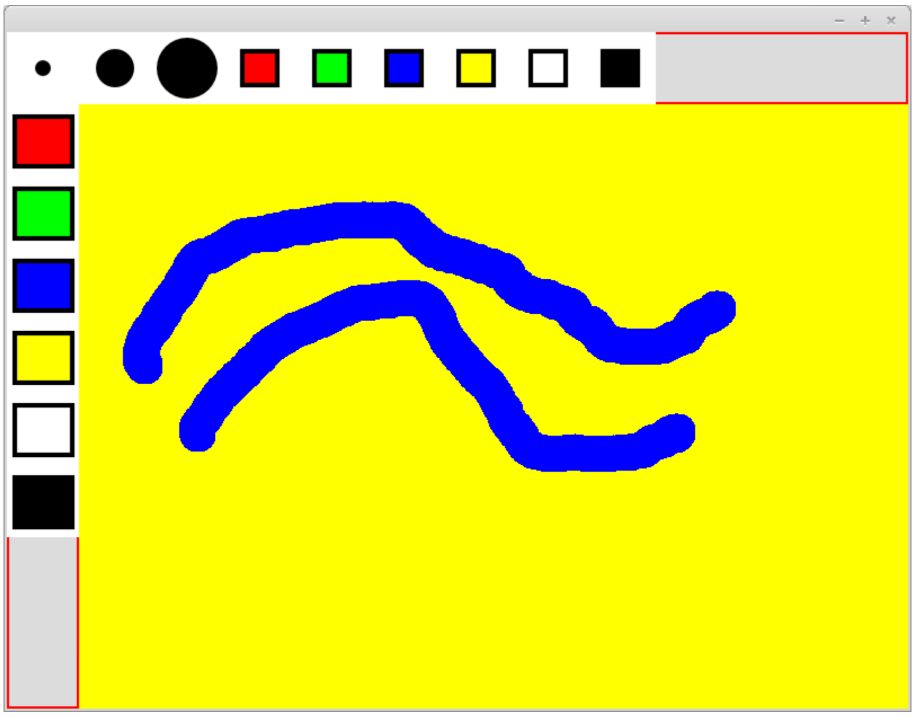

Figure 7: The paint program uses a channel to handle mouse events from the canvas.

An excerpt from the source code:

```
// Connect events from canvas to onMouseEventsFromCanvas method.
event.NewReceiverFor(canvas).SetEvtChanHandler(
    func(ec <-chan interface{}){
        onMouseEventsFromCanvas(ec, canvas) })

func onMouseEventsFromCanvas(
    ec <-chan interface{}, canvas *widget.Canvas){
    for evt := range ec {
        switch evt := evt.(type) {
        case event.PointerEvt:
            switch evt.Type {
            case event.PointerTouchEvt:
                canvas.MoveTo(image.Point{X: evt.X, Y: evt.Y})
            case event.PointerMoveEvt:
                if evt.State == event.PointerStateTouch {
                    canvas.LineTo(image.Point{X: evt.X, Y: evt.Y})
                }
            }
        }
    }
}
```

Connecting events to a function is nearly the same as in the previous example, but this time the event handling function has a channel argument:

```
event.NewReceiverFor(canvas).SetEvtChanHandler(
    func(ec <-chan interface{}){
        onMouseEventsFromCanvas(ec, canvas)})
```

This coding style has two major advantages: First, it is easy to process events in linear style by using the `range` operator. There is no need to react to external function calls, which makes the code very legible.

Second, there is no need for extensive bookkeeping by using class variables. Status information may be tracked by local variables, keeping the code much clearer. The `isDrawing` variable is such a case.

### Example: Old School Event Switching

Some people will wonder, how events are distinguished inside an event handler function. ui2go actually uses the old style of event switching:

```
event.NewReceiverFor(buttonPanel).SetEvtHandler(
    func(evt interface{}){ onCommand(evt) })

func onCommand(evt interface{}) {
    switch evt.Command {
    case "SmallBrush":
        canvas.SetBrushRadius(7)
    case "BigBrush":
        canvas.SetBrushRadius(27)
    }
}
```

Many programmers don't like this style. They find it ugly and inelegant. I agree to that view, when it comes to long `if else` statements, but I think Go `switch` statements are very legible.

