Virtual Component

A Design Pattern for Memory-Constrained Embedded Applications

Angelo Corsaro, Douglas C. Schmidt, Raymond Klefstad, Carlos O'Ryan
{corsaro,schmidt,klefstad,coryan}@ece.uci.edu
Electrical and Computer Engineering Department
University of California, Irvine, CA 92697, USA

Abstract

The proliferation of embedded systems and handheld devices with limited memory is forcing middleware and application developers to deal with memory consumption as a design constraint [1]. This paper uses the POSA *format [2] to present the Virtual Component compound pattern [3] that helps to reduce the memory footprint of middleware, particularly standards-based middleware such as CORBA or J2EE, by transparently migrating component functionality into an application on-demand. This compound pattern applies the Factory Method [4], the Proxy [4], and Component Configurator [5] design patterns to achieve its goals. We describe the Virtual Component pattern as a separate named abstraction since each of these constituent pattern does not independently resolve the set of forces addressed by the Virtual Component pattern.*

Keywords: Design Patterns, Embedded Systems, Memory Footprint, Middleware, CORBA, Abstract Factory pattern, Component Configurator pattern, Proxy pattern.

1 Intent

The Virtual Component design pattern provides an application-transparent way of loading and unloading components that implement middleware software functionality. This pattern ensures that the middleware provides a rich and configurable set of functionality, yet occupies main memory only for components that are actually used.

2 Example

There is a trend to develop memory-constrained applications, such as embedded systems, using feature-rich middleware, such as Java VMs [6] and RMI [7], COM+ [8], or CORBA [9]. Memory-constrained applications have historically been developed manually and hard-coded to use low-level languages and software tools, which is tedious and error-prone. The growing use of middleware provides a more powerful distributed computing model that enables clients to invoke operations on reusable components without hard-coding dependencies on their location, programming language, OS platform, communication protocols and interconnects, and hardware [10].

However, middleware (particularly standards-based middleware) often has many features that are not needed by all applications that use it. For example, in the context of CORBA:

- A server application may not actually use all three versions (*i.e.* version 1.0, 1.1 and 1.2) of the standard CORBA Internet inter-Orb protocol (IIOP).
- A client application may not need to use all the collocation optimizations, interceptors, or smart proxy mechanisms specified in the CORBA standard.
- "Pure client" CORBA applications that invoke requests, but do not service requests from other clients, do not require a portable object adapter (POA) (a POA maps client requests to the appropriate servant in a CORBA server).

Providing all these features in one monolithic implementation increases middleware footprint, which may make it unsuitable for use in memory-constrained applications. Two types of footprint are important to memory-constrained applications:

- **Static footprint**, which is the amount of storage needed to hold an image of an application. This storage can reside in the ROM where the program is stored, on secondary disk storage, etc. The static footprint of a particular version of a particular application is time-invariant since it does not change as the application runs.
- **Dynamic footprint**, which represents the amount of memory used by a running instance of the application. The *dynamic footprint* is essentially the sum of the *code*, the *heap*, and *stack* size. Based on this definition, it is clear that the *dynamic footprint* is time-dependent. For memory-constrained applications, it is essential to have a hard upper bound on the dynamic footprint size.

One way to reduce middleware footprint is to provide compile-time options that produce subsets of middleware tailored for the needs of each specific application. As the number of capabilities in the middleware grows, however, this solution does not scale well since it either

- Forces application developers to know in advance what features they will require or
- Increases the development cycle by forcing application developers to recompile the middleware whenever they change the set of features they require.

In addition, compile-time subsetting approaches may not reduce middleware dynamic memory footprint sufficiently since the resources associated with infrequently used components will be allocated during execution–even if they are not used during a particular run of an application. It is therefore necessary to devise a more effective technique to reduce static and dynamic footprint of middleware by removing unneeded functionality and providing fine-grained control over the different middleware components used by an application at run-time.

3 Context

Componentized middleware, where configurability and customizability is needed to meet memory constraints imposed by application requirements and the run-time platform.

4 Problem

Middleware provides developers with a powerful and reusable set of abstractions for building applications. Implementing memory-constrained applications via reusable middleware remains hard, however, since the following forces must still be resolved:

1. The specifications for features and options for middleware (particularly standards-based middleware) are large and continually growing
2. Middleware implementations can incur a large static and dynamic memory footprint unless they are designed in advance to avoid this and
3. Memory-constrained applications cannot afford to waste storage on unnecessary or rarely used functionality.

In addressing the previous forces, care must be taken to provide a solution that:

1. Presents a clean architecture that is amenable to reuse and can be retargeted easily.
2. Does not introduce accidental complexity in the exposed API, *i.e.* is transparent.

Supporting all the features and options specified by middleware can therefore create a large memory footprint, making the middleware unsuitable for memory-constrained applications.

Few applications use all the functionality provided by middleware. However, implementers of middleware cannot want only eliminate capabilities from their products based on the needs of any particular application. Middleware must therefore be designed to support easy customization to meet application functionality requirements *and* memory constraints.

