UI customisation

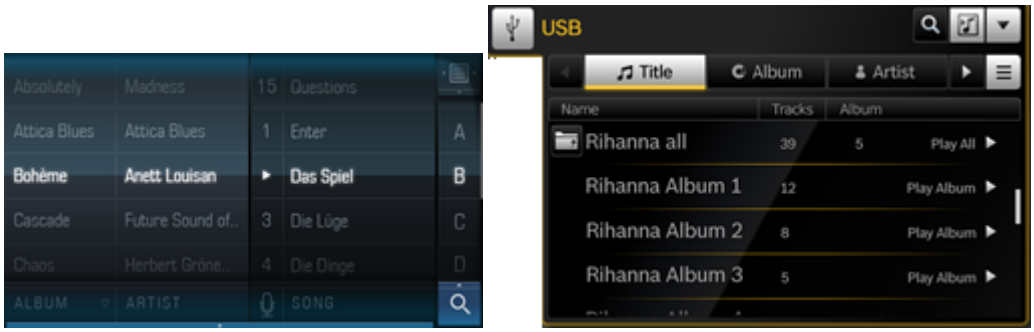
Contents

UI customisation

Introduction

The goal of this user interface customisation design document is to reduce app development time when porting between variants by abstracting the differences between variants into a UI library.

For example, below are designs of an audio player application — on the left is variant A and on the right is the variant B.



The A variant mixes three independently scrollable lists for artist, album, and then track. The B variant uses one scrollable list with columns for the aforementioned details. Using different widgets and styling it should be possible to radically change the user interface as above. More examples of variant changes are shown in Variant differences.

The goal of standardising this process is reduce the amount of code written or changed in customising a variant. It is understood that for system components, code might have to be altered for some requests, but code inside application

bundles should remain as similar as possible and work in variant-specific ways automatically.

Terminology and Concepts

Vehicle

For the purposes of this document, a *vehicle* may be a car, car trailer, motorbike, bus, truck tractor, truck trailer, agricultural tractor, or agricultural trailer, amongst other things.

System

The *system* is the infotainment computer in its entirety in place inside the vehicle.

User

The *user* is the person using the system, be it the driver of the vehicle or a passenger in the vehicle.

Widget

A *widget* is a reusable part of the user interface which can be changed depending on location and function.

User Interface

The *user interface* is the group of all widgets in place in a certain layout to represent a specific use-case.

Roller

The *roller* is a list widget named after a cylinder which revolves around its central horizontal axis. As a result of being a cylinder it has no specific start and finish and appears endless.

Speller

The *speller* is a widget for text input.

Application Author

The *application author* is the developer tasked with writing an application using the widgets described in this document. They cannot modify the variant or the user interface library.

Variant

A *variant* is a customised version of the system by a particular system integrator. Usually variants are personalised with particular colour schemes and logos and potentially different widget behaviour.

View

A *view* is an page in an application with an independent purpose. Views move from one to another, and sometimes also back, to form the workflow of the application. For example, in a photo application the list of photos is one view and the highlight on one photo in particular, perhaps with more metadata from the photo, is another view.

Template

A *template* is a text-based representation of a set of widgets in a view. Templates are for allowing changes and extensions without having to rebuild the actual code.

UI prototyping

UI prototyping is the process of building a mock-up of a UI to evaluate how it looks, and how usable it is for different use cases — but without hooking up the UI to an application implementation or backing code. The idea is to be able to produce a representative UI as fast as possible, so designers and testers can evaluate its usability, and can produce further iterations of the design, without wasting time on implementing backing functionality in code until the design is finalised. At this point, a programmer can turn the prototype into a complete implementation in code.

The process of prototyping is not relevant to UI *customisation*, but is relevant to the process of using a UI toolkit.

Here is an example of some prototype UIs[1], made in Inkscape.

WYSIWYG UI editing

WYSIWYG UI editing is the process of using a UI editor, such as Glade[2], where the UI elements can be composed visually and interactively to build the UI, for example by dragging and dropping them together. The appearance of the UI in the designer is almost identical to its appearance when it is run in production.

This could be contrasted with designing a UI by writing a ClutterScript[3] file, for

[1] https://github.com/gnome-design-team/gnome-mockups/blob/master/passwords-and-keys/passwords-and-keys.png

[2] https://glade.gnome.org/

[3] https://developer.gnome.org/clutter/stable/ClutterScript.html#ClutterScript.description

example, where the UI has to be run as part of a program in order to visualise it.

Use Cases

A variety of use cases for UI customisation are given below.

Multiple Variants

Each system integrator wants to use the same user interface without having to rewrite from scratch (see Variant differences).

For example, in the speller, variant A wants to highlight the key on an on-screen-keyboard such that the key pops out of the keyboard, whereas variant B wants to highlight just the letter within the key with no pop out animation.

Another example, in the app launcher, variant A wants to use a cylinder animation for rolling whereas variant B wants to scroll the list of applications like a flat list.

Fixed Variants

A system integrator wants multiple variants to be installable concurrently on the system, but wants the variant in use to be fixed and not able to change after being set in a configuration option. The system integrator wants said configuration option to be changeable without rebuilding.

