Compound object model for scalable system development in C++

Stepan Orlov\(^1\) and Natalia Melnikova\(^{1,2}\)

\(^1\) Peter the Great St. Petersburg Polytechnic University, St. Petersburg, Russian Federation
majorsteve@mail.ru

\(^2\) ITMO University, St. Petersburg, Russian Federation
naunat@mail.ru

Abstract
We present an object model that is the ground for several software products written by our team in C++ and used by our customers for over 10 years. The object model has allowed us to design software packages of many components and achieve high level of code reuse due to specific features of the model. Besides the object model, we also present several common infrastructure components that could be useful for the development of any general-purpose software system in C++.

Keywords: C++, component based system, object model, object-oriented programming, framework

1 Introduction

Developing a large software system in C++ is tricky. One of the reasons for that is that the language itself does not provide any recommended way to organize the code into modules and packages (like, e.g., Python does). Developers are not limited in how to do it, and each developer team seems to go their own way.

A large software system consists of many modules. A module is usually a static library or a dynamic library, but further we will only consider modules as dynamic libraries (or shared objects in Unix operating systems). System functionality is located in modules, and there are dependencies between modules, which means that one module can need the functionality of one or more other modules.

Technically, there are several ways to organize the access to the functionality of a module. One of traditional ways to do that is to export symbols from the module. Then the module can be linked with the executable, which in turn can be done early or late. In the former case, the module is just linked with the executable during the build process. In the latter case, the module is loaded after executable starts up, then exported symbols are resolved using API (application programming interface) provided by the operating system.
Using the approach mentioned above requires careful separation of the functionality among modules. Typically, it is better to minimize the number of inter-module dependencies, and to have a tree-like module dependency graph.

There is, however, a different approach to accessing module functionality, and it gives some benefits. The idea is to have all modules early linked to a small module providing core functionality for registering objects in a factory. Each module encapsulates its functionality in classes that implement interfaces. A class can be registered in a factory at module loading time. Once a module is loaded, no further symbol resolution is necessary: instances of registered classes can be created using factories, and the invocation of class methods can rely on virtual tables rather than exported symbols. This approach allows to make modules independent from each other (the exception is the core module used by all others) from the linker perspective. Figure 1 shows the dependencies between modules for the two approaches described above.

![Figure 1: Module dependencies in case of explicit usage of exported functionality (left) and in case of relying on the factory provided by our object model (right).](image)

Next sections discuss the design of the core library and several general purpose components built on top of the presented object model; an example of project using this technology is presented; the proposed framework is compared to others.

## 2 Compound object model

### 2.1 Compounds, components, and interfaces

The idea of the compound object model has emerged as an alternative way to provide high level of code reuse. In C++ object-oriented programming, traditional ways to reuse code are based on composition, aggregation, inheritance, and less evident associations, often using templates. This all works well within one module; however, applying composition and inheritance across module boundaries implies additional requirements and limitations for the entire structure of the system. For example, if a class from module A inherits a non-interface class from module B, then B must export the base class, and A must be linked against B. This is a limitation, because we would prefer modules A and B to be independent from each other. Composition of class instances also has some issues, but they are not as critical as for the inheritance.

A compound object consists of one or more components connected with each other in a tree-like graph. The root of the tree is called the primary object; all other components are tear-offs.
A component is an instance of a C++ class that follows several conventions. The class must inherit one of two classes (one for primary, one for tear-off), and it must declare a numeric component identifier (unique for each component within the entire system). In addition, components typically implement interfaces.

A compound object supports a set of interfaces. An interface is a C++ class whose all methods are either pure virtual or inline, so they can never be exported symbols. Interfaces cannot have static fields, and there is also a natural limitation on field types — modules export no symbols! The design of the compound object model specifies additional requirements on an interface: it must virtually inherit a base class (same for all interfaces), and it must declare a numeric interface identifier (unique for each interface within the entire system).

An interface can be compound-wide or component-wide. A compound object supports a compound-wide interface if one of its component classes inherits the interface. The object model has functionality to retrieve a pointer to supported compound-wide interface from a pointer to any other interface of the same compound. Besides, there is a way to override a compound-wide interface by implementing it in a child tear-off (see Interface5 in Fig. 2). In further text, we omit “compound-wide” for the sake of conciseness, so the term interface means compound-wide interface.