But wouldn't it be much better to directly connect events to call back functions? Most GUI libraries use this new style of coding (again in pseudo code):

```
smallBrushButton.onClick = onSmallBrushClick(evt)
bigBrushButton.onClick = onBigBrushClick(evt)

function onSmallBrushClick(evt){
    canvas.SetBrushRadius(7)
}

function onBigBrushClick(evt){
    canvas.SetBrushRadius(27)
}
```

Despite it's widespread use, this style has a number of drawbacks:

1. The problems associated with callback functions that were already discussed in section 3.2.3.
2. Did I already mention, that interface design and program code often get mixed up?
3. Method inflation: Code often consists of an extensive amount of one liners, making it harder to maintain. This problem could be overcome by binding events directly to closures, but this is usually considered bad style, because methods must often be called from different locations in the program code.
4. Program semantics are torn apart. Logically associated things are no longer associated in the source code.

The old school code on the other side features some advantages:

1. Easy to read and maintain.
2. Easy to add scripting capabilities (just connect more event senders to the same event switch). The event switch already is some kind of simple interpreter.
3. Flexible code: For example, adding a new event handler just requires an extra case in the `switch` statement. It keeps the changes local to one class. The new style requires a new event connection and a new method. In bigger program this is usually done in different classes.

4. Network transparency could be achieved just by exchanging the event senders and listeners implementation.

# 4 Layout Management

When confronted with screens of different sizes and pixel densities pixel accurate layout usually results in disaster. Therefore some good method of laying out components on the screen is needed.

In most graphical user interfaces widgets are arranged in some grid like fashion. Distances are often the same for all screens of a single program. So one would expect, that the process of laying out components is highly automated. Unfortunately this is not the case. Especially in the main stream of software development, there is a tendency to change the notation (XML, JSON) without reducing the overall complexity.

The ui2go solution to component layout is a layout manager, called *Combigridlayout*. Combigridlayout was heavily inspired by MiG-Layout. It is not as powerful, but it basically shares the same layout philosophy and there are some important differences, that give it a quite distinctive flavour.

1. Combigridlayout is very small and easy to port.
2. The layout is done using printf-like layout strings.
3. Combigridlayout features an easy to use mock-up-mode, that makes it possible to test a layout without creating graphical components (widgets).

In ui2go Combigridlayout is the basic layout manager, that can be used for nearly all purposes. Components are laid out like printing lines onto a piece of paper using the Addf (add formatted) command. Here is a simple example, that adds a canvas to a window.

```
win := widget.NewWindow()

canvas := widget.NewCanvas()
win.Addf("%c growxy", canvas)

win.Show()
win.Run()
```

The layout work is done in the line

```
win.Addf("%c growxy", canvas)
```

The layout is specified by the layout string `%c growxy`. `%c` is a placeholder for the widget. `growxy` tells the layout manager to use all available space for the component.

One interesting aspect of the `Addf` command is the mock-up mode. It is legal to omit the component parameter. Combigridlayout then creates a dummy widget to provide a preview of the final layout.

Combigridlayout provides just a few basic layout commands, but in conjunction with layout containers, that may be arbitrarily nested, these commands are quite versatile:

| Layout Command | Meaning |
|---|---|
| %c | Placeholder for a component (widget) |
| growx | Eat up the available space in x direction. An additional integer parameter serves as a weight, when there are more than one grow commands. |
| growy | Eat up the available space in y direction. An additional integer parameter serves as a weight, when there are more than one grow commands. |
| growxy | Eat up the available space in x and y direction. This command always comes without a parameter. |
| spanx | Denotes how many grid cells in x direction are covered by the component. |
| spany | Denotes how many grid cells in y direction are covered by the component. |
| wrap | Start a new line |

## 4.1 Border Layout

This example shows the border layout, that is the basic layout for many traditional GUI programs.

