Slicker Redesign

1 Reasons for redesigning Slicker

There are numerous reasons for redesigning Slicker.

- There is no good possibility to share edgealigning-code between taskbar, the slider and the cardstacks. There is an implemented class to solve this, but the rest of the design as it is now does not allow usage of common shared classes for edgealigned widgets.
- The sourcetree. The current sourcetree contains some redundant information in the directory tree, as well as some very high and complex coupling between the cardhandler and carddeskcore. One goal of the redesign would be to try cleaning up the sourcetree somewhat.
- The current design and implementation is very focused on cards. It was a natural choice to focus on the cards from the beginning, since that's one of the outstanding features of Slicker, with little resemblance to any existing project. However, the current design cares little about the slider-applets, any taskbar, or about the launcher.
- The existing plugin-framework is quite tightly bound to the CardApplets. This must be extended to cover all types of required plugins.
- Experience from the existing design and implementation have given knowledge and understanding about several properties of the system-goals. For example, it has been found that maybe there is little reason for differing the cardapplets and the slider applets.
- There has been some changes in the goals. A taskbar has been added, ideas about Slicker as a clipboard has been added, etc.
- Many of the old designers/developer have "left the ship". New developers are having problems understanding the old design and implementation fully.
- There are several more reasons for a redesign, but these are a few of the more important ones.
- The aim of this redesign will of course be about giving Slicker a more solid foundation, providing greater flexibility and a better modular approach.
- This said, let's get on to the actual design-proposal.

2 The sourcetree

As stated above, one of the goals of the redesign would be to somewhat clean up the buildtree.

2.1 Redundancy

The existing contains some redundancy in the form of the path to all sources containing names like "slicker/slicker". My solution to this problem is simple. Since there is really no sources in the top src-tree, let's put our modules in there. Suggested top-level modules are as follow.

2.2 Core

There is also a bad distinction between "cardhandler" and "carddeskcore". The core code, loaded at all times, should of course be located in the "core" directory, but this directory will then have to be divided a little to keep things nice and clean.

2.3 Share

A lot of code will be shared between different classes, between plugins and between plugins and the core. This type of code is typically abstract super-classes and should be located in the "share" directory. This directory will of course also be quite large and will have to be divided into more detailed directories.

2.4 Plugins

There are 3 types of applets. Applets, Taskbars and GPP. Actually, this is only a matter of organizing the files. In reality, since plugins are really just a loadable chunk of code, there are no difference between the plugin-types.

```
Slicker
|-- admin // Build-scripts (KDE-standard)
|-- autom4te.cache // Buildcache, comes with the buildtree
|-- plugins // Maybe keep this in the install-dirs?
| |-- applets // Holds all the applets
| |-- taskbar // Holds the realized Taskbar-plugins
| `-- gpp // General Purpose Plugins
|-- core // main.c
| |-- slider // The slider-widget
| |-- taskmanager // Manage running tasks.
| |-- trayiconmanager // Dispatches trayIcon requests from X11
| |-- mimedispatcher // The MimeDispatcher class.
| |-- pluginmanager // The PluginManager
| `-- cardbase // Cardstack, and card-layouting.
`-- share
|-- edgewidget // Holds EdgeWidget
|-- plugins // Hold the common plugin-base
|-- applets // Shared appletbase.
|-- mimehandler // The shared mimehandler interface.
`-- taskbar // Holds the shared code of the taskbar-plugins.

For later addition:
core/launcher // Code for the launcher-widget.
```

3 Plugins

All plugins[1], nomatter type, has the rules that it must contain a subclassed implementation of Plugin accepting a PluginInfo as a parameter to the constructor, as well as a line saying: EXPORT_PLUGIN(<ClassName>);

So to define a simple plugin, what is required is really:

