



## **Imendio's vision on GTK+**

---

**As presented at the Berlin 2008 hackfest**

*Version 1.1*

# 1. Introduction

At Imendio we have been thinking for a long time where we want GTK+ to end up. We feel that GTK+ has a large user base, who expect the project to move forward in a steady phase and deliver more and more features that are also common in other toolkits. Moving forward also means having a long-term vision what to aim for. In this section we will outline what our long-term vision for GTK+ is.

If we look at other toolkits around us, we are seeing that these are making great strides in creating a nicer user experience. Progress is being made on making the user experience more vibrant and lively by the use of animations, the use of non-standard user interface components and clearer interactions between user interface components. For GTK+ we envision adding new layouting paradigms, such as having alpha-transparent widgets to be able to stack user interface on top of each other. This is seen in for example Apple's iPhoto, where you can retouch images using an alpha-transparent overlay with controls. We want to be able to create subtle transitions between different states of the user interface and not distract the user with sudden updates. Concepts derived from physics, such as gravity and elasticity, should be used to make the interface feel more natural and easier to use.

Compared to other toolkits developing GTK+ applications is complicated for newcomers. We want to improve the life of the application developers by a few orders of magnitude. Ways to do this are by drastically improving the current support and tools for UI building and making it possible to do rapid prototyping by quickly writing a new part of an existing application in a scripting or other higher-level language. Having the full API of GTK+ specified in IDL will enable us to create even higher quality and easier to maintain and use language bindings than we currently have. These language bindings can then become first-class citizens and even be used to implement parts of the library stack in. With a data abstraction layer, it will be very easy to create user interfaces for a given database schema, by automatically generating parts of the UI using the improved tools.

In recent years the cross platform support of GTK+ has been greatly improved. A single GTK+ application will run on Linux, as well as on Windows and Mac OS X. Writing the proper back-ends to make this possible is currently very hard, by reworking GDK and the layer between GDK and the back-ends we want to make it much easier to implement back-ends for GTK+. As well as creating new ports of the GTK+ easier, this will also simplify the current back-ends by a great deal, making them much easier to maintain and more stable. Of course having working user interface elements is only one part of a toolkit. We want to strive for even better platform integration by looking into integrating with the native configurations systems, printing and file chooser dialogs, clipboard, and many other things.

A toolkit cannot survive without good maintenance. GTK+ has been accumulating a lot of new widgets and features, removing the need for keeping the older features around. Once a project starts to be a host to a lot of outdated and unused code, its health will be affected. We can only move the project forward in the areas described above by improving the maintainability of the project. In order to achieve this we believe that we must make it possible to evolve the toolkit in iterations with adding, deprecating and removing API. It will give us the power to get the API right in the end using feedback from hands-on experience and make sure mistakes will be removed and not have an infinite life in the toolkit. Automated UI testing of the core and the refactoring of the inner workings of GDK, GTK+ and the back-ends will lower the barrier for new contributors to join in and help us in creating the perfect cross-platform

toolkit. Finally, having the full API specified in IDL will give us the possibility to move away from a C-based core, should that prove to be a better solution.

In the remainder of this document we will outline our plan to realize this long-term vision. We think that a change in development methodology is needed. We will describe the changes that we feel are required and why. A roadmap will be presented for the migration to 3.0 and beyond. For the beyond part, we will describe a few features that we expect to be able to implement short-term.

## 2. New features short-term

One of the main things that the users of a new toolkit always want are new features. Features that will make the applications more appealing to end-users, features that will make the life of the application developer easier. In this section we will present some of the main items we would like to see appear in the 3.x series.

### Alpha transparency and widget stacking

All widgets should fully support alpha transparency. This will open the doors for creating much more complex and interesting user interfaces. Together with a container that supports overlapping widgets, or widgets that have been stacked on top of each other it will be very easy to create user interfaces that in fact “manipulate” the background. Think of for example the photo editing controls in Apple’s iPhoto. Or the editable widgets of GtkTreeView, while this is already possible using popup windows, the ability to stack widgets on top of each other easily will greatly simplify the implementation. As an additional bonus we could even make the editable widget slightly transparent.

