

Wrapping a complex C++ library for Eiffel

FINAL REPORT

July 1st, 2005

Semester project

Student:

Simon Reinhard

simonrei@student.ethz.ch

Supervising Assistant:

Bernd Schoeller

Supervising Professor:

Bertrand Meyer

Table of Contents

1. Abstract.....	3
2. Overview of the approach.....	3
Classes.....	3
Class members.....	3
Value conversions.....	3
Inheritance.....	4
Inheritance from wrapper classes.....	4
Multiple inheritance.....	5
Multiple inheritance from wrapper classes.....	5
Repeated inheritance.....	5
Naming.....	5
Memory management.....	5
Limitations.....	6
3. Automated wrapping.....	6
Generated code.....	6
Support library.....	7
Customization.....	7
Limitations.....	7
4. Ogre3d library for Eiffel.....	7
Manual director classes.....	7
Singletons.....	8
Example.....	8
5. References.....	8

1. ABSTRACT

The goal of this project was to wrap a C++ library for Eiffel. The library is assumed to be already designed in an object oriented style, so that no modifications to the interface are necessary and the C++ classes can be exposed directly to Eiffel. The first section is an overview of the approach that was used to create such a wrapper, in the second section a tool is presented which largely automates the tedious process of writing the needed code and finally, some results for Ogre3d, an object oriented C++ library for 3d graphics, are shown.

2. OVERVIEW OF THE APPROACH

In this section i will present an approach for wrapping C++ classes to allow them to be used in Eiffel. This approach is largely inspired by the wrapper classes in SWIG[1], also known as proxy classes or shadow classes. However, existing SWIG modules do not support some features of Eiffel like multiple inheritance for which i had to find specific solutions.

This approach only targets a subset of all C++ features which i believe to be particularly important for object oriented libraries. Non class members are ignored totally.

In the following sections i will motivate design decisions and point out problems that arise when wrapping C++ classes.

Classes

For every C++ class that should be accessible from Eiffel, a corresponding wrapper class is created. These wrapper classes have a managed pointer to an instance of the wrapped class and features which allow the elements of this class to be accessed.

Instances of wrapper classes can be created either by supplying a pointer to a C++ object, or by using one of the constructors, which are mapped to creation features in the wrapper class.

Class members

For every member function with public accessibility in a wrapped C++ class, there is a wrapper feature in the Eiffel wrapper class. Additionally there is a C interface function, which implements the call to the member function with given arguments and target. This is necessary because ISEs Eiffel to C++ external interface is not complete - passing arguments of complex types as references or by value for example is not supported - and it's also more convenient because the code generated by the ISE Eiffel compiler can be compiled as C and only needs to include a simple header file containing headers of the interface functions instead of a possibly quite complex C++ header file. This also reduces compile time and the potential for errors due to naming conflicts with Eiffel specific headers.

Member variables can be treated like member functions, except that the interface function does not call a function but reads a variable. One can also implement setter features by creating an interface function which writes the variable.

Constructors also have an interface function which calls the constructor and returns a pointer to the created object. They are wrapped by creation features which store the returned value in the pointer of the wrapper class.

Static member functions are treated the same way as member functions, but they don't need a pointer to the target object. Therefore they can be used by inheriting from a wrapper class even when a pointer to an external object is not properly initialized.

Value conversions

To enable calls from Eiffel to C++, the arguments have to be converted from Eiffel objects to a form C++ understands and the return value has to be converted back to an Eiffel object.

This conversion is done in two steps: The first step happens in the Eiffel wrapper feature where by default basic types like INTEGER are left as they are and references of wrapper class types are converted to pointers to C++ objects or default pointers, if a reference is void. The second step is performed in the interface function which converts pointers of unknown type to the expected type and dereferences pointers if they are to be passed as references or values to the C++ member function.

For return values, this process is reversed - the interface function converts the returned value to a basic type and the Eiffel wrapper feature creates instances of wrapper classes around returned pointers.

This whole process may also be customized to allow better integration. For example it is more convenient to pass instances of the Eiffel standard class STRING to a wrapper feature than to create an instance of a wrapper class for whatever string class the C++ member function expects.

Inheritance

The inheritance of the C++ classes is mapped to the wrapper classes by the following rule:

In a wrapper class for A, the wrapper class for B appears in the inheritance clause if and only if A inherits directly from B.

This allows polymorphic attachment of wrapper classes just like regular Eiffel classes.