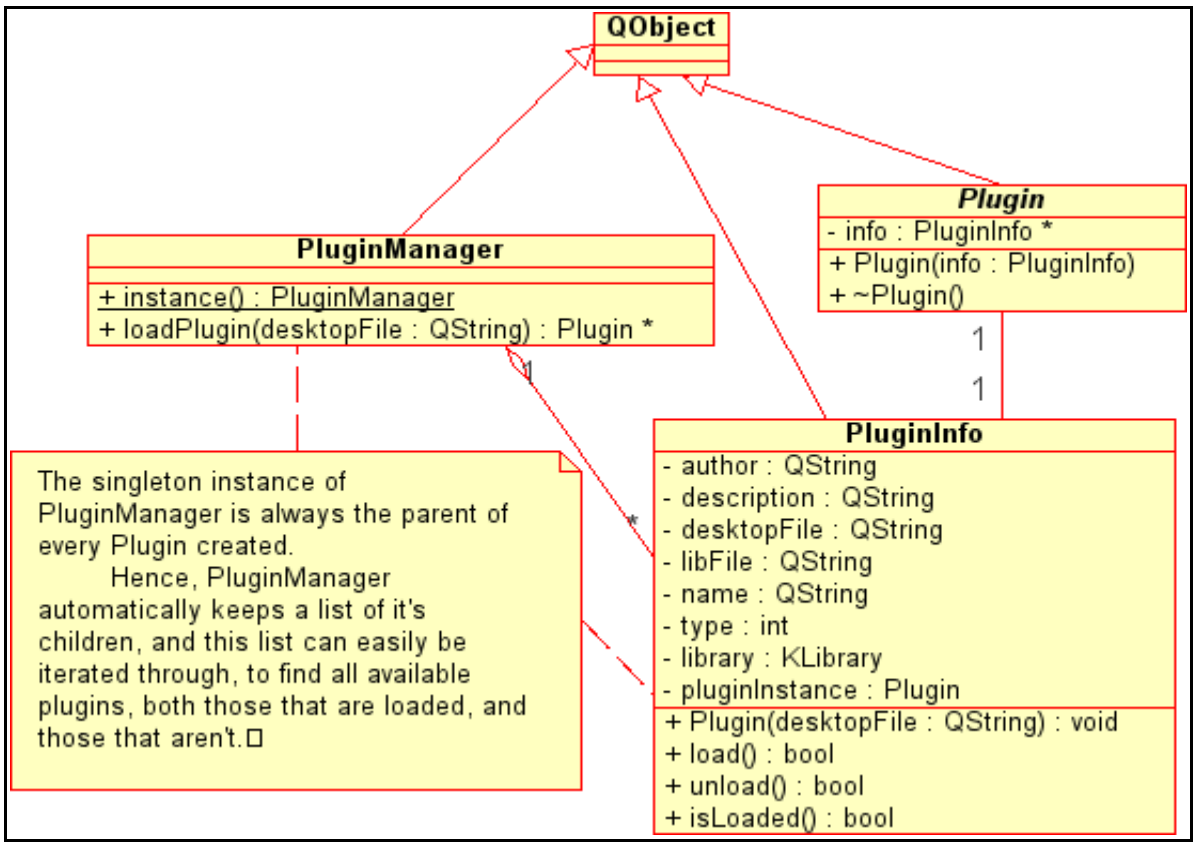
```
#include <plugin.h>
#include <kdebug.h>

class TestPlugin : Plugin {
  TestPlugin(PluginInfo *i) : Plugin(i) {
    kdDebug() << "Plugin loaded. << endl;
  }

  TestPlugin() {
    kdDebug() << "Plugin destroyed. << endl;
  }
}

EXPORT_PLUGIN(TestPlugin)
```

The inner workins of the plugin- architecture is rather easy to describe. Consider the following class- diagram for a while:



To clarify a little more, EXPORT_PLUGIN(ClassName) will be a macro, mapping to something like:

1 A plugin is a binary component, able to be loaded/unloaded at runtime.

```
Plugin *p = NULL;
extern "C"
{
  Plugin * init(PluginInfo *i)
  {
    if (!p)
      p = new ClassName(i);
      return p;
  }
  void destroy()
  {
    delete p;
  }
}
```

So, the loading- process is basically:

- PluginInfo gets created with a desktopFile parameter.
- PluginInfo reads it's info from the desktopFile
- PluginInfo loads the lib specified in the desktopFile.
- The init-function of the library gets called.
- The plugin gets instantiated by init. The constructor gets called.
- Init returns the instantiated Plugin.

