GAL Framework

Radek Bartoň, Martin Hrubý
Faculty of Information Technology
Brno University of Technology
E-mail: `xbarto33@stud.fit.vutbr.cz`, `hrubym@fit.vutbr.cz`

Keywords: design, GIS, GRASS, open source, library

Abstract

GAL (GIS or GRASS? Abstraction Layer) Framework is meant to be multiplatform Open-Source library with certain tools and subsidiary daemons for easy implementation of distributed modules for GIS GRASS[1] in static or dynamic programming languages. This article aims to present some ideas behind this library and bait a fresh meat for this project since its complexity needs more spread development team not just single person. Project homepage can be found at http://gal-framework.no-ip.org.

Motivation

A year ago I was trying to implement some feature to nviz module and I got acquainted with GRASS's core libraries. Accustomed to well-designed and well-documented APIs like Qt[2], Coin[3] or wxWidgets[4] it was shocking to me to meet such old-styled API with underestimated state of documentation, so I gave up that job in few days in anger. But because I like GRASS style from user's point of view and its complexity and scalability of provided modules, I wanted

[1] `http://grass.itc.it/`
[2] `http://doc.trolltech.com/`
[3] `http://doc.coin3d.org/Coin/`
[4] `http://www.wxwidgets.org/manuals/stable/wx_contents.html`

to provide an easy and modern approach to module development along with current ones for those that are not satisfied with present possibilities as me. So I decided to work on GAL Framework as my diploma work and thesis. As actual advancement of common information technology in hardware domain heads to multiprocessored devices and usage expansion of dynamic languages such as Python, Java, C# or Ruby is here, there is need to consider these factors in GRASS development too. GAL Framework should be answer to these trends.

Principles

GAL Framework should be alternative to current way of creating new modules for GRASS and some current modules can be reimplemented in GAL Framework too if some weight reasons occurs. Illustration of future GAL Framework role in GRASS module development can be seen on Figure 1. Current GRASS code is displayed yellow and it is formed by GRASSlib and code of each of GRASS modules. Green GAL Framework code is formed by core GAL library and code of new GRASS modules which can be implemented with it or code of reimplemented modules (v.proj for example). GAL Framework then may depend on GRASSlib itself and other possible libraries such as D-Bus[5] for distributive execution of functions or TerraLib[6] for faster implementation of GIS related functionality (red block).

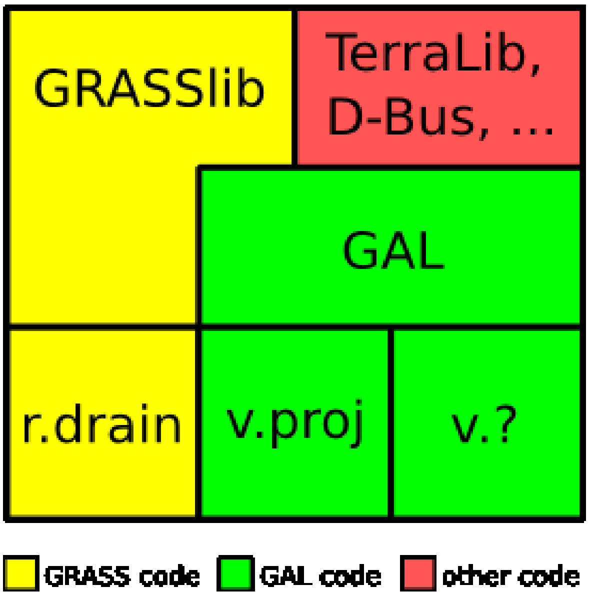

Figure 1: Role of GAL Framework in module development.

GAL Framework internally uses component architecture which means that whole system is formed by bunch of components which communicate which each other by interfaces. Some components declare interfaces, some of them implement interfaces and some of them use interfaces. Using UML syntax this can be displayed as on Figure 2.

There are four components and one interface on a figure. *Component 1* declares an interface *Interface* which indicates stereotype <<*reside*>>. *Component 1* and *Component 2* uses in-

[5] `http://www.freedesktop.org/wiki/Software/dbus`
[6] `http://www.dpi.inpe.br/terralib/`

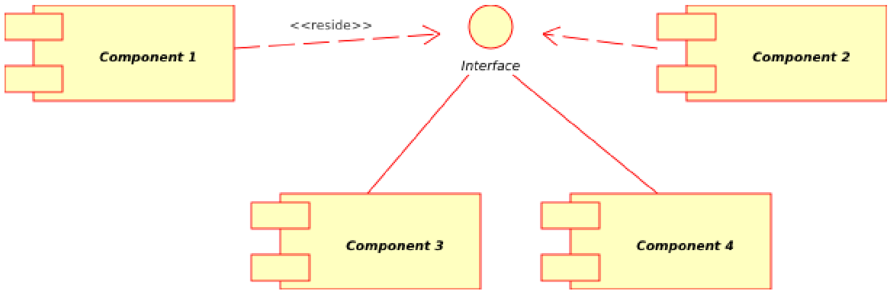

Figure 2: Component architecture in UML syntax.