↪ Implementations of standard CORBA must be prepared to provide all the services in the specification, even though applications rarely use all its features. For example, business applications use only a few transport protocols or QoS policies. Likewise, embedded applications may not use the CORBA `Any` data type, or may choose to use `Any`s in a limited way. Moreover, real-time applications rarely use middleware meta-programming features [11], such as the CORBA dynamic invocation interface (DII) that discovers and invokes operations on objects at run-time.

5 Solution

Identify *components* whose interfaces represent the building blocks of the middleware being developed and implement these capabilities using *concrete components*. Use *factories* to create the concrete components needed by an application via a set of *loading/unloading strategies* that provide different ways to instantiate and manage the concrete components occupying memory at run-time. This pattern *virtualizes* each component since the middleware does not know whether a component is present or when it will be instantiated until its functionality is actually used.

In detail: each component identified during the middleware decomposition should be implemented in a manner that is as decoupled from other components as possible. A concrete component should be associated with a loading strategy to support different application use-cases, *e.g.*, some concrete components should be loaded eagerly, whereas others should be loaded lazily. Component unloading can also be done eagerly (*e.g.*, as soon as the instance reference count goes to zero) or lazily (*e.g.*, when when memory gets low, unload components with zero reference count based on a least-recently-used algorithm, etc.). The Virtual Component pattern provides component-level control over the subset of functionality that is needed for a given application, thereby minimizing the overall static and dynamic footprint by controlling when, how, and what components will be instantiated as the application runs.

6 Structure

The participants in the Virtual Component pattern include the following:

- **Component**, which defines the interface to the capabilities provided by a component and represents the *contract* between a component and its clients.

  ↪ For example, if an ORB's Portable Object Adapter (POA) is designated as a component, its component interface is the `PortableServer` API specified by the OMG CORBA specification [9].

Class Component	***Collaborator***
Responsibility • Define an interface for a well defined and encapsulated service provided by the middleware	

- **Concrete component**, which provides the actual implementation of a component. The Virtual Component pattern avoids loading and/or deploying concrete components into applications that do not require their capabilities.

  ↪ For example, the implementation of a POA in a particular ORB is a concrete component.

Class ConcreteComponent	***Collaborator***
Responsibility • Provide an implementation for the service defined by the Component	

- **Component factory**, defines an interface and a factory method that creates components.

  ↪ For example, the component factory in CORBA is the `CORBA::ORB` interface, whose `resolve_initial_references()` operation is a factory method that returns an object reference to a component based on a string parameter passed to the operation. Other factory methods in CORBA include `CORBA::Object::_narrow()` and `CORBA::ORB::string_to_object()`.

Class ComponentFactory	***Collaborator***
Responsibility • Provide an interface for the creation of Components	

- **Concrete component factory**, a concrete implementation of the component factory that produces concrete components.

↪ For example, the implementations of the `CORBA::ORB` or `CORBA::Object` interfaces provided by a specific ORB are concrete component factories.

Class	***Collaborator***
ConcreteComponentFactory	• Loading Strategy
Responsibility	
• Create ConcreteComponents, using the appropriate loading strategy	

- **Loading strategy**, which defines how and when a concrete component is loaded and instantiated.

  ↪ For example, an embedded application that needs to minimize its dynamic footprint can use a loading strategy that loads and instantiates a POA concrete component lazily, *i.e.*, on-demand at run-time. Conversely, a real-time application that cannot tolerate the jitter introduced by this lazy initialization can use a loading strategy that pre-loads and initializes the POA eagerly, *i.e.*, at ORB initialization time.

Class	***Collaborator***
LoadingStrategy	
Responsibility	
• Provide a way of loading and instantiating components	

- **Unloading strategy**, which defines how and when a concrete component and its associated resources are unloaded.

  ↪ For example, components could be reference counted, and once the instance count reaches zero the associated resources are unloaded, *i.e.*, the component instance can be released and its associated DLL could be unloaded. Unloading may be lazy or eager. Lazy unloading means a component is unloaded only when available memory becomes low, at which point classes with a zero instance count may be unloaded. Eager unloading means a component class is unloaded immediately whenever the instance count goes to zero.

The structure of participants in the Virtual Component pattern is shown in the class diagram in Figure 1.

Class	***Collaborator***
UnloadingStrategy	
Responsibility	
• Provide a way of unloading components	

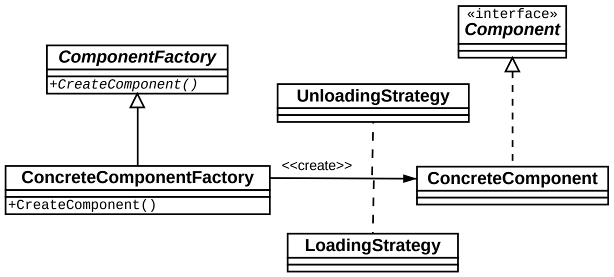


Figure 1: **Virtual Component Static Structure.**

7 Dynamics

To illustrate the collaborations performed by participants in the Virtual Component pattern, we examine three different loading scenarios. In this paper, we distinguish between the *loading* and the *instantiation* of a component:

- *Loading* refers to the activities performed to make the implementation of a component available for instantiation. For example, in a C++ application where components are packaged in DLLs, loading a component corresponds to loading the DLL associated with a component. Conversely, in a Java application the loading phase corresponds to loading the classes that implement the component within the JVM.
- *Instantiation* corresponds to actually creating and initializing an instance of a particular component. Clearly, to be instantiated a component must first be loaded.

