GIRAF Design Manual

Manual for the design of applications in the Giraf software suite

Aalborg University

Contents

Preface

1. Terms

The following terms are used throught this design manual.

1.1 *GIRAF* Software Suite

When referring to the "*GIRAF*-software suite" throughout the manual, we mean the entire *GIRAF* project being developed by the current Software 6 semester at Aalborg University. This extends to all the individual applications and their dependency applications being developed, which at the current time of writing includes:

- Administration (Profile Manager)
- Giraf (Main Launcher)
- Kategorispillet (Category Game)
- Kategoriværktøjet (Category Manager)
- Livshistorier (Life Story)
- Pictooplæser (Picto Reader)
- Pikto søger (Picto Search)
- Piktotegner (Pictocreator)
- Sekvens (Sequence)
- Stemmespillet (Voice Game)
- Tidstager (Timer)
- Ugeplan (Week Schedule)

1.2 Contextuality

Something is contextual if it is dependent on the content of the current window. A contextual button is a button that does only apply some action on a sub part of the current window. If you have a collection of images and the user selects abitrary many of these images to delete them. Then the button to delete the images is contextual.

1.3 Screen Real Estate

The screen real estate or screen estate is the amount of space on the display that is available for some application.

2. Box Model

This design manual utilizes the standard box model that consists of content, padding, border and margins as seen in Figure 2.1. The main difference to notice is that *margin* is the outer spacing and that *padding* is the inner spacing on elements.

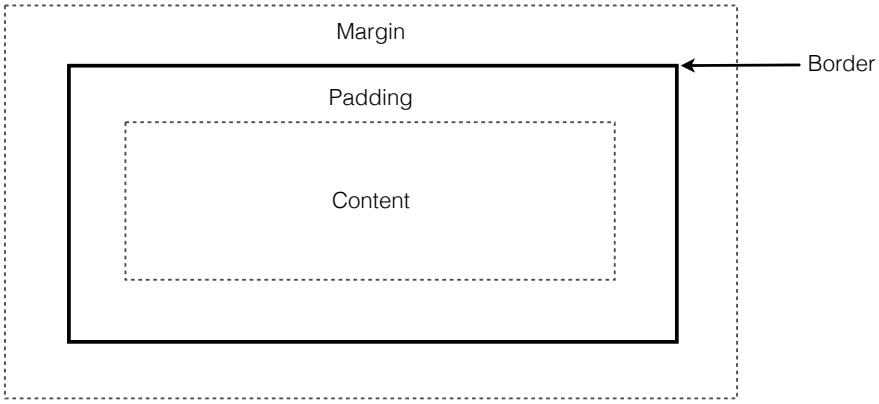


Figure 2.1: The Box Model

One should use *margin* when spacing between elements is desired, and if one would like spacing from the content of the element to the border of the element, *padding* should be used.

II Core Elements

3. Icons

Throughout the *GIRAF*-software suite different icons will be used to reference different applications and functionalities. Refer to the sections below to see how and when to use these icons.

3.1 *GIRAF* Software Suite Icon

The different applications in the *GIRAF* software suite may want to refer to the software suite itself. To do this, the overall application icon seen in Figure 3.1 can be used. This icon will also be used for indicating activity when the *Launcher* starts up. Other activity indicators is often displayed as progressbars (see Chapter 11).

Figure 3.1: Overall application icon for the *GIRAF* software suite

3.2 Application-specific Icons

Each application in the *GIRAF* software suite must have its own icon. This icon should reflect the content and functionality of the application and must not be ambiguous. All application icons should furthermore use the icon-base seen in Figure 3.2.

Rendered icons must be available in sizes defined in the Android Iconography article[1]. The foreground of the icon must be clearly visible in any of these sizes. Furthermore, the icon must not appear smudged or otherwise distorted in any scale.

Figure 3.2: Base for all application specific icon

[1] http://developer.android.com/design/style/iconography.html

3.3 Icons used to represent functionality

Icons may be used in buttons (See `GirafButton` in *GIRAF Components*) to represent functionality. The foreground of these icons must be gray-scaled and be quadratic (square). The functionality that the icon represents should be reflected in the icons itself. The icons should use the icon-base seen in Figure 3.3.

> **Note 3.3.1** Please be aware that the actual icons should not be designed/rendered using the icon-base. Instead use the `GirafButton` from the *GIRAF Components* library. This will allow you to only design the actual foreground and not worry about the background (base).

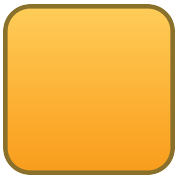
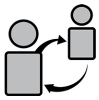
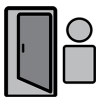


Figure 3.3: Base for all functionality-specific icons

3.4 Icons and when to use them

This section will describe the different icons that must be used throughout the applications in the *GIRAF* software suite. Please notice that icons in this section are not displayed on a button-like background. Icons may be used in different contexts if necessary.

3.4.1 User Management

Switch User: Indicates that the current profile can be switched	
Back: Indicates that the user can go back	
Log out: Indicates that the user can log out of the application	

3.4.2 Camera

Camera: Indicates that a camera can be used. For instance to take a picture	
Switch Camera: Indicates that the camera used can be switched. For instance from rear camera to front camera	

3.4.3 Microphone

Microphone: Indicates that the microphone in the tablet may be utilized. For instance for recording speech 

Microphone (off): Indicates that the microphone is unavailable or disabled 

Microphone (on): Indicates that the microphone is currently in use 

3.4.4 Media

Play: Indicates that something is playing 

Record: Indicates that something is recording 

Stop: Indicates that either something playing or recording can be stopped 

3.4.5 Arrows

There are no current usecase for these arrows, consider removing them from the set of available icons.

Arrow (down): X 

Arrow (left): X 

Arrow (right): X 

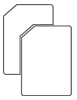
Arrow (up): X 

3.4.6 Version control

Undo: Indicates that an action can be undone 

Redo: Indicates that an undone action can be redone	
Synchronize: Indicates that data can be synchronize to/from the server	
Save: Indicates that actions performed can be saved	
Cancel: Indicates that some action can be canceled	

3.4.7 Data Management

Add: Indicates that a new instance of something can be created	
Rotate: Indicates that something can be rotated	
Resize: Indicates that something can be resized	
Copy: Indicates that something can be copied	
Edit: Indicates that something can be edited	
Delete: Indicates that something can be deleted	
Change Pictogram: Indicates that a pictogram can be changed or set	
Search: Indicates that a search-action can be performed	

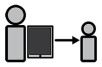
3.4.8 Utility

Help: Indicates that the user might seek help 

Settings: Indicates that the user might adjust settings 

Landscape to Portrait: Indicates that the orientation of the tablet can be changed from landscape mode to portrait mode 

Portrait to Landscape: Indicates that the orientation of the tablet can be changed from portrait mode to landscape mode 

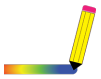
Give Tablet: Indicates that the tablet will be given to another user 

3.4.9 Miscellaneous

Accept: Indicates that something is accepted. For instance when user is presented with a confirmation 

Choose: Indicates that the user should make a choice 

Color Background: Indicates that colors of the background of an element can be specified 

Color Stroke: Indicates that colors of the stroke of an element can be specified 

Eraser: Indicates content can be erased 

4. Image Representation

For the following sections there exist a shared component called `GirafPictogramItemView` which assists one to comply with the following design rules, for a guide on how to use this class see Appendix D.

4.1 Appearance

In the *GIRAF* software-suite there exist various types of images, such as pictograms and profile pictures. Images should be quadratic, have a white background (Color 3.1) regardless of content and have a black border (Color 3.2), as seen in Figure 4.1. The image view should have a 10 dp padding so that the content gets spacing to the border as seen in Figure 4.2.

(a) Pictograms (b) Profile pictures

Figure 4.1: Image representation

Note 4.1.1 Profile pictures, as seen in Figure 4.1b, should, as the first profile (Jacob), fit to the canvas. However, many profile pictures are captured in a different way as seen in the second profile (Peter). It is desired that in the future, when profile pictures are captured, that they only allow for pictures that have the layout of the first profile. For the sake of convenience and backwards compatibility other sizes are allowed.