terface (symbolized by dashed line with arrow). That means that they are calling interface's functions. *Component 3* and *Component 4* implement interface functions which is symbolized by solid line. There is an abstraction over interface functions called slots which may be configured some ways to specify which implementation will be executed when interface slot is called. Sometimes you may want to execute just lastly registered implementation, sometimes you may want to call each of them, etc.

To allow public access to all available components and interfaces in the system, a common access point must be introduced. In this case, it is called component manager and it serves for new component or interface registration as long as registration of interface implementations. Figure 3 shows simplified class diagram for relationships between component manager, components, interfaces and slots:

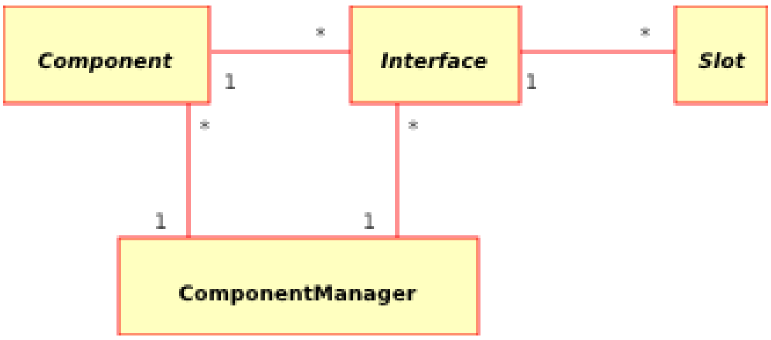

Figure 3: Class diagram of component architecture.

Let's see on Figure 4 how can be this component architecture applied in practice:

There is a `ModuleComponent` component implementing a GRASS module at the top which uses three different (and imaginary for now) interfaces:

- `IVectorLayerProvider`,
- `IRasterLayerProvider`,
- `IMessageHandler` for retrieving of vector or raster data and for message display and logging.

`IVectorLayerProvider` interface is implemented by two different components registered in system. One of them shelters vector layer data in PostgreSQL database, the second shelters for example data in MySQL database. Raster related interface `IRasterLayerInterface` is

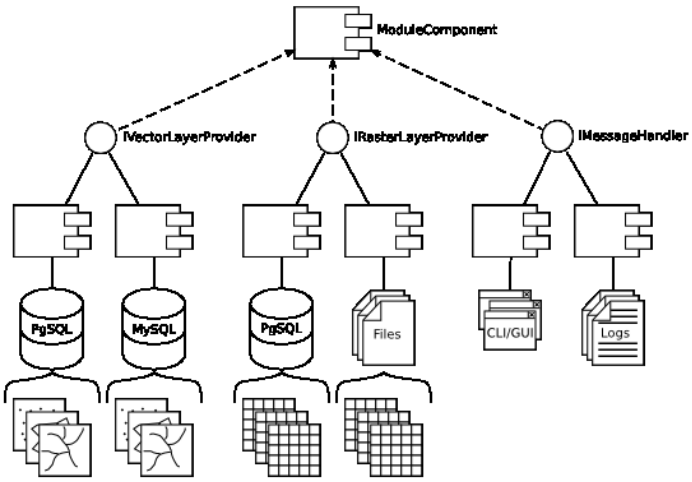

Figure 4: Component architecture practically.

then implemented by two components too. First offers raster data from PostgreSQL and second enwraps raster data in ordinary data files. This diagram should demonstrate that using this approach module component can obtain GIS data and it don't need to care where and how are these data stored.

When module does what it wants with retrieved data it needs to output some informations to console, logs or GUI, it sends messages through `IMessageHandler` interface. In discussed example is this interface implemented by two components which forwards messages to CLI, GUI or writes them to log files. This again demonstrates presentation of outputed data independence.

Usage and Examples

Although bindings to most of commonly used dynamic languages are planned, at least for Python and Java, all of following code examples are written in C++ because it will be main language used during GAL Framework development. Get known that all of this example usages reflect current state of design and may change in future.