And the unloading is something like:

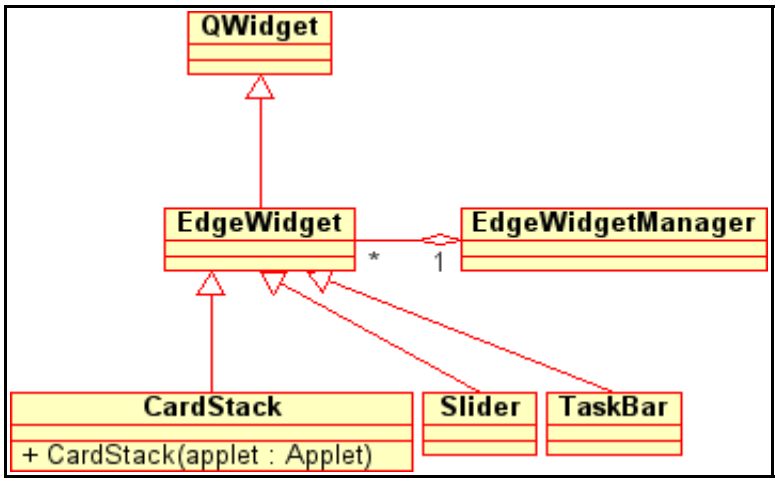
- PluginInfo calls the destroy of the plugin.
- destroy destroys the instantiated Plugin.
- PluginInfo unloads the library.

4 EdgeWidget

Slicker will have several types of widgets[2] that needs to stay connected to a ScreenEdge. To mention a few, there is Taskbar, Slider and CardStack. These will all share some common functionality:

- They will all need to be connected to a screenedge at all times.
- They will all probably need to be relayouted as soon as the edge changes.
- They will all need the functionality to hide.
- They will all need to check so they don't collide with each other.

All this functionality would be rather unpractical to have redundant in each using class. We must somehow find a way to share this functionality between the classes that needs it. The EdgeWidget is one solution.



2 Graphical components, as denoted in the Qt API.

5 Applets

Applets[3] are usually contained within plugin-libs, but not necessarily. Applets may come with a MimeHandler (see below), but are not required to. One applet can be located at several locations (but only one location at once). They could be held inside a card, or they could be held within the slider, or the launcher. Applets has 3 important aggregations:

- Name - defined by QObject
- Icon - a QWidget containing a small icon of the applet. The applet has to render this icon itself. This increases complexity somewhat, since the applet programmer also has to specify how to draw the Icon, but it also increases the flexibility a lot, since the Icon can be animated, SVG-based, calculated on drawal, or whatever.
- Contents - This is the main contents of the applet. For instance, this is what is rendered in the large area of the card, if the applet is used in a card.

In theory, there are no difference between applets used in the slider, in the cards and in the launcher. In practice however, an applet may not be suitable for rendering as a card. In this case, just nulling the content-part will make the CardStacks reject this applet.

For plugins to announce a new type of applet to the AppletManager, it will have to define a new AppletDef. An AppletDef simply contains some informational parameters on a certain class of applets. It also keeps a reference to all the instantiated applets of it's class. AppletDef follows the singleton pattern in the sence that normally there are only one object of each final type in the system.

Basically, Applets, AppletDefs and the AppletManager are all QObjects, meaning they are part of a QObject-tree. The parent for every AppletDef is the AppletManager (singleton-class). The parent for every Applet is it's AppletDef.

The AppletManager also keeps track of all available AppletHosts[4]. But this is not dealt with in a QObject-manner.

