The Extension Interface design pattern allows multiple interfaces to be exported by a component, to prevent bloating of interfaces and breaking of client code when developers extend or modify the functionality of the component.

Example Consider a telecommunication management network (TMN) [ITUT92] framework that can be customized to monitor and control remote network elements such as IP routers and ATM switches. Each type of network element is modeled as a multi-part framework component in accordance with the Model-View-Controller pattern [POSA1]. A view and a controller are located on a management application console. The view renders the current state of a network element on the console and the controller allows network administrators to manage the network element.

A model resides on the network element and communicates with the view and controller to receive and process commands, such as commands to send state information about the network element to the management application console. All components in the TMN framework are organized in a hierarchy. The UniversalComponent interface shown in the following figure provides the common functionality needed by every component, such as displaying key properties of a network element and accessing its neighbors.

```java
public interface UniversalComponent {
    public void render () { ... }
    public int getState () { ... }
    public long getPermissions () { ... }
    public Enumeration getNeighbors () { ... }
    public void lock () { ... }
    public void unlock () { ... }
    public Object getData (String key) { ... }
    public void setData (String key, Object v) { ... }
    // ... many other methods ...
}
```
In theory, this design might be appropriate if the UniversalComponent interface shown above is never changed, because it would allow client applications to access a wide range of network elements via a uniform interface. In practice, however, as the TMN framework becomes increasingly popular, management application developers will request that new functionality and new methods, such as dump() and persist(), be added to the UniversalComponent interface.

Over time the addition of these requests can bloat the interface with functionality not anticipated in the initial framework design. If new methods are added to the UniversalComponent interface directly, all client code must be updated and recompiled. This is tedious and error-prone. A key design challenge is therefore to ensure that evolutionary extensions to the TMN framework do not bloat its interfaces or break its client code.

**Context**

An application environment in which component interfaces may evolve over time.

**Problem**

Coping with changing application requirements often necessitates modifications and extensions to component functionality. Sometimes all interface changes can be anticipated before components are released to application developers. In this case it may be possible to apply the ‘Liskov Substitution Principle’ [Mar95]. This principle defines stable base interfaces whose methods can be extended solely via subclassing and polymorphism.

In other cases, however, it is hard to design stable interfaces, because requirements can change in unanticipated ways after components have been delivered and integrated into applications. When not handled carefully, these changes can break existing client code that uses the components. In addition, if the new functionality is used by only few applications, all other applications must incur unnecessary time and space overhead to support component services they do not need.

To avoid these problems, it may be necessary to design components to support evolution, both anticipated and unanticipated. This requires the resolution of four forces:

- When component interfaces do not change, modifications to component implementations should not break existing client code.
If implementations of our UniversalComponent interface store their state persistently in external storage, clients should not be affected if this functionality is re-implemented differently, as long as the component’s interface is unchanged.

- Existing client code should not break when developers extend a component with new services that are visible externally. Ideally it should not be necessary to re-compile client code.

- It may be necessary to add a logging service to the UniversalComponent interface so that management applications and network elements can log information to a central repository. Existing clients that are aware of the original version of UniversalComponent should not be affected by this change, whereas new clients should be able to take advantage of the new logging functionality.

- Changing or extending a component’s functionality should be relatively straightforward, neither bloating existing component interfaces nor destabilizing the internal architecture of existing components.

- When adding the logging service outlined above, we should minimize changes to existing implementations of the UniversalComponent interface.

- It should be possible to access components remotely or locally using the same interface. If components and their clients are distributed across network nodes, the interfaces and implementations of a component should be decoupled.

- Management applications can benefit from location-transparent access to remote network elements in our TMN system. It should therefore be possible to separate the interfaces of network element management components from their physical implementations. These can be distributed throughout the network.

Solution

Program clients to access components via separate interfaces, one for each role a component plays, rather than programming clients to use a single component that merges all its roles into a single interface or implementation.
In detail: export *component* functionality via *extension interfaces*, one for each semantically-related set of operations. A component must implement at least one extension interface. To add new functionality to a component, or to modify existing component functionality, export new extension interfaces rather than modify existing ones. Moreover, program clients to access a component via its extension interfaces instead of its implementation. Hence, clients only have dependencies on the different roles of a component, each of which is represented by a separate extension interface.

To enable clients to create component instances and retrieve component extension interfaces, introduce additional indirection. For example, introduce an associated *component factory* for each component type that creates component instances. Ensure that it returns an initial interface reference that clients can use to retrieve other component extension interfaces. Similarly, ensure that each interface inherits from a *root interface* that defines functionality common to all components, such as the mechanism for retrieving a particular extension interface. All other extension interfaces derive from the root interface. This ensures that at minimum they offer the functionality it exports.

**Structure**

The structure of the Extension interface pattern includes four participants:

*Components* aggregate and implement various types of service-specific functionality. This functionality can often be partitioned into several independent roles, each of which defines a set of semantically-related operations.

<table>
<thead>
<tr>
<th>Class</th>
<th>Collaborator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component</td>
<td></td>
</tr>
</tbody>
</table>

**Responsibility**

- Plays different roles
- Implements extension interfaces
- Returns initial interface to component factory
Components in our TMN framework play various roles, such as storing and retrieving the state of a network element or managing the persistence of a component’s internal state.

Extension interfaces export selected facets of a component’s implementation. There is one extension interface for each role [RG98] that a component implements. In addition, an extension interface implicitly specifies a contract that describes how clients should use the component’s functionality. This contract defines the protocol for invoking the methods of the extension interface, such as the acceptable parameter types and the order in which methods must be called.

The components in the TMN framework can implement the IStateMemory interface, which allows them to maintain their state in memory. A persistence manager, such as the CORBA Persistent State Service [OMG99e] can use the IStateMemory interface to manage component persistence without requiring components to expose their representational details.