Templates

A system integrator wants to customise the user interface as easily as possible without recompilation of applications. The system integrator wants to be able to choose the widgets in use in a particular application user interface (from a list of available widgets) and have them work accordingly.

For example, in a photo viewing application with one photo selected, system integrator A might want to display the selected photo with nothing else displayed, while system integrator B might want to display the selected photo in the centre of the display, but also have the next and previous photos slightly visible at the sides.

Template Extension

A system integrator wants to use the majority of an Apertis-provided template, but also wants to add their own variant-specific extensions. The system integrator wants to achieve this without copy and pasting Apertis-provided templates to retain maintainability, and wants to add their own extension template which merely references the Apertis-provided one.

For example, said system integrator wants to use an Apertis-provided button widget, but wants to make it spin 360° when clicked. They want to just override the library widget, adding the spin code, and not have to touch any other code relating to the internal working of the widget already provided in the library.

Custom Widget Usage

A system integrator wants to implement custom widgets by writing actual code. The system integrator wants to be integrate the new custom widgets into the user interface and into the developer tooling.

Template Library

A system integrator wants to be able to add new templates to the system via over the air (OTA) updates. The system integrator does not want the template to be able to reload automatically after being updated.

Appearance Customisation

Each system integrator wants to customise the look and feel of applications by changing styling such as padding widths, border widths, colours, logos, and gradients. The system integrator wants to make said modifications with the minimum of modifications, especially to the source code.

Different Icon Themes

Each system integrator wants to be able to trivially change the icon theme in use across the user interface not only without recompilation, but also at runtime.

Different Fonts

Each system integrator wants to be able to trivially change the font in use across the user interface, and bundle new fonts in with variants.

OTA Updates

System integrators want to be able to add fonts using over the air (OTA) updates. For example, the system integrator wants to change the font in use across the user interface of the variant. They send the updated theme definition as well as the new font file via an update and want it to be registered automatically and be immediately useable.

Language

The user wants to change the language of the controls of the system to their preferred language such that every widget in the UI that contains text updates accordingly without having to restart the application.

Right-to-Left Scripts

As above, the user wants to change the language of the controls of the system, but to a language which is read from right-to-left (Arabic, Persian, Hebrew, etc.), instead of left-to-right. The user expects the workflow of the user interface to also change to right-to-left.

OTA Updates

A system integrator wants to be able to add and improve language support over over the air (OTA) updates. For example, the system integrator wants to add a new translation to the system. They send the translation via an update and want the new language to immediately appear as an option for the user to select.

Animations

A system integrator wants to customise animations for the system. For example, they want to be able to change the behaviour of list widgets by setting the visual response using kinetic scrolling and whether there's an elastic effect when reaching the end of items. Another example is they also want to be able to customise the animation used when changing views in an application. Another example is the how button widgets react when pressed.

The system integrator then expects to see the changes apply across the entire system.

Prototyping

An application author wants to prototype a UI rapidly (see UI prototyping), using a WYSIWYG UI development tool (see WYSIWYG UI editing) with access to all the widgets in the library, including custom and vendor-specific widgets.

Day & Night Mode

A user is using the system when dark outside and wants the colour scheme of the display to change to accommodate for the darkness outside so not be too bright and dazzle the user. Requiring the user to adapt their eyes momentarily for the brightness of the system could be dangerous.

View Management

An application author has several views in their application and doesn't want to have to write a system of managing said views. They want to be able to add a workflow and leave the view construction, show and hide animations, and view destruction up to the user interface library.

Display Orientation

A system integrator changes the orientation of the display. They expect the user interface to adapt and display normally, potentially using a different layout more suited to the orientation.

Note that the adaptation is only expected to be implemented if easy and is not expected to be instantaneous, and a restart of the system is acceptable.

Speed Lock

Laws require that when the vehicle is moving some features be disabled or certain behaviour modified.

Geographical Customisation

Different geographical regions have different laws regarding what features and behaviours need to be changed, so it must be customisable (only) by the system integrator when it is decided for which market the vehicle is destined.

System Enforcement

Due to restrictions being government laws, system integrators don't want to rely on application authors to respect said restrictions, and instead want the system to enforce them automatically.

Non-Use Cases

A variety of non-use cases for UI customisation are given below.

Theming Custom Clutter Widgets

An application developer wants to write their own widget using the Clutter library directly. They understand that standard variant theming will not apply to any custom widget and any integration will have to be achieved manually.

Note that although unsupported directly by the Apertis user interface library, it is possible for application authors to implement this higher up in the application itself.

Multiple Monitors

A system integrator wants to connect two displays (for example, one via HDMI and one via LVDS) and show something on each one, for example when developing on a target board like the i.MX6. They understand this is not supported by Apertis.

DPI Independence

A system integrator uses a display with a different DPI. They understand that they should not expect that the user interface changes to display normally and not too big/small relative to the old DPI.

Display Size

A system integrator changes the resolution of the display. They understand that they should not expect the user interface to adapt and display normally, potentially using a different layout more suited to the new display size.

Dynamic Display Resolution Change

A system integrator wants to be able to change the resolution of the display or resize the user interface. They understand that a dynamic change in the user interface is not supported in Apertis.

Requirements

Variant set at Compile-Time

Multiple variants should be supported on the system but the variant in use should be decided at application compile-time such that it cannot be changed later (see Fixed variants).