### Easier to use layouting

Layouting using GTK+ is very powerful, but can sometimes be painful. We still believe that GTK+’s box model is one of the best paradigms for layouting widgets out there. In order to improve this even more, we want to create an easier to use container, with sane defaults. By including flags for setting hspread or vsread (which will make the widget in the box fully extend horizontally and vertically without having to set the correct parameters on all parent boxes) and strings and struts we hope to make this powerful layouting even easier to use.

### Transitions, physics and other animations

In order to make user interfaces more appealing and natural to use, we want to work on getting animations throughout the core widgets of the toolkit. Many widgets will have to be changed internally for this. We want to have support for creating easy transitions between different states of the user interface. The user will then be presented with a more smooth and comfortable user experience. As examples, one can think of notebook tabs that fade in/out or morph. Or smooth expand and collapse animations in GtkTreeView and GtkExpander.

Physics in the user interface will also make the user feel more comfortable with the interface. When dragging a few sentences from one GtkTextView to another, the items at the drop spot can smoothly move away and some amount of gravity will be applied to the mouse cursor helping the user to drop the data where he wants to have it. Another famous example is the unlock slider on the iPhone, on release it should shoot back to its original position, resembling the physics of a spring.

### Usage of an IDL

We envision the API of future releases of GTK+ to be fully expressible in an Interface Description Language (IDL). This will greatly simplify things for the maintainers of

language bindings and improve the quality and completeness of bindings significantly. This way, other languages than C can become full first-class citizens.

### Theming improvements

In discussions and meetings with artists creating themes for GTK+ it has become apparent that the theming support in GTK+ can use some improvements. Because of the original design of GtkStyle and it being at the very heart of GTK+, it is very hard to make changes in a compatible way. In the future, we want to look at improving the support for theming, working on this together with authors of the major theme engines available. Artists have also been saying that the gtkrc file format is very hard to use. This could possibly be improved by trying to change it towards something that artists are familiar with, like for example CSS.

## 3. The need for a change in development model

We believe, together with many others, that the way we currently develop GTK+ is not future-proof. Our promise of not breaking the ABI of the toolkit has put GTK+ in a street with a dead-end, at some point development will slow down and eventually cease, because it will become impossible to change things. We see this already happening and constantly aggravating<sup>1</sup>. This is not a healthy policy for a software project as large as GTK+. In our opinion this should change.

The project should get the possibility to break both API and ABI on a regular basis. Then old, deprecated code can be removed from the library, instead of accumulating. All structures with public fields should be sealed -- that is, all public fields will be removed. As long as we expose public fields of object structures, we will be severely limited in making functional changes to the code and even plain refactoring, which does not change the functionality. This is hurting us already in several areas. We could not update the old implementation of the tooltips API, public fields were exposed (although marked "private") that made it impossible for us to change the internal workings of the tooltips code in a compatible fashion. Even the simple change to use a hash table instead of a list for the tips data to improve performance did not occur without any outcries. Right now the old field with a pointer to the list for tips data is still there, but unused. The application developers actually using this field better have some NULL pointer checks in place.

More problematic areas can be found. For example the GtkStyle structures, or extending/replacing GtkStateType. Almost every application accesses several of the fields in GtkStyle directly. It is impossible to migrate this to a purely cairo-based GtkStyle API. We cannot replace the fields, change of the ordering or types of the fields. The GtkColor and GdkGC arrays cannot be replaced with cairo equivalents, no new state types can be added. The application programmers can also happily change the values of these fields, this makes it impossible to have a "compatibility" layer where we have the data in cairo structures in private and provide a copy of that in the old public fields. Next to requiring us to keep all old legacy API for GdkColor and GdkRC in the library, we cannot even catch all write accesses to these public fields, which are required to properly update the private fields and thus ensure compatibility. If these public fields could only be accessed via functions, we could provide a feasible migration path.

---

<sup>1</sup> See <http://blogs.gnome.org/timj/2006/12/20/20122006-gtk-core-maintainer-shortage/> and <http://mail.gnome.org/archives/gtk-devel-list/2006-December/msg00074.html>

Using accessor functions instead of direct field accesses we can provide a much better migration path. In contrast to public structure fields, we can easily change the data type of the private, hidden, fields and update all functions accessing them. The accessor functions will not change, compatibility will not be broken at all. In contrast to structure fields, functions can be easily modified to implement their functionality on top of a new, replacement, API. This will provide a working solution until the application developer has fully migrated to the new API. A function-only API is also very easy to express in an IDL and will greatly simplify the creation of language bindings.