Similarly, a component supports a component-wide interface if it inherits the interface. Our object model provides functionality to retrieve a pointer to the supported component-wide interface from a pointer to any other interface inherited by the same component.

Notice that the interface casting functionality does not rely on RTTI (run-time type identification) and hence is free of its limitations on some systems (e.g., the `dynamic_cast` operator may unexpectedly fail on some systems if the class inheritance diagram involves classes from different modules).

Importantly, the components of a compound object can be instances of classes from different modules.

Figure 2 illustrates the concept of a compound object and its interfaces.

### 2.2 Object lifetime, smart pointers, and interface casting

Tear-off components can have different lifetime. Some of them can be preloaded (created and destroyed together with the primary object) and some can be temporary, created on demand, when an interface pointer is requested.

Components implement reference counting. A reference counter is a non-negative integer number, incremented each time by the user to indicate that the component is in use, and decremented by the user to indicate that the component is not needed anymore. As soon as a reference counter reaches zero, the component is deleted from memory.

Preloaded tear-offs share the reference counter with the primary component, while temporary tear-offs implement their own reference counters in addition to incrementing and decrementing the reference counter of the parent component.

The reference counting is managed by smart pointers [1]. Smart pointer is a template class parametrized by interface type. It can be initialized by a raw interface pointer (because the reference counter is not outside the managed component), and in that sense is similar to `boost::intrusive_ptr` [2].

Additional smart pointer functionality is the interface pointer casting. Thus, when a specific interface pointer of a compound object is necessary, it can be obtained by initializing the smart pointer of appropriate type by a pointer to any interface of the compound object. When this is done, the underlying calls find the necessary component, or create it if is not created yet, and
even load the module implementing the component if necessary.

Finally, the object model supports weak pointers. A weak pointer does not increment component reference counter, therefore letting the pointed component to be deleted. Once that happens, the weak pointer becomes aware of the component deletion and sets the pointer value to zero. Weak pointers are used to avoid dangling pointers.

2.3 Object model configuration

Our object model provides functionality to create components and compounds in memory (see next subsection). But before it can be used, the configuration has to be specified. Object model configuration is a text file that lists known modules, components in each module, and interfaces supported by each component. Modules are identified by their name; components and interfaces are identified by their numeric identifiers. This information lets the object factory know which module needs to be loaded when a specific component instance needs to be created. In addition, the configuration file contains the compound object configuration. For each primary object, its tree of tear-offs is specified. This makes the object model know which components constitute a compound object.

2.4 Object factory

Once the object model configuration is loaded, objects can be created using the object factory. The factory can create individual components or entire compound objects. Notice that the configuration information allows to load modules on demand, as soon as the first component from a specific module has been requested.

When a module is loaded, it performs the initialization of its static variables. We use this initialization stage to register all component classes of the module in the class registry. After initialization, the factory is able to create components. So when the factory is asked to create a component specified by its numeric identifier, it takes the following steps. First, the component identifier is looked up in the class registry. If the identifier is not found there, it is looked up in the configuration file. If the identifier is not found in the configuration file, an error is generated. Otherwise, the module implementing the component is loaded, and the identifier is looked up in the class registry once again. Finally, a component instance is created.

The object factory can also create compound objects specified by the numeric identifier of the compound primary object. To create a compound object, the factory creates its primary object component first, and then creates and binds all preloaded tear-off components. The necessary information is read from the compound object configuration. Binding of a tear-off component is the process of its association with the parent component (remember that the compound object is a tree-like graph). It’s not only the factory who can bind tear-offs — they can be bound manually when creating necessary compounds on the fly.

2.5 Globally accessible memory area

Global variables are typically discouraged by programming guides because there are several architectural and technical downsides related to them. At the same time, sometimes it’s necessary to have something similar to global variables in a software system.

Our object model provides an API to manage globally accessible objects. The idea is to have a pool of objects (one per execution thread or shared among several threads) and to provide functions that register an object in the pool, find object in the pool, and unregister a previously registered object.
An object is identified by two integer numbers: interface identifier (see sec. 2.1) and so called service identifier, whose meaning is defined by the user. The pool of objects is a dictionary where keys (pairs of interface and service identifiers) are mapped to stacks of objects. Each registration of an object pushes the object on top of the stack; each lookup returns object at the top of the stack; each unregistration removes object from the stack. Using stacks allows objects registered with the same key to shadow each other, which is quite reasonable in many cases.