CSS Styling

The basic appearance of the widgets should be stylable using CSS, changing the look and feel as much as possible with no modifications to the source code required (see Appearance customisation, Different icon themes).

The changes possible using CSS do not need to be incredibly intrusive and are limited to the basic core CSS properties. For example, changing colour scheme (background-color, color), icon theme & logos (background-image), fonts (font-family, font-size), and spacing (margin, padding).

More intrusive changes to the user interface should be achieved using templates (see Templates) instead of CSS changes.

For example, a system integrator wants to change the colour of text in buttons. This should be possible by changing some CSS.

Templates

CSS is appropriate for changing simple visual aspects of the user interface but does not extend to allow for structural modifications to applications (see CSS styling). Repositioning widgets or even changing which widgets are to be used is not possible with CSS and should be achieved using templates (see Templates).

There are multiple layers of widgets available for use in applications. Starting from the lowest, simplest, level and moving higher, encapsulating more with each step:

- buttons, entries, labels, …
- buttons with labels, radio buttons with labels, …
- lists, tree view, …
- complete views, or *templates*.

Templates are declarative representations of the layout of the user interface which are read at runtime by the application. Using templates it is possible to redesign the layout, look & feel, and controls of the application without recompilation.

The purpose of templates is to reduce the effort required by an application author to configure each widget, and to maintain the same look and feel across the system.

Catalogue of Templates

There should be a catalogue of templates provided by the library which system integrators can use to design their applications (see Template library). The layouts of applications should be limited to the main use cases.

For example, one system integrator could want the music application to be a simple list of albums to choose from, while another could want the same information represented in a grid. This simple difference should be possible by using different templates already provided by the user interface library.

Template Extension

In addition to picking layouts from user interface library-provided templates, it should also be possible to take existing templates and change them with the minimal of copy & pasting (see Template extension).

For example, a system integrator could want to change the order of labels in a track information view. The default order in the library-provided template could be track name and then artist name, but said system integrator wants the artist name first, followed by the track name. This kind of change is too fundamental to do in CSS so a template modification is required. The system integrator should be able to take the existing library-provided template and make minimal modifications and minimal copy & pasting to change the order.

Template Modularity

Templates should be as modular as possible in order to break up the parts of a design into smaller parts. This is useful for when changes are required by a system integrator (see Templates, Template extension). If the entire layout is

in one template, it is difficult to make small changes without having to copy the entire original template.

Fine-grained modularity which leads to less copy & pasting is optimal because it makes the template more maintainable, as there's only one place to change if a bug is discovered in the original library-provided template.

Custom Widgets in Templates

A system integrator should be able to use custom widgets they have written for the particular variant in the template format (see Custom widget usage). The responsibility of compatibility with the rest of the user interface of custom widgets is on the widget author.

Documentation

With a library of widgets and models available to the system integrator, the options of widgets and ways to interact with them should be well documented (see Template library). If signals, signal callbacks, and properties are provided these should all be listed in the documentation for the system integrator to connect to properly.

Widget Interfaces

When swapping a widget out for another one in a template it is important that the API matches so the change will work seamlessly. To ensure this, widgets should implement core interfaces (button, entry, combobox, etc.) so that when swapped out, views will continue to work as expected using the replacement widget. Applications should only use API which is defined on the interface, not on the widget implementation, if they wish for their widgets to be swappable for those in another variant.

As a result, system integrators swapping widgets out for replacements should check the API documentation to ensure that the interface implemented by the old widget is also implemented in the new widget. This will ensure compatibility.

GResources

If an application is loading a lot of templates from disk there could be an overhead in the input/output operation in loading them. A way around this is to use GResource[4]s. GResources are useful for storing arbitrary data, such as templates, either packed together in one file, or inside the binary as literal strings. It should be noted that if linked into the binary itself, the binary will have to be rebuilt every time the template changes. If this is not an option, saving the templates in an external file using the `glib-compile-resources` binary is necessary.

[4] https://developer.gnome.org/gio/stable/GResource.html

The advantage of linking resources into the binary is that once the binary is loaded from disk there is no more disk access. The disadvantage of this is as mentioned before is that rebuilding is required every time resources change. The advantage of putting resources into a single file is that they are only required to be mapped in memory once and then can be shared among other applications.

MVC Separation

There should be a functional separation between data provider (*model*), the way in which it is displayed in the user interface (*view*), and the widgets for interaction and data manipulation (*controller*) (see example in Templates). The model should be a separate object not depending on any visual aspect of the widget.

Following on from the previous example (in Templates), the model would be the list of pictures on the system, and the two variants would use different widgets, but would attach the same model to each widget. This is the key behind being able to swap one widget for another without making code changes.

This separation would push the *model* and *controller* responsibility to the user interface library, and an application would only depend on the *model* in that it provides the data to fill said model.

Language Support

All widgets should be linked into a language translation system such that it is trivial not only for the user to change language (see Language), but also for new translations to be added and existing translations updated (see Ota updates).

Animations

Animations in use in widgets should be configurable by the system integrator (see Animations for examples). These animations should be used widely across the system to ensure a consistent experience. Applications should expose a fixed set of transitions which can be animated so system integrators can tell what can be customised.

