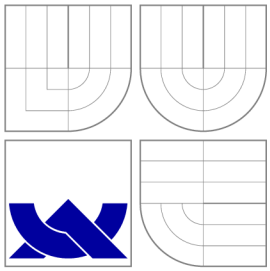
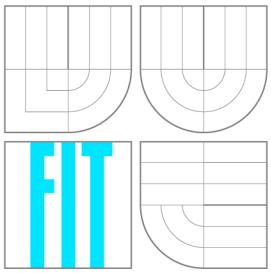


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

MODERNIZACE GIS SYSTÉMU GRASS

GRASS GIS MODERNIZATION

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. RADEK BARTOŇ

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. MARTIN HRUBÝ, Ph.D.

BRNO 2008

Abstrakt

Geografický informační systém GRASS se stal za 26 let své existence standardem na poli modelování geografických jevů. Jeho vnitřní struktura však odpovídá době jeho vzniku. Tato práce chce navrhnout možnou podobu modernizace interních částí zavedením komponentní architektury a objektových návrhových vzorů, jakož i podporu distribuovaných výpočtů a dynamických jazyků, ale z uživatelského pohledu chce ovládání zachovat. Výsledek návrhu je rozveden do prototypové implementace knihovny nazvané GAL Framework.

Klíčová slova

GIS, GRASS, modelování a simulace, komponentní architektura, návrhové vzory, dynamické jazyky, distribuované výpočty

Abstract

The geographical information system GRASS has become a standard on the field of geographical phenomenon modeling during its 26 years old lifetime. However, its internal structure follows practices from the date of its creation. This thesis aims to design a possible shape of internal parts modernization using a component architecture and object-oriented design patterns with distributed computing and dynamic languages support in mind. The designed system should stay identical from the user's point-of-view. Design results are proven on a prototype library implementation called the GAL Framework.

Keywords

GIS, GRASS, modelling and simulation, component architecture, design patterns, dynamic languages, distributed computing

Citace

Radek Bartoň: Modernizace GIS systému GRASS, diplomová práce, Brno, FIT VUT v Brně, 2008

GRASS GIS Modernization

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Martina Hrubého. Úplný seznam použitých literárních pramenů jsem uvedl v zadní části práce.

.....
Radek Bartoň
May 18, 2008

Poděkování

Poděkování za vznik této diplomové práce patří panu Martinu Hrubému za četné konzultace, přátelský přístup a podporu na GRASS-dev e–mailovém fóru.

Acknowledgment

Thanks for this thesis creation belongs to Mr. Martin Hrubý for countless consultations, friendly attitude and support at GRASS–dev e–mail forum.

Contents

1 Introduction

1.1 Motivation

Geographic information systems (GIS) [1] are becoming more and more significant in many aspects of human life such as industry, engineering, ecology, public administration, sociology or nature sciences. Where people was previously deciding only by their personal judgements, now relies on sophisticated and scientific analyses. Together with the vast expansion of Internet services, Web based geographical information systems are spreading too.

In Open Source domain, the best known and featured software for geographic analyses is the GRASS GIS. Unfortunately, its development is stagnating because of small interest from fresh and young developers. This is partially caused by the fact that its design and concepts are overcomed by modern practices in a software development. This work tries to propose one of many eventual ways of modernization and prepare soil for further feature advancement.

1.2 Contents

The thesis is divided into eight main chapters which concerns in different aspects of solved tasks. Analysis of the current GRASS GIS architecture is placed first. Problem specification is discussed next, followed by explanation of component architecture concepts which is main approach to solve confronted problems. Significant parts of the analytical model are described next but full description of designed classes, interfaces and components is available at [2]. Pinpoint of used design patterns in the design has its own chapter. Explanation of slot execution mechanism implementations follows. The most extensive chapter about each of GAL Framework's subsystems is situated after. Then are notes from implementation of dynamic languages bindings and thesis is concluded with experimental performance results and their analysis.

The work proceeds from the semestral thesis of the same name, uses and extends its results. Theoretical parts of the text (chapters 2, 3, 4, 5 and 7) are fully or partially originated from this source. The master's thesis appends practical results, experiences and a more detailed documentation of a prototype implementation.

2 GRASS GIS Architecture

This chapter summarises GRASS evolvement during years of development and current state of raster and vector subsystems (versions 6.x).

2.1 Brief GRASS GIS History

The Army Corps of Engineers' Construction Engineering Research Laboratory (USA/CERL) in Champaign, Illinois, USA started in the 1980s work on an inexpensive raster based GIS software for UNIX systems lately called the Geographic Resources Analysis Support System (GRASS) as an opposite for the ESRI's ARCINFO software. To better understand acquisition tendencies of the GAL Framework and this paper, it is needed to list brief history of GRASS development with architecture innovations of each GRASS evolution step.

1982 [3] The Fort Hood Information System (FHIS) was developed. It was from today's measures a simple raster processing program running on the PDP-11 mainframe and communicating with a remote terminal via a serial link. It used concepts of categories and subcategories, an area of interest, overlapping 100×100 meters cell rasters and a mask raster layer.

1983 [4] Due to slow communication with the terminal, the FHIS was ported to the SUN-1 microcomputer and called the Installation GIS (IGIS). It used a monochrome monitor for command input, a color monitor for data visualization and it also separated data from a program. These two innovations survided in form of monitors (d.mon) and map locations.

1984 [4] The first GRASS called package of 20 programs was released. From this point the project focused on public comunity development although only in a college area.

1990 [4] After six years of growth, current module organization to letter–dot format, command–line argument parsing and module descriptions was introduced with the GRASS 4.0. First attempts for a user interface fall to this period too. A source code is organized to directories for general, miscellaneous, display, raster, imagery and sites modules and core libraries sources. Vector data support is done by additional set of tools for analog map digitization and conversion to rasters.

1997 [5] The GRASS 4.2.1 was released with new modules, code cleanup and a Tcl/Tk user interface.

1999 [5] First GNU/GPL GRASS 5.0 version with floating point and `NULL` raster data support added.

2002 [6] A new arc-based vector architecture with more likely modern approaches and database support introduced.

2006 [7] Realization of need for dynamic language support with a SWIG interface prototype for Python and Perl (GRASS 6.1.0).

2.2 General Concepts

From the users' point–of–view, the GRASS GIS [8] is a collection of rather independent command–line *modules* performing individual data transformation or analysis tasks. Some functionality is done by coupling certain number of modules using Bourne shell scripting. Lately, much effort to develop the comfortable graphical user interface which executes such modules was done.

Although formerly was a GRASS GIS's codebase mix of C and Fortran language and shell scripts, Fortran was abandoned and replaced with C during years. An experimental Python interface to core libraries was introduced recently. Nevertheless, its code purely follows the functional programming paradigm.

Data storage is organized with the hierarchical directory structure of the operating system. The root directory is referred to as `GISDBASE` and may contain *locations* which represent mapped areas. Each location is formed by *mapsets* that contains map *layers* of common meaning. For example, one mapset could have map layers for hydrological analyses and a second mapset could have layers for forestry models. Only one location and one mapset is active in a single moment. There is a special mapset called `PERMANENT` which must be present in every location. It holds unchanging map data as well as some additional metadata.

The mapset directories may hold subdirectories and files with differently typed data and information. Some of them could be:

cell/ – Integer data of 2D rasters.

fcell/ – Floating data of 2D rasters.

cellhd/ – Information about a projection, dimensions and a resolution.

cats/ – Names of categories assigned to data values.

colr/ – Color tables and rules for data visualization.

cell_misc/ – Information about `NULL` valued data and other metadata.

hist/ – Metadata with history of commands that was used to create stored data.

grid3d/ – Data of 3D rasters.

vector/ – A vector data geometry.

dbf/ – Vector data attributes.

```
struct Cell_head
{
  int format;          /* max number of bytes per cell minus 1          */
  int compressed;      /* 0 = uncompressed, 1 = compressed, -1 pre 3.0 */
  int rows;            /* number of rows in the data 2D                 */
  int rows3;           /* number of rows in the data 3D                 */
  int cols;            /* number of columns in the data 2D              */
  int cols3;           /* number of columns in the data 3D              */
  int depths;          /* number of depths in data                      */
  int proj;            /* Projection (see #defines above)               */
  int zone;            /* Projection zone                               */
  double ew_res;       /* East to West cell size 2D                     */
  double ew_res3;      /* East to West cell size 3D                     */
  double ns_res;       /* North to South cell size 2D                   */
  double ns_res3;      /* North to South cell size 3D                   */
  double tb_res;       /* Top to Bottom cell size                       */
  double north;        /* coordinates of layer                          */
  double south;
  double east;
  double west;
  double top;
  double bottom;
};
```

Code 2.1: Structure describing raster layer.

2.3 Raster Architecture

2D raster layers data can be of three types: integer, float and double. 3D rasters can be only float and double [9]. A layer is accessed as one big grid, row by row and when lower resolution is required it is resampled with nearest neighbour method the same way.

Each raster layer is described with `Cell_head` structure [10] (see code 2.1). It is bounded by a region in a specified cartographic projection. Its dimensions (number of rows and columns) are precomputed from this region, north-south and east-west resolution. Supported compression algorithm is RLE.

The GRASS supports only a limited number of metadata for raster layers. If a special raster layer called `MASK` is created, all raster operations are masked by `NULL` valued cells of this layer.

2.4 Vector Architecture

Vector data are represented by composition of *nodes* and *arcs* [6]. The arcs are *paths* created with multiple line *segments* and the nodes are boundary *vertices* (although interior nodes are supported too). Multiple arcs forms a *line*. This structure is intended for modeling of linear objects like streams or roads. Connected arcs with a *centroid* vertex form an *area* which represents areal objects like forests or lakes. Option to insert interior holes and isles in the areas is present too. Point objects are implemented in an own library and stored in an internal file format.

Geometry, topology and attributes are stored separately. Geometry can be loaded from a native format, shapefiles [11], or the PostgreSQL [12] database with the PostGIS [13] extension. Loading from many other file formats with the OGR [14] library is available as well. Topology is stored in a native format, the PostGIS database or constructed during file loading. The attributes may be placed in the DBF, SQLite, PostgreSQL, MySQL or ODBC database through a common DBMI [15] interface. Geometry of the objects have two or three spatial coordinates and 3D objects like *faces* and *volumes* can be created too but with limited topology.

The attributes are associated to geometry with *category* numbers and *field* numbers also called *layers*. The field number determines a database table and the category number determines a table row where look for attribute values. A special text file describes this association of fields to databases and tables.

3 Task Specification

Follow brief listings of intentions, ideas and problems that occurred after the project analysis and a summary of objectives that was or wasn't solved during given time with an explanation.

3.1 Intended Objectives

The next list of key objectives draws up desired intentions of the GAL Framework project in the time of its formation:

- Design a flexible, platform independent and extensible environment for development of new largely analytical modules for the GRASS GIS based on a component architecture.
- Design and implement an internal and external representation of raster and vector data for use in the GAL Framework.
- Use present GRASS libraries beneath an abstraction interface for transient support of the current raster and vector representation.
- Achieve compatibility with the GRASS GIS at module level so that any possible GUI frontend for modules developed using the GRASSlib could be used with modules developed using the GAL Framework as well.
- Prepare detailed documentation of the entire system design and possible usage before and during the implementation.
- Provide as complete as reasonable bindings to dynamic languages, especially for Python and Java.
- Implement an example implementation of certain data loading and processing modules to show example usage of the GAL Framework as programming environment.
- Prepare set of tutorials showing aspects of possible usage of the GAL Framework.
- Publicize project's aims and intentions on the Web and at meetings.
- Discuss all concepts and ideas with comunity to acquire wishes and needs of majority of the people from the GIS domain.

3.2 Obstacles

After brief consideration of the previous list of intentions these restrains appeared:

- Restrictions emergent from C/C++ as statically typed compiled languages.