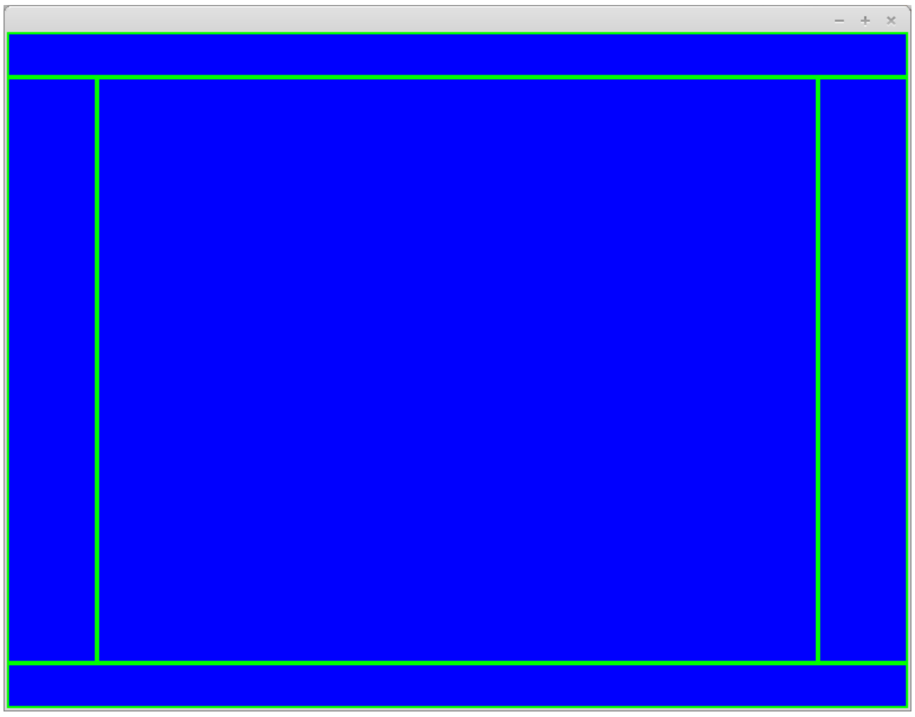

Figure 8: Classical border layout.

```
win := widget.NewWindow()

win.Addf("%c spanx 3 wrap")
win.Addf("%c %c growxy %c wrap")
win.Addf("%c spanx 3 ")

win.Show()
win.Run()
```

The easiest way to grasp the layout is to scan for the `%c` placeholders. In the above example there are three lines, the first containing one component, the second containing three components and the last containing one component.

In the lines with only one component, the component spans three grid cells (`spanx 3`). In the second line the middle component eats up all available space (`growxy`). So the other components are made as small as possible.

When creating a new layout it is recommended to first write some lines containing only %c and add the constraints thereafter.