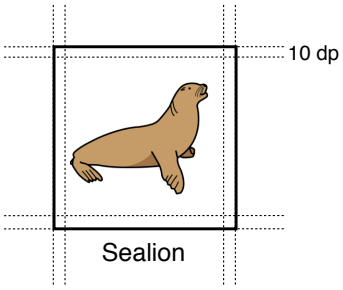
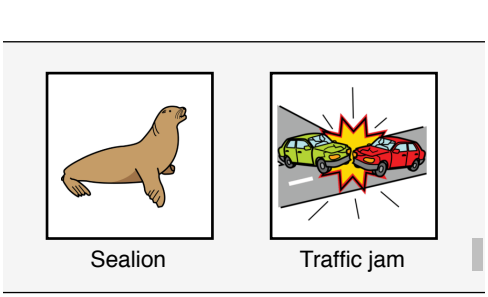


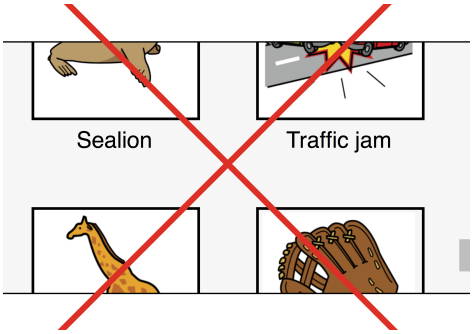
Figure 4.2: Image representation

4.1.1 Citizen view

An image is not allowed to be incomplete when shown to a citizen. See Figure 4.3 for an example of how an image should be shown to citizens and an example (Figure 4.3a) of how it should not be shown to citizens (Figure 4.3b).



(a) Correct citizen appearance



(b) Wrong citizen appearance

Figure 4.3: Citizen appearance explanation

Update colors used to mark selection in the figure to the correct colors

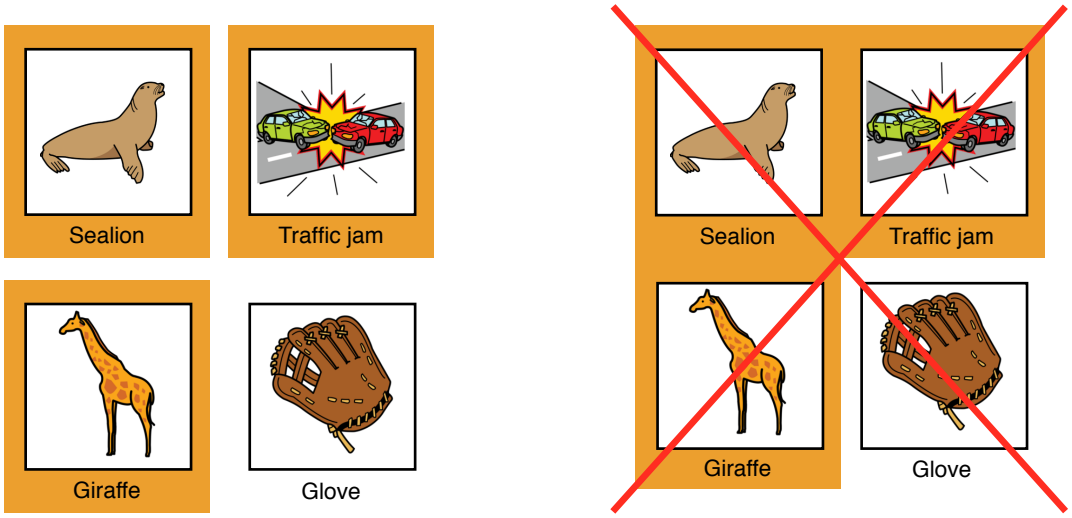
4.2 Marking

It is often possible to mark images throughout the suite, and whenever this happens one should mark the background of the selected item with an orange color (Color 3.3), as seen in Figure 4.4.

> **Note 4.2.1** It is important that when having multiple marked images that there are some margin between these elements. This should be implemented as seen in Figure 4.4a, and not as seen in Figure 4.4b.

4.3 Indicator overlay

If one wants to indicate that an image is editable, for instance when picking an icon for a category, the image should have an overlay indicating this as seen in Figure 4.5a. Other types of indicators should look similar, e.g. when indicating that an image is a placeholder for a category it is indicated as seen in Figure 4.5b.



(a) Corrected marking

(b) Wrong marking

Figure 4.4: Marking of images

(a) Editable indicator

(b) Category indicator

Figure 4.5: Indicator overlays

5. Typography

5.1 Fonts, Sizes and Colors

When creating text fields in the *GIRAF* software suite, is it imperative that they look alike across the entire suite. For this reason one should at all times use the default font-type in the applications.

The size of any text across the applications should always be given in the scale-independent pixel unit “sp”, and should always be readable on both 7 and 10 inch tablets. It is up to the developers of the individual applications to decide upon a specific text-size, but it should fit the general layout of the suite.

All text should by default be black (Color 1.1) across the suite when placed in buttons, views, etc. When creating hint text, e.g. when there are no applications present in the launcher, one should use a gray text-color (Color 1.2). When using the showcase library to highlight features, the text color of the help text should be white (Color 1.5), so it is readable on the dark overlay.

Insert a preview of the font

6. Tone of Voice

When displaying messages to the user the following rules must be respected.

6.1 Short and Precise

All messages must be short and precise. If a part of a sentence is redundant, remove it. However, you must provide enough information to cover the whole situation.

■ **Example 6.1 — Correct use.** The application can only be started from a guardian profile

■ **Example 6.2 — Incorrect use.** The application could not be started

■ **Example 6.3 — Incorrect use.** The application could not be started unless you start it from a guardian profile. Please log onto a guardian profile to gain access to this application.

6.2 Addressing Users

When referring to people with Autism Spectrum Disorder (ASD), use the word *citizens*. When referring to either institutional- or legal guardians, use the word *guardian*.

■ **Example 6.4 — Correct use.** The application cannot be started from a citizen-profile

■ **Example 6.5 — Incorrect use.** The application cannot be started from a child-profile

7. Application structure

7.1 Top Bar

All activities in every application, except the home-activity in the Launcher-application, must have a top bar. This top bar should have an orange gradient (Color 4.1) as in the example in Figure 7.1. This top bar should provide a short and simple title with enough information to allow the user to know where in the application he or she is. The height of the top bar must be $56dp$. Furthermore, this top bar must include at the two buttons described in Section 7.1.1 and Section 7.1.2. A guide for customization of the action bar can be seen in Appendix A.

Figure 7.1: Top bar example

> **Note 7.1.1** Such a top bar is easily achieved by letting all of your `Activity` extend the `GirafActivity` from the *GIRAF Components* library. Doing this will also implement the back button described in Section 7.1.1.

7.1.1 Back Button

The left-most button in the top bar must be a back button. This button must have exactly the same functionality as the back button on all Android devices. The icon of this button must be the back icon as described in Section 3.4.1.

7.1.2 Help Button

The left-most button in the right side of the top bar bar must be a help button. This button must provide the user with some useful help information regarding the current screen which the user is presented with. For instance if the user is assigning applications to users, the help might be a guide on how to do this correctly.

7.2 Side Bar

Side bars need not, but may be contextual. Side bars may be used to switch between content of applications - for instance if the application is split into two parts (side bar and content). A side bar should have a gradient a orange gradient (Color 4.1) and look like Figure 7.2. Side bars may not use more than 20% of the total screen estate. When using a side bar it must appear on the left side of the screen. The height of the side bar must be exactly the same as the container that it is inside. Most collections of items must have margins between them, the sidebar however is an exception regarding margin, meaning that there should be no margin between the markings of the items in the sidebar.

Note 7.2.1 When using the *GIRAF Components* library one can access the layout resource `R.layout.giraf_sidebar_layout` and inflate it. The parent of this is a `RelativeLayout`.

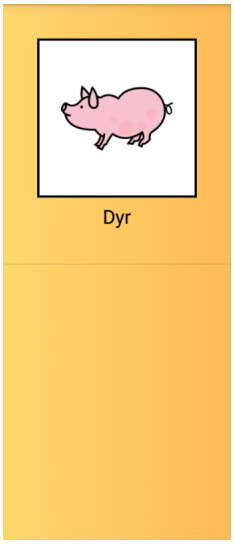


Figure 7.2: Side bar example

7.3 Bottom bar

Bottom bars must be entirely contextual and depend on the current content displayed in the current activity. A bottom bar should have a gradient a orange gradient (Color 4.1) and look like Figure 7.3.

Note 7.3.1 When using the *GIRAF Components* library one can access the layout resource `R.layout.giraf_bottom_layout` and inflate it. The parent of this is a `RelativeLayout`.