If a virtual member function is overridden, the corresponding wrapper feature in the wrapper class is redefined. Because Eiffel requires all redefined features to be declared in the inheritance clause, we also need a rule for a wrapper feature to appear in the inheritance clause for a base class:

In a wrapper class for A, the wrapper feature for f appears in the redefine part of the inheritance clause for the base class B if and only if the wrapped member function is overridden in A and B is a descendant of the class where f is originally defined.

Pointers returned by C++ may be polymorphic, so Eiffel has no knowledge about the dynamic type of the objects returned by C++, without additional overhead. Instances of wrapper classes may therefore not have the dynamic type corresponding to the C++ object they are representing, but a more restrictive one. This may cause an assignment attempt not to succeed even though the dynamic type of the C++ object would allow it. Possible solutions to this problem are to check the dynamic type of all objects returned by C++ and creating an appropriate instance, which would cause a serious overhead, or to customize the assignment attempt for wrapper classes.

Inheritance from wrapper classes

Wrapper classes can also be used as ancestors for Eiffel classes. Calls from Eiffel to a feature of the wrapper class are then routed to the correct implementation with dynamic binding. However dispatching calls from C++ to the correct implementation requires an additional mechanism, which can be implemented by creating new C++ classes which inherit from the wrapped classes and implement all virtual member functions by a call to the corresponding Eiffel feature. As introduced in SWIG[1], I will use the term director class to refer to such classes.

When a class inherits from a wrapper class it has to call a special initialization feature, which creates an instance of a director class instead of a normal instance of the wrapped class. Instances of such classes can then be passed to wrapper features where an instance of the inherited wrapper class is expected and subsequent calls from C++ to virtual member functions are correctly routed to the Eiffel implementation. There is still one problem left though: If a feature is not redefined or performs a call to its precursor, this call will be routed to the director class which again calls the Eiffel feature instead of the C++ precursor, potentially creating an infinite loop. To break this loop, a rule is needed to be able to tell when a call to an external function is to be routed to the precursor in C++ instead of a call to an Eiffel feature:

A call to an interface function is routed to the C++ precursor if and only if the call has passed a wrapper feature with an instance of a wrapper class pointing to an instance of a director class as target.

The reasoning behind this rule is, that a wrapper feature of such an object represents the C++ implementation of the wrapped class. On the other hand all calls not meeting this condition may still have to end up in an Eiffel implementation.

Since there is not enough information available in the C interface function for this decision, an additional boolean argument is added to all interface functions, which is true whenever the call has to be routed to the C++ precursor. Moreover, wrapper classes have a boolean attribute which they pass along with all calls to their interface functions. This attribute is set to true in the initialization feature described above and set to false in all other initialization features.

Multiple inheritance

Multiple inheritance is generally treated just like single inheritance. However there are some issues which are only coming up if multiple inheritance is used.

C++ allows pointers to the same object to be different if they are cast to void pointers, in case the static type of the pointer is not the same. This is generally not the case for the usual implementation of single inheritance, but almost surely happens if multiple inheritance is used. Since void pointers are needed to pass values from C++ to Eiffel this has to be taken into account.

To support this, there are additional interface functions which cast pointers to derived classes to pointers of their base classes. Wrapper classes have a feature for every ancestor including itself which returns a pointer to the wrapped C++ object of the desired type, using the aforementioned interface functions. These features are then used, whenever access to a pointer to the wrapped object is needed.

Multiple inheritance from wrapper classes

Multiple inheritance involving wrapper classes should be as natural as possible. If all wrapper classes inherit from a common ancestor, some care has to be taken not to cause unnecessarily complex inheritance clauses or - even worse - unresolvable conflicts. The latter occurs if a pointer to the external object is a feature in the common ancestor. A class inheriting from multiple wrapper classes should also have multiple features for pointing to the corresponding external objects, but there is only a single one present. To solve this, pointers are not defined in a common ancestor but only defined in wrapper classes at the top of the inheritance hierarchy.

Repeated inheritance

Unfortunately, Eiffel and C++ have different semantics for repeated inheritance [see 2 and 9]. If a class inherits repeatedly from a base class, the default semantic in C++ is, that this class has multiple instances of all members defined in the repeatedly inherited class, while Eiffel always has only a single instance of a feature for a class. C++ has a construct called virtual inheritance, which has the same semantics as Eiffel, but this feature is rarely used, since most C++ compilers have a rather inefficient implementation.

As an effect of this mismatch, it seems impossible to accurately represent the C++ inheritance structure in Eiffel wrapper classes in the presence of non-virtual repeated inheritance.