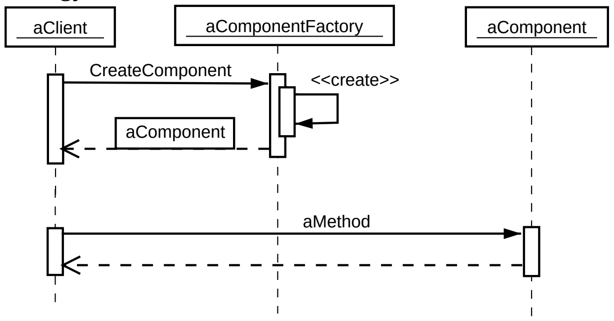
Scenario 1. This scenario shows an *eager static* loading strategy, where the application loads the entire concrete component during program startup.

- At application startup time, all needed components are loaded and initialized.
- The application code creates a concrete component via a component factory, which can either provide a pre-initialized component or create a new one.

No subsequent dynamic loading is necessary in this case.

- The application code then uses the component.
- In this scenario the unloading policy does nothing, *i.e.*, it is a no-op. Unloading a component that is not used at a particular time or during a particular path through a program defeats the purpose of eager static loading since additional latency will be required to reload the component if its needed later in the program. The main goal of eager static loading is to ensure that whatever is used by the system will already be in place when it is needed. This property is desirable for applications that cannot tolerate jitter introduced by lazy loading and instantiation.

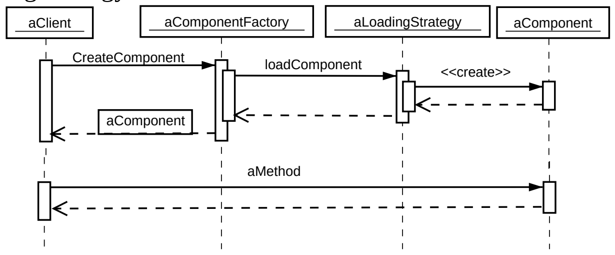
The following figure illustrates the eager static loading strategy:



Scenario 2. This scenario shows an *eager dynamic* loading strategy. In this strategy, a concrete component factory loads the entire concrete component when the application resolves the component at run-time, rather than loading it eagerly during program startup.

- The application code creates a concrete component via a component factory.
- The eager dynamic loading strategy associated with the concrete component factory loads the concrete component. Depending on how the middleware was configured, the factory can create different types of concrete components.
- The application code then uses the component.
- Component use is reference counted, and when a component reference count drops to zero the unloading strategy associated with the component is invoked.

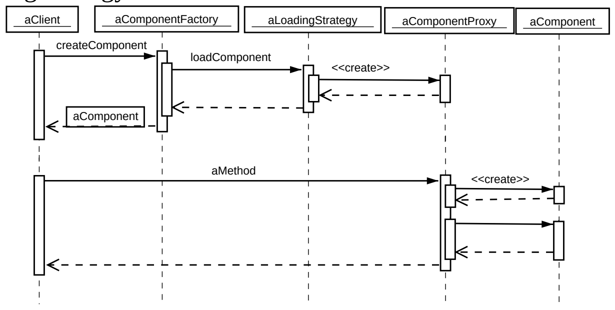
The following figure illustrates the eager dynamic loading strategy:



Scenario 3. This scenario shows a *lazy dynamic* loading strategy. This strategy defers loading the concrete component until it is actually accessed by a client, rather than loading it when it is requested by the application.

- The application code creates a concrete component via a component factory.
- The concrete component factory creates and returns a reference to a proxy for the concrete component. This proxy knows how to load the component's functionality and data when it is accessed.
- The application code then uses the component via its proxy, which triggers the proxy to load the component's functionality and data. Depending on how the middleware was configured, the proxy can load different types of concrete components.
- Component use is reference counted, and when a component reference count drops to zero the unloading strategy associated with the component is invoked.

The following figure illustrates the lazy dynamic loading strategy:



8 Implementation

This section describes the activities associated with implementing the Virtual Component pattern.

1. Partition the middleware into a set of decoupled components with well-defined interfaces. *Required components* are those required by any application that might use the middleware since they are fundamental to its internal operational behavior. In contrast, *optional components* are those that may be used by specific applications. Identify the optional components that exist in the middleware and make them into virtual components. Each virtual component will be represented by a well-defined interface, a concrete implementation, and a loading strategy.

↪ The ACE ORB (TAO) [12], which is an open-source implementation of CORBA, decomposes its C++ implementation of CORBA [9] into many core ORB components with well-defined IDL interfaces, such as the Portable Object Adaptor (POA), Real-time CORBA, CORBA Messaging, transport protocols, and interface repository, that are optional for memory-constrained applications. TAO's loading strategies are described in implementation activity 3.2.