Obviously, while breaking ABI is easy to do for the library developers, it incurs a lot of pain on application developers and distributors. Things get nasty when one is writing a GTK+ 3 application that depends on a GTK+ 2 library. Then GTK+ 2 will be pulled in, causing numerous symbol clashes between GTK+ 2 and GTK+ 3. One solution to this problem is to move to another namespace for GTK+ 3, when both GTK+ 2 and 3 are pulled in, symbol clashes will not occur. We are of the opinion that this does not solve the problems at all. While you can now include GTK+ 2 and 3 at the same time, it will not be very usable. Problems will turn up by having multiple connections to the X server, multiple main loops, multiple thread initializations, etc. These problems are very hard to solve or even unsolvable, so there is no gain in changing namespace.

Another way to move GTK+ forward and avoid an ABI break is to add a new widget system, independent from the existing one, and then integrate both ways: make it possible to use new widgets in an application using the old widget system and vice versa. Apart from the full integration between `GtkWidget` and "NewWidget" not being trivial at all, we believe that the work that is required to realize this amounts to almost the same as writing a new toolkit. In addition, we will not change the development model, making chances high that we will get stuck at the same problems as now in the future.

In our opinion the best solution is to move to GTK+ 3 by breaking ABI and make sure that the entire library stack is parallel installable. Old GTK+ 2 applications will continue to function properly. Applications and libraries will be incrementally ported. The only issue that will turn up are GTK+ 2 dependent libraries that are not parallel installable. During the course of the 2.x development line, the GTK+ library package demonstrated the needed measures to be taken to allow parallel installability. Dependent libraries will need to be adopted in a similar fashion in order to make use of GTK+ 3. With past experience the GTK+ team is able to assist library developers and distributors to achieve the necessary adoption steps.

In short, sealing structures and breaking API and ABI regularly will allow us to adopt a new development model wherein we can incrementally remove old code and replace core code with new implementations, with better APIs and more flexibility. The APIs can still be improved and refactored once they have been in a few GTK+ release already. Old code can really be removed now, making the library smaller and easier to maintain and get into.

## 4. A new development policy

Based on the findings in the last section, we have started to draft a development policy. A lot of GTK+ development right now happens using an "unwritten" policy that all core maintainers know about. New developers only learn about this policy by writing patches, getting patch review and in such a way gradually becoming part of the core pool of GTK+ contributors. We think that it is very important to document the new development policy if we are going to adopt a new development model. Such a docu-

ment will avoid confusion later on and basically provide application developers with a "contract" of what they can, and cannot, expect from the GTK+ team.

The points below have been drafted in the name of the GTK+ team, therefore "we" indicates the GTK+ team and not Imendio.

### **Possible to break API and ABI with every major release**

We need the possibility to break API and ABI every few years with major releases and will make active use of it. We will ensure to make the migration to a new major release as smooth as possible, possibly by proving source compatibility from the last release in the previous major series to the new major series. In order for this to work without causing too much breakage amongst the platform, the entire library stack has to be parallel installable. All libraries depending on glib and/or GTK+ should aim to be parallel installable as well to lower the amount of pain for the users. This is as far as we can help as the GTK+ team, the actual packaging is the task of the Linux distributors. We will aim to put out a new major release every regularly (For instance 3 years is an overseeable period). When we release a new major release we announce a deadline for the next major release. A major release only breaks ABI by only removing API that has been deprecated in the previous minor release and does not add new deprecations.

### **Remove deprecated code after a defined period of time**

API that has been marked as deprecated will be removed after a defined period. One possible option is to remove deprecated API in the next major release. We will provide developers with tools to make it easy to spot the usage of deprecated code. For example at run-time spit out messages for each deprecated function used. The deprecation process will work as follows:

- In each minor release API can be deprecated. Deprecated API will cause compile time warnings unless compatibility switches are provided. In addition, we will add run-time warnings to these deprecated functions. The developer will then be notified of the usage of deprecated code. These run-time warnings are only visible when enabled via an environment variable.
- During the next major release the functions will be removed. ABI compatibility is now broken.