If new network element components are added that also implement IStateMemory, the persistence manager can manage their persistence without requiring any changes. The IStateMemory interface contains methods to prepare the component for reading and writing its state, as well as its read and write operations. The implicit contract between the interface and its users therefore prescribes that the prepare() method must be called before either readState() or writeState().

The root interface is a special extension interface that provides three types of functionality:

• Core functionality that all extension interfaces must support, for example functionality that allows clients to retrieve the interfaces they request. This functionality defines the basic mechanisms a component must implement to allow clients to retrieve and navigate among its interfaces.

• Domain-independent functionality, such as methods that manage component life-cycles.

• Domain-specific functionality that should be provided by all components within a particular domain.
Although the root interface must implement core functionality, it need not support domain-independent or domain-specific functionality. However, all extension interfaces must support the functionality defined by the root interface. Each extension interface can thus play the role of the root interface, which guarantees that every extension interface can return any other extension interface on behalf of a client request.

### UniversalComponent Interface

A UniversalComponent interface can be defined as the root interface in our TMN framework. Unlike the multi-faceted—and increasingly bloated—UniversalComponent interface outlined in the Example section, however, this root interface only defines the minimum set of methods that are common to all components in the TMN framework.

Clients access the functionality provided by components only via extension interfaces. After a client retrieves a reference to an initial extension interface, it can use this reference to retrieve any other extension interface supported by a component.

A management application console client can use components in the TMN framework to render the state of network elements and their relations visually on the screen, as well as to store and retrieve their state using persistent storage.

<table>
<thead>
<tr>
<th>Class</th>
<th>Collaborator</th>
<th>Class</th>
<th>Collaborator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Root Interface</td>
<td>Component</td>
<td>Extension Interface</td>
<td>Component</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Responsibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>Defines a role-specific interface offered by a component</td>
</tr>
</tbody>
</table>

To retrieve an initial reference, clients interact with a component factory associated with a particular component type. This component factory separates the creation and initialization aspects of a component from its processing aspects. When a client creates a new
component instance, it delegates this task to the appropriate component factory.

After a component is created successfully, the component factory returns a reference to an extension interface to the client. A component factory may allow clients to request a specific type of initial extension interface. Factories may also provide functionality to locate and return references to existing component instances.

The class diagram below illustrates the participants in the Extension Interface pattern. This diagram emphasizes logical rather than physical relationships between components. For example, extension interfaces could be implemented using multiple inheritance or nested classes, as described in implementation activity 6.1 (155). Such implementation details are transparent to clients.
Dynamics We illustrate the key collaborations in the Extension Interface pattern using two scenarios. **Scenario I** depicts how clients create new components and retrieve an initial extension interface:

- The client requests a component factory to create a new component and return a reference to a particular extension interface.
- The component factory creates a new component and retrieves a reference to its root interface.
- The component factory asks the root interface for the requested extension interface, then returns a reference to the extension interface to the client.

Note that the factory could return any interface to the client, instead of retrieving a specific extension one. Such a design can incur additional round-trips in a distributed system, however, which increases the overhead of accessing the required interface.

**Scenario II** depicts the collaboration between clients and extension interfaces. Note that the component implementation itself is not visible to the client, because it only deals with extension interfaces:

- The client invokes a method on extension interface A, which can either be the root interface or an extension interface.
- The implementation of extension interface A within the component executes the requested method and returns the results, if any, to the client.
- The client calls the `getExtension()` method on extension interface A and passes it a parameter that specifies the extension interface in which the client is interested. The `getExtension()`
method is defined in the root interface, so it is supported by all extension interfaces. The implementation of extension interface A within the component locates the requested extension interface B and returns the client a reference to it.

- The client invokes a method on extension interface B, which is then executed within the component implementation.

**Implementation**

This section describes the activities associated with implementing the Extension Interface pattern. This pattern should be familiar to anyone who has programmed with Microsoft’s Component Object Model (COM) [Box97], Enterprise JavaBeans (EJB) [MaHa99], or the CORBA Component Model (CCM) [OMG99a], because it captures and generalizes the core concepts underlying these component technologies.

1 **Determine the stability of the design and the long-term application requirements.** Before applying the Extension Interface pattern, it is important to determine whether it is really needed. Although this pattern is a powerful solution to a particular set of forces, it is non-trivial to implement. It can also complicate a software design significantly if applied unnecessarily.

We therefore recommend that the forces outlined in the Problem section are considered carefully. You should ensure that these issues are faced in your software system before applying this pattern. For example, it may turn out that the complete set of methods an interface requires can be determined during system development, and that the interface will not change over time as application requirements evolve. In this case, it may be simpler to use the Liskov

2 Analyze the domain and specify a domain-specific component model. Assuming that the Extension Interface pattern is necessary, the next activity involves analyzing domain-specific application requirements. In particular, this activity focuses on identifying application-specific entities, such as the network elements in our TMN example, the roles a particular entity provides to the system, and the functionality that supports the different roles. The result is a domain model that identifies which components to implement, as well as the functionality they must provide.

For the management application console, every type of entity to be controlled is implemented as a separate managed object [ITUT92], which is an abstraction used to represent hardware units, such as routers, computers, bridges, or switches. Managed objects can also represent software elements, such as applications, ports, or connections. Management applications use managed objects to control and monitor the state of network elements, display debugging information, or visualize system behavior on a management console.

After devising a domain model, it is necessary to specify a component model to implement the identified components:

- If the components are restricted to a single application domain or a small set of related domains, consider specifying a domain-specific component model that is tailored for the application or family of applications being developed.