Naming

Eiffel has very clear naming conventions, which are almost universally used, whereas C++ libraries usually apply the so called camel style or variations thereof to separate words. To make the wrapper classes more consistent with other Eiffel software, names in the C++ library are converted to the Eiffel style. The basic approach is to detect when a new word starts, for example by capitalization and special characters, and insert an underscore between the words.

C++ also supports overloading for functions, where more than one function can have the same name but different signatures. To disambiguate the features in Eiffel, which does not support overloading, a unique postfix is added to the feature names. Eiffel for .NET[10] for example uses the function's signature to generate such a postfix.

Memory management

Since the garbage collector in Eiffel can't handle external objects, they have to be disposed by other mechanisms. To make sure an object is only disposed once, pointers with a notion of ownership are used. This procedure is quite common in C++ development and also used in other wrapper approaches like the

ones in SWIG[1] and EWG[4]. An external object is disposed, if the pointer that owns this object is collected. If a pointer is not owning an object, this pointer is defined to be shared. If all pointers are shared, the object is assumed to be disposed by some mechanism in the external library.

By default, pointers retrieved from a C++ constructor are assumed to be unshared. All other pointers are assumed to be shared.

Since pointers are responsible for disposing external objects, they have to call the destructors of the class. Because C++ destructors cannot be called without knowledge of the object's type, there is a pointer class for every wrapped C++ class. Just like the wrapper classes, these pointer classes have the same inheritance structure as the C++ classes. Every wrapper class has a pointer of corresponding type. These pointers are covariantly redefined to the corresponding type under inheritance and merged together under multiple inheritance.

An instance of a director class has a reference to its associated Eiffel object, and this Eiffel object has a reference back to the director instance, forming a cyclic dependency. However, the garbage collector sees the reference of the director class instance as an external reference and thus adds it to the root set. This may lead to a wrapper-director pair never being collected, even though there is no reference left, pointing to any of them. This issue could be solved by replacing the reference of the director class by some kind of weak reference, if the Eiffel object is responsible for disposing the external object. This approach is also used in some modules for SWIG[1]. Unfortunately Eiffel does not currently support weak pointers, even though some proposals have been made[8].

Limitations

This approach does not cover throwing and handling exceptions across language borders. In order to support this, one would need additional code in the interface functions to catch C++ exceptions and translate them to Eiffel exceptions. Also wrapper features would need the possibility to throw C++ exceptions when an Eiffel exception is detected. The same has to be done for director classes.

I did not investigate the potential of contracts in wrapper classes.

Eiffel, similar to C++ supports the definition of standard arithmetic operations for a class. It might be possible to use wrappers for the C++ operators to define the corresponding Eiffel operators.

3. AUTOMATED WRAPPING

The amount of code needed for wrapping even moderately sized libraries is rather large. Moreover, this code is very repetitive, which makes it unpractical to write manually. Therefore, I looked for possibilities to automatically generate wrapper code for a library.

Unfortunately, existing wrapper generators for Eiffel either only support C, like EWG[4] and a seemingly abandoned Eiffel module for an old version of SWIG[7], or they do not support enough features, like the Legacy++ tool shipped with ISE Eiffel[12].

The most difficult part of the process is parsing and analyzing the C++ header files containing the interface description for the classes to be wrapped. I found two tools which were able to do these steps: GCC-XML [11] is a modified version of gcc, which only parses its input and outputs the relevant parts of the internal parse tree in an xml format. SWIG[1] is a wrapper generator for different higher level languages, ranging from scripting languages like Python over statically typed languages like Java and even to functional languages like Ocaml. SWIG has its own C++ parser and an enhanced preprocessor, specifically built for the needs of a wrapper generator. Moreover, the semantic analyzer is able to handle incomplete type information, making it suitable for only wrapping interesting parts of a library.

Because of the strong orientation towards the target domain, I decided to write a SWIG module for Eiffel, which generates the needed wrapper code automatically. In the following sections, I will give an overview of this module's functionality.

Generated code

For every class in the interface definition passed to SWIG, an Eiffel wrapper class is generated in a cluster named after the module name of the SWIG interface file. Additionally, a C++ file and a corresponding C

header file containing the functions needed by the wrapper classes to interface with the C++ library is generated.

Unknown or incomplete types are handled with a special wrapper class, which only has a pointer to an external object. This is convenient to only wrap interesting parts of a library.

Support library

As every SWIG module, the Eiffel module has some Library files which define the handling for basic types and standard types like strings.

Additionally there are some Eiffel classes, which are needed to compile the generated wrapper classes.