Figure 7.3: Bottom bar example

OBS: The gradient is in the wrong direction. It should be light in the bottom darker in the top.

7.4 Content

The main content of applications should be focused in the center of the layout and any menu bars should be above, under, and to the sides of the main content.

Note 7.4.1 We recommend using Android `Fragment` instances to manage content of an `Activity` if the main content of an Android `Activity` needs to change between different content that needs to be controlled differently.

7.5 Combinations of Bars

You might want to combine usages of the bars mentioned in the previous sections. This section will describe the different possible combinations of bars.

7.5.1 Combination #1

This is the most commonly used “combination” of bars (just the top bar actually). Since the top bar is required for all layouts this will be the base for the next combinations. Figure 7.4 shows an illustration of the layout using only the top bar. Please notice that the dark gray area is the top bar while the lighter gray is the actual content of the activity.

Figure 7.4: Layout example with only the top bar

7.5.2 Combination #2

If the content of the application needs to be switched out or replaced at some point, one may want to use a side bar. Figure 7.5 shows an illustration of such a layout using a top (darkest gray) bar and a side bar (dark gray). The content of the application is the light gray color.

Figure 7.5: Layout example with both a side bar and a top bar

7.5.3 Combination #3

If the content of the application needs to be switched out or replaced at some point, one may want to use a side bar. If the content that is replaced needs to be managed somehow, a contextual bottom bar can be used. Figure 7.6 shows an illustration of such a layout using a top (darker gray) bar, a side bar (dark gray), and a bottom bar (darkest gray). The content of the application is the light gray color.

Figure 7.6: Layout example with both a side bar, a top bar, and a bottom bar

7.6 Clickable Elements

All elements that are clickable must have a safety-distance to other elements. This will ensure that the user does not accidentally press the wrong thing and ultimately does something wrong. This safety distance may be achieved using several different methods. Please refer to the following sections. Figure 7.7a shows an example of correct item spacing while Figure 7.7b shows an example of incorrect item spacing.

Elements must have a safety distance to ...

- Other clickable elements
- Borders of it's container
- Borders of the tablet

(a) Correct item spacing

(b) Incorrect item spacing

Figure 7.7: Examples of correct and incorrect element spacing

7.6.1 Element Margin

Elements may be spaced apart from each other using margin on the individual elements. The distance between the elements should be consistent throughout all activities of any given application. Figure 7.8 shows an example of the margin for a given element..

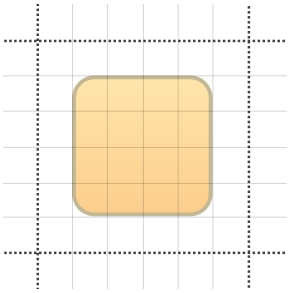


Figure 7.8: Example of element margin

Note 7.6.1 If margin is used inside a container each element with margin will also be a certain distance from the borders of that specific container. If, for instance, an element has a margin of 10, then this element would be a distance of 10 from the borders of the container. This means

that there *might* not be need for any padding on the given container.

Consistent Margin

Elements of the same type appearing in the same context must have the same distance to other elements. However, if the elements appear in an order, for example a horizontal list, the first and last element may differ. For instance, the first element may have a smaller left-margin and the last element may have a smaller right-margin. Figure 7.9 shows an illustration of this example.

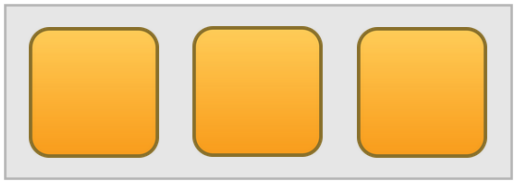


Figure 7.9: Example of element margin

7.6.2 Container Padding

Each container should provide some padding for its content. This padding should be somewhat identical to the spacing between elements inside the container. Figure 7.10 shows an example of a container with padding.

Figure 7.10: Example of a container with padding

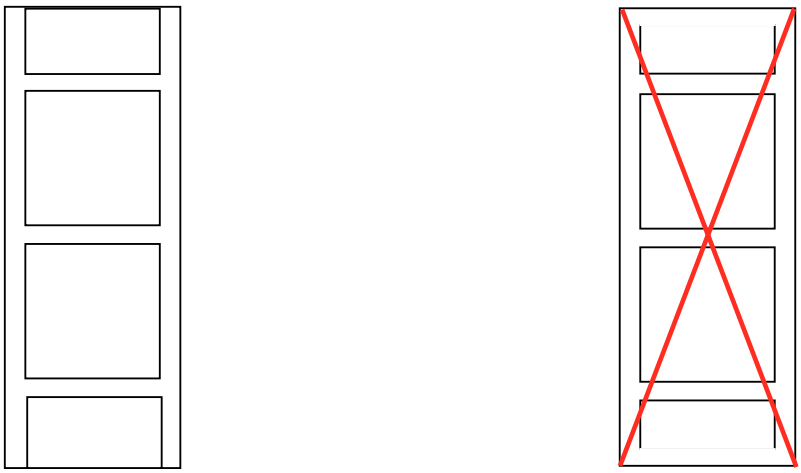
Indsæt indhold i figuren

Note 7.6.2 Whenever the content of the container can be scrolled through (for example a `GridView`) a property called `clipToPadding` must be set to `false` as seen in Figure 7.11a. Please refer to the Android documentation for additional information.

7.7 Item familiarity

It is important that the users are familiar with the items they are manipulating. One should make sure that a given item looks the same when navigating, creating and editing. For instance a category is displayed by an icon and a title below it. In the cases of creating and editing a category the layout and graphical presentation should be similar.

Insert image of the three phases of a category: create, display, update



(a) `clipToPadding` set to false

(b) `clipToPadding` set to true

Figure 7.11: Illustration of the property `clipToPadding`

8. Colors

Throughout this chapter a single color in each entry will denote a solid color, while two colors in an entry will denote a gradient between the two colors.

8.1 Text and Background

These colors should be used throughout any application.

1.1	Regular text-color	#000000
1.2	Text-color used to indicate placeholder or hint-texts	#AAAAAA
1.3	Background-color for any applications window background	#000000
1.4	Background-color for any activity	#E9E9E9
1.5	Showcase help text-color	#FFFFFF

8.2 Buttons

All buttons in the *GIRAF* software suite should use these colors for buttons. Please note that all gradients defined below are from top to bottom. Also note that the colors for the disabled button must be slightly transparent (65%).

2.1	Regular button background	#FFCD59	#FF9D00
2.2	Regular button stroke/border		#8A6E00
2.3	Pressed button background	#D4AD2F	#FF9D00
2.4	Pressed button stroke/border		#493700

2.5	Focused button background	#FF9D00	#FF5900
2.6	Focused button stroke/border		#8A6E00
2.7	Focused button background	#FAD355	#FEBE40
2.8	Focused button stroke/border		#E4AE4E

8.3 Images

3.1	Image background-color	#FFFFFF
3.2	Image border-color	#000000
3.3	Image marking-color	#FED76C

8.4 Bars

All applications that using bars must use the following colors.

4.1	Background of any bar	#FDBB55	#FED76C
4.2	Stroke/border of the bar		#E5BE53

Note 8.4.1 The gradient for the topbar is from top to bottom.

Note 8.4.2 The gradient for the side is from left to right.

Note 8.4.3 The gradient for the bottom is from bottom to top.

8.5 Week indicators

These colors must be used whenever a certain weekday is referenced. Note that these colors are primarily used to increase the usability for citizens because they are already familiar with this color scheme.

5.1	Monday	#007700
5.2	Tuesday	#800080
5.3	Wednesday	#FF8500

5.4	Thursday	#0000FF
5.5	Friday	#FFDD00
5.6	Saturday	#FF0000
5.7	Sunday	#FFFFFF

8.6 Page Indicator

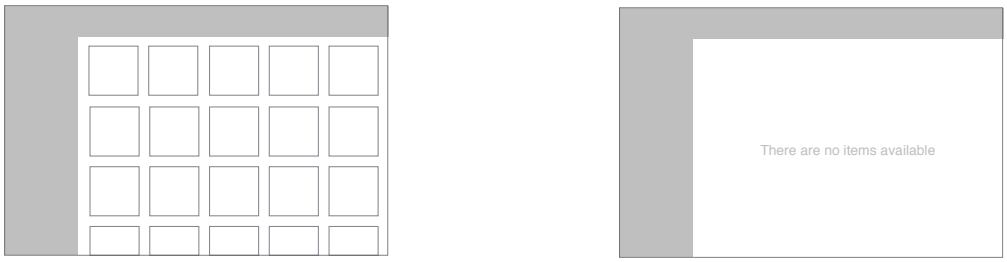
These colors must be used for indicating which page the user is currently on.