- Conversely, if the components must be applied to a wide range of applications, or even across multiple domains, consider using an existing component technology, such as Microsoft COM [Box97], EJB [MaHa99], or the CORBA Component Model [OMG99a].

In the latter case, the next implementation activity can be skipped, because these component models define the infrastructure it specifies.

3 Specify the root interface. Determine if each type of functionality identified in implementation activity 2 above should form part of the root interface, or be separated into an extension interface.
Extension Interface

With this criteria in mind, iterate through three sub-activities:

3.1 Specify core functionality. Several issues must be addressed when defining the core functionality of the root interface:

- **Extension interface retrieval.** At minimum, the root interface must include a method that returns extension interface references to clients. The type of information returned from this method depends largely on the programming language. For example, Java clients expect to retrieve an object reference, whereas pointers are an appropriate choice for C++.

- **Unique naming.** Extension interfaces must be named using integer values or strings. Strings can be read more easily by programmers and simple management tools, but integer values can be smaller and processed more efficiently. To prevent name clashes, interface identifiers can be generated algorithmically. For instance, Microsoft COM uses 128-bit globally unique identifiers (GUIDs) based on the address of the network interface, the date, and the time.

- **Error handling.** Component developers must determine what a component should do when a client requests an extension interface that is not supported. For example, a component could either return an error value or raise an exception. The Implementation section of the Wrapper Facade pattern (47) discusses several strategies for implementing an error-handling mechanism and evaluates their trade-offs.

3.2 Specify domain-independent services. In addition to defining a method for retrieving extension interfaces, the root interface can provide methods for various domain-independent services. Here are two possibilities:

- **Reference counting.** In programming languages that do not provide automatic garbage collection, such as C or C++, clients are responsible for deleting extension interfaces they no longer need. However, multiple clients may share the same extension interface. Components can thus provide a reference counting mechanism to prevent the accidental deletion of resources used to implement extension interfaces.

Reference counting enables components to track the number of clients accessing specific extension interfaces. After an extension interface is no longer referenced by any clients, the resources used
by the component’s implementation of the interface can be released automatically. The Counted Pointer idiom [POSA1] [Cope92] presents several options for implementing a reference counting mechanism.

- **Run-time reflection.** Another example of a domain-independent service is a run-time reflection mechanism. This mechanism allows components to publish information about the specific roles, extension interfaces, and methods they support. Using this knowledge, clients can construct and send method invocations dynamically [GS96]. This enables scripting languages to integrate components into existing client applications at run-time. Reflection mechanisms can be instantiated using the Reflection architectural pattern [POSA1].

### 3.3 Specify domain-specific functionality

The root interface can also export domain-specific functionality if all components implementing the root interface provide this functionality.

- In our management application console, the drawing functionality could be moved to the root interface.

The decision about which domain-specific services to specify in the root interface should normally be deferred until after implementation activity 4 (153). Ideally, all domain-specific functionality should reside in separate extension interfaces. If all components end up implementing a particular extension interface, refactor the current solution [Opd92] [FBBOR99] and move the methods of that particular extension interface to the root interface.

- In our TMN framework example, extension interfaces are identified by unique integer constants. We use Java as the implementation language because it provides automatic garbage collection, which simplifies memory management. The only common functionality therefore required in the root interface—which we call IRoot—is a method that allows client to retrieve any interface they need.

```java
// Definition of IRoot:
public interface IRoot {
    IRoot getExtension (int ID) throws UnknownEx;
}
```
IRoot serves as a generic base interface for all extension interfaces. If a component does not support a particular interface, it throws an UnknownEx exception:

```java
public class UnknownEx extends Exception {
  protected int ID;
  public UnknownEx (int ID) { this.ID = ID; }
  public int getID () { return ID; }
}
```

The unique identifier of the requested interface is passed as an argument to the UnknownEx constructor. This allows a client to determine which interface caused the exception.

Another potential candidate for inclusion in the root interface is a persistence mechanism. However, there are many different strategies and policies for handling persistence, such as managing component state in databases or flat files, which makes it hard to anticipate all possible use cases. Therefore, components can choose to support whatever persistence mechanism they consider appropriate by implementing specific extension interfaces.

4 Introduce general-purpose extension interfaces. General-purpose extension interfaces contain functional roles that must be provided by more than one component and that are not included in the root interface. A separate extension interface should be defined for each role. For example, extension interfaces can be defined to handle persistence aspects of components, as discussed at the end of implementation activity 3 (150).

- Our management application console helps to control and monitor remote network entities via managed objects. Managed objects are implemented as components that send information to the management application console and receive commands from it.
Every managed object therefore implements the following interface, IManagedObject:

```java
// Definition of IManagedObject:
import java.util.*;
public interface IManagedObject extends IRoot {
    public void setValue (String key, Object value);
    public Object getValue (String key) throws WrongKeyEx;
    public void setMultipleValues (Vector keys, Vector values);
    public Vector getMultipleValues (Vector keys) throws WrongKeyEx;
    public long addNotificationListener (INotificationSink sink);
    public void removeNotificationListener (long handle);
    public void setFilter (String expr);
}
```

This example illustrates managed objects that are visualized on a management console. We therefore introduce two additional extension interfaces, IDump and IRender, which are implemented by all components that print debug information on the console or draw themselves.

```java
// Definition of IDump:
public interface IDump extends IRoot {
    public String dump ();
}

// File IDraw.java.
public interface IRender extends IRoot {
    public void render ();
}
```

If a particular general-purpose extension interface must be supported by all components, it may be feasible to refactor the root interface specified in implementation activity 3 (150) and integrate this functionality there. Note, however, that refactoring the root interface may bloat it with functionality or break existing applications, thereby defeating the benefits of the Extension Interface pattern.