2. Implement the concrete components. Some components identified in implementation activity 1 may have more than one implementation alternative. Where alternative implementations exist for a component, or where a set of different subcomponents work together to implement a component, apply the Strategy pattern [4] to define each concrete component implementation for the virtual component interface.

↪ For example, there may be three different types of POAs: a standard POA, a real-time POA, and a multicast POA [13]. When the POA is needed, only the concrete class of the desired type of POA is loaded and instantiated. In general, the types of optional middleware components in TAO include:

- **Excess components.** Certain applications may or may not need a particular CORBA feature, such as `Any` data types or portable interceptors. The middleware—and in some cases even application developers—may not be able to determine before run-time if a feature is needed.
- **Role-dependent components.** The CORBA standard defines a large set of related capabilities, such as CORBA IIOP message marshalers/demarshalers, portable object adapters (POAs), or interoperable object reference (IOR) format parsers. Only a subset of these capabilities are actually used at run-time, depending on the role that an application play, *e.g.*, client, server, or both client and server. However, all capabilities must be available in case they are needed.
- **Component implementation alternatives.** CORBA capabilities, such as buffer allocation, object adaptor, network protocol, or dynamic thread scheduling, can be implemented using a variety of different strategies, yet only one (or in some cases, none) is actually needed at any given time.

These optional implementations of TAO's components described above are implemented as strategies that reside on secondary storage until they are needed or wanted, at which point they are linked into the application's address space.

3. Define the component loading strategies. Define a means to configure which loading strategies should be used for each concrete component in the middleware. There are several sub-activities involved here:

3.1. Determine how to associate component identity with concrete component implementations. Concrete components can be named using integer values or strings. Strings can be read more easily by programmers and simple management tools, but integer values require less space and may be compared more efficiently.

↪ For example, CORBA's `resolve_initial_references()` operation is passed a string containing the component to be resolved.

To prevent name clashes, interface identifiers can be generated algorithmically.

↪ For instance, Microsoft COM identifies components using 128 bit globally unique identifiers (GUIDs) based on the address of the network interface, the date, and the time.

The component factory includes a method that returns component references to clients. The type of information returned from this method depends largely on the programming language.

↪ For example, CORBA and Java clients will receive an object reference, whereas pointers may be an appropriate choice for C++.

3.2. Determine the loading strategy for concrete components. Some applications are more concerned about footprint, while others may be more concerned about predictable real-time performance. We therefore differentiate the following binding times:

- **Eager static**, which loads the entire concrete component immediately during program initialization. For composite components, this loading strategy takes care of loading the closure of components. For example some JVM support option to enable eager class loading, in this case the closure of the classes referenced by the application is loaded eagerly at startup time to avoid the latency introduced by loading on demand. This strategy is intended for those applications concerned with real-time performance. In this case, the loading strategy may pre-load user-selected concrete components at initialization time to eliminate jitter that would result if a lazy component faulting strategy were used. The implementation of this strategy relies on implementation language mechanisms to perform static initialization. For example Java provides *static* blocks, while in C++ there are idioms that can be used to simulate a Java *static* block. The code that is executed at application startup time then ensures that all components are loaded and instantiated.
- **Eager dynamic**, where the concrete component factory loads the entire concrete component when it is instructed to resolve the component at run-time. This strategy is suited for applications that are concerned about minimal footprint, but can tolerate initial delays from dynamic loading. The implementation of this strategy could use the Component Configurator pattern [5] to implement the different components. When a component needs to be created, the component factory will use the services provided by the component configurator to load and initialize it.
- **Lazy dynamic**, which defers the loading of a concrete component until it is actually accessed by the client. This strategy is suited for footprint-sensitive applications that can tolerate the latency of loading a concrete component into memory on-demand. The implementation of this strategy would require the component factory to create component proxies that will lazily create their associated concrete components only when a component is actually accessed.

$\hookrightarrow$ TAO uses the eager dynamic scheme in which the ACE Service Configurator [14] framework (which implements the Component Configurator pattern [5]) is used to load and initialize components, which are created via factory methods. For example, a POA provides a series of services needed to dispatch upcalls to CORBA servants in a portable, efficient, and type-safe manner. This optional component need not be loaded and initialized in the ORB until an application tries to resolve it via the `CORBA::ORB::resolve_initial_reference()` operation. At this point, the appropriate POA class is loaded and an instance is created. A similar approach is used for the other TAO components mentioned above.

Likewise, ZEN [15] is a Java implementation of CORBA that uses Java's dynamic class loading (or JVM support for on-demand class loading) to load concrete components at the appropriate time. ZEN uses naming conventions to construct class names from the values, such as version number, message type, or IOR format name, that lead to the use of the class, so that the ORB can know what classes to load. These uniform naming conventions enable new classes to be added easily later.