3 Messaging system

The concept of a notification message is extremely fruitful in software development because it helps to decouple software components from each other and make them more abstract and usable in wide ranges of application.

For example, the Qt framework [9] provides the mechanism of signals and slots that can be connected to each other in order to perform an action in response to an event (here event and action do not refer to the Qt’s concepts of the same names). The signals and slots mechanism is very powerful; however, it relies on the Qt meta-object compiler that parses C++ source code and generates auxiliary source code. In our non-Qt applications, we use a simple messaging system that does not require a meta-object compiler, but in exchange it only supports fixed signatures for message receivers and hence compile-time type checking cannot be done.

The messaging system consists of three interfaces, which are message receiver, message target, and message delivery. Message receiver is a component-wide interface that allows to expose message handler methods. The methods are regular methods of the class that implements a component of a compound object. Message target is an interface that allows a compound object to receive messages. Once received, a message is dispatched to all message receivers of the compound that are interested in messages of the corresponding type. Message delivery is an interface that allows to send messages and to subscribe message targets for certain messages. Once a message is sent to a message delivery, the delivery dispatches the message to all targets subscribed for messages of the corresponding type. Importantly, the message target and message delivery interfaces only need to be implemented once. Tear-offs implementing these interfaces can be attached to any compound objects in the compound object configuration. On the other hand, the message receiver interface has to be implemented by each component that has message handlers. But the implementation is trivial and is done using helper macros.

A global message delivery object is one of the applications of the globally accessible objects described in sec. 2.5: sometimes it is more natural to have a global message delivery than to find it somewhere else.

4 Document data as tree of properties

A running program operates on data in memory; it is important for users to make these data persistent. In other words, there is a concept of a document — the data that can be written to and read from a stream (e.g., a file) and that represents all editable objects that user works with using a software system. The organization of document data in memory has to be correlated with its organization in stream, and certain decisions need to be made. Often, document data can be represented by a tree-like data structure; to represent a tree-like document data structure in a stream, one can use formats like XML and JSON.
To implement the tree-like data organization in memory, we introduce the following three interfaces. The property set interface represents a dictionary of properties. Within a property set, elements are identified by their names (strings). In addition, a property can be identified by a unique number. The reason for introducing numerical identifiers will be explained below, in sec. 8. The property interface represents an element of a property set. A property is the placeholder for a value or an array of values. Property values can be of different types, including numbers, strings, property sets, objects, and object references. Property is aware of its parent property set, so it only can be nested in at most one property set; it is possible to walk from any property up to the root property set of the tree. Finally, property set binder is a component-wide interface that allows a component to expose its properties.

The implementation of the property set interface is bulky, but one standard implementation tear-off is enough. When the tear-off is bound to a compound object, it collects properties from all components using the property set binder interface. The binder interface has to be implemented by each component, but the it is lightweight and is trivially implemented.

Using the above interfaces allows to represent all document data in a tree-like structure, with leaves of the tree being values of simple types. In certain cases, a “black box data” object can be a leaf of the tree of properties.

5 Persistent storage

In-memory document data cannot be serialized directly to a stream, such as a file. One of the reasons for that is that object references stored in memory are essentially pointers, i.e., addresses of object locations in memory. There are other serialization issues, such as creation of an object instance with known type identifier.

To address the reference serialization problem, the concept of a cached identifier is introduced. The idea is to identify each referenced object instance by an integer number and to store the identifier as the value of the object reference. There is a convenience class for representing object references. An instance of this class can be turned into a pointer and into an identifier. The instance stores just one 32-bit value treated as pointer if it is even (all pointers to objects are even due to alignment) and as identifier if it is odd (we force identifiers to be odd). The conversion from pointer to identifier is done by requesting it from the object pointed to; the conversion from identifier to pointer is done by requesting it from object collection — a globally available dictionary that maps identifiers to pointers. Notice that the conversion from identifier to pointer is only possible after all objects have been read from the data stream.

Other serialization issues have been solved, too; however, their description is beyond the scope of this paper. For example, creation of an object instance by its type identifier is performed in a natural way by using the object factory (see sec. 2.4).