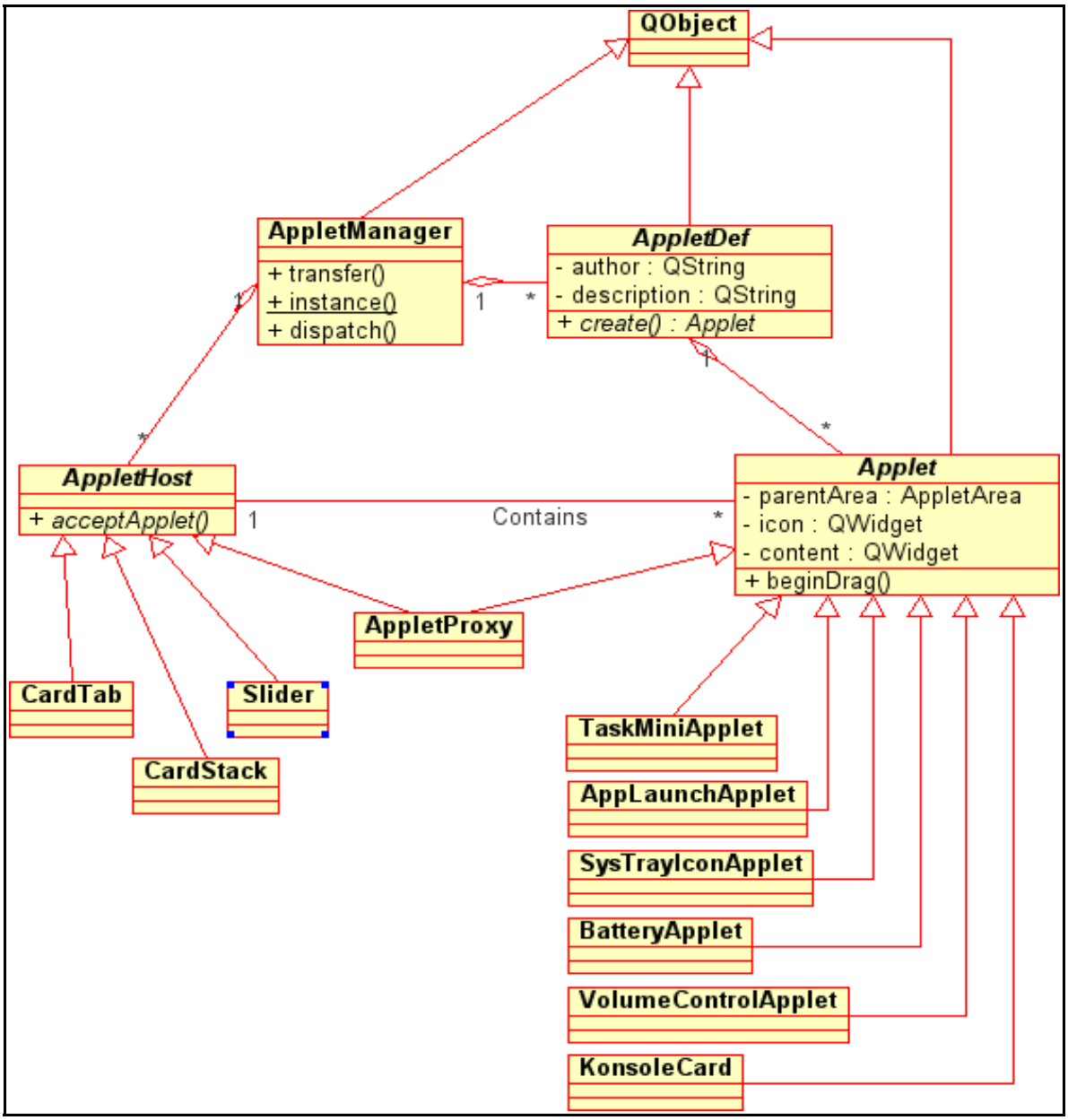
Example AppletHosts are the CardTab, the Slider, and the AppletProxy[5].

---

3 An applet is a "graphical" component with the ability to relocate to a area of the desktop.

4 AppletHost are the abstract class that defines required methods to accept and contain applets.

5 An AppletProxy is an applet that may contain other applets.


QObject
AppletManager
+ transfer()
+ instance()
+ dispatch()
AppletDef
- author : QString
- description : QString
+ create() : Applet
AppletHost
+ acceptApplet()
Contains
Applet
- parentArea : AppletArea
- icon : QWidget
- content : QWidget
+ beginDrag()
AppletProxy
CardTab
Slider
CardStack
TaskMiniApplet
AppLaunchApplet
SysTrayIconApplet
BatteryApplet
VolumeControlApplet
KonsoleCard

6 Cards

Cards is a big and imortant part of Slicker. Cards are the one feature that really distinguishes Slicker from the rest of the crowd. Therefore, it's essential for us to have a working Cards-system. The problem with the cards are, as I see it, that it's a problem with three important topics/issues. Let's break it down. (Divide and Konqueror, isn't it what they say?)