In both these CORBA implementations, optional middleware components can either be

- **Eager static**, *e.g.*, when a developer knows a particular set of core ORB services are required or
- **Eager dynamic**, *i.e.* loaded on-demand only when needed and optionally cached in primary memory for fast reuse.

4. Define the concrete component unloading strategies. Depending on application characteristics, different unloading strategies can be used. At one end of the spectrum, we could have an unloading strategy that does nothing, *i.e.*, it could be a Null Object [16]. Conversely, we could have an unloading strategy that unloads all the resources associated with a component, including the state of the component and the code instructions residing in its DLL. Naturally, unloading DLLs must be performed in a way that is consistent with how DLLs are opened, loaded, and used, as discussed in the following sub-activities.

4.1. Determine the reference counting strategy. One approach to component unloading is to use the reference counting provided by the OS for opened DLLs.

Another approach would implement this reference counting in the middleware via patterns such as Counted Pointer [2], to avoid unnecessary system calls and provide greater control over resource management.

↪ For instance, under Linux it is possible to load and unload DLLs using respectively the functions `dlopen()` and `dlclose()`. Linux keeps a reference count of the library opened, and takes care of unloading the DLL once the number of `dlclose()` matches the number of `dlopen()`. Based on this, one way of implementing loading/unloading of components, would be that of always performing a `dlopen()` when the component is created, and always a `dlclose()` when the component is unloaded. In this case the OS would perform the DLL reference-counting. This solution has obvious advantages of delegating the responsibility of DLLs reference counting to the OS, but has the disadvantage that many superfluous call to `dlopen()` and `dlclose()` are made. The other way of implementing loading and unloading, is that of implementing the DLLs' reference-counting in the middleware, and have only one `dlopen()`, and a matched and `dlclose()` per component.

4.2. Determine whether the components are stateless or stateful. The developer need to understand whether (1) component are stateless, or (2) their state can be discarded once the component is unloaded, or (3) the component state has to persist across multiple load and unload. In the latter case, two solution can be envisioned. The component can be made serializable, so that the state can be saved and restored at each loading and unloading. The other solution that can be pursued is that of avoiding the unloading the component at all.

9 Example Resolved

Applications written using CORBA rely on a *stub* to make operations invoked on CORBA objects appear local, even if they are remote. A stub is an instance of the Proxy pattern [4, 2] that marshals and demarshals parameters and forwards client requests to a target object. Internally, an ORB may need to use different stub implementations for the same CORBA object or for different instances of the same CORBA object. For example, different stub implementations can be used to implement collocation optimizations [17], interceptors [18], or smart proxies [19].

The Virtual Component pattern has been applied extensively to TAO. One of the use of this pattern within TAO, is to control how stubs associated with CORBA objects are loaded and initialized. To see how TAO uses the Virtual Component pattern, consider the following code that defines the IDL interface for a Bank Account application. The use of the Virtual Component pattern in this context helps application with many objects of many different types to minimize their dynamic footprint.

```
interface Account {
  float get_balance ();
  float withdraw (in float amount);
  void deposit (in float amount);
};
```

The class diagram depicted in Figure 2 represents the structure of the classes created by the TAO's IDL compiler for the `Account` interface. This set of collaborating

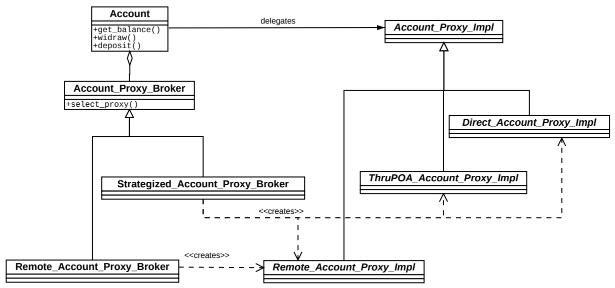

Account
+get_balance()
+widraw()
+deposit()
delegates
Account_Proxy_Impl
Account_Proxy_Broker
+select_proxy()
Direct_Account_Proxy_Impl
Strategized_Account_Proxy_Broker
ThruPOA_Account_Proxy_Impl
<<creates>>
<<creates>>
Remote_Account_Proxy_Broker
Remote_Account_Proxy_Impl

Figure 2: **Static Structure of the Virtual Component Pattern Applied in TAO**

classes implements the Virtual Component pattern. Figure 2 and Figure 1 shows how the `Account_Proxy_Impl` class plays the component role, while the implementation of this class plays the concrete component role. The `Account_Proxy_Broker` plays the role of the component factory, while the `Remote_Account_Proxy_Broker` and the `Strategized_Account_Proxy_Broker` play the role of the concrete component factory. In this incarnation of the Virtual Component pattern, the loading strategy is associated with the concrete factory rather than the concrete component.

Figure 3 shows the interaction diagram for this instance of the Virtual Component pattern. In this use case, the proxies are loaded lazily.