Object serialization is implemented using the object streamer interface. The interface declares methods for reading and writing object data to/from a stream. Importantly, there is usually no need for developer to care about the implementation of this interface: provided all serializable object data is exposed through its properties of simple types, standard serializer can handle it, so the serialization goes “for free”. The object streamer interface has to be implemented only for “black box data” objects (see sec. 4), which is not common.
6 Other infrastructural components

Among other common infrastructural components we would like to mention those that are important for scripting. When we expose document data as a tree of properties, we already make the outer world aware of the design of objects inside our system. But object oriented programming encourages the use of methods — functions bound to object data. In some languages, such as JavaScript, any property can be a method just because the type of a value can be function (functions are first-class citizens). However, in our design, properties are considered as data chunks, and if we need methods, we have to add them in a way independent from properties. This is implemented much like properties: There is a method launcher interface that allows to request information about all methods that the compound object supports, and to invoke any of these methods. It relies on the method launcher binder component-wide interface exposing actual methods and implemented by the components. One tear-off implementation of method launcher is practically sufficient and is configured for use by all components with methods.

To further expose the design of our system to the outer world, we associate some compound object types with type names. These type names can be used to create instances of corresponding types. The type names are nested in namespace trees, so, for example, there can be a type name like mech.geom.point (mech and geom are namespace names). Instances of named types can be created in a scripting environment.

Next, we propose to consider an embedded language engine as the “outer world”. We have experience of embedding JavaScript, particularly the V8 implementation [10]. In the world of JavaScript, a property set is (almost) an object, and methods are functions that are properties of object prototype. Namespace tree for named types is a set of nested objects, and the named types are functions (leaves of namespace tree) acting as object constructors. Exposing objects to JavaScript has the downside that we can no longer control object lifetime explicitly (in JavaScript, objects are subjects to garbage collection that takes place from time to time), but in exchange, we get powerful scripting language to manipulate underlying objects. For example, scripting language can be used for the journalization of user actions. Such journals are extremely useful because users know what they actually do with their documents; moreover, journals can be edited and replayed, which can save user time when an error in document is found and fixed. Advanced users are able to apply scripting to prepare document data more efficiently.

The generation of journal commands can be implemented by GUI (Graphical User Interface) components, so the user does not have to care about syntax or other details of the embedded language. In this sense, a journal of user actions comes “for free” from the user perspective.

7 System configuration tool

It is difficult to create and maintain the object model configuration file (see sec. 2.3) manually, by hand. A special tool for efficient management of the object configuration has been developed. First of all, the tool provides commands to load and unload modules. When a module is loaded, the types of its component objects are registered in the class registry (see sec. 2.4). These types are tracked and associated with the module loaded. Interfaces implemented by each component are inspected and collected. This functionality allows to generate the first part of the configuration file, covering everything except the compound configuration.

Types can be filtered and grouped, and the groups can be connected by links. This allows to generate compound configuration. The crucial point here is the filtering, because it determines how flexible the tool is. For example, to implement persistent storage through property sets
for all objects, one can first select into one group all primary components whose compound objects currently support property set interface and do not support object streamer interface; then into the other group select standard property set serializer; then link the second group to the first group. This will configure standard serialization of all property sets.

The configuration can be organized as a set of levels in order to easily manage multiple configurations, e.g., one for console front end, and another one for GUI front-end.

8 User documentation generation system

This section gives an example illustrating the profits of our approach to software design. Creating good user documentation is a challenge. Where to start from? How to organize documentation pages? How to make sure that we did not forget to document something? These issues are addressed by our user documentation generator.

User documentation is typically structured as a tree of nodes; each node is one documentation page. The purpose of help generation engine is to generate this tree according to some rules and to document automatically as much as possible. The tool should also be able to add any manually added documentation content and assemble the file with user documentation.

Before running help generator, the developer configures it by specifying certain plugins. A plugin is responsible for certain part of documentation. Each plugin can be parametrized by some parameters specific to it. Since user can see named types, objects, properties, and methods, those all should be documented. In addition, it’s good to document other things, including modules that supply compound objects, selected interfaces, module version history and others. The plugins plug into each other and make a hierarchy.

Each plugin is responsible for invoking underlying plugins and feeding them with the required input data and providing them with a context that determines, in particular, the position of current node in the documentation tree. Given all necessary inputs, a plugin starts generating documentation in the current page or creates nested pages. Within each page, the information is structured into blocks. A plugin writes its output into certain page blocks.