6.1	Active page	#FF9D00
6.2	Inactive page	#FFCD59

9. Item collections

9.1 Empty content indicator

If the content is a dynamic collection of items and that collection is empty one should indicate that there are no items avaiable. This indicator should be a text with a gray text-color (Color 1.2) like seen in Figure 9.1b.



(a) Content with items

(b) Content with no items

Figure 9.1: Empty content indicator

9.2 Overscroll

When a view has a collection of items and the user tries to scroll further than there are elements, a gray shadow should be display as seen in Figure 9.2.

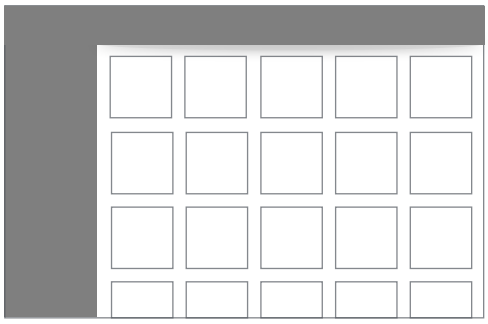
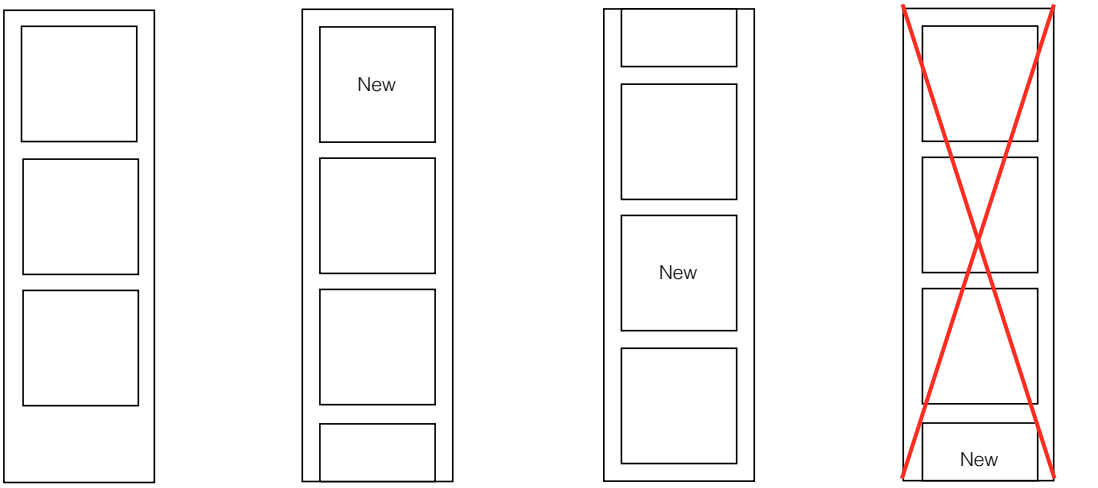


Figure 9.2: Content with overscroll shown in top

9.3 Updates in the collection

When updating a collection of items one should notify the user that some change in the collection have happened. There are two legal ways to this:

1. Add the item as the first element in the collection as seen in Figure 9.3b.
2. Move the view of the collection to place where the item is added as seen in Figure 9.3c.



(a) Item collection before update

(b) Item collection after update option #1

(c) Item collection after update option #2

(d) Wrong item collection af update

Figure 9.3: Update of item collection

9.3.1 Adding items

Describe what icons should be used to add items to a list. Sekvens, Ugeplan og Brugerhåndtering use three types of indicators.

9.4 Reordering in the collection

Describe how one may reorder in a collection of items. For instance the bottom bar in the Pictoreader

III

Components

10. Buttons

The following sections there exist a shared component called `GirafButton` which assists one to comply with the following design rules, for a guide on how to use this class see Appendix B.

10.1 States

A button should have three sates: default, pressed and disabled. The background of a default button should have a orange/yellow (Color 2.1) gradient background with a darker border (Color 2.2) as seen in Figure 10.1a. When a button is pressed the background should become darker (Color 2.3) alongside the an even darker border (Color 2.4) as seen in Figure 10.1b. When a button is disabled the background and the border should be identical to the default button however this button should be 65% opaque as seen in Figure 10.1c.

(a) Default (b) Pressed (c) Disabled

Figure 10.1: Sates of a button

OBS! the disabled button does not, at the moment, reduce the opacity on the content of the button only the background.

10.2 Content

A button can either contain text (Figure 10.2a), and icon (Figure 10.2b) or both (Figure 10.2c). If one wants a button that does something that can be symbolized easily with an icon (See Chapter 3.4) which the user is familiar with, one should use an icon only button. In other cases where some text is needed to explain an action one should do so, how ever one should only have both icon and text in cases where the icon helps to understand the text. In the cases where there are no icon symbolizing what the text describes one should use a text only button.

(a) Button with text (b) Button with icon (c) Button with both

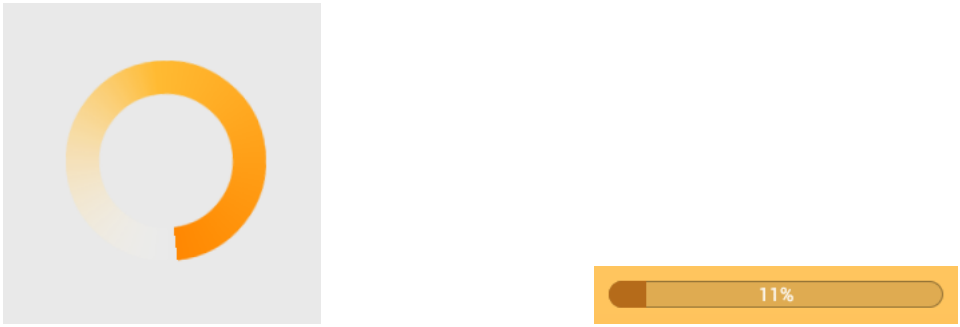
Figure 10.2: Content of a button

10.3 Contextual Ordering

When a collection of buttons is shown in the same context the buttons should be ordered by negative and positive actions. Leftmost buttons should cause negative actions and the rightmost should cause positive buttons. An example of this can be seen in the dialog displayed in Figure 12.3b on Page 48.

11. Progress Bar

The Android framework includes a widget called `ProgressBar` which can be used both as an activity indicator (see Figure 12.4), and as an actual progress bar. Both usages of the word will henceforth be referred to as just `ProgressBar`, unless otherwise made explicit. Both are used to indicate that the application is not frozen and that something is going on. The ProgressBar (as a horizontal progress bar) should generally be used when there is a reliable way of calculating the actual progress on the running task a seen in Figure 11.1b. The activity indicator should generally be used when there is no way of telling how long there will be until the task is complete as seen in Figure 11.1a.



(a) A spinning progressbar (activity indicator)

(b) A horizontal progressbar

Figure 11.1: Examples of progressbars

Describe a progressbar or indicator that should be used when synchronizing.

12. Dialog

When a user has to respond to an event, dialogs should be used. A dialog consists of a title, a description, some buttons and possibly some additional views. In *GIRAF Components* there exists some classes for this purpose. The general layout of a dialog can be seen in Figure 12.6. For a guide of how these dialogs can be implemented see Appendix C.

12.1 Confirm dialog

When a user needs to confirm that some action is going to happen, the confirm dialog should be used as it looks in Figure 12.1.

Figure 12.1: Confirm dialog

12.2 Notify dialog

When a user needs to be notified that some action has happened, the notify dialog should be used as it looks in Figure 12.2.

Figure 12.2: Notify dialog

12.3 Profile selector dialog

When a user needs to select a profile in some context, eg. change the current citizen on the tablet, one should use the dialog as it looks in Figure 12.3a. In other use cases where a user might need to select multiple profiles, one should use a dialog as it looks in Figure 12.3b.

(a) Single profile selector

(b) Multi profile selector

Figure 12.3: Profile selectors

12.4 Waiting dialog

In cases where the system needs to perform some action that takes a long time to execute (See Section 12.4.1), to indicate that the system is not frozen, the waiting dialog, as it looks in Figure 12.4, can be used.