The simplest interface usage example which doesn't create component and can be used only as a client module is listed here:

```
    #include <iostream>

    #include <core/GAL.h>
    #include <core/ComponentManager.h>
    #include <core/Interface.h>
    #include <exceptions/Basic.h>

    int main(int argc, char * argv[])
    {
      try
      {
01:     GAL::initialize();

        // Obtain slots of IExampleInterface functions.
```

```
02:     ComponentManager & component_manager = GAL::getComponentManager();
03:     Interface & interface = component_manager.getInterface("IExampleInterface");
04:     Function1Slot & function1 = dynamic_cast<Function1Slot &>(interface.getSlot("function1"));
05:     Function2Slot & function2 = dynamic_cast<Function2Slot &>(interface.getSlot("function2"));

        // Call functions' implementation.
06:     std::cout << "Function 1 returned: " << function1() << std::endl;
07:     std::cout << "Function 1 returned: " << function2("Example", 10) << std::endl;

08:     GAL::finalize();
      }
      catch (Exception exception)
      {
        std::cout << exception.getMessage() << std::endl;
        return EXIT_FAILURE;
      }
      return EXIT_SUCCESS;
    }
```

Code is enclosed between initialization and deinitialization functions: `GAL::initialize()` and `GAL::finalize()`. First we obtain reference to publicly available component manager at line 02. Then we get interface object at line 03. At line 04 and 05, we get slot objects which can be remembered during whole program run time for interface's functions execution. This can minimalize needed overhead. Lines 06 and 07 do this execution. In this example we used only imaginary interface `IExampleInterface` but in real case it can be for example interface which provides access to raster layer and we will compute some statistical values over returned data. Note, that from this point of view it doesn't matter if interface's slots are implemented in statically linked C++ code, loaded and executed from plug-in in shared library or they are executed distributively in Python code or on distant machine via RPC call. All of this is only up to slot implementations and its configuration.

The second example shows how can be certain interfaces implemented by components:

```
    #include <iostream>
    #include <string>

    #include <core/GAL.h>
    #include <core/ComponentManager.h>
    #include <core/Component.h>
    #include <core/Interface.h>
    #include <exceptions/Basic.h>

    // ExampleComponent class definition.
01: class ExampleComponent: public Component
    {
      public:
02:     virtual void initialize();
03:     virtual void finalize();
04:     virtual void * getFunction(const char * name);
05:     static const char * function1();
06:     static const char * function2(const char * first, cont int second);

      private:
07:     Interface * interface;
    };

    // Component initialization. Gets instance of interface and registers it
    // in component manager.
08: void ExampleComponent::initialize()
    {
09:   ComponentManager & component_manager = GAL::getComponentManager();
10:   this->interface = &(component_manager.getInterface("IExampleInterface"));
```

```
  component_manager.registerImplementation(*(this->interface), *this);
}

// Component deinitialization. Unregisters interface.
void ExampleComponent::finalize()
{
  ComponentManager & component_manager = GAL::getComponentManager();
  component_manager.unregisterImplementation(*(this->interface), *this);
  this->interface = NULL;
}

// Implementation of first interface function.
const char * ExampleComponent::function1()
{
  std::cout << "IExampleInterface::function1" << std::endl;
  return "Example";
}

// Implementation of second interface function.
const char * ExampleComponent::function2(const char * first, const int second)
{
  std::cout << "IExampleInterface::function2" << std::endl;
  return "Example";
}

// Overriden virtual function which returns function pointer to requested
// implementation of interface function.
void * ExampleComponent::getFunction(const char * _name)
{
  std::string name = std::string(_name);
  if (name == "function1")
  {
    return (void *) &(this->function1);
  }
  else if (name == "function2")
  {
    return (void *) &(this->function1);
  }
  else
  {
    return NULL;
  }
}

int main(int argc, char * argv[])
{
  try
  {
    GAL::initialize();

    // Create component instance.
    Component * component = new ExampleComponent();
    component->initialize();

    GAL::daemonize();

    // Destroy component instance.
    component->finalize();
    delete component;

    GAL::finalize();
  }
  catch (Exception exception)
  {
    std::cout << exception.getMessage() << std::endl;
    return EXIT_FAILURE;
```

```
  }

  return EXIT_SUCCESS;
}
```

There is defined `ExampleComponent` first at lines 01-07 in this example. Each component needs to override three virtual methods: `Component::initialize()` at lines 08-11 for component initialization and registration of interface implementation, `Component::finalize()` at lines 12-15 for component deinitialization and interface implementation unregistration and `Component::getFunction()` at lines 22-29 for access to interface functions implementations. Main program block (lines 30-36) then contains code which creates instance of component and then turns program to daemon with `GAL::daemonize()` method. Interface implementation can be then called by external modules. In case where implemented interface is used only internally as in this example, no daemonization is needed and program would just use interface as in first example.