- Limitations of the SWIG automatic wrapper generator, especially with variable length arguments functions and callbacks. This appears to be solvable with little effort.
- Lack of current GRASS libraries re-entrance safety since many of internal structures are `static`. There is necessity of locks when executing a GRASS code in parallel threads or processes which will affect performance and may lead to deadlocks.
- Overall scope and time requirements of the project. There is need to invite other developers to make this project reasonable.
- Unpleasant attitude of GRASS developers to the object–oriented programming.

3.3 Solved Objectives

Sumarization of project status when this document was created and discussion of unrealized goals is following:

- The design of the core component management and communication system forms content of this thesis, especially of the chapter 5 and it is also distributed to several wiki pages at the project's homepage [2]. The prototype implementation is stored in a SVN source code management system and browsable through a web interface at the same place.
- The external representation of raster data was shaped with `RasterTile`, `ColorRules` and `ColorTable` classes and it is accessed with an `IRasterLayerProvider` interface (see section 8.7). After further consideration, the internal representation was left in a native GRASS format of the GRASSlib library and used in a `GRASSRasterLayerComponent` component. The vector subsystem was kept to other person's responsibility as were consulted with the mentor of the thesis.
- Implemented example modules accepts the same command–line arguments as the GRASS modules if they provides the same functionality.
- A comprehensive conceptual documentation as well as the GAL Framework library reference is available at the project's homepage [2] or on an attached CD.
- Bindings of the library was developed for Python and Java languages. More about them in the chapter 9.
- GAL Framework tutorials are placed in the appendix A of this document.
- A full–featured Web site for the project management and propagation was established, the project was introduced at the GRASS–dev mailing list and an article for the Geoinformatics FCE CTU 2007 Workshop [16] was published.
- Unfortunately, no positive response was received from the comunity and no other attention was given. Therefore, the proposed design is a product of a single mind (of course, inspired from many sources) and not the product of a diverse group which the system of such extent requires.

4 Component Architecture

To be as flexible and extensible as possible, the GAL Framework uses the *component architecture* of software design similar to [17]. Here will be described what does this term mean in a context of the project and how it influences a library structure and usage at the level of individual GRASS modules development.

4.1 Components and Interfaces

Using UML [18] notation such an architecture can be shown as on the figure 4.1. There are four *components* connected to one *interface* with different relationships. You may consider that the components are groups of objects or classes which form compact sub-systems doing some job. For example a raster component which loads raster data from different files to the memory or an analytical component which performs some computation over loaded data.

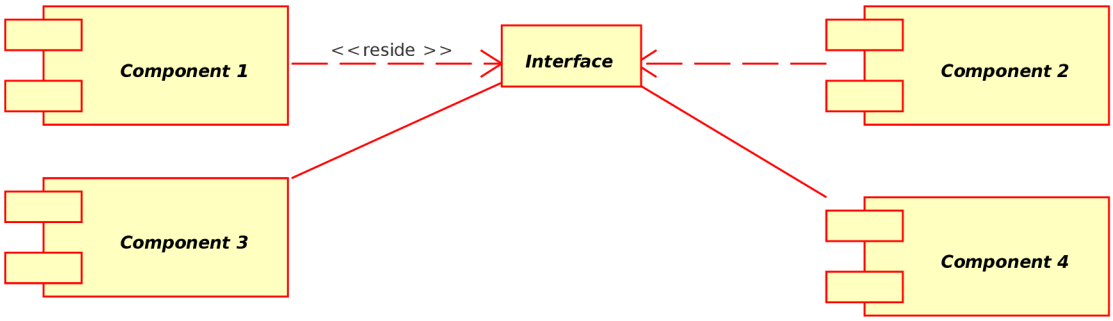


Figure 4.1: The component diagram of the component architecture.

The components may own, use or implement certain interfaces which describes how these components would like to communicate with the others. On the figure, it's the `Component 1` that declares (owns) an interface `Interface` which is indicated by a stereotype ≪`reside`≫. The `Component 1` and the `Component 2` use this interface (symbolized by dashed lines with an arrow). The `Component 3` and the `Component 4` implement interface functions which is represented by solid lines.

Owning an interface means that a component is responsible for its creation, destruction and registration in the system, using an interface means that a component may call one or more of the interface functions. In the most cases, a component which owns an interface is using it at the same time but there is only single component owning single interface. On

the other side of the interface, stay components which implement them. They listen what the components using the interface say and respond to their requests. This implies that the communication is entirely directed by the components that use the interface but the subsidiary components can use other interface which could be implemented by the superior components to reverse this subordination.

4.2 Slots

There is an abstraction over the interface functions called a *slot* that may be configured some way to specify which implementation will be executed. Sometimes you may want to execute just the lastly registered implementation, sometimes you may want to call each of them, etc. The slot is meant to be a functor class that forwards interface function calls everytime it is evoked.

Components using the interface can pick which implementations should be used. A question what the available implementations are chosen when the interface function is evoked could be separated into three cases. It is a little analogy to the TCP/IP unicast, multicast and broadcast:

1 to N - Only a single implementation is chosen among the others. It can be the first idle component or the lastly registered one. An example of such interface may be the component which loads data from certain file format.

M to N - A subset of the registered implementations is chosen. An example of this are data processing components that are used to balance CPUs usage.

N to N - All the implementations are evoked and results (if any) are collected together. An example could be two components that receives error or debug messages. One displays them on the screen, a second logs them to a file but they are both notified.

Different slot implementations abstract a mechanism of interface function execution. There is a static method callback and a D–Bus RPC library slot implementation currently supported. More about the slot implementations is discussed in the chapter 7.

This layout brings flexibility to the framework because any component using a particular interface can choose which implementation of the interface wants to utilize. Furthermore, any component can engage to implement the interface and lately it can abandon its obligation.

4.3 Component Manager

To allow public access to all the available components and interfaces in the system, a common access point must be introduced. In this case, it is called a component manager and it serves for a component or interface registration as long as the registration of the interface implementations. The figure 4.2 shows a simplified class diagram of relationships between the component manager, components, interfaces and slots.

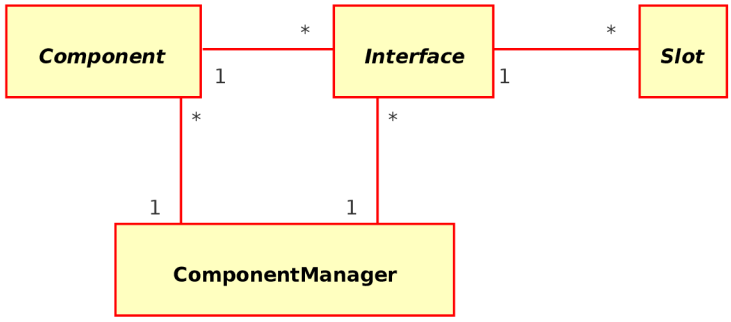


Figure 4.2: The class diagram of the component architecture.

Every component can then ask the component manager to receive a particular interface and use it or commit to implement it. Interfaces, when they are used, contains capabilities to manage all their implementing components so that the component using the interface can decide which of them wants to use in a manner presented before.

This brings a powerful extensibility to the system since the components can be loaded from dynamic libraries like plugins on the system start or even in runtime. Furthermore, they can be spread over computer network or parallel environment and executed by a slot mechanism implemented using some RPC library.

4.4 Example

A conceptual example of the component architecture applicability [16], [19] is shown on the figure 4.3. There is a component `ModuleComponent` implementing some GRASS module at the top which uses three different interfaces: the `IVectrLayerProvider` for retrieving vector data, the `IRasterLayerProvider` for accessing raster data and the `IMessageHandler` for message display and logging.

The `IVectorLayerProvider` interface is implemented by two different components registered in the system. One of them shelters vector layer data in the PostgreSQL database, the other provides for example data from the MySQL database.

The raster related interface `IRasterLayerProvider` is implemented by two components too. The first offers raster data from the PostgreSQL database and the second enwraps raster data in ordinary data files. This diagram should demonstrate that using this approach a module component can obtain GIS data and it don't need to care where and how are these data stored.

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

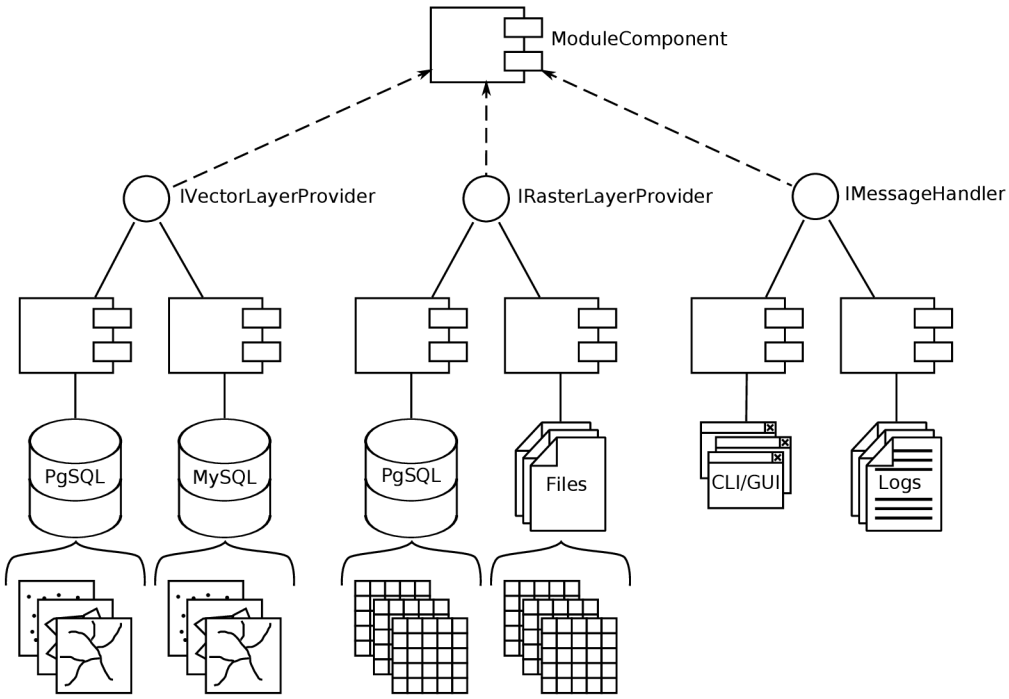


Figure 4.3: The component architecture practically.

5 Analytical Model

To realize deeper consequences between the GAL Framework core components, an analytical model was created. It consists of an use case diagram (fig. 5.1) and an analytical class diagram (fig. 5.2). Detailed description of each use case and each analytical class is available at [20] and [21] and updated continuously. Following lists are constrained to note about the most important of them.

5.1 Use Cases

Intended behavior and usage of the GAL Framework is modeled in the use cases. Relations between the individual use cases are displayed on the use case diagram on figure 5.1. Follows a list of some of them with an explanation. Only role present is a *programmer* who develops new GRASS modules using the framework.

DefineInterfaceUseCase summarizes steps needed to define a new interface object. This is done by the programmer and it includes definition and instantiation of all slot objects of the interface which forms interface functions signatures. The custom slots can be defined entirely or they can be renamed from pre–defined slots.

RegisterInterfaceUseCase registers the interface object instance in the component manager with a given name. This identifier must be unique in the whole system or locally if the interface is not intended to be used with a RPC based slot mechanism.

DefineComponentUseCase describes the course of actions needed to define a new component by the programmer. Aggregated objects and interface objects with their slots are instantiated and initialized first, interface function implementations are prepared as methods of the components then.

RegisterInterfaceImplementationUseCase connects the interface function implementations of the component with an interface object prototype of the component manager. This is done with the component manager method call.

ImplementInterfaceUseCase is performed during the component initialization at a module start. The component registers the aggregated interface objects in the system if it owns them and thus they aren't known to the system yet. Afterwards, the interface function implementations are registered by inclusion of the **RegisterInterfaceImplementationUseCase** use case.

UseInterfaceUseCase retrieves an interface object clone from the component manager by an interface name, gets a slot object of a desired interface function by its name and then executes the slot object as a functor.