7. Note that a component may provide interfaces, such as IManagedObject, that are accessed locally by the client, while their actual implementation resides on a remote network node. By using the Proxy pattern [POSA1] [GoF95], distribution can be transparent to clients. For clarity we assume that all interfaces have local implementations in this example. For information on how proxies can be introduced to support distributed environments, refer to the Distributed Extension Interface variant.
Define component-specific extension interfaces. The extension interfaces needed to export generic component functionality were specified in implementation activities 3 (150) and 4 (153). This implementation activity defines additional interfaces that are specific to a particular component or that are applicable to a limited range of components.

For our TMN framework, we specify the extension interfaces IPort and IConnection. Managed objects that represent ports on a particular host implement IPort:

```java
// Definition of IPort:
public interface IPort extends IRoot {
    public void setHost (String host);
    public String getHost ()
    public void setPort (long port);
    public long getPort ()
}
```

Likewise, objects that represent the connection between two ports implement IConnection:

```java
// Definition of IConnection:
public interface IConnection extends IRoot {
    public void setPort1 (IPort p1);
    public IPort getPort1 ()
    public void setPort2 (IPort p2);
    public IPort getPort2 ()
    public void openConnection () throws CommErrorEx;
    public void closeConnection () throws CommErrorEx;
}
```

Implement the components. The implementation of components involves five sub-activities:

6.1 Specify the component implementation strategy. This activity determines how extension interface implementations should be linked, in accordance with the following three strategies:

- **Multiple inheritance.** In this strategy a component class inherits from all of its extension interfaces.

- **Nested classes.** In this strategy extension interfaces can be implemented as nested classes within the component class. The component class instantiates a singleton instance [GoF95] of each nested class. Whenever the client asks for a particular extension interface, the getExtension() method implementation returns the appropriate nested class object.
• **Separate interface classes.** Extension interfaces can use the Bridge or Adapter patterns [GoF95] to implement separate classes that are independent of the component itself. This strategy is particularly useful when applying the Extension Interface pattern to refactor an existing component that does not implement the pattern.

The ‘tie’ adapter [SV98a] defined in the CORBA IDL mappings for Java and C++ is an example of this component implementation strategy. In CORBA a tie adapter inherits from an automatically-generated servant class, overrides all its pure virtual methods, and delegates these methods to another C++ object, the so-called ‘tied object’. A server application developer defines the tied object.

Regardless of which component implementation strategy is selected, the client is unaffected, because it only accesses the component via references to extension interfaces.

6.2 **Implement the mechanism to retrieve extension interfaces.** When implementing the generic method that retrieves extension interfaces on behalf of clients, ensure that the method implementation conforms to three conventions:

• **Reflexivity.** When clients query extension interface A for the same extension interface A, they must always receive the same reference A.

• **Symmetry.** If a client can retrieve extension interface B from extension interface A, it also must be able to retrieve extension interface A from extension interface B.

• **Transitivity.** If a client can retrieve extension interface B from extension interface A and extension interface C from extension interface B, it must be possible to retrieve extension interface C directly from extension interface A.

Following the conventions above ensures that a client can always navigate from a specific extension interface of a component to any other extension interface of the same component. In other words, each extension interface can be connected with every other extension interface via navigation.
6.3 Implement a reference counting mechanism (optional). If the root interface requires the reference counting mechanism discussed in implementation activity 3.2 (151), specify the resources in the component implementation that must be managed by this mechanism. There are two common options for implementing a reference counting mechanism:

- If each interface implementation requires or uses separate resources, or if there are separate implementations for each interface, introduce a separate reference counter for each extension interface. If a particular reference counter drops to zero, release all resources used by the corresponding extension interface. After the last reference counter has fallen to zero, all resources associated with the component can be released.

- If all interface implementations share the same resources, introduce a global reference counter for the entire component. After this global reference counter reaches zero, release the component’s resources.

The first option can optimize resource management more effectively than the second option, because in the second option all resources must always be available. In the first option, in contrast, extension interfaces and their required resources may be activated and deactivated on demand. Only those extension interfaces and resources actually used by clients are activated. The disadvantage of maintaining extension interface-specific reference counters, however, is their complex implementation within the component.

We can apply reference counting to activate and deactivate extension interface implementations on-demand in our TMN framework. This avoids the unnecessary commitment of resources such as memory or socket handles. For example, when a management application client accesses an extension interface whose reference counter is zero, the component can activate the interface implementation and its resources transparently. When no clients access the extension interface, the corresponding implementation and resources can be deactivated and released selectively. The COM [Box97] component model implements this strategy.
6.4 Select a concurrency strategy. In concurrent or networked systems, multiple clients can access a particular extension interface simultaneously. Implementations of different extension interfaces may share state and resources within the component. Critical sections and state within the component’s implementation must be serialized therefore to provide corruption from concurrent access by clients.

The Active Object (369) and Monitor Object (399) concurrency patterns, as well as the Scoped Locking (325), Strategized Locking (333), and Thread-Safe Interface (345) synchronization patterns, define various strategies and mechanisms for protecting critical sections and state within components.

6.5 Implement the extension interface functionality using the selected component implementation strategy. This implementation activity is largely domain- or application-specific, so there are no general issues to address.

In our TMN framework example we implement components using multiple interface inheritance. Our components do not require explicit reference counting, because Java provides automatic garbage collection.

For simplicity, we do not illustrate the component concurrency strategy. To identify different extension interfaces uniquely, we define an InterfaceID class that enumerates all interface identifiers. These are defined to be integers via the following types:

```java
// Definition of InterfaceID:
public class InterfaceID {
    public final static int ID_ROOT = 0;
    public final static int ID_MANOBJ = 1;
    public final static int ID_DUMP = 2;
    public final static int ID_RENDER = 3;
    public final static int ID_PORT = 4;
    public final static int ID_CONN = 5;
}
```

A more sophisticated implementation could use a repository of interface identifiers. In this case, unique identifiers could be generated automatically by tools to prevent name clashes when different component providers define different interfaces. We could also use a String as the identifier type rather than an int. This might improve the readability and debuggability of the component
system, but at the expense of larger memory footprint and slower lookup time.