## 4.2 Grid Layout

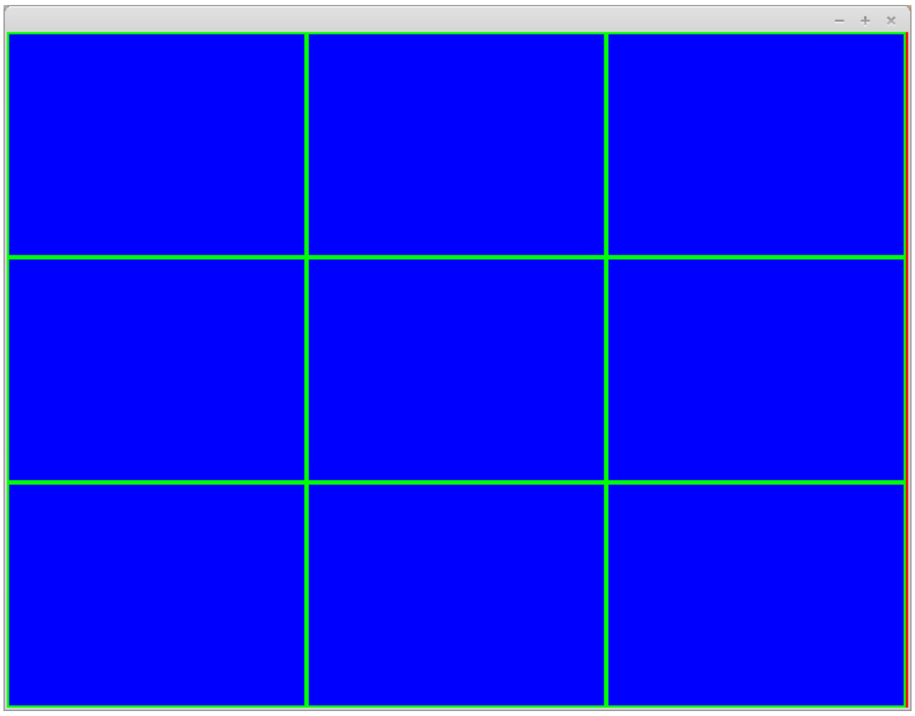

Figure 9: Grid layout with equally sized cells.

```
win := widget.NewWindow()

win.Addf("%c growxy %c growx %c growx wrap")
win.Addf("%c growy  %c       %c wrap")
win.Addf("%c growy  %c       %c     ")

win.Show()
win.Run()
```

The grid layout is somewhat similar to the border layout, but now there are three lines containing three widgets. All widgets should be equal in size. So the `grow` command is provided. Note, that for some components `grow` is not specified. The components automatically adapt to the grid size, which is already fully specified by other components.

## 4.3 Classical Input Mask

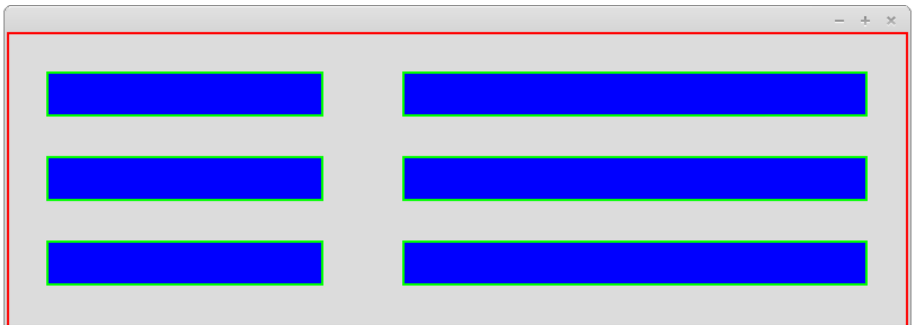
Figure 10: Mock-up of a traditional input mask.

```
win := widget.NewWindow()

gaps := widget.GridGaps{
    Top:            1,
    Begin:          1,
    End:            1,
    Bottom:         1,
    BetweenColumns: 2,
    BetweenRows:    1}.Unit(widget.Cm)
win.SetGaps(gaps)

win.Addf("%c growx 1 %c growx 2 wrap")
win.Addf("%c         %c         wrap")
win.Addf("%c         %c")

win.Show()
win.Run()
```

The input mask example features one interesting topic: grid gaps. Usually distances to the window border and between components are always the same. So it is easy to specify them for the whole window using `SetGaps`.

# 5 Font Size

## 5.1 Distance Dependency

When creating graphical user interfaces one recurring question is about what font sizes to use and what unit of measure is appropriate? Many people still stick to pixel units, but as pixel densities vary greatly across different devices (100 to 300ppi is quite common) it became clear that pixel size is not a good solution. Some companies introduced device independent pixels, but that is just the same as introducing a new length unit like meters.

The real problem is that there is an increasing amount of different devices around like smartphones, computer screens and digital billboards, that are all viewed from different distances. The perceived size of something depends on the viewing distance and this must be taken into account.