- How Cards relate to Applets
- How CardStacks relate to the Cards
- How CardStacks relate to EdgeWidget.

6.1 Cards/Applets

A card must always be created with an applet, and must always contain this applet. When the applet is removed, the Card should be destroyed.

A card consists of two major graphical components.

- A CardTab
- Some content.

The content is pretty simple. It basically maps to the content of the Applet contained in the card. The CardTab is slightly more complex, but not much. It consists of three major parts.

- An icon
- A name
- An AppletHost

The icon maps directly to the Icon-widget of the applet and the name is of course the name of the applet. The AppletHost however, is a graphical area that should be able to contain applets that accepts to render it's Icon to very small regions. (Typically 16x16.)

6.2 CardStacks/Cards

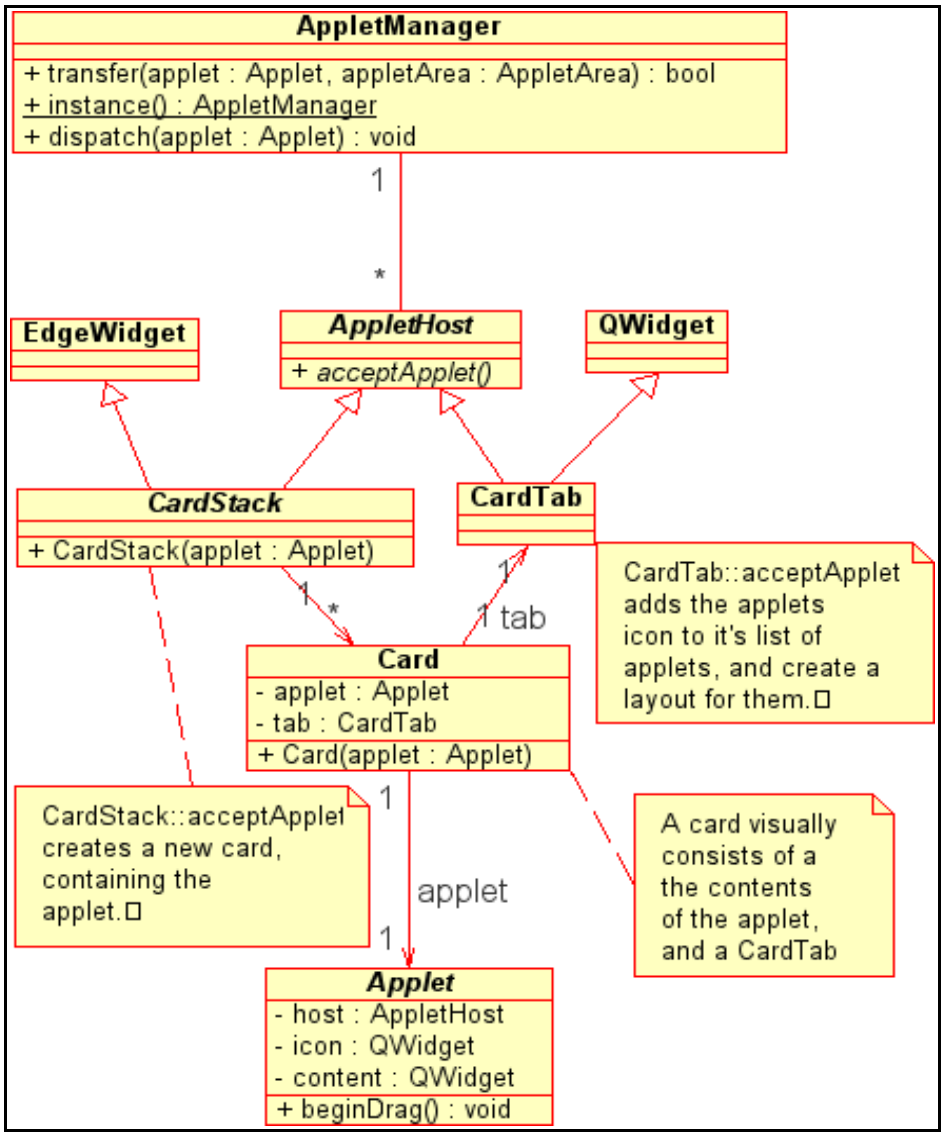
A CardStack is essentially a collection of cards. The CardStack is responsible for laying out it's cards, render a default background according to current colorscheme, and setup a mask to create the rounded corners.