Scripting Support

The widgets and templates should be usable from a UI design format, such as GtkBuilder[5] or ClutterScript[6]. This includes custom widgets. This would enable application authors to quickly prototype applications (see Prototyping).

[5] https://developer.gnome.org/gtk3/stable/GtkBuilder.html#GtkBuilder.description

[6] https://developer.gnome.org/clutter/stable/ClutterScript.html#ClutterScript.description

Day & Night Mode

The user interface should change between light and dark mode when outside the vehicle becomes dark in order to not shine too brightly and distract the user (see Day night mode).

View Management

A method of managing application views (see View) should be provided to application authors (see View management). On startup the application should provide its views to the view manager. From this point on the responsibility of constructing views, switching views, and showing view animations should be that of the view manager. The view manager should pre-empt the construction of views, but also be sensitive to memory usage so not load all views simultaneously.

Speed Lock

Some features and certain behaviour in the user interface should be disabled or modified respectively when the vehicle is moving (see Speed lock). It should be possible to customise whether each item listed below is disabled or not as it can depend on the target market of the vehicle (see Geographical customisation). Additionally, it should be up to the system to enforce the disabling of the following features and should not be left completely up to application authors (see System enforcement).

Scrolling Lists

The behaviour of gestures in scrolling lists should be altered to remove fast movements with many screen updates. Although still retaining similar functionality, gestures should cause far fewer visual changes. For example, swiping up would no longer start a kinetic scroll, but would move the page up one tabulation.

Text

Text displayed should either be masked or altered to remove the distraction of reading it while operating the vehicle, depending on the nature of the text.

- SMS messages and emails can have dynamic content so they should be hidden or masked.
- Help text or dialog messages should have alternate, shorter messages to be shown when the speed lock is active.

List Columns

Lists with columns should limit the number of columns visible to ensure superfluous information is not distracting. For example, in a contact list, instead of

showing both name and telephone number, the list could should show only the name.

Keyboard

The keyboard should be visibly disabled and not usable.

Additionally, default values should be available so that operations can succeed without the use of a keyboard. For example when adding a bookmark when the vehicle is stationary the user will be able to choose a name for the new bookmark before saving it. When the vehicle is moving the bookmark will be automatically saved under a default name without the user being prompted for the name. The name (and other use cases of default values) should be modifiable later.

Pictures

Superfluous pictures used in applications as visual aids which could be distracting should be hidden. For example, in the music application, album covers should be hidden from the user.

Video Playback

Video playback must either be paused or the video masked (while the audio continues to sound).

Map Gestures

As with kinetic scrolling in lists (see Scrolling lists), the gestures in the map widget should make fewer visual changes and reduce the number of distractions for the user. Similar to the kinetic scroll example, the map view should move by a fixed distance instead of following the user's input.

Web View

Any web view should be masked and not showing any content.

Insensitive Widgets

When aforementioned functionality is disabled by the speed lock, it should be made clear to the user what has been modified and why.

Approach

Templates

The goal of templates is to allow an application developer to change the user interface of their application without having to changing the source code. These

are merely templates and have no way of implementing logic (if/else statements). If this is required, widget code customisation is required (see Custom widgets).

Specification in ClutterScript

ClutterScript is a method for creating user interfaces from JSON files. An example is shown below which describes variant A application chooser user interface:

```
[{
  "id": "model-categories",
  "type": "LightwoodAppCategoryModel"
},
{
  "id": "model-apps",
  "type": "LightwoodAppModel"
},

{
  "id": "window",
  "type": "LightwoodWindow",
  "children": [
    {
      "id": "roller-categories",
      "type": "LightwoodRoller",
      "model": "model-categories",
      "app-list": "roller-apps",
      "signals": [
        { "name": "activated", "handler": "category_activated_cb" }
      ]
    },
    {
      "id": "roller-apps",
      "type": "LightwoodRoller",
      "model": "model-apps",
      "signals": [
        { "name": "activated", "handler": "app_activated_cb" }
      ]
    }
  ]
}]
```

The first two objects created (`model-categories` and `model-apps`) are models for the application categories available on the system, and the applications available on the system—due to their class names (`LightwoodAppCategoryModel` and `LightwoodAppModel` respectively). These models are not widgets visible in the user interface, but proper widgets will refer to them later in the template.

The next entry describes the main window in the user interface, inside of which there is a horizontal box (with some style properties set), with two children that are both of type `LightwoodRoller`. Although these widgets are of the same type, they are different instances and they have been given different models. The first roller widget (on the left) has been given the `model-categories` model and the second roller widget (on the right) has been given the `model-apps` model, both created at the beginning of the JSON file.

Additionally the `LightwoodRoller::activated` signal is connected on both rollers to different callbacks. The signal callback names are listed in the application documentation. In this case, when the left-hand roller with categories is changed (activated), the right-hand roller with applications is updated (set by the `app-list` property).