ExecuteModuleUseCase defines and executes a module for the GRASS GIS developed in the GAL Framework. This use case contains: definition and initialization of all needed components either a single–purpose for this module execution or a general–purpose from GAL subsystems, request for needed slots to their interfaces which are received from the component manager, intended module computation with slots as functor or dynamically using their methods and finally release of the all obtained resources.

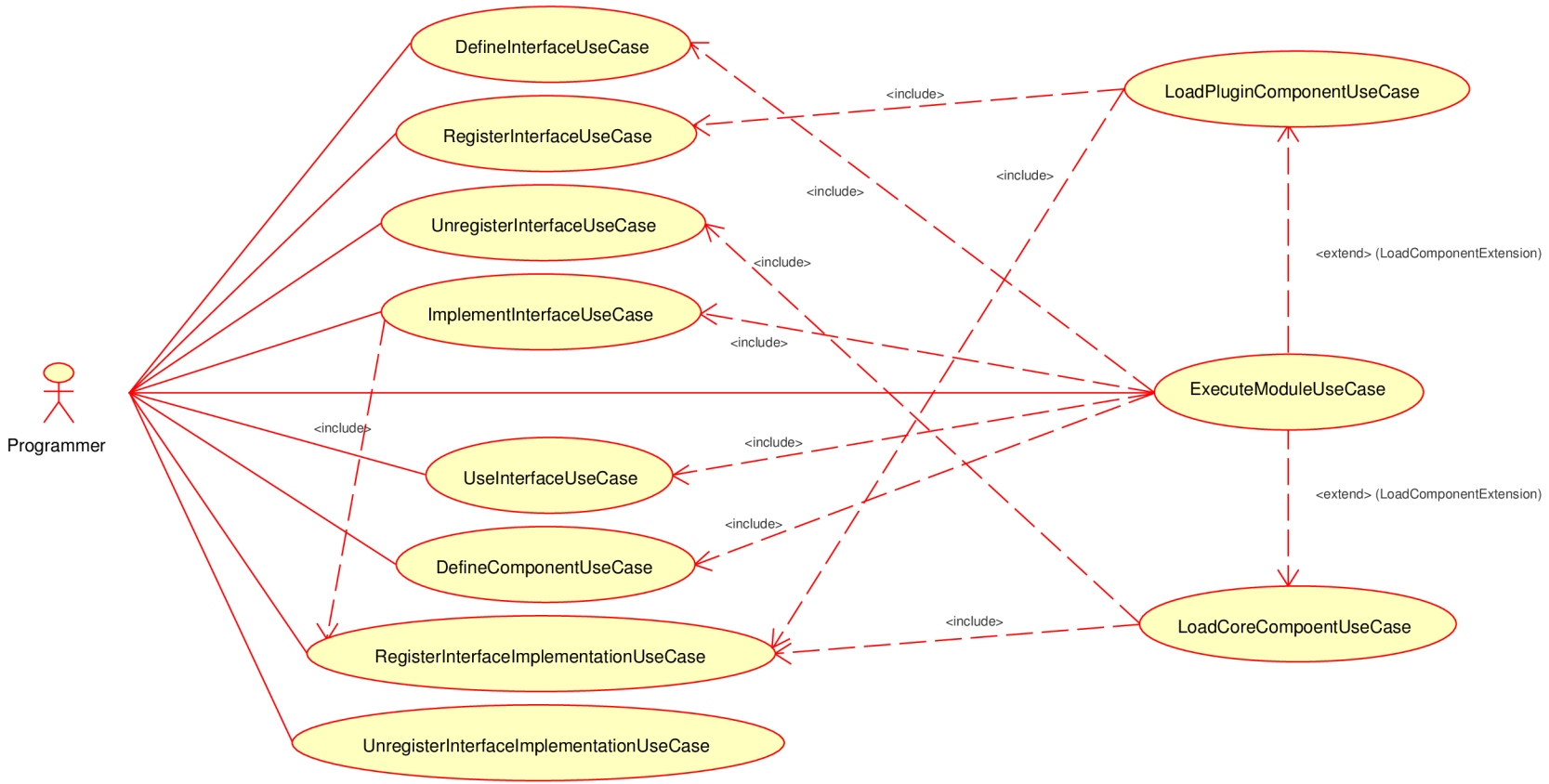


Figure 5.1: The use case diagram of the GAL Framework analytical model.

5.2 Analytical Classes

Analytical classes are the product of realization of the use cases. They can be viewed in the analytical class diagram on the figure 5.2 as well as relationships between them. Here is a list of them with brief description of some of their methods.

GAL is a static class for library initialization, deinitialization and component manager instance retrieval. It also takes care of event loop based subsystems for modules running as a daemon or as a tool with user interface. Some of its significant methods are:

initialize initializes the GAL Framework. It must be called at the start of a module.

finalize deinitializes the framework. It must be called before module end.

demonize turns a program calling this method to a daemon that handles slot implementation execution requests and other events.

quit exits the event processing loop started with the **demonize** method.

getComponentManager provides access to a publicly available component manager.

ComponentManager is a singleton class that serves for components, interfaces and interface implementations registration and retrieval.

getInterface returns a cloned interface object of a registered interface. This local copy can be then configured and used in modules to execute interface functions.

getInterfacePrototype gives access to an internal prototype of a registered interface. It can be used to create a clone for further usage but should not modified.

registerInterface registers a new interface to the system with an interface object prototype.

registerImplementation assigns an interface with a component providing its implementation. Implementing methods can be then called by components using the interface.

registerObject registers a new object prototype in the system that can be then used as a interface function argument or as a return value.

createObject returns a cloned and deserialized instance of an object prototype requested by a class name.

Component is the superclass for all components that can be registered in a component manager. All subclasses have to implement initialization and finalization methods where specify which interface functions may implement.

initialize is a pure virtual method for the initialization of aggregated objects and interfaces and registration of declared or implemented interfaces. Available interface function implementations of the component should be published in this method too using a **setImplementation** method.

finalize is a pure virtual method for deinitialization of aggregated objects and interfaces of the component.

getFunction allows the component manager ask for the interface function implementations of the component. This method is necessary for interconnection of the component and the interface object prototype during interface implementation registration.

setImplementation claims that there is a static method implementation of some interface function in the component. It is used in initialization of the component in a **initialize** method.

ConcreteComponent is the realization of the abstract **Component** class. This class is just exemplary in the model and shows how concrete components could be derived in practice.

Interface is a base class for interface objects. It contains slot object instances representing its functions and it also have methods for their administration as well as methods for a slot implementation registration and selection. Each interface has its own unique identifier.

clone returns a cloned instance of the interface object and allows use this class as a prototype design pattern.

setSlotType changes the way how slots of this interface will be executed, concretely it exchanges used slot executor in all aggregated slot objects.

getSlot returns a slot object representing an interface function identified by its name. This method is often used in modules when certain interface is used.

addSlot is used only in a derived concrete interface objects' constructor to append new functions to the interface.

registerComponent appends a component as an implementing component for the local interface object instance. This function has global impact when applied on an interface object prototype.

IConcreteInterface is again an exemplary realization of the abstract class **Interface** showing what is needed to be done when deriving from its superclass in a custom interface definition.

Slot is a base class that serves as an abstraction over interface functions and their execution mechanisms. Each slot instance has its name which is similar to the interface function name where the slot belongs to.

clone returns a cloned instance of the slot object similarly like the method with the same name of the **Interface** class because interface objects are composed from the slots.

addArgument defines a slot's input signature by appending new arguments of the interface function.

addReturnValue defines a slot's output signature with appended return values.

execute executes a slot implementation with the previously setted arguments using a configured slot executor and fills specified return value variables with result of the called interface function.

call is a pure virtual call operator which may be implemented in descendant slots to support a direct execution of the interface functions.

ConcreteSlot1, ConcreteSlot2 are custom slots extending the basic slot object with a call operator implementation. This allows static execution of interface functions with a defined signature in statically typed languages with a fixed number of function arguments.

SlotExecutor is a simple abstraction class that allows configure slots to use different execution mechanisms in runtime. New mechanisms are added to the system by inheriting from this class and overriding a **execute** method.

getType returns a type of slots which execution the implemented slot executor supports.

execute accepts a slot object that interface function the slot executor will call with its arguments and its return values.

EventHandler helps abstract different libraries and subsystems with event processing loops and merge them into the single loop. Derived classes must implement both variants of event processing methods.

waitEvent is a blocking variant of the event processing method. It serves at least one pending event and returns the number of actually processed events. If no event is available, it waits until some is.

processEvent is a non-blocking variant of the previous method. It does nothing and returns zero, if no events are prepared in a queue for this event handler.

Object is a base class for objects that can be used as interface function arguments and that can be transferred between processes or over network using a serialization.

getClassName gives the name of the class that acts for runtime object type identification of derived classes.

clone returns a copy of the object instace. The objects are also prototypes because they must be created dynamically.

serialize method returns a string representation of a object state either in a text or a binary form.

deserialize is an inverse operation for the previous. It accepts the string representation from which it builds a new object state.

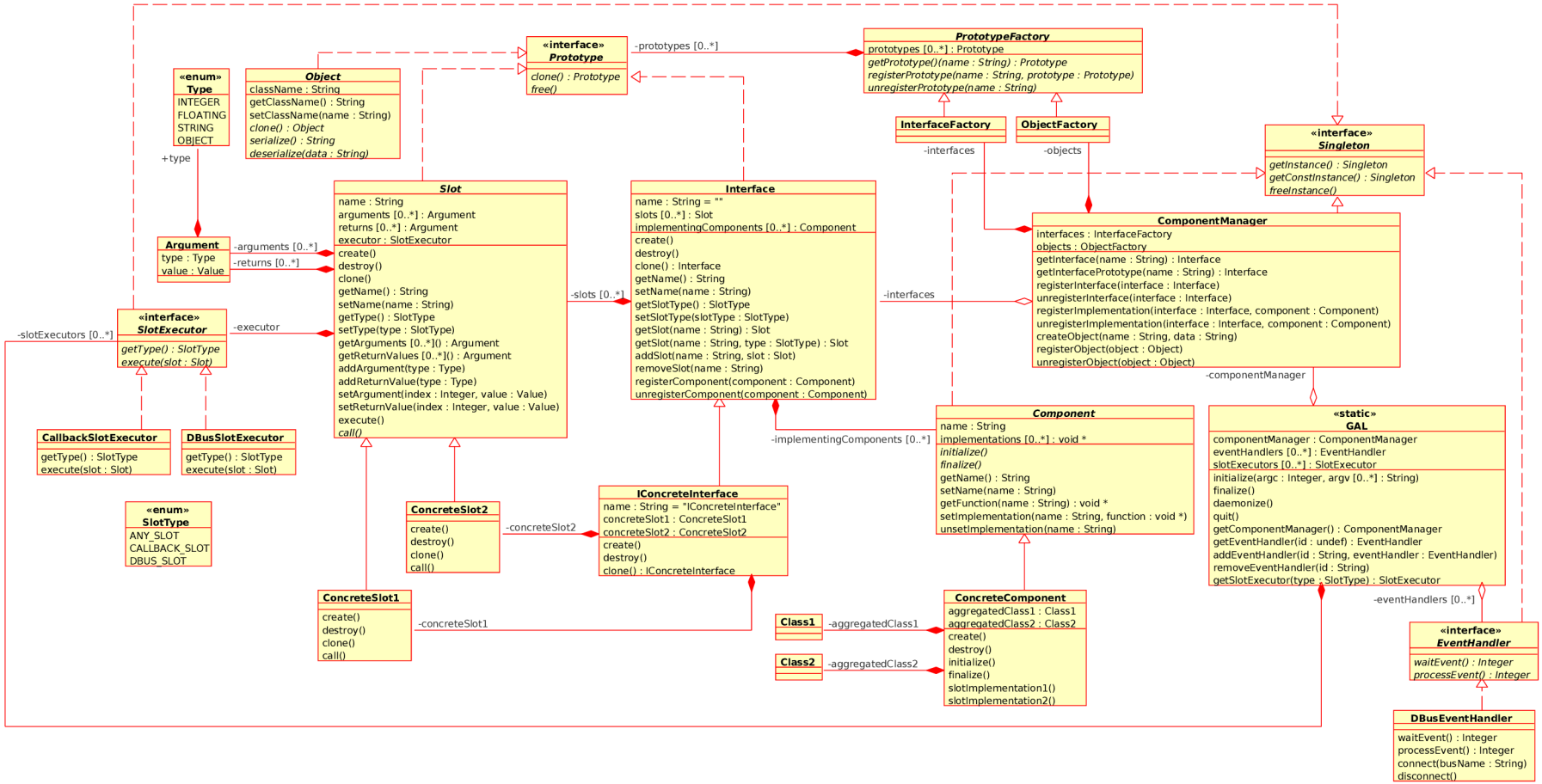


Figure 5.2: The class diagram of the GAL Framework analytical model. Note that the **create()** and **destroy()** methods denotes a regular constructor and a destructor and the **call()** is a call operator.