One of the component types in the management application console represents a connection between two ports. This component supports the extension interfaces IManagedObject, IRender, IConnection, and IDump. We implement all extension interfaces using Java interface inheritance:

```java
// Definition of ConnectionComponent:
public class ConnectionComponent implements IManagedObject, IRender, IDump, IConnection {
    // <table> contains all properties.
    private Hashtable table = new Hashtable ();

    // <listener> contains event sinks.
    private Hashtable listeners = new Hashtable ();
    private long nListeners = 0;

    private IPort port1, port2;
    private String filterExpression;

    // <IRoot> method.
    public IRoot getExtension (int ID) throws UnknownEx {
        switch(ID) {
            case InterfaceID.ID_ROOT:
            case InterfaceID.ID_MANOBJ:
            case InterfaceID.ID_DUMP:
            case InterfaceID.ID_RENDER:
            case InterfaceID.ID_CONNECT:
                return this;
            default:
                throw new UnknownEx (ID);
        }
    }
}
```

Note how the getExtension() interface uses a switch statement to determine which interface is supported by the component. Had the identifier type been defined as a String rather than an int, we would have used a different type of lookup strategy such as linear search, dynamic hashing, or perfect hashing [Sch98a].

```java
// Definition of IManagedObject:
public void setValue (String key, Object value) {
    table.put (key, value);
}
```
public Object getValue (String key)
    throws WrongKeyEx {
    WrongKeyEx wkEx = new WrongKeyEx ();
    if (!table.containsKey (key)) {
        wkEx.addKey (key); throw wkEx;
    }
    return table.get (key);
}

// Additional methods from <IManagedObject>.
public void setMultipleValues
    (Vector keys, Vector values) { /* ... */ }
public Vector getMultipleValues
    (Vector keys) throws WrongKeyEx { /* ... */ }
public long addNotificationListener
    (INotificationSink sink) { /* ... */ }
public void removeNotificationListener
    (long handle) { /* ... */ }
public void setFilter
    (String expr) { /* ... */ }

// <IDump> and <IRender> methods.
public String dump () { /* ... */ }
public void render () { /* ... */ }

// <IConnection> methods.
public void setPort1 (IPort p1) { port1 = p1; }
public IPort getPort1 () { return port1; }
public void setPort2 (IPort p2) { port2 = p2; }
public IPort getPort2 () { return port2; }
public void openConnection () throws CommErrorEx { }
public void closeConnection () throws CommErrorEx { }

7 Implement component factories. Every component type must
implement a factory that clients can use to obtain instances of the
component type. This involves three sub-activities:

7.1 Define the association between component factories and components.
For every component type, a singleton [GoF95] component factory can
be defined to create instances of this component type. Two strategies
can be applied to implement this association:

• One component factory interface per component type. In this
strategy a separate factory interface is defined for every component
type and used to instantiate the component type. One component
type could offer a component factory interface with a single method
create()). Another component type could offer a selection of
different methods for creating components. The Factory Method
pattern [GoF95] can be used to implement this strategy. It requires clients to handle many different component factory interfaces, however.

- **One component factory interface for all component types.** In this strategy there is only one component factory interface that all concrete component factories must implement. This design enables clients to create different components in a uniform manner. For example, when a client creates a new component, it only must know how to invoke the generic component factory interface. The Abstract Factory design pattern [GoF95] can be used to implement this strategy.

7.2 **Decide which functionality the factory will export.** Regardless of the strategy selected in implementation activity 7.1 (160), the following issues must be addressed when specifying the interface of a particular component factory:

- There could be one or more different methods for creating new components. These creation methods are similar to constructors in object-oriented programming languages, such as C++ or Java, in that they instantiate and initialize component types. Different types of initialization information might be necessary to construct a new component instance. For each of these alternatives, a separate create method is introduced with its own, possibly empty, set of initialization parameters.

- Methods could be available for finding existing components rather than creating them for each invocation. If component instances are already available and if they can be identified uniquely, a factory can be implemented using the Manager pattern [Som97]. In this case, its find() methods are passed a set of conditions as arguments, such as the primary key associated with each component. It then retrieves one or more components that adhere to the condition arguments.

- Clients can specify component usage policies. For example, one policy could provide a singleton implementation for a particular component type. Another policy could determine whether a specific component is expected to maintain its state persistently.

- Life-cycle management support for components is another candidate for the component factory interface. For example,
methods to release existing components might be included in the component factory.

For every managed object in our TMN framework, we provide a separate component factory, implemented as a singleton. The interface IFactory is generic and is supported by all concrete component factory implementations. It contains the `create()` method that clients use to instantiate a new component and to return the IRoot interface to the caller:

```java
// Definition of Factory:
public interface Factory {
    IRoot create();
}
```

Every concrete component factory must implement this factory interface:

```java
// Definition of ConnectionFactory:
public class ConnectionFactory implements Factory {
    // Implement the Singleton pattern.
    private static ConnectionFactory theInstance;

    private ConnectionFactory() { }

    public static ConnectionFactory getInstance() {
        if (theInstance == null)
            theInstance = new ConnectionFactory();
        return theInstance;
    }

    // Component creation method.
    public IRoot create() {
        return new ConnectionComponent();
    }
}
```

7.3 **Introduce a component factory finder.** As the number of component types increases, the problem of how to find the associated component factories arises. One way to resolve this is to define a global component factory finder. This finder could maintain the associations between component types and their component factories, as specified in implementation activity 7.1 (160).