Another example to compare is given below with the B-variant application chooser user interface:

```
[{
  "id": "model-apps",
  "type": "LightwoodAppModel"
},

{
  "id": "window",
  "type": "LightwoodWindow",
  "children": [
    {
      "id": "list-apps",
      "type": "LightwoodList",
      "model": "model-apps",
      "signals": [
        { "name": "activated", "handler": "app_activated_cb" }
      ]
    }
  ]
}]
```

The differences of the B-variant application chooser in comparison to the A-variant application chooser are:

1. There is no categories model and no categories roller.
2. There is no more box inside the main window widget.
3. The list widget is a `LightwoodList` instead of a `LightwoodRoller`. This is a visual difference dictated by the widget implementation and chosen for this variant, but the data backend for both lists (in the model `model-apps`) is unchanged. Both widgets should implement a common `LightwoodCollection` interface.

These are just two examples of how an application chooser could be designed. The user interface files contain minimal theming as that is achieved in separate CSS files (see Theming).

Typically, applications will come with many templates for system integrators to either use, or take guidance from.

Properties, Signals, and Callbacks

The GObject properties that can be set, the signals that can be connected to, and the signal callbacks that can be used, should be listed clearly in the application documentation. This way, system integrators can customise the look and feel of the application using already-written tools.

When changing a template to use a different widget it might be necessary to change the signal callbacks. This largely depends on the nature of the change of widget but signals names and signatures should be as consistent as possible across widgets to enable changing as easily as possible. If custom callbacks are used in the code of an application, and the callback signature changes, recompilation will be necessary. The signals emitted by widgets and their type signatures are defined in their interfaces, documented in the API documentation.

For example, in the examples above, a `LightwoodRoller` widget for listing applications was changed to a `LightwoodList` and the activated signal remained connected to the same `app_activated_cb` callback.

Template Inheritance

At the time of writing, ClutterScript has no way of referring to objects from other JSON files or of making an object a modified version of another. A proposal to modify `ClutterScriptParser` to support this feature is as follows (this change would take a couple of days to implement):

An `app-switcher.json` file contains the following objects defined:

```
[{
  "id": "view-header",
  "type": "LightwoodHeader",
  "height": 100,
  "width": 200
},

{
  "id": "view-footer",
  "type": "LightwoodFooter",
  "height": 80
},

{
```

```
  "id": "app-switcher",
  "type": "LightwoodWindow",
  "color": "#ff0000",
  "children": [ "view-header", ··· , "view-footer" ]
}]
```

Header and footer objects are defined (`view-header` and `view-footer`) each with `height` properties (100 and 80 respectively). An `app-switcher` object is also created with the `color` and `children` properties set. Note that the `children` property is a list referring to the header and footer objects created before. (The ellipsis between said objects marks the omission of other objects between header and footer for brevity.)

If a system integrator wanted to give the same appearance to their app switcher view but wanted to change the height of the header and the colour of the main app switcher, without copy & pasting a lot of text to redefine all these objects, objects can simply extend on previous definitions. For example, in a `my-app-switcher.json`:

```
[{
  "external-uri": "file:///path/to/app-switcher.json",
  "id": "view-header",
  "height": 120
},

{
  "external-uri": "file:///path/to/app-switcher.json",
  "id": "app-switcher",
  "color": "#ffff00"
}]
```

Referencing objects defined in other files can be achieved by specifying the `external-uri` property pointing to the other file, and the `id` property for selecting the external object.

In this example, the `view-header` object is extended and the `height` property is set to 120. All other properties on the original object remain untouched. For example, the `width` property of the header remains at 200.

It is possible to simply to refer to objects in other files without any changes. Each external object must be referred to separately as they are not automatically brought into scope after the first `external-uri` reference. For example:

```
{
  "id": "example-with-children",
  ...
  "children": [
    "first-child",
    {
```

```
        "id": "second-child",
        "type": "ExampleType"
      },
      {
        "external-uri": "file:///path/to/another.json",
        "id": "third-child"
      }
    ]
}
```

In this example, this object has three children:

1. An object called `first-child`, defined elsewhere in the JSON file.
2. An object called `second-child`, defined inline of type `ExampleType`.
3. An object called `third-child`, defined in `another.json`.

In these examples, the `external-uri` used the `file://` URI scheme, but others supported by GIO can be used. For example, templates in GResources can be used using the `resource://` URI scheme.

Application authors should not use templates and inheritance excessively such that every single object is in a separate file. This will cause more disk activity and could potentially slow down the application. Templates should be broken up when clarity is in question or when a non-trivial object is to be used across in other views.

Widget Factories

If a system integrator wants to replace a widget everywhere across the user interface, they can use a widget factory to replace all instances of said old widget with the new customised one. This is achieved by using a factory which overrides the type parameter in a ClutterScript object.

For example, if a system integrator wants to stop using `LightwoodButtons` and instead use the custom `FancyButton` class, there are no changes required to any template, but an entry is added to the widget factory to produce a `FancyButton` whenever a `LightwoodButton` is requested. Templates can continue referring to `LightwoodButton` or can explicitly request a `FancyButton` but both will be created as `FancyButtons`. If an application truly needs the older `LightwoodButton`, it needs to create a subclass of `LightwoodButton` which is not overriden by anything, and then refer to that explicitly in the template.

Custom Widgets

Apertis widgets can be subclassed by system integrators in variants and used by application developers by creating shared libraries linking to the Apertis widget library. Applications then link to said new library and once the new widgets are registered with the GObject type system they can be referred to

in ClutterScript user interface files. If a system integrator wants a radically different widget, they can write something from scratch, ensuring to implement the appropriate interface. Subclassing existing widgets is for convenience but not technically necessary.