6 Applied Design Patterns

Following sections of this chapter introspect deeper consequences in the designed and implemented parts of the GAL Framework core system and emphasizes appliance of object-oriented design patterns as were described in the well-known book *Design Patterns: Elements of Reusable Object-Oriented Software* [22]. They refer to a terminology used in the book and explain differences between the patterns presented in the book and their usage in the library. Mentioned class names are taken from the library prototype implementation and you can see their detailed documentation at [23].

6.1 Singleton

The singleton design pattern ensures that classes following it have only a single instance and offers public access to it.

Clear candidate for this pattern was a component manager `ComponentManager` class but after further consideration components (`Component`), event handlers (`EventHandler`) and slot executors (`SlotExecutor`) were assigned to the pattern too. All these classes are unique in a process context and except of the `SlotExecutor`, which is an read-only element, need a private structure locking when race conditions will occur. The singletons are realized by a derivation from a `Singleton` C++ template where it is possible. This allows receiving an instance of a final type directly instead of using casting.

6.2 Prototype

The prototype pattern allows for creation of a cloned instance of an object by a virtual method defined in the base class and implemented in subclasses. It is not as restricted to a concrete type as a regular copy constructor.

The GAL Framework uses the prototypes together with prototype managers implemented in a `PrototypeManager` C++ template. They allows a registration, an unregistration and an access to registered prototypes using methods parametrized with a string prototype identifier. The returned reference to the prototype instance may be used to create the clone but it must be dynamically casted to its end type before its use.

The pattern is concretely applied in an `Interface` and an `Object` class with prototype managers in a `InterfaceManager` and a `ObjectManager` classes. A copy method of the `Interface` class performs a deep copy of an interface with all its slots so a `Slot` class is a

prototype as well. Behavior of the copy method of the `Object` class is up to derived class creator but it should be deep in most cases.

6.3 Abstract Factory

Offers a way how to create a concrete object instance without knowledge of its final type. Only common base class type awareness is necessary for a complete class hierarchy. The set of generated object instances can be dynamically reconfigured. The pattern is closely related and cooperative with the prototype and a factory method design pattern.

As noted before, the abstract factories can be combined with the prototype design pattern to create new instances of registered objects. We can say that from this perspective the `ComponentManager` class serves as such factory to create the `Interface` and the `Object` classed objects. The desired interface object instance is requested with an identifier of the interface and cloned from the registered interface prototype.

6.4 Strategy

The strategy is a simple design pattern which detaches diverse algorithms operating on single data from their storage. It supports a dynamically configurable exchange of behavior. The pattern consists of a strategy class which specifies behavior and a context class that describes data.

This is very suitable for various implementations of the slot execution mechanism. The strategy object of the pattern, which defines a common interface of the algorithm, is represented by a `SlotExecutor` class in the framework. A `Slot` class represents the context passed to a strategy method as long as a client that invokes the strategy algorithm method when the slot is executed. Theoretically, this pattern can be changed to a command design pattern to open a further possibilities if a future development imply.

6.5 Iterator

The iterator is certainly the most frequently used design pattern in many applications. It provides sequential access to elements of a compound container object while it keeps its internal structures hidden. It may carry an additional information about the traversal than just an actual item or a position.

Since the library uses associative arrays implemented with STL maps at many places, the iterator pattern is utilized even here. To simplify the STL iterator usage, a `MapIterator` template was created. It basically enwraps the standard iterator received from the container map and extends it with a direct access to a value part of an item and with a boolean cast operator which allows test it in conditions directly. Few cases of the `MapIterator` template utilization are a `SlotIterator` in the interface objects, a `InterfaceIterator` in the component manager and a `PrototypeMapIterator` in the prototype managers.

7 Slot Implementations

The first primary aim of the GAL Framework is to bring a distributed or multiprocessor computation of GIS related tasks into the GRASS. An efficient singleprocessor slot execution is crucial too. To satisfy this contradiction, two mechanisms for the interface function execution was chosen and the others may be introduced on demand. The first is a slot abstracting usual callbacks and the second is a slot calling remote procedures with a D-Bus library [24]. Other alternatives may be the ORBit2 [25], the High Performance Virtual Machines (HPVM) [26], the XML-RPC [27] or the Open MPI [28]. The spirit of the slot design should help incorporate these libraries into the framework.

The slot objects defined in a `Slot` class and belonging to some interface object of a class `Interface` declare methods for an interface function signature management and a method for a slot type setup. Change in the slot type replaces an assigned slot executor of a class `SlotExecutor` which is responsible for an actual slot implementation invocation. An object diagram of this is on figure the 7.1. It shows that the currently configured slot executor is a D-Bus slot executor (the `DBusSlotExecutor` class).

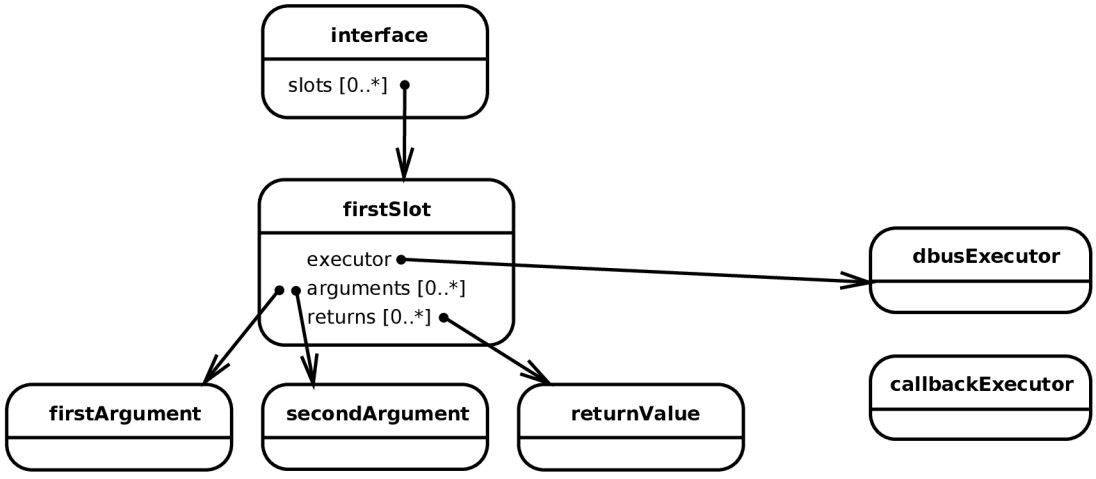


Figure 7.1: The object diagram of a slot execution mechanism selection.

From a module developer position, the communication mechanism for the slot execution is selected by a configuration of the slot type in the interface. This configuration is performed before the interface object is used in modules and may be changed during usage if it is needed so. A called implementation is determined by an appropriate component loaded

locally in the case of a local execution mechanism or by a master process in the case of a D-Bus mechanism. The master process is the one that have registered its name on a D-Bus session bus as the first.

Because a new communication types of the slots are added to the system by a derivation of the `SlotExecutor` base class, a derivation of the `Interface` and the `Slot` base class may be utilized to extend more general behavior. An example of this may be *composed interfaces* and *composed slots* implemented in a `ComposedInterface` and a `ComposedSlot` class. The composed slots offers one additional string argument which identifies an affected subelement of the target component and the composed interfaces has a method for selection of this element and one implicit interface function which queries a list of available elements. This interface can be used when the component hides many objects with the same behavior and a selection of the concrete object is not too frequent.

For the demonstration of present abilities of the library the described configuration options are sufficient but for the future there could be for example configuration methods to execute an implementation of a component specified by its name, a global last registered component, all components with arguments and return values coupled to arrays, etc. A load balancing is the next possible exploitation of these possibilities.

7.1 Callback Slots

The callback implementation of the slot executor in a `CallbackSlotExecutor` class simply returns execution to the slot object which holds a function pointer to a registered component's static method but due to a dynamic nature of a slot signature specification, a foreign function interface library libffi [29] is used a for stack frame construction in runtime. The static methods must be used because taking a function pointer to a regular method and coverting it to a void pointer is illegal in C++. Arguments are passed to the static method preppended with a pointer to the component instance to simulate the object method call. Multiple return values are bound to a single structure returned by the method and then unpacked to slot's return values.

7.2 D–Bus Slots

The D-Bus library was used as a primary library for a remote procedure execution because it is a desktop oriented, living and spread project which will soon become a standard on its field as soon as the KDE 4 and the GNOME will utilize it more. Whole mechanism of the D-Bus slot execution is slightly complex but a schema on the figure 7.2 tries to illustrate it. It shows an example with one object argument and one object return value but basic types like integers and strings are supported too.

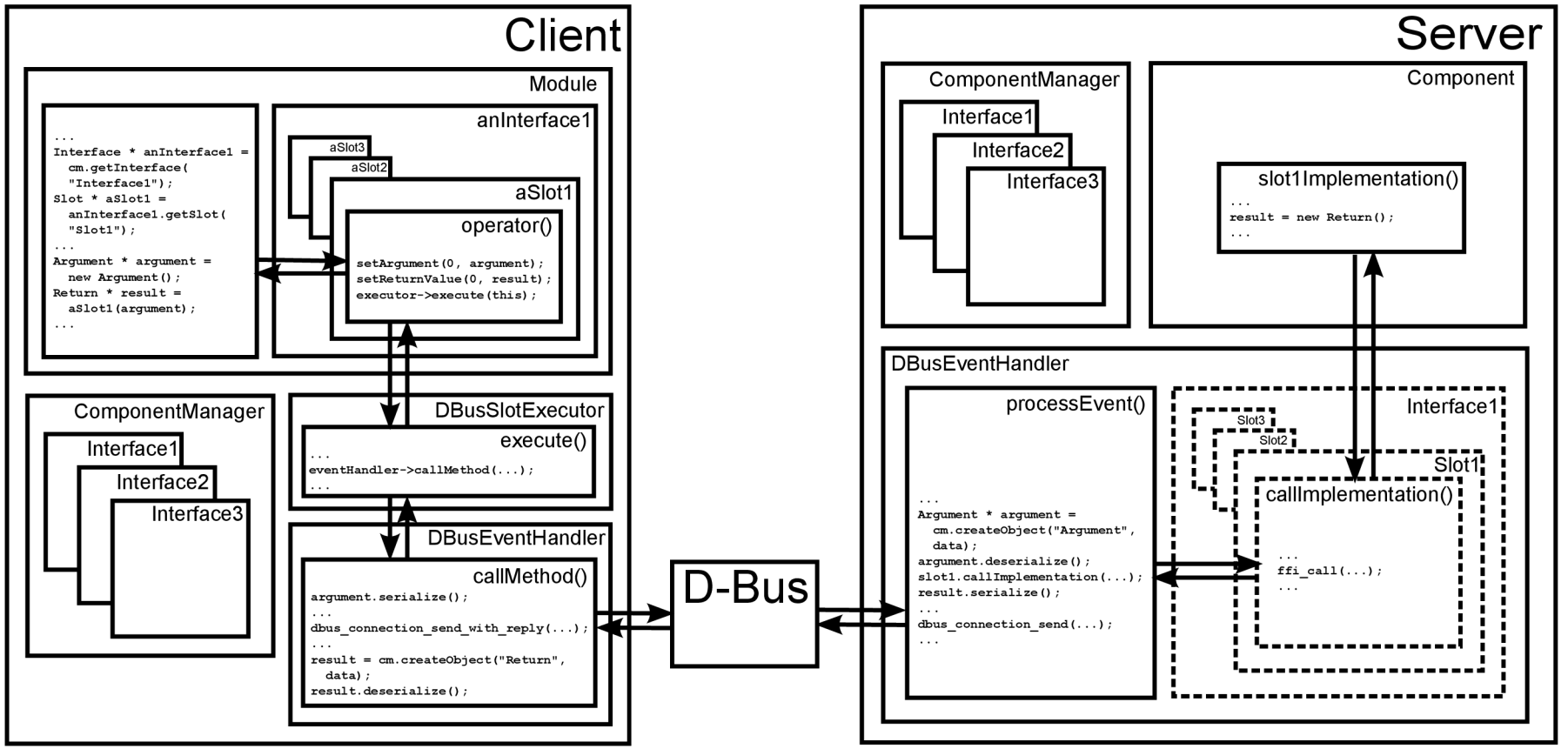


Figure 7.2: The interface function execution schema of a slot configured to a D-Bus communication.