To obtain the component factory for a particular component type, clients must indicate to the component factory finder which component type they require. Component types must therefore be identified uniquely. A common way to implement this identification
mechanism is to introduce a primary key type for every component type. This key type helps to associate component instances with instances of the primary key type uniquely.

For example, each component instance might be associated uniquely with an integer value. This integer value might be passed as an argument to a particular find() method of the component factory, which uses the primary key to obtain the associated component instance. For this purpose, the component factory can apply the Manager pattern [Som97] and map from primary key values to component instances. To simplify client programming, the same primary key type can be used to identify both component instances and extension interfaces, as shown in implementation activity 3.1 (151). In Microsoft COM, for example, globally-unique identifiers (GUIDs) identify both extension interfaces and component types.

When clients request a specific component factory from the component factory finder, the factory finder returns the interface of the component factory. By using this interface, clients can instantiate the components they need. If there is only one global component factory finder in the system, use the Singleton pattern [GoF95] to implement it.

The component factory finder can optionally provide a trading mechanism [OMG98b]. In this case, clients do not pass a concrete component type to the component factory finder. Instead, they specify properties that can be used by the component factory finder to retrieve an appropriate component factory. For example, a client might specify certain properties of extension interfaces in which it is interested to a component factory finder. The component factory finder then locates a component type that implements all the requested interfaces.

Management application clients in our TMN system need not know all component factories. We therefore introduce a component factory finder that is responsible for managing a hash table with component-to-factory associations. Clients need only know where the single component factory finder is located. To identify components uniquely, we apply the same strategy used for interfaces in implementation activity 6.5 (158).
A class `ComponentID` is introduced that contains integer values, each associated with a single component factory:

```java
// Definition of ComponentID:
public class ComponentID {
    public final static int CID_PORT = 0;
    public final static int CID_CONN = 1;
}
```

The component factory finder is implemented as a singleton. It contains two methods that are publicly accessible. The `registerFactory()` method must be called—either by clients or by components—to register component factories with the component factory finder. The `findFactory()` method is used to search for existing component factories.

```java
// Definition of FactoryFinder:
import java.util.*;

public class FactoryFinder {
    // ID/factory associations are stored in a hash table.
    Hashtable table = null;
    // Implement the Singleton pattern.
    private static FactoryFinder theInstance;

    public static FactoryFinder getInstance () {
        if (theInstance == null) {
            theInstance = new FactoryFinder ();
        }
        return theInstance;
    }

    private FactoryFinder () {
        table = new Hashtable ();
    }

    // Component factory is registered with the finder.
    public void registerFactory (int ID, Factory f) {
        table.put (new Integer (ID), f);
    }

    // Finder is asked for a specific component factory.
    public Factory findFactory (int ID) throws UnknownEx {
        Factory f = (Factory) table.get (new Integer (ID));
        if (f == null) throw new UnknownEx (ID);
        else return f;
    }
}
```
8 Implement the clients. Clients use functionality provided by components. They may also act as containers\(^8\) for these components. To implement clients apply the following steps:

- First determine which component functionality they require. For example, determine if there are existing components that cover some or all of the functionality the clients are expected to provide.

- Identify which components should be composed together and determine which components can use other components.

- Determine if there are any subsystems within the client application that might be used in other applications and separate these into new component types.

After evaluating these issues, integrate the client application using the components identified via the analysis outlined in the implementation activities above.

In our example, to localize the initialization of our TMN system we provide a class `ComponentInstaller` within a client that creates all the necessary component factories and registers them with the component factory finder:

```java
class ComponentInstaller {
    static public void install () {
        // First, get the global factory finder instance.
        FactoryFinder finder = FactoryFinder.getInstance ();
        // Ask the factory finder for the comp. factories
        PortFactory pFactory = PortFactory.getInstance ();
        ConnectionFactory cFactory = ConnectionFactory.getInstance ();

        // Register both component factories.
        finder.registerFactory
            (componentID.CID_PORT, pFactory);
        finder.registerFactory
            (componentID.CID_CONN, cFactory);
    }
}
```

8. Typically a component is loaded into the address space of a run-time environment that provides resources such as CPU time and memory to its components. This run-time environment is often called a container, because it shields components from the details of their underlying infrastructure, such as an operating system. In non-distributed use cases, clients can contain components and therefore act as containers.
The main class of the client application defines the methods `dumpAll()` and `drawAll()`. Both methods are passed an array of components as a parameter. They then iterate through the array querying each component for the extension interface IDump and IRender, respectively, calling the methods `dump()` and `render()` if the query succeeds. This example shows that polymorphism can be supported by using interface inheritance rather than implementation inheritance.

```java
// This client instantiates three components: two ports // and a connection between them.
public class Client {
    private static void dumpAll (IRoot components[]) throws UnknownEx {
        for (int i = 0; i < components.length; ++i) {
            IDump d = (IDump) components[i].getExtension((InterfaceID.ID_DUMP));
            System.out.println (d.dump ());
        }
    }

    private static void drawAll (IRoot components[]) throws UnknownEx {
        for (int i = 0; i < components.length; ++i) {
            IRender r = (IRender) components[i].getExtension((InterfaceID.ID_RENDER));
            r.render();
        }
    }
}
```

The `main()` method is the entry point into the client application. It first initializes the TMN system using the initialization component introduced above, then it retrieves the required component factories representing ports and connections between ports:

```java
public static void main (String args[]) {
    Factory pFactory = null;
    Factory cFactory = null;

    // Register components with the factory finder.
    ComponentInstaller.install();

    // access factory finder.
    FactoryFinder finder = FactoryFinder.getInstance();
```
try {
    // Get factories.
    pFactory = finder.findFactory
               (componentID.CID_PORT);
    cFactory = finder.findFactory
               (componentID.CID_CONN);
}
  catch (UnknownEx ex) {
    System.out.println (ex.getID () +
             "not found!");
    System.exit (1);
  }