The third example is just code snippet showing how to define a new interface with its slots:

```
// Definition of CallbackSlot derived slot.
class Function1Slot: public CallbackSlot
{
  public:
    Function1Slot();
    char * execute();
    char * operator()();
};

// Definition of DBusSlot derived slot.
class Function2Slot: public DBusSlot
{
  public:
    Function2Slot();
    char * execute(char * first, int second);
    char * operator()(char * first, int second);
};

// Slot signature initialization.
Function1Slot::Function1Slot():
  CallbackSlot()
{
  this->addReturnValue(STRING);
}

// Slot signature initialization.
Function2Slot::Function2Slot():
  DBusSlot()
{
  this->addArgument(STRING);
  this->addArgument(INT32);
  this->addReturnValue(STRING);
}

// Arguments and return values packing and unpacking for D-Bus call.
char * Function2Slot::execute(char * first, int second)
{
  char * result = NULL;

  this->setArgument(0, (void *) first);
  this->setArgument(1, (void *) second);
  this->setReturnValue(0, (void *) &result);
```

```
22:   this->callFunction();

23:   return result;
    }

    // Syntax sugar for use of slot as a functor.
24: char * Function1Slot::operator()()
    {
25:   return this->execute();
    }

    // Syntax sugar for use of slot as a functor.
26: char * Function2Slot::operator()(char * first, int second)
    {
27:   return this->execute(first, second);
    }

    // Definitions of example interface with two function slots.
28: class IExampleInterface: public Interface
    {
      public:
29:     IExampleInterface();
30:     ~IExampleInterface();

      private:
31:     Slot * function1Slot;
32:     Slot * function2Slot;
    };

    // Interface initialization. Instances of two slots are created and added
    // to interface.
33: IExampleInterface::IExampleInterface():
34:   Interface(),
35:   function1Slot(NULL), function2Slot(NULL)
    {
36:   this->setId("IExampleInterface");

37:   this->function1Slot = new Function1Slot();
38:   this->function2Slot = new Function2Slot();

39:   this->addSlot("function1", *(this->function1Slot));
40:   this->addSlot("function1", *(this->function2Slot));
    }

    // Interface deinitialization. Instances of two slots are removed and
    // destroyed.
41: IExampleInterface::~IExampleInterface()
    {
42:   this->removeSlot("function1");
43:   this->removeSlot("function2");

44:   delete this->function1Slot;
45:   delete this->function2Slot;
    }
```

First of all, two slots are defined at lines 01-04 and 05-08. One of them is derived from `CallbackSlot` which is simple static implementation of slot mechanism and second is derived from `DBusSlot` which implements slot mechanism via D-Bus calls. Slot constructors at lines 09-11 and 12-16 specify their signature by addition of input and output arguments. For `DBusSlot` derived slots is needed to define way of argument packing and unpacking which does `Function2Slot::execute()` method at lines 17-23. `IExampleInterface` is then defined at lines 28-32 and slots are created or destroyed in its constructor at lines 33-40 and destructor at lines 41-45. Finally, to be able to use this interface in whole system, its instance has to be registered in component manager using `ComponentManager::registerInterface()` method.

Current State

At the time of this article creation, GAL Framework is in early stage of design with work on test implementation of designed ideas in progress. Test component management, client-side and partially a server-side of D-Bus slot implementation is finished. Pieces of knowledge gained from test implementation will be fed back to update the current design lately. When all parts of component management and slot mechanisms will be designed and proved by implementation, GAL Framework may proceed to design of appropriate GIS related interfaces which will serve for further implementation of GRASS modules.

Currently GAL Framework uses SCons as a build tool, Subversion as a source management system, Trac as a project management system, wiki and discussion forum, Umbrello for UML modeling, SWIG for bindings generation and libffi and D-Bus as a libraries.

Anyone interested in this approach, who wants to contribute to GAL Framework at least with comments or new ideas, may contact me at `xbarto33@stud.fit.vutbr.cz` or create account at http://gal-framework.no-ip.org to get edit rights for wiki and discussion forum.

References

1. Christopher Lenz, Dave Abrahams and Christian Boos: Trac Component Architecture, web-page[7]
2. Trac – http://trac.edgewall.org/
3. SCons – http://www.scons.org/
4. Subversion – http://subversion.tigris.org/
5. Umbrello – http://uml.sourceforge.net/index.php
6. SWIG – http://www.swig.org/
7. libffi – http://sourceware.org/libffi/
8. D-Bus – http://www.freedesktop.org/wiki/Software/dbus/

[7] `http://trac.edgewall.org/wiki/TracDev/ComponentArchitecture`