There are three main blocks: a `Client` and a `Server` written using the GAL Framework library and a `D-Bus` library system block. The `Client` consists of a code of an implemented GRASS module `Module` and a framework library code. If the module wants to use an interface, it receives a cloned instance `anInterface1` of an interface prototype `Interface1` from a `ComponentManager`. To execute an interface function, the module asks a received interface object for a reference to its slot `aSlot1` by an interface function name. Then it creates an instance of a function argument `argument` and calls the slot as a functor. All of this denotes an unnamed block in the left part of the `Module` block.

The `aSlot1` in a call operator `operator()` deposits the argument object and a variable for the return value object within itself using a `setArgument()` and a `setReturnValue()` method and tells the configured slot executor to execute it. In this case of the D-Bus communication example, a `DBusSlotExecutor` in an `execute()` method calls an `callMethod()` method of a `DBusEventHandler` object which serializes the input argument, sends a D-Bus method call message and deserializes a received return value to the instance `result` created in the component manager from an object prototype. A control flow than returns to the main module code that executed the slot.

On the server side, the `DBusEventHandler` waits for the D-Bus method call messages. When such arrives, a `processEvent()` method of this object gets an instance of the argument object class from the component manager and deserializes it with message data. Next it finds a proper interface and a slot prototype and executes an interface function implementation using a `callImplementation()` method of the slot with the argument instance. This calls a component's static method in the same manner as in the callback communication. After the method returns an allocated return value object, the `DBusEventHandler` serializes it for a D-Bus method call message reply.

8 GAL Framework Subsystems

A directory structure of a source code tree is publicly available at `svn://gal-framework.no-ip.org:3691` or browsable at `http://gal-framework.no-ip.org/browser`. It is currently divided into eight subsystems: `core`, `exceptions`, `dbus`, `general`, `gis`, `display`, `raster` and `vector`. Each of them contains definitions of subsidiary classes as long as component and interface object definitions. Here is listed only a content of the most important of them. For a detailed description see the GAL Framework library reference [23] or the commented source code.

8.1 Core Subsystem

The core subsystem comprise the same classes as mentioned in the section 5.2 about the analytical model plus a *composed slot* base class `ComposedSlot`, a *composed interface* base class `ComposedInterface`, derived basic slots and *composed slots* with predefined interface function signatures and a callback implementation of the slot executor in a `CallbackSlotExecutor` class. More about the *composed slots*, *composed interfaces* and the callback slot executor can be read in the chapter 7.

8.2 Exception Subsystem

Exceptions are from the beginning intended to be a main mechanism for reporting of errors occurred during the interface function execution or an invalid state or operation signalization when manipulating with objects. The exceptions forms a single inheritance tree starting with an `Exception` superclass. Each exception bares a string message explaining its meaning. They are currently defined in two files the first contains general purpose exceptions and the second declares exceptions related to the D-Bus subsystem. Few examples are: `EIOError`, `ENotFound`, `EArgumentError` and `EConnectionError`.

8.3 D-Bus Subsystem

This subsystem implements remote interface function execution with the D-Bus RPC library [24]. Details on this matter and a deeper explanation of this implementation is present in the section 7.2.

8.3.1 Classes

`DBusEventHandler` is connected to a D-Bus system bus and ensures that a local interface function implementation is found and executed when a D-Bus method call event from another process arrives.

DBusSlotExecutor performs the opposite side of the communication — it produces the D-Bus method call event of the slot's interface function when the slot is executed.

8.4 General Subsystem

It comprehends some general purpose classes and interfaces that could be potentially utilized even in non-GIS-related applications together with classes from the core, the exception, the D-Bus and partially the display subsystem.

8.4.1 Classes

ModuleOptions is a container class with boolean module arguments. Each option has a name and a help description.

ModuleArguments class is a list of module arguments with an assigned value obtainend from an `IArgmentsProvider` interface. Items of a `ModuleArgument` class are structures with a name, the value and a description.

Variable is a similar structure to the module argument used in `IEnvironmentProvider` interface functions.

8.4.2 Interfaces

IArgmentsProvider interface can be used to get options and arguments which controls module actions. It also builds a help string from given module arguments and options descriptions.

- **getOptions** method returns the module options formatted in the `ModuleOptions` object.
- **setOptions** sets the list of acceptable options of the module. If this function is not used before the `getOptions` call, the module gets all arguments provided by a implementing component.
- **getArguments** gives the `ModuleArguments` object that contains a list of the requested module arguments.
- **setAruments** is an analogous method to the `setOptions` but for the module arguments.
- **setDefaultArgument** tells the implementing component what argument is a default one.
- **getHelp** creates a formatted help string for the module which can be printed to the standard output or displayed in the user interface. It accepts a module description as a parameter and employs the previously setted descriptions.

IEnvironmentProvider helps abstract foreign sources of environment variables. For example, a GRASS GIS module `g.env` reads the varables from a global configuration file or a local configuration file of a mapset. System-wide varables (an active monitor for example) could be store using this interface too. This interface has a `getVariable` and a `setVariable` function.

8.4.3 Components

DefaultArgumentsProvider returns from the IArgmentsProvider interface functions formatted command-line arguments passed to the GAL Framework initialization method.

DefaultEnvironmentProvider implements the IEnvironmentProvider interface for a GAL internal variables storage.

GRASSRCEnvironmentProvider also implements the IEnvironmentProvider interface but for GRASS's variables saved in the global configuration file.

GRASSMapsetEnvironementProvider is the same as the previous components but for the variables of the mapset.

8.4.4 Modules

g.gald is a daemon module that loads all internal components of the GAL Framework with their implementations and it waits for their interface functions execution over the D-Bus. It is needed for example in cooperation with a d.mon and a d.rast modules because it keeps a component with monitor windows available even after the d.mon module finishes.

g.quit is an auxiliary module that terminates running master module which is in the most cases the g.gald.

g.gisenv supplies some possibilities of the equally called GRASS module and operates with the IEnvironmentProvider interface and its implementing components.

8.5 Display Subsystem

Classes, interfaces and components of the display subsystem provide a support for graphical user interface or data visualization related modules. They are created with the help of the Qt widget toolkit [30] version 4.x but there is a possibility to use any other GUI library if it allows an individual event processing.

8.5.1 Classes

QtEventHanler implements the EventHandler base class and manages the Qt event processing loop and a QApplication object. More on event handlers is in the section 5.2.

Area is a general rectangle region with x and y coordinates, a *width* and a *height*.

RasterImage is a class for a multichannel two dimensional raster image data storage and manipulation. It's used in a IRasterDisplayer interface. Pixels can be accessed as raw data or as Color typed elements at chosen x and y cooordinates. The content or the format can be described with a metadata attribute.

QtMonitorGLWidget inherits from a QGLWidget class and appends an internal image buffer for widget canvas repaint everytime it's needed. As the name may indicate, it uses OpenGL for rendering.

QtMonitorWindow is derived from a QMainWindow class and contains the QtMonitorGLWidget widget as a drawing area.

drawImage is the only method added to the base class. Method converts a RasterImage to a QImage for drawing on the canvas.

8.5.2 Interfaces

IMonitorController serves for modules that controls size, position and appearance of graphical windows. For example, modules like a d.mon module from the GRASS GIS which controls monitors. It is the *composed interface* where the *element* identifies a target monitor window. A list of provided interface functions would be too long and it is rather unimportant to be listed here.

IRasterDisplayer is the *composed interface* that allows modules display raster images on the monitor. Visualized data is just multichannel pixel pictures not raster GIS data as name could insult. The *element* is a target monitor window name. An example module that could use this interface is a d.rast.

getRasterImageArea returns an Area object with dimensions of canvas where the raster images can be drawn. This means that any rendering request beyond the borders will not be visible.

displayRasterImage displays a raster image of the RasterImage class at given coordinates and with specified dimensions.

8.5.3 Components

DefaultMonitorController component implements the IMonitorController and the IRasterDisplayer interfaces for eight monitors named x0 to x7. This respects practices from the GRASS GIS. As long as this component is resident in the memory and initialized, it can accept interface functions implementation calls.

RoamerComponent is a base and only component that implements the IMonitorController and the IRasterDisplayer interfaces for a d.roamer visualization tool. An *element* name of the module window is a roamer for the identification in interface functions.

8.5.4 Modules

d.mon is a simple module that shows, hides or selects as active monitors provided by the DefaultMonitorController component instantiated in the g.gald daemon. Raster layers can be continuously displayed with the d.rast program. Such a monitor window can be seen on the figure 8.1.

d.move allows shift a monitor window from the command-line or a script. It uses one of the IMonitorController interface functions to accomplish that.

d.resize is similar module to d.move but it resizes the monitor window instead.

d.roamer is a more complex 3D visualization tool for the GRASS GIS written using the GAL Framework. The user can freely roam over the displayed terrain with this tool and one of selectable level of detail algorithms for the rendering is the ROAM

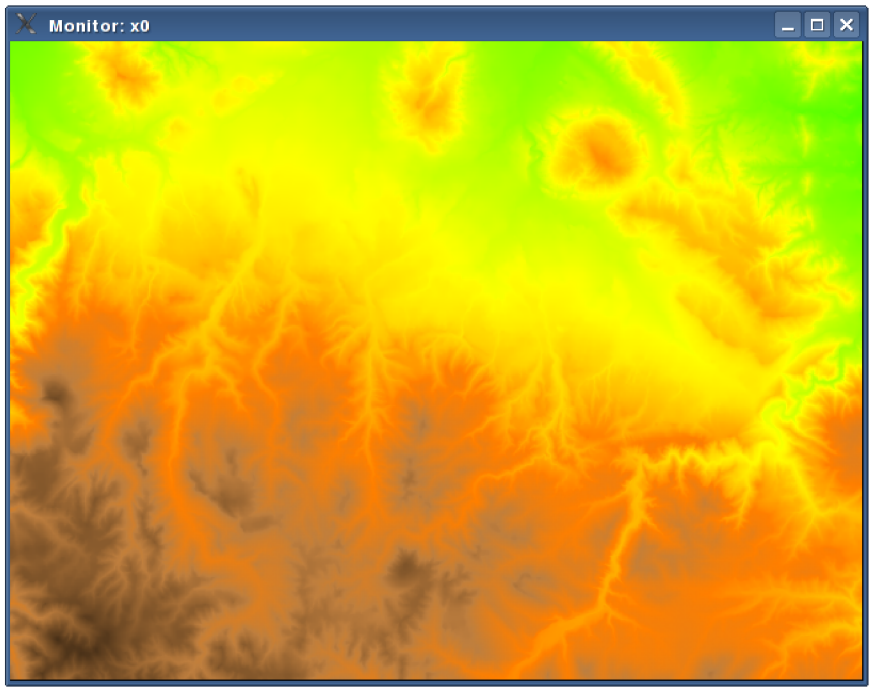


Figure 8.1: The monitor window opened with the `d.mon` module with an `elevation.10m` raster layer from the `spearfish60` test mapset displayed by the `d.rast` module.

algorithm [31] from the SoTerrain library [32]. That's where the module name comes from.