  // Create two ports and a connection.
  IRoot port1Root = pFactory.create ();
  IRoot port2Root = pFactory.create ();
  IRoot connectionRoot = cFactory.create ();

Note that a client could type cast port1Root and port2Root below
instead of calling the getExtension() method, because the
components use interface inheritance to implement the extension
interfaces. However, this design would tightly couple the client
implementation and the component implementation. If we later
restructured the components to use Java inner classes rather than
multiple interface inheritance, for example, all the client code would
break.

try {
  // Initialize port 1.
  IPort p1 = (IPort) port1Root.getExtension
               (InterfaceID.ID_PORT);
  p1.setHost ("Machine A");
  p1.setPort (PORT_NUMBER);
  // ...Initialize port 2 and connection...

  // Build array of components.
  IRoot components[] = { c, p1, p2 };  
  // Dump all components.
  dumpAll (components);
  // Draw all components.
  drawAll (components);
} catch (UnknownEx error) {
  System.out.println ("Interface "+
         error.getID () + " not supported!");
} catch (CommErrorEx commError) {
  System.out.println ("Connection problem");
}
Shortly after delivering the component-based management application console to their customers, the TMN framework developers receive two change requests. The first request requires each component in the TMN framework to load and store its state from a persistent database. The second request requires a new component with a star connection topology. This topology denotes a set of network elements that are all connected to a central element, yielding a star-like shape.

To satisfy these change requests, the TMN framework developers can apply the Extension Interface pattern:

- To support loading and storing component state from a persistent database, a new extension interface called IPersistence is defined:

```java
public interface IPersistence extends IRoot {
    public PersistenceId store ();
    public PersistenceId persistenceId;}
```

Every existing component is then enhanced to implement this interface. In detail, a component implementor must add all methods defined in the new interface to the component implementation. The amount of work necessary to extend a component with a new extension interface directly depends on the particular extension interface added to the component. The persistence example requires just a few database calls to implement the new interface.

- To support star connection topologies, we define an IConnectionStar interface:

```java
public interface IConnectionStar extends IRoot {
    public void setAllPorts (IPort ports[]);
    public void setPort (long whichPort, IPort port);
    public IPort getPort (long whichPort);
}
```

All TMN framework components must then implement the IRender, IDump, IConnectionStar, IMangedObject and IPersistence interfaces.

The InterfaceID class defined in implementation activity 6.5 (158) is extended with identifiers for the new interfaces.
If a client needs to access the new functionality, it can retrieve any extension interface from the component, query the component for a new extension interface and use the new service:

```csharp
IRoot iRoot = /* ... */; // use any component interface
try {
    PersistenceId storage = /* ... */;
    IPersistence iPersistence = iRoot.getExtension (InterfaceID.ID_PERSISTENCE);
    PersistenceId id = iPersistence.load (storage);
} catch (UnknownEx ue) {
    // Provide exception handling code here when
    // <getExtension> fails to return <IPersistence>.
}
```

**Variants**

**Extension Object** [PLoPD3]. In this variant there is no need for a component factory because each component is responsible for returning interface references to clients. Extension objects are well-suited for components that are built using a single object-oriented programming language, such as C++ or Java, where components derive from all interfaces they implement. Type-safe downcasting can be used to retrieve component interfaces. In these language-specific implementations component factories are not needed because component classes map directly to language classes, which are themselves responsible for instance creation.

**Distributed Extension Interface.** This variant features an additional type of participant, *servers*, which host the implementations of components. Each server contains the factory as well as the implementation of all supported extension interfaces. A single server can host more than one component type. In distributed systems, clients and servers do not share the same address space. It is the task of the server to register and unregister its components with a locator service, so that clients or factory finders can retrieve remote components.

In distributed systems there is a physical separation of interfaces and implementations. Client proxies can be introduced to attach clients to remote extension interfaces transparently [POSA1] [GoF95]. Client-side proxies implement the same extension interfaces as the components they represent. They also shield clients from tedious and error-prone communication mechanisms by forwarding method invocations over the network to remote components. Proxies can be defined so that clients can leverage the *Extension Object* variant outlined above. To enhance performance, client proxies can provide
co-located [WSV99] local implementations of general-purpose extension interfaces to reduce network traffic, in accordance with the Half Object plus Protocol pattern [Mes95].

In distributed object computing middleware [OMG98c], proxies can be implemented automatically via an interface definition language (IDL) compiler. An IDL compiler parses files containing interface definitions and generates source code that performs various network programming tasks, such as marshaling, demarshaling, and error-checking [GS98]. The use of interface definition languages simplifies the connection of components and clients written in different programming languages. To ensure this degree of distribution and location transparency, the underlying component infrastructure can instantiate the Broker architectural pattern [POSA1].

*Extension Interface with Access Control.* In this variant the client must authenticate itself to the extension interface. Client access to an extension interface can be restricted by this method. For example, an administrator might be granted access to all interfaces of a component, whereas another client would be allowed to invoke methods on a subset of interfaces that provided specific functionality.

*Asymmetric Extension Interface.* This variant specifies one distinguished interface that is responsible for providing access to all other interfaces. In contrast to the symmetric case, clients are not capable of navigating from an extension interface to any other extension interface. They must instead use the distinguished extension interface to navigate to any other extension interface. This interface may be provided by the component itself, as defined by the Extension Object variant.