There is no such thing as an empty CardStack. CardStacks are always created with an applet, to populate the first card. When the last card of a CardStack is removed, the stack should be deleted.

There is one important thing to notice here. CardStack is an abstract class, meaning it has to be subclassed and cannot itself be instantiated. A Card is not a QWidget, meaning it doesn't render anything to anything. It simply groups Tabs and Contents together and let's the subclassed CardStack layout the cards. The reason for doing things this way is to allow more flexibility in how CardStacks work, in order to address the issue about different types of CardStacks.

6.3 CardStacks/EdgeWidget

This is so simple I hardly know why I'd like to put it in a separate section. A CardStack simply extends the functionality of EdgeWidget. That's it.


AppletManager
+ transfer(applet : Applet, appletArea : AppletArea) : bool
+ instance() : AppletManager
+ dispatch(applet : Applet) : void
1
*
EdgeWidget
AppletHost
+ acceptApplet()
QWidget
CardStack
+ CardStack(applet : Applet)
CardTab
1
1 *
1
1 tab
CardTab::acceptApplet adds the applets icon to it's list of applets, and create a layout for them.
Card
- applet : Applet
- tab : CardTab
+ Card(applet : Applet)
CardStack::acceptApplet creates a new card, containing the applet.
1
applet
1
A card visually consists of a the contents of the applet, and a CardTab
Applet
- host : AppletHost
- icon : QWidget
- content : QWidget
+ beginDrag() : void

7 MimeHandling

From the beginning, it were stated that Cards should be able to move from one cardstack to another. Later on, there were a request for allowing the SysTray icons move between the contents of the SysTrayCard to the Tab of the SysTrayCard.

When redesigning started, it were suggested by Witcomb that applets should not be able to be used both in the Slider, and in a card or basically anywhere else. This were designed as described in section Applets.

This led to the thought that if applets should be able to be placed into cards, and cards should be able to move between stacks, why not also allowing cards to be moved to the Slider, taking the form of Slider-applets? And if SysTray icons should be able to move between CardContents and tabs, why not also make SysTrayIcons an applet?

To support this requirement, we could either do some complex dragging ourselves, or we could support the so popular DND-Scheme[6]. DND in Qt follows the XDND-specification, forcing dragged content to be described using some Mimetype[7]. This essentially means we can't drag any type of information without specifying it's type.

This is not a problem, we could simply assign dragged content the type "slicker/applet" and be happy with that. But let's take it one step further: Let's think about what we should do if anything of type "whatever/whatever" gets dropped on us? I see two possibilites here.

- Slicker simply denies any content dropped, not beeing "slicker/applet".
- Slicker holds a pluggable list of MimeHandlers, that sees if there is something sensible to do to take care of the situation.

Guess which option I like best? ;-)

Think about one example of the second option:

1. Slicker detects the user has dropped something of type "text/plain" on a CardTab.
2. Slicker passes the query on to our MimeDispatcher, which then iterates through a list of active MimeHandlers.
3. It comes to the handler loaded by "TextAppletPlugin". This handler says "Hey, I can do this." and returns a list of proposed Kactions.
4. MimeDispatcher walks through the rest of the MimeHandlers, remembering each proposed action.
5. When all Mimehandlers have been traversed, MimeDispatcher looks at the list.
6. If there are multiple actions suggested, it presents these actions to the user and let's him/her choose. If there is only one or zero actions in the list, there is no need to query the user.