The module displays a raster layer and it is controlled in the same way as the `DefaultMonitorController` managed windows with the `d.mon` and the `d.rast` modules of the framework. The layer is interpreted as a texture but a terrain heightmap can be selected with the `d.rast` module using the different argument during the visualization too as can be seen on the figure 8.2.

The figure 8.3 demonstrates a communication begind the raster layer display. The `d.roamer` module with its `RoamerComponent` is running as a daemon and accepts D-Bus method call events. The `d.rast` module gets raster layer data from the `GRASSRasterLayerProvider` component in the framework and sends them to the listening `RoamerComponent` component which creates and shows a scene graph with the terrain.

8.6 GIS Subsystem

Here in this subsystem, support instruments for performing GIS-specific computation such as map projections or coordinate systems may be present. For the moment, there are available only a rectangular region abstraction class representing an area of interest in the map and an interface and components working with it.

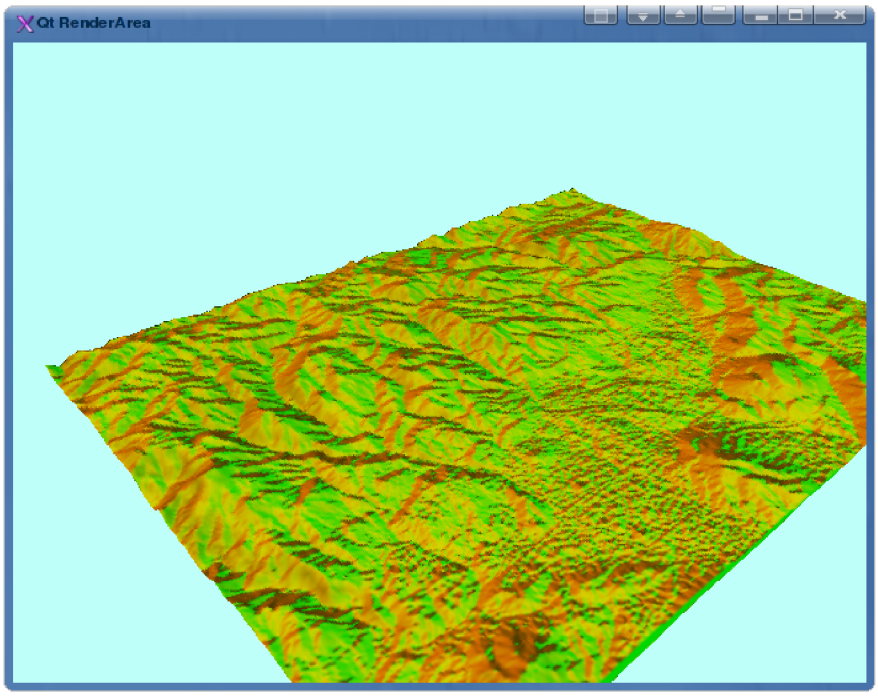


Figure 8.2: The `d.roamer` module user interface with an `aspect` raster layer from the `spearfish60` test mapset displayed as a texture and an `elevation.10m` layer displayed as a heightmap.

8.6.1 Classes

`LayerRegion` bounds cutout of a raster or a vector map layer with an east, a west, a north and a south edge. It also describes number of rows and columns and a north-south and a east-west resolution for convenience although one information can be computed from another.

8.6.2 Interfaces

`IRegionProvider` presents an access point to components providing different areas of interest on the map layers.

`getRegion` returns the requested `LayerRegion` region object.

8.6.3 Components

`GRASSDefaultRegionProvider` manages the default region of the GRASS GIS mapset and offers it to any component or module that uses the `IRegionProvider` interface.

`GRASSUserRegionProvider` is analogous component but for the active region of the user.

8.6.4 Modules

`g.region` module allows read and modify the user region. It reimplements the same-called module from the GRASS GIS package using the `IRegionProvider` interface.

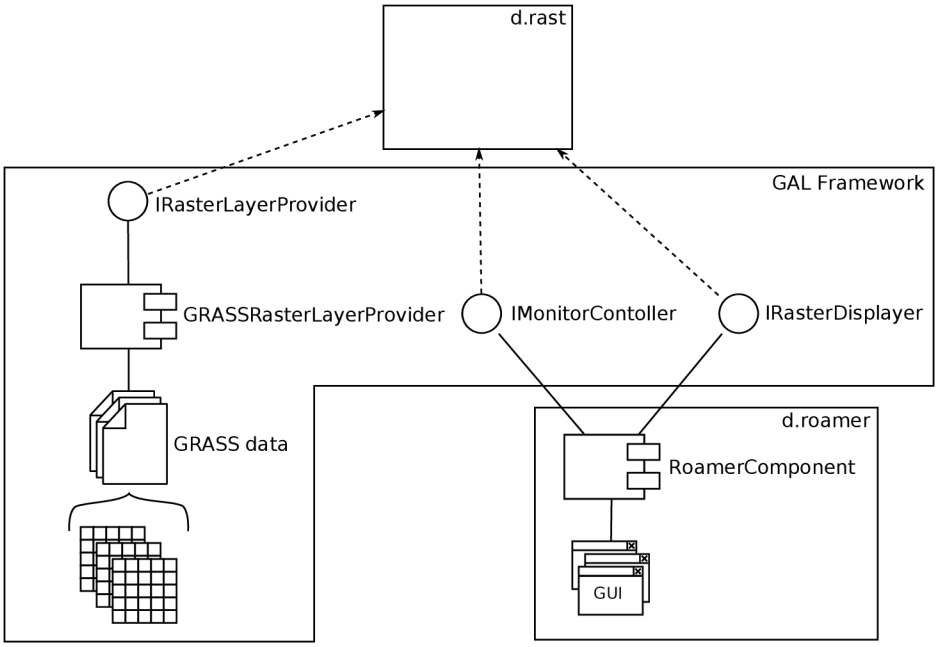


Figure 8.3: The internal architecture of the `d.roamer` module.

8.7 Raster Subsystem

Interaces and components that access or manipulate GIS raster data as well as modules that implement such interfaces belong to this subsystem. A current implementation reads raster data from the native GRASS format using its raster library in tiles.

8.7.1 Classes

`ColorTable` is a linear table which converts raster data values to corresponding colors. It's bounded by a minimal and a maximal value. It's used for a raster layer visualization where each value denotes an independent category.

`getColor` method returns a color of a `Color` class for a given integer value of a raster cell.

`ColorRules` is a set of intervals with assigned boundary colors. The resulting color is computed by their interpolation during the look up. This is more suitable for the visualization of rasters that models continuous phenomenons.

`addRule` appends a new color rule to the set with the specified interval and the boundary colors.

`getRule` provides access to the stored color rules of a type `ColorRule` which offers a color look up method `getColor`.

`RasterTile` is a tile of raster data returned by a `IRasterLayerProvider` interface function. It support various cell data types. Currently it's designed for two dimensional

rasters only but this should be changed for future practical applications of the library. Some of its attributes are the color table, the color rules and metadata string.

getColorTable returns reference to the color table of the tile.

getColorRules gives reference to the object with color rules.

getMedatata returns the string with metadata attached to the tile. It is not decided what form this metadata should take because they are not needed in example modules.

getClolor performs a direct color look up for a raster tile cell at given coordinates.

8.7.2 Interfaces

IRasterLayerProvider is a *composed interface* that could be used in every raster processing module because it could provide access to any raster data in a GRASS mapset and location or any other external format. Source of data depends only on a selected implementing component. Data are requested in tiles of specified position and dimensions by the layer region.

getLayer returns a part of the raster layer represented by a `RasterTile` object instance.

IRasterLayerDisplayer interface may be preferred in the cases where a raster data transfer between components would be too expensive for the interactive visualization. An interface function just passes request for layer display and an implementing component reads the data by its own. Even this interface is the *composed*.

displayLayer tells what raster layer should be displayed.

8.7.3 Components

GRASSRasterLayerProvider implements the `IRasterLayerProvider` interface and provides raster layers from the GRASS *locations* with the `libgrass_gis` library.

8.7.4 Modules

g.list module is able to print to the standard output a list of all available raster layers provided by the `GRASSRasterLayerProvider` component.

d.rast is more related to the raster subsystem even if it serves for the raster layer display on the active monitor. It reads raster data using the `IRasterLayerProvider` interface and sends them to a visualization component in the `g.gald` process using the `IRasterDisplayer` interface.

8.8 Vector Subsystem

The vector subsystem is not present in the library because an internal representation of vector layers should have been elaborated in the Bc. Jan Kittler's master's project and then interfaced to the framework's practices. Unfortunately, he postponed his work to the next academic year. Author can only hope that further cooperation with him will bring outstanding implementation of vectors to the GAL Framework.

9 Dynamic Language Bindings

The second of the primary targets of the GAL Framework is to support an interface function execution from various dynamic languages. This chapter discuss achievements and obstructions of this objective.

9.1 SWIG Utilization

The Simplified Wrapper and Interface Generator (SWIG) [33] tool is suitable for an automatic generation of C/C++ libraries bindings to many dynamic languages. This fully applies for a C code which don't use variable length arguments functions and callbacks. Support of these programming techniques is not at all or at least hardly possible from the principle. Some C++ constructs like nested classes, a `new` and a `delete` operator overloading or an uncommon use of templates are not wrappable as well. Of course, a set of transformed source language properties depends on a target language capabilities. For this reason, a wrapper of the slot execution methods has to be thoroughly customized. Fortunately, this could be done with advanced SWIG features or certain hacks.

9.2 General Customizations

As mentioned before, the nested C++ classes can't be wrapped by the SWIG. To overcome this limitation, nested slot declarations inside the interface object classes had to be renamed to a global GAL namespace. For example, they are accessible under a `GAL.SomeSlot` class in Python instead of a `GAL::ISomeInterface::SomeSlot` class in C++.

Even if wrapped object *proxies* can be thrown as exceptions in the target language, for a more clearly readable exception backtrace it's better to call a predefined SWIG exception evocation function in a *throws typemap* which transforms the C++ exception to an appropriate target language exception of a specified type.

A minor limitation represents need of a template instantiation before their interfacing. This means that there can't be used the templates typed with types declared in the host language.

9.3 Python Bindings

At the beginning, only the Python bindings were developed to prove a GAL Framework's core system possibilities in the dynamic language support. This language was chosen because of its simplicity, clearness, frequent usage and because it's the best implemented

target language in the SWIG. This implies a good knowledge base available on the Internet. The resulting wrapper is a dynamic library written using a Python/C API [34] and a single Python script with all *proxy* classes.

The first needed modification to the wrapper interface for this language was a rename of all used `operator[]` operators to `__getitem__()` methods for a read access and `__setitem__()` methods for a write access. Then, a *in typemap* converting a `argc` and a `argv` argument of the `GAL::initialize()` method to a Python list of strings was created for a convenience because Python stores the command-line arguments in a `sys.argv` list.

The slots obtained from the interface objects must be casted to their final type to allow the interface function call with a defined signature in C++ as you can notice in a library tutorial in the appendix A. The SWIG wraps an `Interface::getSlot()` method to return a *proxy* object containing a pointer to a `Slot` but Python expects that an object reference is always of its final type and thus it doesn't offer any casting mechanism for this. To fix this contradiction, C++ conversion functions would be needed to be written and wrapped if there wasn't other solution. The slots were designed for a dynamic signature specification and execution. The overloaded call operators of derived slots are provided only for convenience. So, only thing needed was extend the Python `Slot` class with a `__call__()` method with variable length arguments which converts Python basic types to C++ basic types and extracts an internal pointer from Python *proxies* when passing the slot arguments and which converts the basic C++ types to the basic Python types and creates the appropriate Python *proxy* object when dealing with the return values.