**Known Uses**  
Microsoft’s COM/COM+ technology is based upon extension interfaces [Box97]. Each COM class implementation must provide a factory interface called IClassFactory that defines the functionality to instantiate new instances of the class. When the COM run-time activates the component implementation, it receives a pointer to the associated factory interface. Using this interface, clients can to create new component instances.

Each COM class implements one or more interfaces that are derived from a common root interface called IUnknown. The IUnknown interface contains the method QueryInterface(REFIID, void**).
which allows clients to retrieve particular extension interfaces exported by a component. The first parameter to QueryInterface() is a unique identifier that determines which extension interface to return to a client. If the component implements the interface requested by a client, it returns an interface pointer in the second parameter, otherwise an error is returned.

This activity is called interface negotiation, because clients can interrogate components to determine whether they support particular extension interfaces. COM/COM+ implements the Distributed Extension Interface variant and allows clients and components to be developed in any programming language supported by Microsoft, including Visual Basic, C, C++ and Java.

**CORBA 3** [Vin98] introduces a CORBA Component Model (CCM) [OMG99a] in which each component may provide more than one interface. Clients first retrieve a distinguished interface, the component’s so-called ‘equivalent’ interface. They then use specific ‘provide’ methods to navigate to one of the extension interfaces, called ‘facets’ in CCM. Every CCM interface must implement the method get_component(), which is similar to COM’s QueryInterface() method described above. It is therefore always possible to navigate from a facet back to the component’s equivalent interface.

To obtain a reference to an existing component, or to create a new component, clients access a so-called ‘home’ interface, which is associated with a single component type. This interface represents the component factory interface, as defined by CORBA components and Enterprise JavaBeans. The factory finder within CCM is implemented by the ComponentHomeFinder, whereas EJB relies on the Java Naming and Directory Interface (JNDI) for the same purpose. CORBA components and the Java-centric subset Enterprise JavaBeans (EJB) [MaHa99] use the Asymmetric Extension Interface variant.

**OpenDoc** [OHE96] introduces the concept of adding functionality to objects using extensions. Functionality is provided to retrieve extensions in the root interface, as well as for reference counting. OpenDoc implements the Extension Object variant of Extension Interface.
**Consequences**

The Extension Interface pattern offers the following **benefits**:

*Extensibility.* Extending the functionality of a component should only require adding new extension interfaces. Existing interfaces remain unchanged, so existing clients should not be affected adversely. Developers can prevent *interface bloating* by using multiple extension interfaces rather than merging all methods into a single base interface.

*Separation of concerns.* Semantically-related functionality can be grouped together into separate extension interfaces. A component can play different roles for the same or different clients by defining a separate extension interface for each role.

*Polymorphism* is supported without requiring inheritance from a common interface. If two components implement the same extension interface, a client of that particular extension interface need not know which component actually provides the functionality. Similarly, multiple components can implement the same set of interfaces, thereby allowing them to exchange component implementations transparent.

*Decoupling of components and their clients.* Clients access extension interfaces rather than component implementations. There is therefore no (tight) coupling between a component implementation and its clients. New implementations of extension interfaces can thus be provided without breaking existing client code. It is even possible to separate the implementation of a component from its interfaces by using proxies [POSA1] [GoF95].

*Support for interface aggregation and delegation.* Components can aggregate other components and offer the aggregated interfaces as their own. The aggregate interfaces delegate all client requests to the aggregated component that implements the interface. This allows the aggregate interfaces to assume the identity of every aggregated component and to reuse their code. However, a pre-condition for this design is that the aggregate interface component and its constituent aggregated components collaborate via the `getExtension()` method.

However, the Extension Interface pattern also can incur the following **liabilities**:

*Increased component design and implementation effort.* The effort required to develop and deploy components can be non-trivial. The
component programming effort is particularly tedious when the Extension Interface pattern is not integrated transparently in a particular programming language. For example, it is relatively straightforward to instantiate the pattern using Java or C++. Implementing it in C is extremely complex, however, due to the lack of key language features such as inheritance or polymorphism.

**Increased client programming complexity.** The Extension Interface pattern makes clients responsible for determining which interfaces are suitable for their particular use case. Clients must therefore perform a multi-step protocol to obtain a reference to an extension interface before using it. A client must also keep track of a variety of bookkeeping details, such as interface or instance identifiers and reference counts, that can obscure the client’s core application logic.

**Additional indirection and run-time overhead.** Clients never access components directly, which may reduce run-time efficiency slightly. Similarly, run-time reference counting of initialized components is complex and potentially inefficient in multi-threaded or distributed environments. In certain cases, however, this additional indirection is negligible, particularly when accessing components across high-latency networks.

**See Also** Components and clients may not reside in the same address space, be written in the same programming language or be deployed in binary form, but it still may be necessary to interconnect them. The Proxy pattern [POSA1] [GoF95] can be applied in this context to decouple a component’s interface from its implementation. For a more sophisticated and flexible solution, the Broker pattern [POSA1] can be applied. In this pattern components act as servers and the broker, among its other responsibilities, provides a globally-available factory finder service.

The Extension Object variant of the Extension Interface pattern is introduced in [PLoPD3]. This variant is applicable whenever the object model of the underlying programming language can be used to implement a non-distributed component extension mechanism. In this case,

- Components and component factories map directly to programming language classes
• Component interfaces map to programming language interfaces and
• The retrieval of component interfaces is implemented using type-safe downcasting

Credits We were pleased to co-operate with Erich Gamma on this pattern description. Erich published the *Extension Object* variant in [PLoPD3] and provided our inspiration for documenting the more general Extension Interface pattern described here. We would also like to thank Don Box, *aka* the ‘COM guy’ [Box97], for providing many insights into the Microsoft COM paradigm, and for identifying the fiduciary benefits of networking and distributed object computing in the days when he was Doug Schmidt’s office-mate in graduate school at the University of California, Irvine.