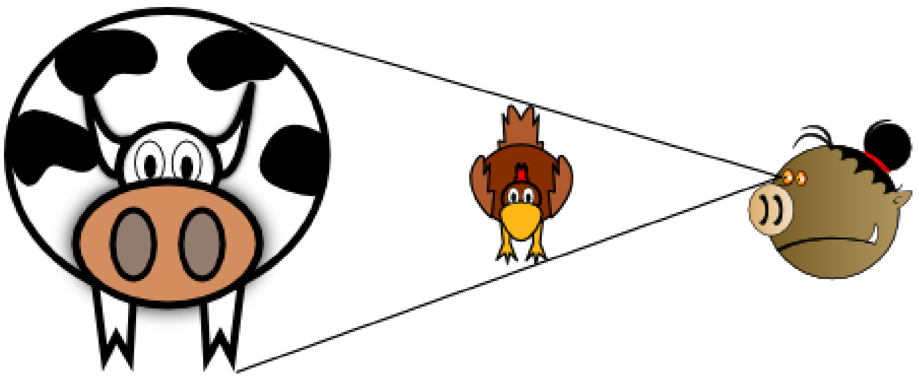

Figure 11: A chicken may be a small animal, but viewed from a nearby distance it seems as big as a cow.

## 5.2 Font Metrics

Body height is defined as the height of the whole font, including ascenders and descenders. In the days of metal type, the metal body of a type had to be large enough to contain the body of a font. Due to mechanical restrictions it also provided some extra space. So type height is a bit larger than body height. In CSS the unit *em* is used for body height.

There is no generally accepted definition of font size. Some people refer to it as the type height, some as the body height and others as the cap height. As far as I know differences between type size and body size are so small that they can be safely neglected here. When I use *font size* in this document I mean *type size* as this is common in DTP programs.

Two other important metrics are cap height and x-height. Cap height is the height of the capital letter *H* while x-height is the height of the small letter *x*. Cap height is important, because it can easily be measured and some experts have used it to recommend ergonomic font sizes. Other people point out that x-height is more important for readability, but as I only know recommendations for cap height, I will stick to cap height instead of x-height throughout this document.

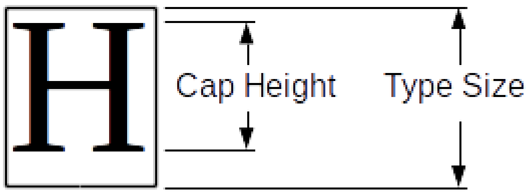

Figure 12: Type size of a font. The type size is larger than the size of a capital letter.

The size of a font is usually measured in point (pt). There are a lot of different point units, but nowadays the most commonly used is the DTP point.

$$1 point = 0.353 mm$$

Many traditional fonts were created according to the same rules of typesetting. Fonts adhering to the American Pica system often satisfy the following equations:

$$cap\ height = type\ size \cdot 0.71$$
$$type\ size = cap\ height \cdot 1.41$$

From this formulas we can calculate the cap height for some type sizes and get this nice table, which will come in handy later:

| Type Size (pt) | Cap Height (pt) | Cap Height (mm) |
|---|---|---|
| 8 | 5.7 | 2 |
| 9 | 6.4 | 2.25 |
| 10 | 7.1 | 2.5 |
| 11 | 7.8 | 2.75 |
| 12 | 8.5 | 3 |
| 13 | 9.2 | 3.25 |
| 14 | 9.9 | 3.5 |
| 15 | 10.7 | 3.75 |
| 16 | 11.4 | 4 |
| 17 | 12.1 | 4.25 |
| 18 | 12.8 | 4.5 |
| 19 | 13.5 | 4.75 |
| 20 | 14.2 | 5 |

LaTeX uses a font size of 10pt as a standard for books and this size is commonly used today. 8pt has traditionally been the smallest size used for book printing.