Importantly, after the documentation generator finishes, in addition to the documentation itself we get one empty file per documentation page. The files reside in subdirectories of corresponding modules and their names are easily recognized by the developer. These files are intended for adding manual page content. Developer can be sure that once all these files are filled with additional manual content, the documentation is complete in terms of current help generator plugin configuration. Nothing will be lost, and rebuilding the documentation once again will produce the final user documentation. Notice that for the documentation generator to work, it’s important to identify object properties by unique numbers (see sec. 4). Technically, we use the Doxygen system [5] (originally intended for generating source code documentation for developers) to format documentation pages.

9 Projects developed using the proposed framework

There is a number of software products based on the proposed object model and many general-purpose infrastructural components, including our own components, the Qt library, and other third-party libraries. However, all of them are in-house software, owned by industrial companies and not publicly available. We can mention one of them, which is the system for the modeling of dynamics of continuously variable transmission (CVT) [8]. The system consists of 70 modules; its source code size, in lines, is about 400000. There is a back-end providing the modeling
functionality and two front ends. The console front end allows user to enter commands in a simple interpreted language and in JavaScript; it is used to run simulation on a remote server. The GUI front end allows user to interactively prepare simulations, run them, and analyze results using plots, spectrograms, 2D and 3D animations. User actions are journalized as JavaScript code; the journal can be edited and replayed. The system has user documentation created with our tool described in Section 8. Figure 3 shows two screenshots of the application main window and a page from the documentation. Besides CVT model, we currently work on another software code in the proposed framework. The software estimates stability of earthen dikes using real-time sensor input (more info on the mathematical model can be found in [6]).

Figure 3: Main window (left) and documentation page (right) of CVT modeling system

10 Comparison with other systems

The proposed compound object model and infrastructural components built on top of it have been developed because we did not find any existing suitable open source system with suitable license in year 2001, when we started our framework. Most noticeably, the Qt library is the alternative to consider first of all; however, it was not freely available on Windows till at least 2005, and the LGPL license for Qt has appeared in 2009. If we started our project now, it would probably be dependent on Qt. There are other systems that might be considered as possible alternatives, but they are not really suitable enough for our use case, and they do not cover all aspects we need. These alternatives are COM (Component Object Model [4]) and CORBA (Common Object Request Broker Architecture [3]) — they suggest their own object models; however, these systems are designed for achieving interoperability of components developed in different languages, and probably communicating via network. Therefore, using them would mean considerable additional overhead, e.g., due to having to perform method parameter marshaling. The MFC (Microsoft Foundation Classes) library [7] can also be mentioned due to its facilities supporting dynamic instance creation and messaging. However, it is not portable and it does not give an efficient solution for object model. Table 1 summarizes the comparison of the proposed framework with other systems.

From this table one can see that the concept of compound object is not supported in any system, designed specifically for C++ development, except ours, or we failed to find such a system. We also have to notice that the architecture of Qt plugins that can be thought of as an alternative for our approach, has quite different architecture and cannot be compared to our system in a straightforward way.
Table 1: Comparison of the proposed object model against object models from other systems

<table>
<thead>
<tr>
<th>System</th>
<th>For C++?</th>
<th>Fast enough?</th>
<th>Has compound objects?</th>
<th>Portable?</th>
</tr>
</thead>
<tbody>
<tr>
<td>proposed system</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>Windows, Linux</td>
</tr>
<tr>
<td>Qt</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>MFC</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>Windows only</td>
</tr>
<tr>
<td>COM</td>
<td>no</td>
<td>no</td>
<td>potentially yes</td>
<td>partially</td>
</tr>
<tr>
<td>CORBA</td>
<td>no</td>
<td>no</td>
<td>potentially yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

Smart pointers are implemented in many systems, including Qt, boost [2], and others; since 2011, they are part of new C++ standard. In all of mentioned cases, however, a smart pointer is different from our implementation because it is not responsible for interface casting. Hence the creation of tear-off objects and the loading of modules on demand is also not handled by these smart pointers.

11 Conclusions and future work

In this paper, we have briefly described the framework that we have been using in the development of several in-house software projects. The framework is based on the compound object model that has many attractive features for developers of scalable software consisting of many modules. The framework solves common problems of representing document data as tree of properties, document serialization, and the automatic generation of user documentation.

Our future plans include refactoring of the source code of the object model and many infrastructural components and making it open source.

References