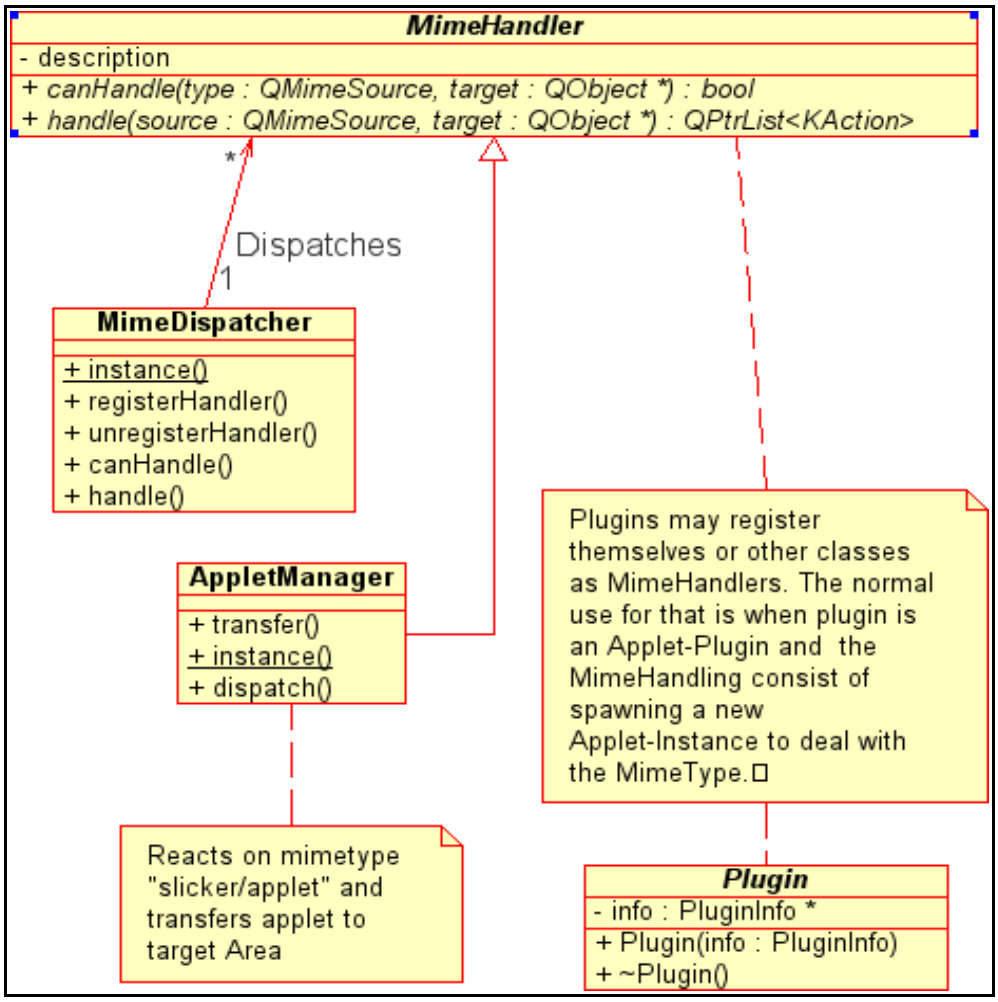
This way we can make sure we maintain a high level of flexibility when dealing with all kinds of DND and Mimehandling.

This approach also has some other strengths. This type of MimeHandling would probably also work with little or no alteration when dealing with cut-and-paste, and could prove useful for other purposes as well.

---

6 DND is short for Drag N' Drop

7 Mimetypes are essentially descriptors of data in the form <class>/<subclass>. Examples on types are "image/gif" or "text/html".


MimeHandler
- description
+ canHandle(type : QMimeSource, target : QObject *) : bool
+ handle(source : QMimeSource, target : QObject *) : QPtrList<KAction>
*
Dispatches
1
MimeDispatcher
+ instance()
+ registerHandler()
+ unregisterHandler()
+ canHandle()
+ handle()
AppletManager
+ transfer()
+ instance()
+ dispatch()
Plugins may register themselves or other classes as MimeHandlers. The normal use for that is when plugin is an Applet-Plugin and the MimeHandling consist of spawning a new Applet-Instance to deal with the MimeType.
Reacts on mimetype "slicker/applet" and transfers applet to target Area
Plugin
- info : PluginInfo *
+ Plugin(info : PluginInfo)
+ ~Plugin()