> **Note 12.4.1** If unused screen real estate is temporarily available at the location onto which elements are currently being loaded, one should place the `ProgressBar` in this unused screen real estate. If there is not enough screen real estate available, or if it does not make sense to place the `ProgressBar` at the available location, one should use a Dialog with a `ProgressBar` instead. An activiy indicator as the `ProgressBar` should generally be used when there is no way of telling how long there will be until the task is complete.

Figure 12.4: Waiting dialog

12.4.1 Long tasks

Long running tasks should generally not block the GUI. Any task that can potentially take a long time should be done on a background thread and NOT on the main GUI thread. See Appendix E for a detailed description of how to enforce this.

12.5 Custom buttons dialog

If one want to have more than two buttons in a dialog the Custom buttons dialog is the best soloution as seen on Figure 12.5.

> **Note 12.5.1** The buttons of the dialog should comply with the rules described in Section 10.3.

Figure 12.5: Custom buttons dialog

12.6 Customizable Dialog

Some uses of dialogs might be more specialized than the ones already existing in *GIRAF Components*, for this reason the inflatable dialog exists. If one wants to add input fields or a custom view one should use this dialog. In Figure 12.6, an example of dialog is shown, this example shows the usecase when a user needs to edit a category.

Figure 12.6: Customizable Dialog

> **Note 12.6.1** It is important that if buttons should be added to this type of dialog it must be placed in the very bottom of the dialog and should be divided as shown in Figure 12.6. Also note that buttons should be 40dp in height.

13. Tabs

Describe Tabs: The design of tabs should be consistent around the giraf software suite. How ever at the moment they only exist in the launcher. There exist both horizontal tabs and vertical tabs in the launcher. Other apps could use these tabs to improve the usability and consistency.

14. Spinners

Describe Spinners (GirafSpinner): This is also known as a dropdown menu. Describe how this should look like, and describe how to inser text into it. Create an appendix that describes how one uses GirafSpinner and GirafSpinnerAdapter. The spinner is not well implemented, but some soloution already exist called a GirafSpinner

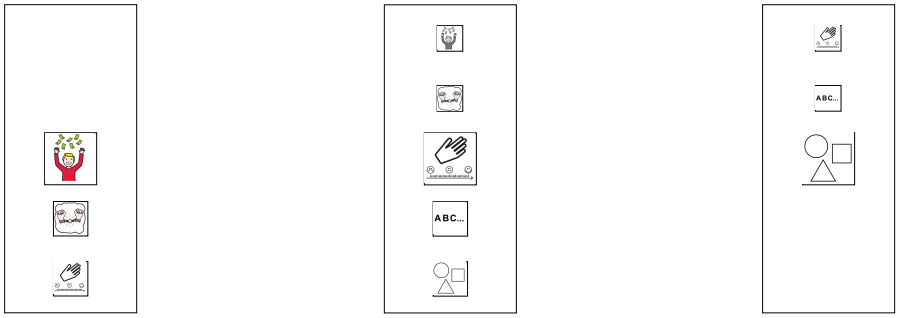
Figure 14.1: Current implementation `GirafSpinner`

15. Sequences

A sequence is a list of pictograms, where it is possible to mark a progress in the sequence of pictograms. This components is very common in the Sekvens app and the Ugeplan app.

15.1 Progress

The progress of the sequence is indicated by the size of pictograms in the sequence. The pictogram currently in progress is always in the middle of the view displaying the sequence as seen in Figure 15.1. When displaying pictograms to a citizen it may not split (Section 4.1.1). Upcomming pictograms should be a bit smaller than the one currently in progress. When progress has been made the pictograms that are done should be smaller and grayed as seen in Figure 15.1b and Figure 15.1c. For the ratio between the pictograms see Table 15.1.



(a) Start of sequence (b) Middle of sequence (c) End of sequence

Figure 15.1: Sequence progress

	Upcomming	In Progress	Done
Ratio	1	1.5	0.75

Table 15.1: Sequence picotgrams ratio

IV

Appendix

A. Customization of action bar

If one wants to customize the action bar, the `GirafActivity` provides the possibility to change the title of the action bar and the possibility of adding `GirafButtons` to the action bar. The action bar looks like the one in Figure 7.1 on Page 29.

> **Note A.0.1** This actionbar is only available to activities that extends the `GirafActivity`.

From default the title in the top bar is the label of the application, how ever if one wants to customize it the `setActionBarTitle` method can be used as seen in Line 10 in Code Snippet A.1.

The action bar always have a back button shown that does the same as the standard android back button. If one wants to add more buttons to the action bar one can use the `addGirafButtonToActionBar` method. One can then instantiate some `GirafButtons` and then use this method as in Lines 13-18 in Code Snippet A.1

Code Snippet A.1: Customized action bar

```
1  public class ExampleActivity extends GirafActivity {
2
3      ...
4
5      @Override
6      protected void onCreate(Bundle savedInstanceState) {
7          super.onCreate(savedInstanceState);
8
9          // Set the title of the actionbar
10         setActionBarTitle("Custom Title");
11
12         // Instantiate buttons for the action bar
13         GirafButton cameraButton = new
               GirafButton(this,getResources().getDrawable(R.drawable.icon_camera));
14         GirafButton helpButton = new
               GirafButton(this,getResources().getDrawable(R.drawable.icon_help));
15
16         // Add the buttons to the action bar
17         addGirafButtonToActionBar(cameraButton,LEFT);
18         addGirafButtonToActionBar(helpButton,RIGHT);
19
20         ...
21
22     }
23 }
```

B. Usage of GirafButtons

This appendix explains how ones used the class `GirafButton` for complying wit the rules described in Chapter 10. Through out this explanation we will be creating a buttons looking somewhat like the one in Figure B.1.

Figure B.1: Visual representaion of a `GirafButton`

Code Snippet B.1: GirafButton example in an XML layout

```
1  <LinearLayout
2     xmlns:app="http://schemas.android.com/apk/res-auto"
3     xmlns:android="http://schemas.android.com/apk/res/android" >
4
5     <dk.aau.cs.giraf.gui.GirafButton
6         android:id="@+id/giraf_button"
7         android:layout_width="wrap_content"
8         android:layout_height="wrap_content"
9         app:text="Camera"
10        app:icon="@drawable/icon_camera" />
11
12 </LinearLayout>
```

A `GirafButton` can be inserted into the GUI both using xml component but also using a java class. If you have layout as in Code Snippet B.1, if you want to access the button you should use the `findViewById` as seen in Line B.2 in Code Snippet 13.

> **Note B.0.2** When using xml to implement a button note that you have to set either the `app:text` property or the `app:icon` property. Otherwise an `IllegalArgumentException` will be thrown.

Code Snippet B.2: GirafButton example in Java code

```
1  public class ExampleActivity extends GirafActivity {
2
3      ...
4
5      @Override
6      protected void onCreate(Bundle savedInstanceState) {
7          super.onCreate(savedInstanceState);
8
9          // Sets the layout of the activity
10         setContentView(R.layout.example_activity);
11
12         // Get a button from xml
```

```
13          GirafButton xmlButton = (GirafButton) findViewById(R.id.giraf_button);
14
15          // Program buttons directly in code
16          GirafButton textButton = new GirafButton(this, "Camera");
17          GirafButton iconButton = new GirafButton(this,
                getResources().getDrawable(R.drawable.icon_camera));
18          GirafButton iconTextButton = new GirafButton(this,
                getResources().getDrawable(R.drawable.icon_camera), "Camera");
19
20      }
21  }
```

If one wants to create various buttons dynamically in code one can use one the three constructors provided for the `GirafButton` class. These three constructors takes either a `String` setting the text on the button, a `Drawable` setting the icon or both as seen in Lines 18-B.2 in Code Snippet 16.

C. Implementation Guide for Dialogs

When using the various dialogs implemented in the *GIRAF Components*, it is important that one uses the support library, this is done by including the support library in the gradle build file as seen in Code Snippet C.1.

Code Snippet C.1: build.gradle

```
dependencies {
   compile ('com.android.support:support-v4:+')
   // Other dependencies
}
```

The dialogs is an extension of the android class `DialogFragment`, meaning that all dialogs is handled as fragments. This means that callbacks from the dialogs is done using interfaces, which the activity starting them should implement. Through out the description of these dialogs there will be talked about a method called `onActionButtonClick` which is an event that is called whenever some gui element is clicked, eg. when a button is clicked. This method will in all of the following examples create an instance of the dialog and show it using the `show` method.