### **Guidelines for new API**

For new API to be introduced to GTK+ we set strict requirements. The API should:

- Never expose structure internals in public header files. Structure definitions have to be confined to .c or private uninstalled header files and should generally be allocated via means like `G_TYPE_INSTANCE_GET_PRIVATE()`. The structure parts publicly exposed to allow object inheritance may only have a `*private` pointer that is not part of the public API to speed up `G_TYPE_INSTANCE_GET_PRIVATE()` lookups.
- Provide a property or getter and setter for each provided field that one should be able to access from the outside. The getter and setter can possibly be generated from the properties in the object.
- Only expose nested opaque structures and function definitions as public API (in public header files and as library symbols). These function definitions should not include C "features" such as flags or varargs, as these are not proper bindable toolkit API. The API needs to be fully expressible in an IDL. Functions using varargs are "C sugar" and can be in the library as addition to the main API for convenience to the people using C.

For existing API, the following changes are allowed and are not seen as API/ABI breaking:

- Functions can be added to interfaces.
- The return value of a function can be changed to something else, only if its current return value is void.

### Deprecated functions are disabled by default

Deprecated functions are disabled in the header files by default. The developer will have to explicitly set the compile flags to be able to use deprecated functions. We expect that developers will catch and fix their uses of deprecated code early, also helping themselves to incrementally migrate code to newer GTK+ versions overtime.

## 5. Roadmap to 3.0 and subsequent 3.x releases

The goals for GTK+ 3.0 are very simple: it is only about putting the new development model and policy into place and nothing more. This means we remove all public structure fields, apply the new development policy and remove all code that has been deprecated in the 2.x series. New features and further development will occur in the following 3.x minor releases. This enables application authors to migrate applications during the 2.x development line and do the final move to 3.0 by just rebuilding against the new GTK+ version.

We understand that getting to GTK+ 3.0 and beyond is not an easy task. Migrating a code base as large as GTK+ requires a lot of thought in advance and careful execution. To help us with this, we've created a roadmap which shows exactly what steps need to be taken in which order to make the migration process a success.

One of the important pieces of this roadmap is the plan to incrementally stop the use of the public structure fields and later remove them. For a smooth transition, we plan to introduce a `GSEAL()` macro. All public fields will first be wrapped using this `GSEAL()` macro. This has no effect on the field and the compatibility of the library unless a special compiler switch is enabled. When compiled with this switch, we can generate compiler warnings or errors about the usage of these structure fields. The definition of this macro will look like this:

```
#ifndef GSEAL
#  ifdef GSEAL_ENABLE
#    define GSEAL(ident)    __g_sealed__ ## ident
#  else
#    define GSEAL(ident)    ident
#  endif
#endif /* !GSEAL */
```

### Final 2.x series

- Provide accessor functions for everything; object and class fields. We will not provide accessors for deprecated widgets, since these will be removed in GTK+ 3.0.
- Accessor functions will be provided for all accessors that are now macros, such as `GTK_WIDGET_GET_FLAGS()`.
- We plan to implement private class data in GObject similar to private instance data, to protect all current state variables in class structures.
- Use `GSEAL()` to deprecate direct accesses to public object and class fields. The object and class fields will later be moved to private structures.

- Introduce a “diagnostic mode” for informing the developer about things that may harm the portability of the application to newer GTK+ releases. These are things such as poking inside composited widgets. This is most likely going to be a means for verbose runtime warnings, configurable via environment variables.

When plan to devise a step-by-step plan for moving all public structure fields to private structures for GTK+ 3.0, when around 70% of this plan has been achieved.

### 3.0

- Seal all structures by removing all the structure fields. Update all the core GTK+ code to use accessor functions instead.
- Remove all code that was deprecated in the 2.x series.

### Post 3.0 (3.1 and beyond)

- Look into deprecating things like the non-multihead functions in GDK, `gdk_draw_*`, etc. We will not deprecate this in 2.x (and thus remove in 3.0) to simplify the transition to 3.0.
- Work on features as outlined in section 2:
  - Transparency and stacked user interfaces.
  - Transitions, physics in UI and other animations.
  - Easier layouting.
  - Express the full API in an IDL.
  - Theming improvements.

## 6. Conclusions

In this document we’ve presented our high level long-term vision for GTK+ and what our ideas are on realizing this. We’ve highlighted problems with the current development model of GTK+ and provided alternatives, with the main goal of making GTK+ a healthy and moving code base again. The draft of the new policy and roadmap are starting points for getting this undertaking off the ground. We welcome all feedback and assistance the community can provide to realize these ideas.