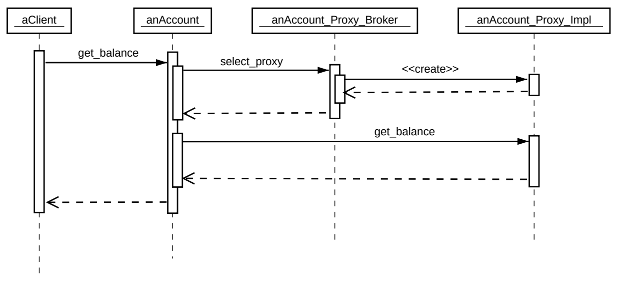


Figure 3: **Interaction Diagram of the Virtual Component Pattern Applied in TAO**

The C++ code fragment below shows the base class for the stub implementation associated with the `Account` interface. This stub code is generated by the TAO IDL compiler. The `TAO_Account_Proxy_Impl` provides exactly the same methods defined in the `Account` interface, but each method has an additional argument of type `CORBA::Object`, which represents the target object on which the method is invoked. This argument is needed to make the `TAO_Account_Proxy_Impl` concrete implementation stateless. Note that all the `TAO_Account_Proxy_Impl` instances are stateless flyweights [4].

```
class TAO_Account_Proxy_Impl
  : public virtual TAO_Object_Proxy_Impl
{
public:
  virtual ~TAO_Account_Proxy_Impl (void) {}

 virtual CORBA::Float get_balance
   (CORBA::Object *obj) = 0;

 virtual CORBA::Float withdraw
   (CORBA::Object *obj,
    CORBA::Float amount) = 0;

 virtual void deposit
   (CORBA::Object *obj,
    CORBA::Float amount) = 0;

protected:
  TAO_Account_Proxy_Impl (void);
};
```

Concrete implementations of `TAO_Account_Proxy_Impl` provide different ways of performing a call on a CORBA object. Decorators [4] could also be used to add behavior to existing concrete implementations.

The `TAO_Account_Proxy_Broker` class shown below is the base class for the various concrete stub implementation factories.

```
class TAO_Account_Proxy_Broker
{
public:
  virtual ~TAO_Account_Proxy_Broker (void);

  virtual TAO_Account_Proxy_Impl &
    select_proxy (Account *obj) = 0;

protected:
  TAO_Account_Proxy_Broker (void);
};
```

The role of this class is to create the appropriate stub to perform a call on a given CORBA object, based on properties of the running application. In TAO, this factory represents the portion of code that encapsulates the loading strategy for stub implementations. In fact, depending on how the application is compiled, different instances of the component factory will allow either eager or lazy loading.

The actual instance of the `TAO_Account_Proxy_Broker` subclass created for an `Account` object depends on both static configuration and runtime properties. TAO then dynamically uses the `CORBA::Object::_narrow()` operation as a factory method to create the appropriate subclass of `TAO_Account_Proxy_Broker`. The appropriate `TAO_Account_Proxy_Broker` factory will be created when the following client code runs:

```
  // Get the object reference somehow.
  CORBA::Object obj = ...

  // Narrow the object down to the right type.
  Account_var an_account =
    Account::_narrow (obj);

 // ...
```

After this operation is performed, the appropriate `TAO_Account_Proxy_Broker` implementation will have been set for the `Account` object.

Now let's consider what happens when the following code is executed:

```
  // ...
  CORBA::Float balance =
    an_account->get_balance ();
  // ...
```

This fragment of code results in the invocation of the following method:

```
CORBA::Float Account::get_balance () {
  TAO_Account_Proxy_Impl &proxy =
    this->the_TAO_Account_Proxy_Broker_->
      select_proxy (this);

  return proxy.get_balance (this);
}
```

At this point, depending on the instance of the `TAO_Account_Proxy_Broker` subclass that was associated with the `Account` object, the most appropriate `TAO_Account_Proxy_Impl` will be returned. The code executed by the concrete instance of the `TAO_Account_Proxy_Impl`, (*e.g.*, if we consider the `TAO_Account_Strategized_Proxy_Broker`) would behave as described below:

```
TAO_Account_Proxy_Impl &
TAO_Account_Strategized_Proxy_Broker::
  select_proxy (Account *object)
{
  int strategy =
    TAO_ORB_Core::collocation_strategy
        (object);

  if (this->proxy_cache_[strategy] != 0)
    return *this->proxy_cache_[strategy];

  // This call loads and creates the
  // appropriate instance of the proxy
  // depending on the strategy.
  this->create_proxy (strategy);

  return *this->proxy_cache_[strategy];

}
```

In this code fragment, the `TAO_Account_Strategized_Proxy_Broker` uses a strategy to determine the most appropriate proxy for performing the requested operation. It then lazily obtains an instance of the needed proxy implementation. Depending on how the `create_proxy()` method is implemented, a combination of both lazy creation and lazy loading are possible. In the current implementation, no unloading take place.

As shown in this example, the Virtual Component pattern can be used to control the way in which different concrete components in a software system are loaded and initialized. In the context of TAO, this pattern helps to reduce the dynamic footprint for CORBA applications, especially for applications that have many different CORBA object instances. Moreover, the Virtual Component patterns frees application developers from having to know which features it will need, while providing a mechanism to ensure that what is needed will be in the right place at the right time.