Also note that in all of the following code snippets there is declared a string tag (`DIALOG_TAG`) this is a tag that the Android system needs to handle fragments. However in many of the code snippets there also exist an integer (`DIALOG_ID`) which is used to call a method in the activity from the fragment.

> **Note C.0.3** It is important that whenever one creates a dialog one uses the `newInstance` method on the specific dialog.

> **Note C.0.4** Remember to call the `show` method on the dialog when the dialog should be displayed. Simply instantiating it is not enough.

> **Note C.0.5** It is important to check that it is the correct `dialogIdentifier` that is used in the methods implemented from interfaces in order to determine which dialog was actually responded to.

C.1 Confirm dialog

The confirm dialog is used to make the user confirm an action before it is executed. This section describes how one could implement an confirm dialog as described in Section 12.1. An example of such an implementation can be seen in Code Snippet C.2.

Using the `GirafConfrimDialog.Confirmation` interface (Line 1 in Code Snippet C.2) allows the activity (`ExampleActivity`) to implement the method `confirmDialog`, as in Line 24 in Code Snippet C.2, that will be called from the dialog whenever a response has been made for it (when the user press the acceptance button).

Code Snippet C.2: Implementaion of confirm dialog

```
1  public class ExampleActivity extends GirafActivity implements GirafConfirmDialog.Confirmation {
2
3      // Identifier for callback
4      private static final int CONFIRM_DIALOG_ID = 1;
5
6      // Fragment tag (android specific)
7      private static final String CONFIRM_DIALOG_TAG = "DIALOG_TAG";
8
9      ...
10
11     /**
12      * When some button is clicked in the gui
13      * @param view the view that was clicked
14      */
15     public void onActionButtonClick(View view) {
16         // Creates an instance of the dialog
17         GirafConfirmDialog confirmDialog = GirafConfirmDialog.newInstance("A good title", "This
               describes the dialog", CONFIRM_DIALOG_ID);
18
19         // Shows the dialog
20         confirmDialog.show(getSupportFragmentManager(), CONFIRM_DIALOG_TAG);
21     }
22
23     @Override
24     public void confirmDialog(int dialogIdentifier) {
25         if (dialogIdentifier == CONFIRM_DIALOG_ID) {
26             // Perform some action after the confirmation
27         }
28     }
29 }
```

The default behavior of buttons in the dialog is to dismiss it. The acceptance button will furthermore call the `confirmDialog` with the identifier `CONFIRM_DIALOG_ID`, as in Lines 24-28 in Code Snippet C.2.

C.2 Notify dialog

The notify dialog is used to make the user aware of some event. This section described how one could implement an notify dialog as described in Section 12.2. An example of such an implementation can be seen in Code Snippet C.3.

Using the `GirafNotify.Notification` interface (Line 1 in Code Snippet C.2) allows the activity (`ExampleActivity`) to implement the method `noticeDialog`, as in Line 24 in Code Snippet C.3, that will be called from the dialog whenever a response has been made for it (when the user press the button).

Code Snippet C.3: Implementaion of notify dialog

```
1  public class ExampleActivity extends GirafActivity implements GirafNotifyDialog.Notification {
2
3      // Identifier for callback
4      private static final int NOTIFY_DIALOG_ID = 1;
5
6      // Fragment tag (android specific)
7      private static final String NOTIFY_DIALOG_TAG = "DIALOG_TAG";
8
```

```
9       ...
10
11      /**
12       * When some button is clicked in the gui
13       * @param view the view that was clicked
14       */
15      public void onActionButtonClick(View view) {
16          // Creates an instance of the dialog
17          GirafNotifyDialog notifyDialog = GirafNotifyDialog.newInstance("A good title", "This
                describes the dialog", NOTIFY_DIALOG_ID);
18
19          // Shows the dialog
20          notifyDialog.show(getSupportFragmentManager(), NOTIFY_DIALOG_TAG);
21      }
22
23      @Override
24      public void noticeDialog(int dialogIdentifier) {
25          if(dialogIdentifier == NOTIFY_DIALOG_ID) {
26              // Perform some action after the notification
27          }
28      }
29  }
```

The default behavior the button in the dialog is to dismiss it. The button will furthermore call the `confirmDialog` with the identifier `NOTIFY_DIALOG_ID`, as in Lines 24-28 in Code Snippet C.2.

C.3 Profileselector dialog

The profile selector dialog is used to allow for user to select one ore more profiles. This section describes how one could implement an confirm dialog as described in Section 12.3. Examples of such an implementation can be seen in Code Snippet C.4 and Code Snippet C.5.

This dialog is used in two use cases. One where the user wants to select exactly one profiles, and another use case where one wants to select arbitrary many profiles. Using the `GirafProfileSelectorDialog.OnSingle` interface allows for the activity (`ExampleActivity`) to respond on a single selected profile as in Line 1 in Code Snippet C.4. Where as the `GirafProfileSelectorDialog.OnMultipleProfilesSelectedListen` allows for the activity to respond on multi selected profiles an in Line 1 in Code Snippet C.5.

> **Note C.3.1** Observe that the main difference between the instantiating of the single and multi version of the profile selector dialog is the fourth argument called `selectMultipleProfiles`. That determines which interface will be called. If this boolean is set to `true` the activity should implement `GirafProfileSelectorDialog.OnMultiProfileSelectedListener` otherwise the activity should implement `GirafProfileSelectorDialog.OnSingleProfileSelectedListener`. Obviously if an activity have one dialog of each type both interfaces should be implemented.

Code Snippet C.4: Implementaion of single profile selector dialog

```
1   public class ExampleActivity extends GirafActivity implements
        GirafProfileSelectorDialog.OnSingleProfileSelectedListener {
2
3       // Identifier for callback
4       private static final int PROFILE_SELECT_DIALOG_ID = 1;
5
6       // Fragment tag (android specific)
7       private static final String PROFILE_SELECT_DIALOG_TAG = "DIALOG_TAG";
8
9       ...
10
11      /**
12       * When some button is clicked in the gui
```

```
13      * @param view the view that was clicked
14      */
15     public void onActionButtonClick(View view) {
16         // Creates an instance of the dialog
17         GirafProfileSelectorDialog singleSelectDialog =
               GirafProfileSelectorDialog.newInstance(this,getGuardianIdentifier(),false,false,"Select
               a profile for some purpose", PROFILE_SELECT_DIALOG_ID);
18
19         // Shows the dialog
20         singleSelectDialog.show(getSupportFragmentManager(), PROFILE_SELECT_DIALOG_TAG);
21     }
22
23     @Override
24     public void onProfileSelected(int dialogIdentifier, Profile profile) {
25         if(dialogIdentifier == PROFILE_SELECT_DIALOG_ID) {
26             // Perform some action based on the profile selected
27         }
28     }
29
30     private long getGuardianIdentifier() {
31         // Some code that returns the wanted guardian identifier
32     }
33 }
```

When a profile is clicked in a single profile selector the `onProfileSelected` method is called, with the profile that was selected and the identifier `PROFILE_SELECT_DIALOG_ID` as in Lines 24-28 in Code Snippet C.4.

Code Snippet C.5: Implementaion of multi profile selector dialog

```
1  public class ExampleActivity extends GirafActivity implements
       GirafProfileSelectorDialog.OnMultipleProfilesSelectedListener {
2
3      // Identifier for callback
4      private static final int PROFILE_SELECT_DIALOG_ID = 1;
5
6      // Fragment tag (android specific)
7      private static final String PROFILE_SELECT_DIALOG_TAG = "DIALOG_TAG";
8
9      ...
10
11     /**
12      * When some button is clicked in the gui
13      * @param view the view that was clicked
14      */
15     public void onActionButtonClick(View view) {
16         // Creates an instance of the dialog
17         GirafProfileSelectorDialog multiSelectDialog =
               GirafProfileSelectorDialog.newInstance(this,getGuardianIdentifier(),false,true,"Select
               some profiles for some purpose", PROFILE_SELECT_DIALOG_ID);
18
19         // Shows the dialog
20         multiSelectDialog.show(getSupportFragmentManager(), PROFILE_SELECT_DIALOG_TAG);
21     }
22
23     @Override
24     public void onProfilesSelected(int dialogIdentifier, List<Pair<Profile, Boolean>>
           checkedProfileList) {
25         if(dialogIdentifier == PROFILE_SELECT_DIALOG_ID) {
26             // Perform some action based on the list of profiles and check status
27         }
28     }
29
30     private long getGuardianIdentifier() {
31         // Some code that returns the wanted guardian identifier
32     }
33
34 }
```

When a profile is clicked in a multi profile selector the `onProfilesSelected` method is called, with a list of profiles and a boolean telling of they were marked or not alongside the identifier `PROFILE_SELECT_DIALOG_ID` as in Lines 24-28 in Code Snippet C.5.