Apertis widgets should be as modularised as possible, splitting functionality into virtual methods where a system integrator might want to override it. For example, if a system integrator wants the roller widget to have a different activation animation depending on the number of items in the model, they could create a roller widget subclass, and override the appropriate virtual methods (in this case activate) and update the animation as appropriate:

```
G_DEFINE_TYPE (MyRoller, my_roller, LIGHTWOOD_TYPE_ROLLER)
static void
my_roller_init (MyRoller *self)
{
}

static gboolean
my_roller_activate (MyRoller *self,
gint item_id)
{
  LightwoodRoller *roller;
  LightwoodRollerClass *roller_class;
  LightwoodModel *model;

  roller = LIGHTWOOD_ROLLER (self);
  roller_class = LIGHTWOOD_ROLLER_GET_CLASS (roller);
  model = lightwood_roller_get_model (roller);

  if (lightwood_model_get_n_items (model) > 5) {
    /* change animation */
  } else {
    /* reset animation */
  }

  /* chain up */
  return roller_class->activate (roller, item_id);
}

static void
my_roller_class_init (MyRollerClass *klass)
{
  LightwoodRollerClass *roller_class = LIGHTWOOD_ROLLER_CLASS (klass);

  roller_class->activate = my_roller_activate;
}
```

Another example is if the system integrator wants to change another part of the roller when scrolling starts, the appropriate signal can be connected to:

```
G_DEFINE_TYPE (MyRoller, my_roller, LIGHTWOOD_TYPE_ROLLER)

static void
my_roller_scrolling_started (MyRoller *self,
                             gpointer user_data)
```

```
{
  /* scrolling has started here */
}

static void
my_roller_constructed (GObject *obj)
{
  /* chain up */
  G_OBJECT_GET_CLASS (obj)->constructed (obj);
 g_signal_connect (obj, "scrolling-started", G_CALLBACK (my_roller_scrolling_started), NULL);
}

static void
my_roller_class_init (MyRollerClass *klass)
{
  GObjectClass *object_class = G_OBJECT_CLASS (klass);
  object_class->constructed = my_roller_constructed;
}
```

In the template, this variant would stop referring to `LightwoodRoller` and instead would use `MyRoller`, or update the widget factory entry (see Widget factories).

Models

Data that is to be displayed to the user in list widgets should be stored in an orthogonal model object. This object should have no dependency on anything visual (see MVC separation).

The actual implementation of the model should be of no importance to the widgets, and only basic model interface methods should be called by any widget. It is suggested to use the GListModel[7] interface as said model interface as it provides a set of simple type-safe methods to enumerate, manipulate, and be notified of changes to the model.

As `GListModel` is only an interface, an implementation of said interface should be written, ensuring to implement all methods and signals, like `GListStore`.

Theming

Using the `GtkStyleContext` object from GTK+ is wise for styling widgets as it can aggregate styling information from many sources, including CSS. GTK+'s CSS parsing code is advanced and well tested as GTK+ itself switched its Adwaita[8] default theme to pure CSS some time ago, replacing theme engines that required C code to be written to customise appearance.

[7] https://developer.gnome.org/gio/stable/GListModel.html
[8] https://git.gnome.org/browse/gtk+/tree/gtk/theme/Adwaita

Said parser and aggregator support multiple layers of overrides. This means that CSS rules can be given priorities and rules are followed in a specific order (for example theme rules are set, and can be overridden by variant rules, and can be overridden by application rules, where necessary). This is ideal for Apertis where themes set defaults and variants need only make changes where necessary.

Clutter Widgets

The `GtkApertisStylable` interface is a mixin for any GObject to enable use of a `GtkStyleContext`. It is an Apertis-specific interface and therefore not candidate for upstreaming, and will need maintaining as the CSS machinery in GTK+ changes over time.

Every existing Clutter widget will have to be manually taught to use the style context and any special requirements will also need to be applied from the style context as necessary.

Theme Changes

Applications should listen to a documented GSettings[9] key for changes to the theme and icon theme. Changes to the theme should update the style properties in the `GtkStyleContext` and will trigger a widget redraw and changes to the icon theme should update the icon paths and trigger icon redraws.

Language Support

GNU gettext[10] is a well-known system for managing translations of applications. It provides tools to scan source code looking for translatable strings and a library to resolve said strings against language files which are easily updated without touching the source code of said applications.

Language Changes

Applications should listen to a documented GSettings key for changes to the user-chosen language, then re-translate all strings and redraw.

Updating Languages

Language files for GNU gettext saved into the appropriate directory can be easily used immediately with no other changes to the application. Over the air (OTA) updates can contain updated language files which get saved to the correct location and would be loaded the next time the application is started.

[9] https://developer.gnome.org/gio/stable/GSettings.html
[10] https://www.gnu.org/software/gettext/

Day & Night Mode

Inspired by GTK+'s *dark mode*[11], variant CSS should provide a `dark` class for widgets to be used in night mode. If the `dark` class is not set the user interface should be in day mode. CSS transitions[12] should make the animation smooth.

A central GSettings key should be read to know when the system is in day or night mode. It will be modifiable for testing and in development.