Customization

To select which classes are wrapped, so called interface files are generated, which normally include C++ header files and have additional SWIG preprocessor statements to customize certain aspects of the wrapper generation. This is extensively documented in the SWIG Manual[1], so I will only mention options specific to the Eiffel module here.

One special feature of SWIG are typemaps, which allow the code generation for argument conversions to be customized. This construct associates code fragments with C++ types by a matching algorithm. The Eiffel module supports typemaps for argument and return value conversions in the interface functions and in the wrapper features.

Limitations

Director classes are not generated automatically but the generated code supports their manual addition. The work required to add this functionality to the module could not have been done during the time of this project without affecting the implementation quality. To create such classes manually, the following is needed: A C++ director class has to be inserted into the generated code. This class inherits from the wrapped class and from a base class for directors. The constructor takes a reference to an Eiffel object, which it passes to the director base class constructor, a number of pointers to callback features and additional arguments which are passed to the base class constructor. All virtual member functions of the base class are implemented by calls to the corresponding callback feature, or, if a special field in the director base class is set, by calling the precursor in the base class. Additionally an interface function for the directors constructor is needed, and an Eiffel class, which inherits from the corresponding wrapper class, and implements the callback features and has a feature for constructing the external director class. For an example of this, consult the Ogre3d library presented in the next chapter.

While multiple inheritance is supported, the use of repeated inheritance in the C++ library causes the module to generate invalid Eiffel code. This is unfortunate but a correct treatment of repeated inheritance is generally not possible because of the mismatch of semantics described earlier. Manual changes to the Eiffel class can probably fix the compilation errors and make it work, but still, not the whole functionality is exposed.

4. OGRE3D LIBRARY FOR EIFFEL

The library is currently only consisting of a very minimal amount of classes needed to create simple applications. A more complete library should be relatively easy to create with the presented tool though. The library is partially inspired by PyOgre, a project to wrap the Ogre3d library for the Python scripting language, hosted at the Ogre3d homepage[3].

Manual director classes

To support callbacks from C++, so called director classes are implemented for selected classes. These are currently only the FrameListener class, which enables actions being executed at frame rate, and the input listener classes, to enable interaction.

Singletons

Ogre makes extensive use of the singleton design pattern[13]. All singleton classes inherit from a class template, instantiated with their own type as parameter. This class has a static member function to access the Singleton instance.

In the interface files, this template is instantiated for every wrapped singleton class, and is named after the scheme "CLASS_NAME_SINGLETON". The wrapper feature of the access function is also renamed to the scheme "class_name_singleton". This feature can then be called to get access to the singleton instance.

Example

The example demonstrates how to setup Ogre3d, basic rendering of an object and some interaction through the standard input devices.

5. REFERENCES

- [1] *Simplified Wrapper Interface Generator*, Online at: <http://www.swig.org/>, consulted in June 2005.
- [2] Bertrand Meyer: *Object-Oriented Software Construction, 2nd edition*, Prentice Hall, 1997.
- [3] OGRE 3D : *Open source graphics engine*; Online at: <http://www.ogre3d.org>, consulted in March 2005.
- [4] *Eiffel Wrapper Generator*, Online at <http://ewg.sourceforge.net>, consulted in March 2005.
- [5] *Cecil*, Online at <http://docs.eiffel.com/eiffelstudio/technologies/cecil/index.html>, consulted in March 2005.
- [6] David Abrahams: *Building Hybrid Systems with Boost.Python*, Online at <http://www.boost-consulting.com/writing/bpl.html>, consulted in June 2005.
- [7] Alex Cozzi: *SWIGEIFFEL*, Online at <http://efsa.sourceforge.net/archive/cozzi/swigeiffel.htm>, consulted in June 2005.
- [8] Frederic Merizen, Olivier Zendra and Dominique Colnet: *Designing efficient and safe weak references in Eiffel with parametric types*, 2003, Online at <http://www.loria.fr/~zendra/publications/rr2003b.pdf>.
- [9] *ANSI/ISO C++ Draft Standards*, 1996, Online at <http://www.csci.csusb.edu/dick/c++std/cd2/index.html>.
- [10] Raphael Simon, Emmanuel Stapf and Bertrand Meyer: *Full Eiffel on the .NET Framework*, 2002, Online at http://msdn.microsoft.com/library/en-us/dndotnet/html/pdc_eiffel.asp.
- [11] *GCC-XML*, Online at <http://www.gccxml.org/HTML/Index.html>.
- [12] Eiffel Software, Online at <http://www.eiffel.com>.
- [13] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: *Design Patterns*, Addison-Wesley, 1995.