9.4 Java Bindings

The second selected dynamic language is Java for its vast usage although it is explicitly compiled, with less dynamism and more language restrictions than Python. The SWIG generates a dynamic library with bindings for this language using a Java Native Interface (JNI) [35] and it creates Java *proxy* classes for all defined types in separate files.

Java doesn't have operator overloading, that's why an `operator()`, an `operator[]`, an `operator++`, an `operator--`, an `operator+`, an `operator-`, an `operator*` and an `operator/` operator had to be renamed to an `call()`, an `get()`, an `inc()`, an `dec()`, an `add()`, an `sub()`, an `mul()` and an `div()` method. Even here, the *in typemap* was written to allow pass a string array with arguments to the library initialization method. Another trouble with Java was the fact that a name of the destructor collides with the name of a framework deinitialization method therefore it was renamed to a `GAL::_finalize()`.

For experimental and presentational reasons, a conversion using a *out typemap* of the SWIG tool was picked as a solution to the slot type problem for this language. The typemap determines a slot type with a `Slot::getClassName()` method, creates a *proxy* for this type and the *proxy* can be then safely casted using native casting operator in a Java code.

9.5 Other Bindings

Despite of the fact that only Python and Java was chosen from a long list of languages that the SWIG supports, C#, Perl and Ruby was other candidates and presents a potential field of evolvement of the GAL Framework. Tcl could be considered also because it's widely used in a GRASS GIS user interface.

10 Experimental Results

As you may point out, the performace will never be the strongest side of the slot execution because of its dynamic nature. Expensive operations should be methods of objects returned from the slots rather than the interface functions itself. The selected execution mechanism affects an interface function call overhead significantly. To make a better image about its impact on the performace, a set of tests was created and performed. Results are presented and discussed in this chapter.

The testing machine was a Intel Core 2 Duo laptop at 1.8 GHz frequency with a 2 GB RAM memory. There was defined an experimental interface with slots accepting differently typed arguments and giving the same return values. That means for example that there was a slot with a single integer argument and a single integer return value. Other tested data types was double, string, object and a slot with no arguments or return values. A string argument value was the "test" word. The object argument was a simple `Object` class derived instance with one integer attribute. Testing applications and scripts are in a `GAL/test/` directory of the root directory on the CD.

The measured results are listed in the table 10.1. Columns denote type of the slot argument and the return value and rows represent used execution mechanism and language. Values are time that costed one execution of an interface function implementation with an empty body.

Table 10.1: The performace of the slot execution.

Type	Void	Integer	Double	String	Object
C++ local	200 ns	287 ns	285 ns	296 ns	359 ns
C++ D-Bus	1.08 ms	1.10 ms	1.09 ms	1.09 ms	1.11 ms
Python local	2.07 μs	2.48 μs	2.53 μs	2.54 μs	24.4 μs
Python D-Bus	1.08 ms	1.10 ms	1.10 ms	1.10 ms	1.12 ms

The first and the fastest row is a callback implementation of the slot mechanism called from a C++ testing module locally. The first cell should describe a raw overhead of execution, the others are the raw overhead plus an overhead per argument and return value of the appropriate type. The integer, the double and the string slots are almost equal. Only an object argument cost is a little bit higher because of an instance creation. Process ran on a single core.

The next is a C++ module calling the implementation in a server process using the D-Bus. It's four orders slower because of the low bandwidth of the D-Bus. Although there

was enabled both cores during the testing, overall system usage was very low (about the 10% per core). This is caused by limitations of a D-Bus synchronization and implies that there is a need to introduce another RPC based slot implementation which don't suffer with this problem. Using less messages in a single moment or bigger ones is more than advised when using the D-Bus slots.

The Python bindings with a direct implementation have proven only one order slower when using the basic types but the object arguments are more expensive than that. This is probably caused by a proxy object creation and destruction.

The cost of Python interpretation has no visible influence comparing to the time losses in the D-Bus synchronization which shows the last line of the table although both client and server was written in Python.

11 Conclusion

Although many work on the design and the prototype implementation was done, there is still much things that could or should be appended to the framework to be generally usable. The author believes that the point of the project was to present an idea and prove it on a pilot implementation not to create a final full–featured system. Future of the project now depends on the intersest of the community.

Lastly, here is just mention about the support tools used during the project development. The Trac [36] was picked up as a project management tool, the Subversion [37] as a source code management system, the SCons [38] as a build system, the Doxygen [39] as the library documentation generator and the many others that was noted previously in the text. Thanks belongs to their developers for the help they have granted.

12 References

[1] The Wikimedia Foundation. Geographic Information System. http://en.wikipedia.org/wiki/Geographic_information_system, March 2008.

[2] Radek Bartoň. GAL Framework Homepage. http://gal-framework.no-ip.org/, July 2007.

[3] William D. Goran, William E. Dvorak, Lloyd Van Warren, and Ronald D. Webster. Fort Hood Geographic Information System: Pilot System Development and User Instructions. Technical Report N-154, USA Construction Engineering Research Laboratory, Champaign, IL., May 1983.

[4] James Westervelt. GRASS Roots. In *Proceedings of the FOSS/GRASS Users Conference*. FOSS/GRASS Users Conference, September 2004.

[5] GRASS Development Team. GRASS History. http://grass.itc.it/devel/grasshist.html, September 2007.

[6] Radim Blažek, Markus Neteler, and Roberto Micarelli. The New GRASS 5.1 Vector Architecture. In *Proceedings of the Open source GIS – GRASS users conference 2002*. University of Trento, September 2002.

[7] GRASS Development Team. GRASS GIS 6.1.0 Released. http://grass.itc.it/announces/announce_grass610.html, August 2006.

[8] GRASS Development Team and Markus Neteler. GRASS 5.0 Programmer's Manual. http://grass.itc.it/grass50/progmangrass50.pdf, January 2004.

[9] GRASS Development Team. Raster Data Processing in GRASS GIS. http://grass.itc.it/grass63/manuals/html63_user/rasterintro.html, April 2008.

[10] GRASS Development Team. GRASS 6 Programmer's Manual. http://download.osgeo.org/grass/grass6_progman/, April 2008.

[11] The Wikimedia Foundation. Shapefile. http://en.wikipedia.org/wiki/Shapefile, April 2008.

[12] PostgreSQL Global Development Group. PostgreSQL. http://www.postgresql.org/.

[13] PostGIS Development Team. PostGIS. http://www.postgis.org/.

[14] GDAL Development Team. OGR. http://www.gdal.org/ogr/.

[15] GRASS Development Team. GRASS DBMI DataBase Management Interface. http://download.osgeo.org/grass/grass6_progman/dbmilib.html, March 2008.

[16] Radek Bartoň and Martin Hrubý. GAL Framework. In *Proceedings of the workshop Geoinformatics FCE CTU 2007*. Czech Technical University in Prague, September 2007.

[17] Christopher Lenz, Dave Abrahams, and Christian Boos. Trac Component Architecture. http://trac.edgewall.org/wiki/TracDev/ComponentArchitecture, July 2007.

[18] Jim Arlow and Ila Neustadt. *UML and the Unified Process: Practical Object-Oriented Analysis and Design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[19] Radek Bartoň. Component Architecture. http://gal-framework.no-ip.org/wiki/ComponentArchitecture, September 2007.

[20] Radek Bartoň. Use Cases. http://gal-framework.no-ip.org/wiki/UseCases, July 2007.

[21] Radek Bartoň. Analytical Classes. http://gal-framework.no-ip.org/wiki/AnalyticalClasses, July 2007.

[22] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, Reading, MA, USA, 1995.

[23] Radek Bartoň. Analytical Classes. http://gal-framework.no-ip.org/doxygen/, May 2008.

[24] freedesktop.org. D-Bus. http://www.freedesktop.org/wiki/Software/dbus.

[25] GNOME Foundation. ORBit2. http://www.gnome.org/projects/ORBit2/.

[26] Andrew A. Chien, Daniel Reed, and David Padua. High Performance Virtual Machines. http://www-csag.ucsd.edu/projects/hpvm.html.

[27] XML-RPC Development Team. XML-RPC. http://www.xmlrpc.com/.

[28] Open MPI Development Team. Open MPI. http://www.open-mpi.org/.

[29] Anthony Green and Gianni Mariani. libffi. http://sourceware.org/libffi/.

[30] Trolltech ASA. Qt. http://trolltech.com/products/qt.

[31] Mark A. Duchaineau, Murray Wolinsky, David E. Sigeti, Mark C. Miller, Charles Aldrich, and Mark B. Mineev-Weinstein. ROAMing terrain: Real-time optimally adapting meshes. In *IEEE Visualization*, pages 81–88, 1997.

[32] Radek Bartoň. SoTerrain. http://blackhex.no-ip.org/wiki/SoTerrain, October 2007.

[33] SWIG Development Team. Simplified Wrapper and Interface Generator. http://www.swig.org/.

[34] Python Software Foundation and Fred L. Drake, Jr. Python/C API Reference Manual. http://docs.python.org/api/api.html, February 2008.

[35] The Wikimedia Foundation. Java Native Interface. http://en.wikipedia.org/wiki/Java_Native_Interface, February 2008.

[36] Edgewall. Trac. http://trac.edgewall.org/.

[37] Tigris.org. Subversion. http://subversion.tigris.org/.

[38] The SCons Foundation. SCons. http://www.scons.org/.

[39] Dimitri van Heesch et al. Doxygen – Source Code Documentation Generator Tool. http://www.stack.nl/~dimitri/doxygen//, May 2008.

Appendix A

Library Tutorial

This appendix contains a tutorial of a GAL Framework library appliance in a GRASS GIS module development. First we will show a basic utilization of a predefined functionality in general on an imaginary interface. Then we will define the used object, the slot, the interface and the component on our own which may be interesting for those who wants extend framework's features. Finally, we will create some more practical module that simulates functionality of the `g.list rast` command. Complete source codes of these examples are placed in a `GAL/doc/Examples/` directory on an attached CD.

A.1 Imaginary Interface

We are going to write a client-side module which calls interface functions to get some data or perform a computation over them and then exits. A structure of such module can be reduced to the following code skeleton:

```
// GAL Framework includes.
#include <GAL/core/Basic.h>

// Local includes

int main(int argc, const char * argv[])
{
  try
  {
    // Initialize GAL Framework.
    GAL::initialize(argc, argv);

    // Create and initialize components.
    ...

    // Get interface objects form component manager.
    ...

    // Get slots from intefaces
    ...

    // Do the job.
    ...

    // Free recieved objects and interfaces.
    ...
```

```
    // Deinitialize and free component.
    ...

    // Deinitialize GAL Framework and exit.
    GAL::finalize();
  }
  catch (Exception exception)
  {
    std::cerr << "Unexpected error: " << exception.getMessage()
      << std::endl;
    return EXIT_FAILURE;
  }

  return EXIT_SUCCESS;
}
```

All significant code is enclosed in a `try` statement with an appropriate exception handler and between a `GAL::initialize()` and a `GAL::finalize()` method. They prepare or free internal structures of the GAL Framework so code that uses it should be within.