View Management

At the time of writing, Clutter does not have a built-in view management system. GTK+ has GtkStack[13] for managing views and displaying animations when moving between one view and another. The useful parts of `GtkStack` could be migrated to Clutter (subject to suitable licensing) to re-use the functionality and user testing and not waste effort in reimplementing everything from scratch. Existing view management systems (for example, in `libthornbury`) should also be considered for this migration task.

Speed Lock

There should be a system-operated service that determines when the vehicle is moving and when it is stationary. From this point the Apertis widgets and applications should change when and where appropriate.

There should be a GSettings key which indicates whether the speed lock is active or not. This key should only be modifiable by said system-operated service and should be readable by the entire system.

Apertis Widgets

Applications that use Apertis widgets extensively should have very little to modify to support the speed lock. Apertis widgets should read and monitor the GSettings key to change their content when necessary:

- Lists with kinetic scrolling – disable the kinetic scrolling (see Scrolling lists).
- Text – very long texts in text views or label widgets should be hidden if there is no alternative provided (see Text).
- Keyboard – do not show the keyboard and provide feedback to the application as to whether the keyboard could appear or not (see Keyboard).
- Pictures – mast the picture shown in the picture widget (see Pictures).

[11] https://developer.gnome.org/gtk3/stable/GtkSettings.html#GtkSettings--gtk-application-prefer-dark-theme
[12] http://www.w3schools.com/css/css3_animations.asp
[13] https://developer.gnome.org/gtk3/stable/GtkStack.html

- Video playback – either pause the video completely or just mask the video and keep the audio sounding (see Video playback).
- Map – disable the kinetic scrolling (see Map gestures).
- Web view – mask the contents entirely (see Web view).

Apertis widgets should fill text widgets with contents that can differ depending on whether the speed lock is active or not.

List Columns

The number of columns visible should be reduced to remove superfluous information when the speed lock is active (see List columns). The nature of every list can be different and the detection of superfluous information is impossible automatically. There should be a way of either application authors specifying which columns should be hidden, or it should be left up to the application itself. If the latter is not an option (see enforcement comments in Speed lock), the entire list widget should be masked to hide its contents.

Keyboard

As mentioned in Keyboard, applications should deal with the possibility that the keyboard may not be available at any given time, if the speed lock is active. In the case that the keyboard request is denied, the application should change its user experience slightly to accommodate for this, such as the example with bookmarks given previously.

The change of user experience also means there must be other ways in which users can edit named items using default values after the speed lock has been disabled.

Templates

Apertis-provided templates should have versions for when the speed lock is activated and Aperis widgets should switch to these templates accordingly.

Insensitive Widgets

As highlighted in Insensitive widgets, it should be made obvious to the user when functionality is disabled, and why. There should be a uniform visual change to widgets when they have been made insensitive so users can immediately recognise what is happening.

A documented CSS class should be added to widgets that are made insensitive by the speed lock so that said widgets follow an identical change in display.

Notifications

Pop-up notifications or a status bar message should make it clear to the user that the speed lock is active and if appropriate, highlight the current functionality that has been disabled.

Masking Unknown Applications

Applications can technically implement custom widgets and not respect the rules of the speed lock. As a result, applications which haven't been vetted by an approved authority should not be able to be run when the speed lock is active. When they are already running and the speed lock is activated, they should be masked and the user should not be able to interact with them.

This behaviour should be customisable and possibly only enabled in a region in which laws are very strict about speed lock restrictions.

References

GTK+ Migration

In an older version of this document it was posed that a move from Clutter to GTK+ might be wise. It has been decided that for the time being a move is unwise due to the immature nature of the GTK+ Scene Graph Kit.

The following sections were removed from the previous sections in this document and have been left here for future reference.

GTK+ or Clutter

The following suggestions are possible using either the GTK+ or Clutter libraries. Existing code is currently written in Clutter, but a move to GTK+ could be wise because GTK+ is still highly used and maintained, whereas Clutter is less used and less maintained. The maintainers of Clutter have even announced that planned future additions to GTK+ would deprecate Clutter[14]. Although not in stone, the deprecation is planned[15] for the 3.20 release of GTK+ which is planned in March 2016.

It is worth noting that Clutter widgets can be embedded inside GTK+ applications, and GTK+ widgets can be embedded inside Clutter applications, but there are many problems with input and ensuring GTK+ functions are only called from GTK+ callbacks, so following this path is likely not worth the eventual problems.

For completeness, the following sections with toolkit-specific approaches are split into two such that both GTK+ and Clutter paths can be considered.