C.4 Waiting dialog

The waiting dialog is used to indicate to the user that some task is being executed and that he/she should wait for that task to end. This section describes how one could implement an waiting dialog as described in Section 12.4.

> **Note C.4.1** Note that in this dialog the dialog is instantiated in the `onCreate` method (see Line 40 in Code Snippet C.6), and is called showed implicitly by the `onActionButtonClick` method which starts the async task (see Line 50 in Code Snippet C.6.

This dialog is often used when some thread syncing tasks is running (see Appendix E). This dialog requires not interface to work properly. Good practice with this dialog is to show it before the long task is executed. This can be done using the `onPreExecute` method (see Line 16 in Code Snippet C.6). One should dismiss the dialog after the task is done which can be handled in `onPostExecute` method (see Line 29 in Code Snippet C.6).

Code Snippet C.6: Implementaion of waiting dialog

```
1  public class ExampleActivity extends GirafActivity {
2
3      private static final String WAITING_DIALOG_TAG = "DIALOG_TAG"; // Fragment tag (android
           specific)
4      private GirafWaitingDialog waitingDialog;
5
6      /**
7       * An async task that performs a task that takes time
8       */
9      private class ExampleTask extends AsyncTask<Void, Void, Void> {
10
11         @Override
12         protected void onPreExecute() {
13             super.onPreExecute();
14
15             // Shows the dialog
16             waitingDialog.show(getSupportFragmentManager(), WAITING_DIALOG_TAG);
17         }
18
19         @Override
20         protected Void doInBackground(Void... params) {
21             // Perform the task the user needs to wait for
22         }
23
24         @Override
25         protected void onPostExecute(Void aVoid) {
26             super.onPostExecute(aVoid);
27
28             // Hides the dialog
29             waitingDialog.dismiss();
30         }
31     }
32
33     ...
34
35     @Override
36     protected void onCreate(Bundle savedInstanceState) {
37         super.onCreate(savedInstanceState);
38
39         // Initialize the waiting dialog
40         waitingDialog = GirafWaitingDialog.newInstance("Please wait", "We are currently performing
               a task that takes time");
```

```
41      }
42
43      /**
44       * When some button is clicked in the gui
45       * @param view the view that was clicked
46       */
47      public void onActionButtonClick(View view) {
48
49          // Starts a tasks
50          new ExampleTask().execute();
51      }
52
53  }
```

C.5 Custom buttons dialog

The custom buttons dialog is used when the developer wants to provide the user with more than two buttons in the dialog. This section described how one coul dimplement an custom buttons dialog as described in Section 12.5.

Using the `GirafCustomButtonsDialog.CustomButtons` (Line 1 in Code Snippet C.7) interface allows for the developer to add abitrary many buttons to the dialog.

> **Note C.5.1** There is no limit on buttons that can be added to the dialog, but one should be careful that the design is consistent. The width should not exceed the width of other dialogs.

Code Snippet C.7: Implementaion of custom buttons dialog

```
1   public class ExampleActivity extends GirafActivity implements
        GirafCustomButtonsDialog.CustomButtons {
2
3       // Identifier for callback
4       private static final Integer CUSTOM_BUTTONS_DIALOG_ID = 1;
5
6       // Fragment tag (android specific)
7       private static final String CUSTOM_BUTTONS_DIALOG_TAG = "DIALOG_TAG";
8
9       ...
10
11      /**
12       * When some button is clicked in the gui
13       * @param view the view that was clicked
14       */
15      public void onActionButtonClick(View view) {
16          // Creates an instance of the dialog
17          GirafCustomButtonsDialog customButtonsDialog = GirafCustomButtonsDialog.newInstance("Pick
                an action","Choose what should happen to the element", CUSTOM_BUTTONS_DIALOG_ID);
18
19          // Show the dialog
20          customButtonsDialog.show(getSupportFragmentManager(),CUSTOM_BUTTONS_DIALOG_TAG);
21      }
22
23      @Override
24      public void fillButtonContainer(int dialogIdentifier, GirafCustomButtonsDialog.ButtonContainer
            buttonContainer) {
25          if(dialogIdentifier == CUSTOMBUTTONS_DIALOG_ID) {
26              GirafButton renameButton = new GirafButton(this,"Rename");
27              GirafButton copyButton = new GirafButton(this,"Copy");
28              GirafButton deleteButton = new GirafButton(this,"Delete");
29
30              // Set onClickListeners to the buttons
31
32              buttonContainer.addGirafButton(renameButton);
33              buttonContainer.addGirafButton(copyButton);
```

```
34              buttonContainer.addGirafButton(deleteButton);
35          }
36      }
37  }
```

Before the dialog is created the `fillButtonContainer` method is called, see Lines 24-36 in Code Snippet C.7. The identifier `CUSTOM_BUTTONS_DIALOG_ID` determines which dialog is being filled with buttons.

C.6 Inflatable dialog

The inflatable dialog is used to create more custom dialogs and want some custom gui elements to be shown in the dialog. In this example we have created a custom dialog that contains an AnalogClock element and a button to dismiss the dialog.

When creating an instance of this type of dialog the `newInstance` method takes an layout resource as third argument. The argument `R.layout.example` as seen in Line 17 in Code Snippet C.8 comes from the layout seen in Code Snippet C.9.

Code Snippet C.8: The implementation of the inflatable dialog

```
1  public class ExampleActivity extends GirafActivity implements
       GirafInflatableDialog.OnCustomViewCreatedListener {
2
3      // Identifier for callback
4      private static final Integer CLOCK_DIALOG_ID = 1;
5
6      // Fragment tag (android specific)
7      private static final String CLOCK_DIALOG_TAG = "DIALOG_TAG";
8
9      ...
10
11     /**
12      * When some button is clicked in the gui
13      * @param view the view that was clicked
14      */
15     public void onActionButtonClick(View view) {
16         // Creates an instance of the dialog
17         GirafInflatableDialog clockDialog = GirafInflatableDialog.newInstance("Time", "Here is the
               time on an analog clock", R.layout.example_layout, CLOCK_DIALOG_ID);
18
19         // Show the dialog
20         clockDialog.show(getSupportFragmentManager(),CLOCK_DIALOG_TAG);
21     }
22
23     @Override
24     public void editCustomView(ViewGroup customView, int dialogIdentifier) {
25         if(dialogIdentifier == CLOCK_DIALOG_ID) {
26
27             // Find the close button defined in xml
28             GirafButton closeButton = (GirafButton) customView.findViewById(R.id.close_button);
29
30             // Do something with the close button eg. set an oClickListener
31
32         }
33     }
34 }
```

Using the `GirafInflatableDialog.OnCustomViewCreatedListener` interface (Line 1 in Code Snippet C.8) allows for the developer to access the custom created layout by the `editCustomView` method. See Lines 24-33 in Code Snippet C.8 for an example of how one access the custom view.

Note C.6.1 One cannot access the custom inflated view before the `editCustomView` is called from the dialog it self.

