# DynamicModules: Programmer Guide

DocumentId:GradSoft-PR-r-01.11.2001-v1.0.0

Ruslan Shevchenko

May 8, 2003

## Contents



# 1 Introduction

**DynamicModules** is cross-platform component written in C++, designed for shared libraries dynamic loading ( `.so` for UNIX or `.dll` for Windows )

Package **DynamicModules** is designed and supported by GradSoft company and supplied in source code. Developer home webpage is http://www.gradsoft.kiev.ua/

# 2 Common information

DynamicModules package allows to organize program building as set consisting of head module and shared libraries loading while execution.

Loadable module developer just supplies the module and set of header files.

There is must be static object in the module body, inherited from `DynamicModule` class or verb|DynamicModuleTag| class.

Program, which is using shared libraries should be compiled with `[lib]DynamicModules.{a,so,lib}` library and should use `DynamicModules` class methods (load, unload, getModule) described in the library.

# 3 Trivial example

## 3.1 Components

Trivial example described here consist of followins components:

1. common header file HelloInterface.h, including DynamicModules.h header file in one's turn
2. loadable module HelloModule.cpp
3. head programm Main.cpp

## 3.2 Common Header File

This file contains data used by both library and client:

1. class HelloInterface - this is interface through which head programm will operate for its specific purpose.
2. class HelloModule - this is interface through which head programm will call to root object of the service

```
#include <GradSoft/DynamicModules.h>

/**
 * Interface for direct work of programm:
 **/
 class HelloInterface
 {
 public:
   virtual ~HelloInterface() {}
   virtual void hello() = 0;
 };

/**
 * Interface for call to root object of the service:
```

```
 **/
 class HelloModule: public GradSoft::DynamicModule
 {
 public:
  /**
   * return reference to HelloInterface instance:
   **/
   virtual HelloInterface* create(const char* name) = 0;
 };
```

### 3.3 Loadable module

This file contains realisation for HelloModule and HelloInterface:

```
#include <HelloInterface.h>
#include <iostream>

// realize direct functionality:

class Hello: public HelloInterface
{
public:

  Hello(const char* x) {}
  virtual ~Hello() {}
  virtual void hello()
  {
    std::cerr << "Hi! I'm Hello" << std::endl;
  }
};

// realize root object (or the Factory):

class Hello1Module: public HelloModule
{
public:

////
// overload next methods of DynamicModule parent class:
////

  // service name
  const char* name() const { return "Hello"; }

  // version information
```

```
  int versionMajor() const { return 1; }
  int versionMinor() const { return 0; }
  int versionSubMinor() const { return 0; }

  // author
  const char* author() const { return "Grad-Soft LTD"; }

////
// realize specific method of HelloModule:
////

  HelloInterface* create(const char* args)
  {
    return new Hello(args);
  }

};

// create module:
Hello1Module tagHelloModule;

// export object:
EXPORT_OBJECT(tagHelloModule)
```

### 3.4 Head programm

```
#include <HelloInterface.h>
#include <memory>
#include <iostream>

using namespace GradSoft;

int main()
{
 try {

  // loading Hello module from current directory
  DynamicModule& dmHello = DynamicModules::load("tagHelloModule","./Hello");

  // casting to type HelloModule in order to call HelloModule::create
  HelloModule& helloModule = dynamic_cast<HelloModule&>(dmHello);

  // using
  {
   std::auto_ptr<HelloInterface> h1 ( helloModule.create("xxx") );
```

```
   h1->hello();
  }

  // unloading
  DynamicModules::unload("tagHelloModule");

 }catch(const DynamicModules::Error& ex){

  // error notification
  std::cerr << "Error during loading DynamicModule" << std::endl;
  std::cerr << ex.what() << std::endl;
  return 1;
 }
 return 0;
}
```

### 3.5 Puting Everything Together

To obtain executable file, make following:

1. Compile `Hello.{so,dll}` library using command like this:
   (a) For UNIX with GCC compiler:
   ```
   g++ -shared -o Hello.so HelloModule.cpp
   ```
   (b) For Windows NT:
   ```
   cl Hello1Module.cpp /MD /GR /GX /D "WIN32" [...] /link -DLL /out:Hello.dll
   ```
   (Note: /MD /GR /D "WIN32" flags must be used obviously)
2. Compile Main.cpp using `libDynamicModules.{a,so}` for UNIX or `DynamicModules.lib` for Windows NT:
   (a) For Linux with GCC compiler:
   ```
   g++ Main.cpp -ldl [...] libDynamicModules.a
   ```
   (b) For Windows NT:
   ```
   cl Main.cpp /MD /GR /GX /D "WIN32" [...] /link DynamicModules.lib
   ```

Executing obtained programm, you can read from standard output:

```
Hi! I'm Hello
```

# 4 API DynamicModule

## 4.1 DynamicModule Class

DynamicModule is abstract C++ class, and certain instance of the class must be defined in the shared library.

Programmer has to overload next methods of the class:

- `const char* name() const` - returns module name.
- `const char* author() const` - returns module athor's name.
- `int versionMajor() const` - returns main version number.
- `int versionMinor() const` - returns second number version.
- `int versionSubMinor() const` -returns third number version.

Several words about version numeration: in our products we use three-digit version number with the next sense of digits:

1 - main version number. It is changed if we change programmer API incompatibly.

2 - second version number. It is changed if we change binary representation or data transmission protocol (in net case).

3 - third version number. Third number alteration should retain binary compatibility.

Note, class sizes influence binary compatibility in shared libraries case, so if you added internal variable in an interface class you should change *second version number* . Therefore we recommend to include only abstract classes into shared interface headers or to use `Pimpl` idiom.

## 4.2 Library Tag

One of C++ library modules should define global variable with module type and unique name may being constracted such as `tag<ModuleName>` or `<moduleName>Tag`.

As you could see in above examples we write then

```
EXPORT_OBJECT(tag<ModuleName>)
```

It is syntactic sugar to conceal Windows API features (in which you should use special key word to export library functionality)

One feature: the definition should be *outside* any C++ namespace.

I. e. your file with module definition should look like underlying:

```
 ... includes ..

namespace MyCompany {

....

class MyModule: public DynamicModule
{
 ...
};

} // end of namespace (MyCompany)

MyCompany::MyModule tagMyModule;
```

By the way, it makes sense to add namespaces names to module names.
I. e. it's better to do next:

```
MyCompany::MyModule tagMyCompany_MyModule;
```

### 4.3 DynamicModules Class

DynamicModules class is singleton, which provides library management.
Main methods are:

```
DynamicModules::load(const char* moduleName, const char* fname);
```

- loads `<fname>.so` module for UNIX, or `<fname>.dll` for Windows NT, and sets `moduleName` as name for the module (note, for UNIX `moduleName` must be the same as variable name having been exported via `EXPORT_OBJECT` macro).

```
DynamicModules::unload(const char* moduleName);
```

- decrements internal reference counter and unloads module, if the reference counter comes to zero.

After module unloading, reference to the module, returned by `load`, becomes invalid and reference use leads to the undefined behavior of the program.

```
DynamicModules::get(const char* moduleName);
```

- the same as `load` except for if module is loaded already reference counter doesn't change.

# 5 Error Processing

DynamicModules methods can generate exception of `DynamicModules::Error` type.

As usually , what() method returns string - error description. Concrete list of possible messages depends on operational system.

# 6 Final Notes

The library gives facility to use weakly connected C++ components model. It is more convenient than COM or XPCOM to implement plugin functionality in consequence of it's simplicity, on the other hand it doesn't make any limitations to a programmer in other component models use or implementing - you may combine instead of choosing.

By the way, one of the library application fields is choice realization between two technologies [in standard `GradSoft` services architecture you may choose underlying COM or CORBA service, just by definition one of two libraries-layers in the system].

# 7 Changes List

24-01-2002
1. trivial example:
   (a) placed in personal section;
   (b) simplified;
   (c) maked compilable;
2. some misfit in API description removed.

01-11-2001 created.