Remember that the above table is not valid for all fonts. The actual cap height depends on the individual font. Especially fonts designed for reading on a computer screen often utilize larger cap and x-heights to increase readability.

## 5.3 Viewing Distances

Lets take a look at some typical viewing distances for different activities:

| Activity | Viewing Distance |
|---|---|
| Book reading | 30 to 40cm |
| Working at a desktop computer | 60 to 80cm |
| Looking at a billboard | many meters |

When people are reading a small book they often get immersed and hold it quite near before their eyes. 30cm is the distance used by ophthalmologists, when conducting tests with the Amsler grid. It is also the nearest point people in the mid forties typically can focus. This is why older people usually need glasses for reading.

60 to 80cm as a viewing distance for desktop computers may seem quite large to many readers, but it is commonly recommended, for example by the German VBG (Verwaltungs Berufsgenossenschaft). The reason for such distances is to reduce strain on the eye and to enable a healthy and relaxed

working position.

One important lesson is, that font sizes for desktop computer screens should be about twice as large as font sizes for books. The smallest print used for books throughout the centuries has been 8pt (2mm cap height). So the recommended minimum font size for effortless reading from a screen is 16pt (4mm cap height). Some people may think, that this is ridiculously large, but viewed from the recommended distance a 16pt font on a screen appears the same size as an 8pt font in a book.

## 5.4 Recommended Font Sizes

Experts have conducted experiments to find out the optimal font sizes for reading. The German VBG (Verwaltungs Berufsgenossenschaft) recommends a cap height of 22 to 31 minutes of arc. For typical viewing distances this results in the following table:

| Viewing Distance | Min Cap Height | Max Cap Height |
|---:|---:|---:|
| 30cm | 1.93mm | 2.72mm |
| 35cm | 2.26mm | 3.18mm |
| 40cm | 2.58mm | 3.64mm |
| 50cm | 3.23mm | 4.55mm |
| 60cm | 3.87mm | 5.45mm |
| 80cm | 5.16mm | 7.27mm |
| 1m | 6.45mm | 9.09mm |

This does not mean that smaller sizes are unreadable, but smaller sizes are harder to read and require more time. The same goes for fonts that are too large.

## 5.5 Cap and Rem Units

ui2go tries to make life easier for programmers, so that they don't have to think much about different font sizes. Therefore it introduces two additional units of measure (sorry for that): cap and rem. Cap is the recommended height for capital letters in a standard font and Rem is the recommended size of em (or type height as I assume, that they are nearly the same).

$$1cap = viewing\ distance(in\ mm) \cdot 7/1000$$

$$1cap = 24minutes\ of\ arc$$

$$1rem = viewing\ distance(in\ mm)/100$$

The standard layout manager supports both units, so that they can be used directly. But does this really work in practice? Here are some examples to give you the idea:

1. For a viewing distance of 30cm one cap results to 2.1mm. That's slightly larger than the smallest size used in traditional print.

   30cm is also a typical smartphone distance and one rem at this distance is 3mm. This corresponds to 16dp (device independent pixel) on an Android device. It is the typical type size for text on a button.

2. Imagine yourself sitting in a diplomatic conference, important papers lying in front of you on the table. Of course you don't want to pick up the papers and hide yourself or even worse, bow down (in front of whoever) to read them.

   What would you do? The typical viewing distance is about 50cm, the result for one cap is 3.5mm. This corresponds to a 14pt font. It comes as no surprise that US diplomatic documents actually use 14 point fonts.

3. At 60cm we get a cap height of 4.2mm, a typical value recommended for effortless reading on a desktop computer screen.

4. When looking at a billboard viewing distances are typically many meters. At 100 meters one rem (type size) is 1 meter. This translates roughly to 30 inches of cap height for a distance of 300 feet.

   This is the well known rule of thumb known to US billboard designers: 1 inch of letter height provides best impact at a distance of 10 feet.

Conclusion: Caps and rems really do the trick and might prove as a valuable help for interface design in the future.