[14] https://www.bassi.io/articles/2014/07/29/guadec-2014-gsk/
[15] https://wiki.gnome.org/Projects/GTK%2B/Roadmap

Specification in GtkBuilder

GtkBuilder is a method for creating user interfaces from XML files. An example is shown below which describes the A-variant application chooser user interface:

```
<interface>
  <object class="LightwoodAppCategoryModel" id="model_categories" />
  <object class="LightwoodAppModel" id="model_apps" />
  <object class="LightwoodWindow" id="window">
    <child>
      <object class="GtkBox" id="hbox1">
        <property name="homogeneous">True</property>
        <property name="orientation">GTK_ORIENTATION_HORIZONTAL</property>
        <child>
          <object class="LightwoodRoller" id="roller_categories">
          <property name="model">model_categories</property>
          <property name="app-list">roller_apps</property>
          <signal name="activated" handler="category_activated_cb" />
          </object>
        </child>
        <child>
          <object class="LightwoodRoller" id="roller_apps">
            <property name="model">model_apps</property>
            <signal name="activated" handler="app_activated_cb" />
          </object>
        </child>
      </object>
    </child>
  </object>
</interface>
```

The first two objects created (`model_categories` and `model_apps`) are models for the application categories available on the system, and the applications available on the system—due to their class names (`LightwoodAppCategoryModel` and `LightwoodAppModel` respectively). These models are not widgets visible in the user interface, but proper widgets will refer to them later in the template.

The next entry describes the main window in the user interface, inside of which there is a horizontal box (with some style properties set), with two children that are both of type `LightwoodRoller`. Although these widgets are of the same type, they are different instances and they have been given different models. The first roller widget (on the left) has been given the `model_categories` model and the second roller widget (on the right) has been given the `model_apps` model, both created at the beginning of the XML file.

Additionally the `LightwoodRoller::activated` signal is connected on both rollers to different callbacks. The signal callback names are listed in the application documentation. In this case, when the left-hand roller with categories is changed

(activated), the right-hand roller with applications is updated (set by the `app-list` property).

Another example to compare is given below with the B-variant application chooser user interface:

```
<interface>
  <object class="LightwoodAppModel" id="model_apps" />
  <object class="LightwoodWindow" id="window">
    <child>
      <object class="LightwoodList" id="list_apps">
        <property name="model">model_apps</property>
        <signal name="activated" handler="app_activated_cb" />
      </object>
    </child>
  </object>
</interface>
```

The differences of the B-variant application chooser in comparison to the A-variant application chooser are:

1. There is no categories model and no categories roller.
2. There is no more box inside the main window widget.
3. The list widget is a `LightwoodList` instead of a `LightwoodRoller`. This is a visual difference dictated by the widget implementation and chosen for this variant, but the data backend for both lists (in the model `model_apps`) is unchanged.

These are just two examples of how an application chooser could be designed. The user interface files contain minimal theming as that is achieved in separate CSS files (see Theming).

Typically, applications will come with many templates for system integrators to either use, or take guidance from.

GTK+ Widgets

Support for `GtkStyleContext` inside GTK+ widgets is already present. Widgets inside the GTK+ library (and therefore also their subclasses) already talk to the style context and are drawn according to custom styling.

New GTK+ widgets with special requirements would need to get the appropriate style information from the style context and apply it as necessary. This is documented in the GTK+ documentation and it is easy to find examples of it in the source code.

Appendix

Variant Differences

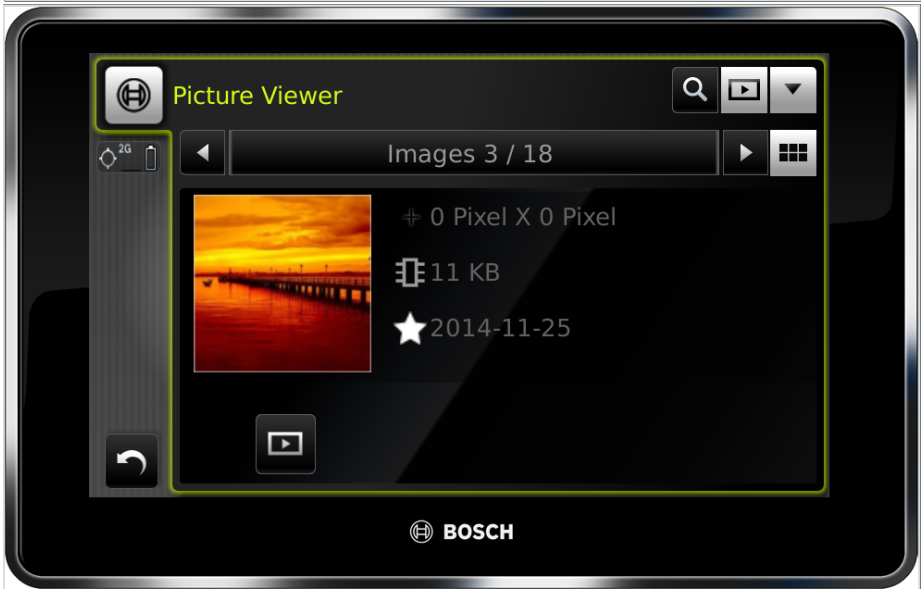
Thumbnail View

- Re-used:
  - Roller

  - Views drawer

- Differences:
  - In variant A, one needs to go back to the app launcher to start the photo viewer with tags/title/date as they are all separate apps; in B it is in the same app.

Detail View



- Re-used: nothing
- Differences:
  - In variant A it is a roller; in B it is an individual image.
  - In variant B there is a media info widget; in A it's a roller on the right.

List View

- Re-used: nothing
- Differences:
  - The roller is different (it displays different information).
  - In variant B one can delete; in A the feature is not present.

Full Screen

- Re-used: nothing
- Differences:
  - The full screen is a roller in variant A; in B it is a single snapshot.
  - In variant B there are many extra functions; in A these functions are not present.