Let's suppose that there is a `ICustomInterface` interface and a `CustomComponent` component that implements it in the framework and we want to call this implementation to set or get some data. First we must load the component with the required interface implementation.

```
// Create and initialize component with implementation.
CustomComponent * component = new CustomComponent();
component->initialize();
```

Then we get a reference to a component manager and request for the interface object by an interface name.

```
// Get interface object form component manager.
ComponentManager & cm = GAL::getComponentManager();
Interface * interface = cm.getInterface("ICustomInterface");
```

The `ICustomInterface` interface has three functions. The first, called a `foo()`, doesn't have any arguments or return values, the second function a `bar()` has a single integer argument and the third a `getPoint()` function returns a point object of a class `Point`. To call these functions, we must obtain slot objects representing them from the interface.

```
// Get slot of ICustomInterface inteface functions.
FooSlot & foo = reinterpret_cast<FooSlot &>(
  interface->getSlot("foo"));
BarSlot & bar = reinterpret_cast<BarSlot &>(
  interface->getSlot("bar"));
GetPointSlot & getPoint = reinterpret_cast<GetPointSlot&>(
  interface->getSlot("getPoint"));

// Call them.
foo();
```

```
bar(99);
Point * point = getPoint();
```

When the work is done we should free the received point object and the interface object.

```
// Free recieved object and interface.
point->free();
interface->free();
```

The component with an implementation is no longer needed so we can deinitialize and release it too.

```
// Deinitialize and free component.
component->finalize();
delete component;
```

That's all for this case, you can see the joint code snippets of this module example in a file `GAL/doc/Examples/core_example.cpp` on the CD.

A.2 Custom Object

We saw the point object of the class `Point` in the previous example. Now we will show how can be such custom object declared and implemented. Every object that can be used as an interface function argument or a return value must be derived from a `Object` base class and override a `clone()`, a `serialize()` and a `deserialize()` method in order to be be transferable between processes or hosts with remote procedure call libraries.

```
class Point: public Object
{
```

A default constructor and a constructor with point coordinates is defined next. Notice that both constructors sets a name of the object's class with a `setClassName()` method. This is crucial for runtime determination of a object type.

```
  public:
    /* Constructors. */
    Point()
    {
      this->setClassName("Point");
    };
    Point(int _x, int _y):
      x(_x), y(_y)
    {
      this->setClassName("Point");
    }
```

Every `Object` is a prototype (see chapter 6). That's why we must override the `clone()` method to get the right instance of the `Point` class. This can be done with the copy constructor. Because we won't define any pointer-linked internal attributes, we don't need to write the copy constructor and we use an implicit one.

```
/* Prototype methods. */
virtual Object * clone() const
{
  // Clone with copy constructor.
  return new Point(*this);
}
```

Override the serialization and the deserialization method is what is left. The serialization method returns a string object with (in this case) a binary representation of a `Point` class state and the deserialization takes that string and restores the state. In other words: they must be complementary.

```
/* Serialization methods. */
virtual String serialize() const
{
  // Serialize point to string.
  String data;
  data.append(reinterpret_cast<const char *>(&this->x), sizeof(int));
  data.append(reinterpret_cast<const char *>(&this->y), sizeof(int));
  return data;
}
virtual void deserialize(const String & data)
{
  // Deserialize point from string.
  const char * bytes = data.data();
  this->x = *reinterpret_cast<const int *>(bytes);
  bytes += sizeof(int);
  this->x = *reinterpret_cast<const int *>(bytes);
  bytes += sizeof(int);
}
```

Now comes a definition of an actual behavior of the point object which is only a `getString()` method returning a string with point's coordinates and attribute access methods.

```
/* Attribute access methods. */
int getX() const { return this->x; }
void setX(const int X) { this->x = X; }
int getY() const { return this->y; }
void setY(const int Y) { this->y = Y; }

/* Object methods. */
String getString()
{
  // Return string with point coordinates.
  std::ostringstream stream;
  stream << "[" << this->getX() << ", " << this->getY() << "]";
  return stream.str();
};
```

And finally a declaration of the internal attributes.

```
  private:
    /* Internal attributes. */
    int x;
    int y;
};
```

Complete source code of this part is in `GAL/doc/Examples/custom_object.h` file on the CD.

A.3 Custom Slot

Now we will define the custom slot `GetPointSlot` that returns our `Point` classed object. It is the same that is used in the `ICustomInterface` from the first part of the tutorial and derived from the basic slot object class `Slot`.

```
class GetPointSlot: public Slot
{
```

In a slot constructor, there is specified an interface function signature using a `addArgument()` and a `addReturnValue()` methods. In this case, it is only a single return value of a `OBJECT` type since the slot just returns a point object derived from the `Object` class. See the library reference for further information about the methods and the other possible types.

```
  public:
    /* Constructor. */
    GetPointSlot():
      Slot()
    {
      this->addReturnValue(OBJECT);
    }
```

The slot objects also honours the prototype design pattern thus the cloning method needs to be implemented in the same way as in the `Object` derived classes.

```
    /* Prototype methods. */
    virtual Slot * clone() const
    {
      return new GetPointSlot(*this);
    }
```

The most notable part of the custom slot implementation is their function call operator. To allow a direct slot execution as a functor in C++ language, the call operator must set pointers to variables that holds arguments or where is a space for return values with a `setArgument()` and a `setReturnValue()` method and then call an `execute()` method. Here is the way how to accomplish this for the `GetPointSlot` slot:

```
    /* Call operator. */
    Point * operator()()
    {
      Point * result = NULL;
      this->setReturnValue(0, &result);
      this->execute();
      return result;
    }
};
```

That's it. The point object will be allocated and returned from the library and when it won't be needed, it may be freed with a `free()` method by the module. A full slot definition can be seen in a `GAL/doc/Examples/custom_slot.h` file on the attached CD.

A.4 Custom Interfce

The next step is to create the `ICustomInterface` interface object. We will use the `GetPointSlot` slot from the previous example and two slots a `FooSlot` and a `BarSlot` predefined in a header file `GAL/include/core/BasicSlots.h` as a `V_V_Slot` and a `V_I_Slot`. As you may notice, names of these slots contain a shortcut for their signature. For example, the first `V` in the `V_V_Slot` name means a void return value and the second means that the slot has no arguments. We just rename them according to the interface function name.

```
// Name predefined slots.
typedef V_V_Slot FooSlot;
typedef V_I_Slot BarSlot;
```

Then we inherit from the `Interface` base class of the interface objects.

```
// Custom component class.
class ICustomInterface: public Interface
{
```

In a default constructor, we first set an interface object class name for its identification in the component manager at runtime and then we create and append the slot object instances with an interface function names using a `addSlot()` method. This declares the available functions of the `ICustomInterface` interface and every component that wants to implement it must implement all of them.

```
  public:
    /* Constructor. */
    ICustomInterface():
      Interface()
    {
      // Set interface name.
      this->setName("ICustomInterface");

      // Append predefined slot instances.
      this->addSlot("foo", new FooSlot());
      this->addSlot("bar", new BarSlot());
```

```
      this->addSlot("getPoint", new GetPointSlot());
    }
```

A copy constructor in this case must be defined even empty because there must bee called an `Interface`'s copy constructor which makes a copy of all aggregated slot objects.

```
    /* Copy constructor. */
    ICustomInterface(const ICustomInterface & interface):
      Interface(interface)
    {
      // Nothing more since interface doesn't have internal attributes.
    }
```

A destructor does the inverse action to the default constructor. That is remove and deallocate the previously appended slots with a `removeSlot()` counterpart of the `addSlot()` method.

```
    /* Destructor. */
    virtual ~ICustomInterface()
    {
      // Remove and free slots.
      Slot * fooSlot = &(this->getSlot("foo"));
      Slot * barSlot = &(this->getSlot("bar"));
      Slot * getPointSlot = &(this->getSlot("getPoint"));
      this->removeSlot("foo");
      this->removeSlot("bar");
      this->removeSlot("getPoint");
      delete fooSlot;
      delete barSlot;
      delete getPointSlot;
    }
```

Finally and again, the prototype cloning method have to be defined.

```
    /* Prototype methods. */
    virtual Interface * clone()
    {
      // Redefine prototype clonning method using copy constructor.
      return new ICustomInterface(*this);
    }
};
```

A file with this part is a `GAL/doc/Examples/custom_interface.h`

A.5 Custom Component

Now we have gotten through all preparation steps to bring the new functionality to the GAL Framework. In reality, the previous three steps won't be so often necessary because in the most cases we implement already defined interfaces. Only left is to specify the own component with the interface implementation. We start deriving from a `Component` class.

```
class CustomComponent: public Component
{
```

Then we set a component's name and tell what static methods of the component implement what interface functions to the base class in a copy constructor. The component manager will ask for this information during an implementation registration. A destructor of this component is empty.

```
  public:
    /* Constructor and destructor.*/
    CustomComponent():
      Component()
    {
      // Set unique component name.
      this->setName("CustomComponent");

      // Add implementation methods in Component class.
      this->setImplementation("ICustomInterface::foo",
        (void *) &(this->foo));
      this->setImplementation("ICustomInterface::bar",
        (void *) &(this->bar));
      this->setImplementation("ICustomInterface::getPoint",
        (void *) &(this->getPoint));
    }
    virtual ~CustomComponent() {}
```

In an initialization method of the component, we create and register in the component manager an owned interface object, a point object prototypes as well as an interface implementation.

```
    /* Component methods that have to be implemented. */
    void initialize()
    {
      ComponentManager & cm = GAL::getComponentManager();

      // Register Point object.
      this->objectPrototype = new Point();
      cm.registerObject(*this->objectPrototype);

      // Register CustomInterface.
      this->interfacePrototype = new ICustomInterface();
      cm.registerInterface(*this->interfacePrototype);

      // Register CustomInterface interface implementation.
      cm.registerImplementation(*this->interfacePrototype, *this);
    }
```

A finalization method unregisters the prior registrations and deletes the allocated instances of the prototypes.

```
    void finalize()
    {
```

```
    ComponentManager & cm = GAL::getComponentManager();

    // Unregister interface implementation.
    cm.unregisterImplementation(*this->interfacePrototype, *this);

    // Unregister and free interfae.
    cm.unregisterInterface(*this->interfacePrototype);
    delete this->interfacePrototype;

    // Unregister object.
    cm.unregisterObject(*this->objectPrototype);
    delete this->objectPrototype;
  }
```

Now it comes actual interface function implementations. In this example, they just prints that the interface function was called and with what arguments but you may fill them with whatever you want to do. They are static methods with a component instance as the first argument because taking a pointer to a C++ object method and converting it to a void pointer is illegal.

```
private:
  /* Inteface functions implementations. */
  static void foo(Component * self)
  {
    std::cout << "ICustomInterface::foo()" << std::endl;
  }
  static void bar(Component * self, int argument)
  {
    std::cout << "ICustomInterface::bar(" << argument << ")" << std::endl;
  }
  static Point * getPoint(Component * self)
  {
    Point * result = new Point(15, 33);
    std::cout << "ICustomInterface::getPoint(): " << result->getString()
      << std::endl;
    return result;
  }
```

Attributes serves only for a pointer to the registered prototypes storage.

```
  /* Internal attributes. */
  Point * objectPrototype;
  ICustomInterface * interfacePrototype;
};
```

We have finished rather imaginary but significant serie of tutorials. This part may be viewed in a file `GAL/doc/Examples/custom_component.h` on the attached CD.

A.6 List of Raster Layers

The final tutorial presents possibility write simple GRASS modules using the GAL Framework in Python programming language. In this case its a module which gets list

of available raster layers in the GRASS and prints them to the standard output. Displayed constructs are almost identical to that presented in the first tutorial except there is no deallocation and the `GAL.initialize()` method accepts a list of module arguments instead of the C–like arguments of a main function due to nature of Python. It's also in a `GAL/doc/Examples/list_rasters.py` file.

```
#!/bin/env python

# Standard imports.
import sys

# GAL Framework imports.
from GAL import *

# Initialize GAL Framework.
GAL.initialize(sys.argv)

# Initialize component with asscess to GRASS rasters.
raster_layer_provider = GRASSRasterLayerProvider()
raster_layer_provider.initialize()

# Get IRasterLayerProvider interface from component manager.
cm = GAL.getComponentManager()
i_raster_layer_provider = cm.getInterface('IRasterLayerProvider')
getElements = i_raster_layer_provider.getSlot('getElements')

# Get list of available raster layers.
layers = getElements()

# Print them.
for layer in layers:
    print layer,
print

# Deinitialize component.
raster_layer_provider.finalize()

# Deinitialize GAL Framework.
GAL.finalize()
```