10 Known Uses

The ACE ORB (TAO) [12] is a C++ implementation of CORBA that uses the Virtual Component pattern to implement its portable object adapter (POA), client-side support for its Interface Repository, pluggable protocols, handling of multiple IOR formats, and to support the CORBA dynamic invocation interface (DII) and dynamic skeleton interface (DSI).

The ZEN ORB [15] is a Java implementation of CORBA that uses the Virtual Component pattern for a wide range of optional capabilities, including pluggable object adapters, object resolvers, IOR parsers, GIOP message handlers, message buffer allocation, CDR stream reader/writers, protocol transports, and Any data types. Since ZEN's design was heavily influenced by the lessons learned on the TAO project, it is more aggressive in its uses of the Virtual Component pattern throughout the ORB.

Java Virtual Machines (JVMs) use a variation of this pattern in which a component is essentially represented by a class. In order to avoid loading classes that may not be used, JVMs defer the loading of a particular class up until its first active use. Moreover, JVMs unload classes for which there are no instances. Unfortunately, there is no standard way of specifying eager loading of classes, so JVMs always use lazy loading. For applications that cannot tolerate the jitter introduced by lazy class loading, the Virtual Component pattern could be used to provide eager class loading in Java middleware and applications.

11 Consequences

The Virtual Component design pattern has the following **benefits**:

- **Smaller footprint** – The static and dynamic footprint of the middleware can be adapted to suit the needs of the application. For example, any particular component that is known to be unused can be eliminated from the static footprint. In addition, only those components in active use are included in the dynamic footprint.
- **Increased flexibility** – It allows middleware developers to offer alternative implementations for components of their system, which improves middleware flexibility by supporting different application requirements. For example, alternative algorithms for buffer allocation may be offered, via the Strategy pattern [4], yet only one alternative at a time must be "plugged in."
- **Better timing control** – It provides fine-grained control over the timing of component loading and unloading. The strategy for loading and unloading a component can be customized to suit the needs of each application. For example, components that cause jitter when loaded lazily can be configured to be loaded eagerly. Likewise, those not contributing to jitter may be loaded and instantiated lazily to minimize the dynamic footprint.

However, this pattern also incurs the following **liabilities**:

- **Higher overhead** – The use of a decoupling layer between the components in the middleware introduces some overhead. Whether this overhead is acceptable or excessive depends on the application, implementation language, compiler technology, and the role that the component plays in the critical path of the application. Advanced software techniques, such as global compiler optimizations or aspect-oriented programming, can be used to reduce or eliminate much of this overhead.
- **Greater jitter** – Certain loading and unloading strategies—*i.e.*, eager dynamic and lazy dynamic—can cause processing delay and jitter that may be unacceptable for real-time applications. Eager static loading may be used to eliminate these delays, however, if adequate memory is available for the resulting dynamic footprint expansion.

12 See Also

The Virtual Component pattern can be viewed as a compound pattern [3] that combines elements of the Factory Method [4], Proxy [4], and Component Configurator [5] patterns. The Factory Method pattern defines an interface for creating an object, but allows subclasses to decide the particular type. Essentially, a factory method defers the instantiation of an object to subclasses. The Proxy pattern provides a surrogate or placeholder for another object to control access to it. The Component Configurator pattern is a dynamic component configuration mechanism that uses a *scripting* mechanism to define what components are created, brought into the system, and removed from the system. The Virtual Component compound pattern combines these patterns to create a dynamic component configuration mechanism that relies on a implicit *faulting* mechanism to load in a required component.

It is worth noting that the patterns constituting the Virtual Component pattern used in isolation do not address all the forces addressed by the Virtual Component pattern. It is the synergy provided by this compound pattern that make it possible to address all the forces outlined in Section 4.

One variation of the Proxy pattern [2, 4] has a simple implementation of an object that can be substituted with the full implementation of that object upon demand. The Proxy pattern can be used to implement component proxy in the lazy dynamic variant of the Virtual Component pattern. In particular, a factory would instantiate a component proxy instead of a concrete component. The proxy would then create the concrete component at its first active use *i.e.*, when the first method call is invoked on the object.

The Strategy design pattern [4] defines a common interface with alternative implementations so different objects may be plugged-in to allow variations in desired behavior.

Some virtual components may be defined by a strategy applied to optional components identified by application developers.

The Lazy Acquisition [20] design pattern provides a way to defer resource acquisition. This pattern could be used in synergy with the Virtual Component pattern to design the lazy loading strategies.

Early real-time operating systems provided programmer-controlled memory segment overlays. For example, DEC RT-11 allowed programmers to define segments of code and/or data to load at different times to overlay the same area of memory. These memory segments were hard to implement, however, because programmers had to decide *before run-time* which functions to group into segments. They also had to decide *at run time* how to explicitly switch from segment to segment. This approach and its corresponding Segmentation pattern is described by [1].