Code Snippet C.9: The custom layout for an inflateable dialog

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    <!-- An analog clock to be shown in the dialog -->
    <AnalogClock
        android:id="@+id/analogClock"
        android:layout_width="match_parent"
        android:layout_height="100dp"
        android:layout_gravity="center" />

    <!-- The container for buttons -->
    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="horizontal">

        <!-- A button to close the dialog -->
        <dk.aau.cs.giraf.gui.GirafButton
            android:id="@+id/close_button"
            android:layout_width="wrap_content"
            android:layout_height="40dp"
            app:icon="@drawable/icon_cancel"
            app:text="Close clock dialog" />

    </LinearLayout>

</LinearLayout>
```

D. Implementation Guide for Pictures

To accommodate the need for displaying pictures throughout the different applications in the *GIRAF* software suite, a shared component was build in the *GIRAF Components* library. This chapter will cover the use and functionality of this component. The component implemented is named `GirafPictogramItemView`.

Consider if it would be better to rename it to something like `GirafImageEntityView` to indicate that the component may be used to more than just displaying pictograms

> **Note D.0.2** Throughout this chapter different examples will reference constructors of the `GirafImageEntityView`. The class contains a lot of different constructors with a lot of different, optional, parameters. The examples will only cover some of these constructors, so please reference the documentation for the class if in doubt.

D.1 General Usage

The component may be used to display anything that extends the `ImageEntity`-interface. Examples of such classes are `Pictogram`, `Category`, and `Applications`. This section will only cover the most basic use of the component. Please refer to the following sections to see more specific usages of the components.

Write that the only use for this component is programmatically and it's XML implementations are very limited

> **Note D.1.1** Some classes cannot be used because of a poor structure. Example of such a class is `Sequence`, which does not have its own image but instead a reference to a `Pictogram`. When using the the component, please make sure that the class extends the `ImageEntity` interface otherwise refer to Section D.6 to see how to fix this issue.

The most general use of this component is to simply display an image. This image "comes from" objects (instances) of classes that extend the `ImageEntity`. To create a new instance of the view, simply call its constructor which will require a context and the object that should be displayed. Code Snippet D.1 shows an example of a `getView`-method in an adapter which is used to display pictograms. Please note that this example does not include a title below the picture. To do this, additional parameters for the constructors are required. Code Snippet D.2 shows the same example, however including the title parameter. The title can also later be changed (or set) using the `setTitle`-method on the `GirafPictogramItemView` object. The title can furthermore be shown or hidden using the methods `showTitle` and `hideTitle`.

Code Snippet D.1: Basic use

```
1 @Override
```

```
public View getView(int position, View convertView, ViewGroup parent) {
    // Find the pictogram to show
    final Pictogram pictogram = pictogramList.get(position);

    // Generate a view that will be used to display the pictogram
    GirafPictogramItemView itemView = new GirafPictogramItemView(context, pictogram);

    // ...

    return itemView;
}
```

Code Snippet D.2: Basic use (with title)

```
@Override
public View getView(int position, View convertView, ViewGroup parent) {
    // Find the pictogram to show
    final Pictogram pictogram = pictogramList.get(position);

    // Generate some title to display below the picture (for instance be the name of the pictogram)
    String title = ...

    // Generate a view that will be used to display the pictogram
    GirafPictogramItemView itemView = new GirafPictogramItemView(context, pictogram, title);

    // ...

    return itemView;
}
```

D.2 Indicator Overlay

In some situations it may be required to indicate something about a picture - for instance when differentiating between pictograms and categories or that the picture can be edited. This can be achieved by using a small indicator overlay in the bottom right corner of the picture. (See Section 4.3 for additional information).

D.2.1 Overlay to indicate editable status

To indicate that a picture is editable use the method `setEditable` on a `GirafPictogramItemView` object. This method requires a `boolean` as argument. If set to `true`, the picture will appear as editable, while `false` will display the picture as normal.

D.2.2 Custom indicator overlay

To use a custom indicator overlay, use the method `setIndicatorOverlayDrawable` on a `GirafPictogramItemView` object. This method requires a `Drawable` as argument.

> **Note D.2.1** It is not possible to use both an editable-overlay along with a custom indicator. If you try to do this, you will receive an exception.

D.3 Fallback Drawable

If you suspect that some of the `ImageEntity`-objects does not actually have an image, a fallback-image may be used. This fallback image will only be displayed in the case that the original objects does not have any images. Code Snippet D.3 shows an example of a `getView`-method in an adapter. This example will use the drawable `icon_fallback` if the variable `pictogram` does not have an image.

The fallback drawable can also be set when calling the `setImageModel`-method. This method require two parameters, the first being an `ImageEntity` (the object that should be shown) and the second is a `Drawable`, which will be used as a fallback.

Code Snippet D.3: Usage with fallback

```
@Override
public View getView(int position, View convertView, ViewGroup parent) {
    // Find the pictogram to show
    final Pictogram pictogram = pictogramList.get(position);

    // Find the drawable that will be used as a fallback
    Drawable fallback = context.getResources().getDrawable(R.drawable.icon_fallback);

    // Generate a view that will be used to display the pictogram
    GirafPictogramItemView itemView = new GirafPictogramItemView(context, pictogram, fallback);

    // ...

    return itemView;
}
```

D.4 Grayscaled Pictures

If needed, the `GirafPictogramItemView` can display all of the pictures grayscaled. From default, this is disabled but may be enabled by adding an additional boolean parameter (set to true) to the constructor or by adding the same parameter to the `setImageModel`-method.

D.5 Marking

Marking of `GirafPictogramItemView` can be done through one of the following methods. The method `setChecked` requires a boolean, indicating if the view should be marked (`true`) or not (`false`). `toggle` required no parameters and will simply invert the current checked/marked state. If needed, you may call `isChecked` which will return a `boolean` indicating if the view is marked.

D.6 Workarounds

> **Note D.6.1** These workarounds are not something that should be used ideally, and should only be temporarily fixes until the structure of the different classes are fixed. This workaround only works for classes that do not implement the interface `ImageEntity`. If the class that you want to display implements `ImageEntity`, please refer to Section D.1.

Most classes that do not implement the interface `ImageEntity` have some kind of relation to something that does. So to display the wrongly structured class, you must first locate this relation. For instance when displaying a `Sequence`, you must use its reference to a `Pictogram` in order to display it. Please refer to Code Snippet D.4 to see an example of how to implement this workaround.

Code Snippet D.4: Workaround example

```
...

// Create a database-helper, which will allow us to find the pictogram
Helper helper = new Helper(context);

// Find the sequence that we want to display
Sequence sequence = ...

```

```
9  // Find the pictogram to be displayed
10 long pictogramId = sequence.getPictogramId();
11 Pictogram pictogram = helper.pictogramHelper.getById(pictogramId);
12
13 // Use this pictogram to display the sequence as a picture
14 GirafPictogramItemView itemView = new GirafPictogramItemView(context, pictogram);
15
16 ...
```

E. Threading AsyncTask

This chapter introduces good practice to managing the GUI thread of Android applications.

E.1 Android GUI Thread

The Android system imposes real time constraints on applications to keep them smooth and lag free. This section covers how one should make sure not to violate these constraints across all application code.

E.1.1 GUI Thread

The real time constraints are implemented on a special GUI thread which runs the user interface event loop, which is basically a queue of tasks (Java Runnable). Applications, e.g. activities, in the Android eco system run, i.e updates their views, on this GUI thread. Any updates to views not coming from the GUI thread will result in an exception. In Android, these real time constraints on the GUI thread have been implemented by simply setting a hard limit on how long, in real time, tasks on the GUI thread are allowed to run. Any violation of the constraints will result in an Application Not Responding (ANR) message to the user [1]. Users will not be able to distinguish an unresponsive application, e.g. the result of a deadlock, from a task taking too long on the GUI thread, which may cause the user to terminate an otherwise functioning application that was doing some complicated database query or downloading some file from the Internet.

E.1.2 AsyncTask

The Android framework provides a very useful abstraction which makes it easy to run long tasks in the background with a class called AsyncTask. The AsyncTask provides four non final methods that make it easy to synchronize between a background thread and the GUI thread. The three methods `onPreExecute`, `onProgressUpdate`, `onPostExecute` are always all guaranteed to run on the GUI thread. The last method, `doInBackground`, always runs on an unspecified background thread from a pool of threads maintained by the system. The class then ensures an order on these method calls being: `onPreExecute`, `doInBackground`, `onPostExecute`. The `onProgressUpdate` method can be run multiple times while the doInBackground method is running and is started by a call to a method called `publishProgress` from the `doInBackground` method. En example of an `AsyncTask` can be found Code Snippet C.6 on Page 67.

E.1.3 Long Tasks

Not causing the application to show an ANR message is one thing, making the user explicitly aware that a long task is running, through some kind of feedback, is considered good practice as

[1] Keeping Your App Responsive - http://developer.android.com/training/articles/perf-anr.html

well [2]. We have tried to provide this feedback through Android `ProgressBar` widgets whenever long operations take place. The name "ProgressBar" is a bit misleading since the style of the `ProgressBar` we use, which is the default style, is more like a spinning activity indicator as seen in Figure 12.4 on Page 48.

[2] David Benyon - Designing Interactive Systems 2nd Edition

V

Index

Index

VI

TODOS

Todo list