Systems with abundant primary and secondary storage and virtual memory can rely on operating system virtual memory mechanisms to subset the footprint of middleware and application software. In turn, OS virtual memory mechanisms are based on patterns such as Copy-on-Write and Paging described in [1]. Virtual memory may not be predictable enough for many types of real-time embedded systems, however. In addition, many embedded systems have primitive operating systems and hardware that make conventional virtual memory solution infeasible.

13 Concluding Remarks

As embedded applications are increasingly developed with standards-based, reusable middleware, there is a growing mismatch between what is provided by the middleware and what is needed by any particular application. This mismatch can yield wasted memory resources for situations where the middleware does not provide an effective level of configurability. For example, if a CORBA ORB is used to support an embedded application it is crucial to avoid paying memory footprint costs for functionality that is not needed by the application.

Developers of middleware must therefore make hard choices about what functionality to include and what functionality to omit. If too much functionality is included, middleware footprint will be unsuitable for embedded applications that have stringent constraints on the size of their EPROM and RAM memory. Conversely, if too little functionality is included, the middleware may not support application needs adequately, which pushes more development effort and cost onto application developers.

The Virtual Component pattern described in this paper allows developers of standards-based middleware to offer a large set of functionality to their users while keeping the static and dynamic memory footprints proportional to the features they actually use. This pattern *virtualizes* each component since the middleware does not know whether a component is present or when it will be instantiated until its functionality is actually used. The Virtual Component pattern has been applied successfully in a variety of standards-based middleware, including TAO, ZEN, and Java virtual machines.

14 Acknowledgments

Thanks to the DOC Group at Washington University, St. Louis and the University of California, Irvine for applying the Virtual Component pattern in the TAO and ZEN ORB middleware projects. Thanks also to Don Hinton, and to our PLoP shepherds Michael Kircher and Prashant Jain, for their helpful comments on the paper.

References

[1] J. Noble and C. Weir, *Small Memory Software: Patterns for Systems with Limited Memory*. Boston: Addison-Wesley, 2001.

[2] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture – A System of Patterns*. New York: Wiley and Sons, 1996.

[3] J. Vlissides, *Pattern Hatching: Design Patterns Applied*. Reading, Massachusetts: Addison-Wesley, 1998.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Massachusetts: Addison-Wesley, 1995.

[5] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2*. New York: Wiley & Sons, 2000.

[6] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*. Reading, Massachusetts: Addison-Wesley, 1997.

[7] A. Wollrath, R. Riggs, and J. Waldo, “A Distributed Object Model for the Java System,” *USENIX Computing Systems*, vol. 9, November/December 1996.

[8] D. Box, *Essential COM*. Addison-Wesley, Reading, Massachusetts, 1997.

[9] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 2.6 ed., Dec. 2001.

[10] M. Henning and S. Vinoski, *Advanced CORBA Programming With C++*. Reading, Massachusetts: Addison-Wesley, 1999.

[11] N. Wang, D. C. Schmidt, O. Othman, and K. Parameswaran, “Evaluating Meta-Programming Mechanisms for ORB Middleware,” *IEEE Communication Magazine, special issue on Evolving Communications Software: Techniques and Technologies*, vol. 39, Oct. 2001.

[12] D. C. Schmidt, D. L. Levine, and S. Mungee, “The Design and Performance of Real-Time Object Request Brokers,” *Computer Communications*, vol. 21, pp. 294–324, Apr. 1998.

[13] R. Klefstad, A. Krishna, and D. C. Schmidt, “Design and Performance of a Modular Portable Object Adapter for Distributed, Real-Time, and Embedded CORBA Applications,” in *Submitted to the 4th International Symposium on Distributed Objects and Applications*, (Irvine, CA), OMG, October/November 2002.

[14] D. C. Schmidt and T. Suda, “The Service Configurator Framework: An Extensible Architecture for Dynamically Configuring Concurrent, Multi-Service Network Daemons,” in *Proceedings of the Second International Workshop on Configurable Distributed Systems*, (Pittsburgh, PA), pp. 190–201, IEEE, Mar. 1994.

[15] R. Klefstad, D. C. Schmidt, and C. O’Ryan, “The Design of a Real-time CORBA ORB using Real-time Java,” in *Proceedings of the International Symposium on Object-Oriented Real-time Distributed Computing*, IEEE, Apr. 2002.

[16] B. Woolf, “The Null Object Pattern,” in *Pattern Languages of Program Design* (R. Martin, F. Buschmann, and D. Riehle, eds.), Reading, Massachusetts: Addison-Wesley, 1997.

[17] N. Wang, D. C. Schmidt, and S. Vinoski, “Collocation Optimizations for CORBA,” *C++ Report*, vol. 11, pp. 47–52, November/December 1999.

[18] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith, “Using Interceptors to Enhance CORBA,” *IEEE Computer*, vol. 32, pp. 64–68, July 1999.

[19] N. Wang, D. C. Schmidt, M. Kircher, and K. Parameswaran, “Towards a Reflective Middleware Framework for QoS-enabled CORBA Component Model Applications,” *IEEE Distributed Systems Online*, vol. 2, July 2001.

[20] M. Kircher, “Lazy acquisition.”