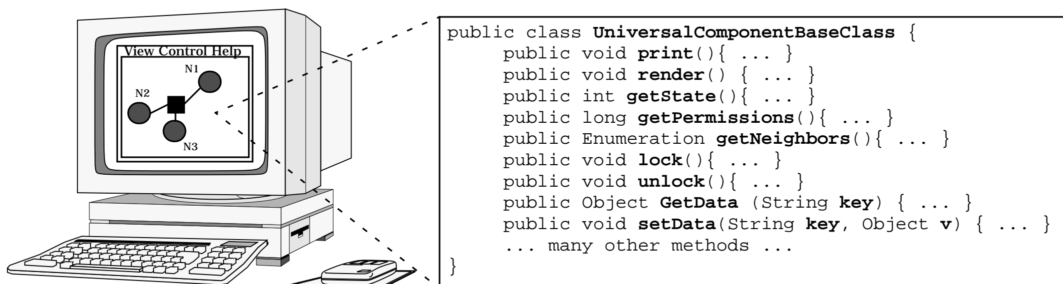
Extension Interface

The *Extension Interface* design pattern[1] prevents bloating of interfaces and breaking of client code when developers add or modify functionality to existing components. Multiple extensions can be attached to the same component, each defining a contract between the component and its clients.

Example Suppose, you are developing a telecommunications management network system to monitor and control remote network elements such as routers or switches. The system must evolve to support new requirements and use cases, therefore each network element is modeled as a separate component. As you are applying the Model-View-Controller pattern, a component is partitioned into two parts: view and control are located on the management console and are responsible to draw the current state of the network element on the screen as well as to interact with the user. The model resides on the network element. It communicates with the view and control to receive commands and send state information. Clients are not aware of this functional partitioning. To separate concerns, generic functionality, such as drawing the network element, is provided by a common base class that all concrete components in the management framework must inherit from.



1. If you ever had some experience with programming Microsoft's Component Object Model, then this pattern will strongly remind you of COM and COM+. The pattern tries to capture the basic ideas behind these technologies at a more general level. As you will see in the *Known Uses* section, the pattern was also applied to other software systems.

© Douglas C. Schmidt 1998, 1999, all rights reserved, © Siemens AG 1998, 1999, all rights reserved

After the project is finished, developers using the framework request that new methods, such as `dump()` and `print()`, should be added to the generic component base class. Over time, incorporating these user requests bloats the component base class with *additional functionality not anticipated in the initial design*. Unfortunately, each time new methods is added to the base class implementation, all user code must be updated and recompiled, as well.

Context Building flexible, stable, and extensible components.

Problem It is hard to anticipate how software components will be used or how they will interact. User demands frequently require modifications and/or extensions to component functionality. However, these changes often happen *after* components have been delivered and integrated into applications. Thus, changes may affect existing user code that is based on the modified components. In addition, the architecture of a component may be destabilized by changing or adding functionality. Consequently, deploying and (re)using components becomes tedious and error-prone.

To avoid these problems, the following *forces* must be considered when building components:

- Components should support inevitable evolution. In particular, when component interfaces remain unchanged, modifications to their implementation should not impact clients.
- Existing client code must never break when developers add new functionality to a component. In the best case it should not even be necessary to re-compile the client code.
- Changing or extending component functionality should be relatively straightforward and should not bloat existing interfaces.
- Remoting of components should be supported. If components and their clients are distributed across network nodes, interfaces and implementation of a component must be physically separated.

Solution Export component functionality to clients via *extension interfaces*. Group each semantically related functionality of a component into a separate extension interface. Provide common and generic functionality in a root interface, especially the functionality required to retrieve a particular extension interface. The root interface is a

© Douglas C. Schmidt 1998, 1999, all rights reserved, © Siemens AG 1998, 1999, all rights reserved

special extension interface all components must support and all other extension interfaces must derive from. To 'derive from an interface' means to syntactically inherit all of its declarations. A component class implements at least one extension interface.

➥ In our example, the functionality necessary for setting and getting the properties of a particular network element is a candidate for an extension interface. ❑

Clients access only interfaces but never component implementations. Hence, clients only see the different roles of a component. A role in this context describes a semantic grouping of functionality that a client can treat as a separate and cohesive unit.[2]

To add new functionality or change existing functionality, integrate new extension interfaces rather than modifying existing ones.

To separate usage aspects of a particular component type from creation aspects, introduce an additional indirection layer: for each component type an associated factory is responsible for creating component instances and returning interface references to the client. The factory is used to instantiate a particular component type on behalf of a client. After instantiating it returns an initial interface to the newly created component. With this interface, the client is able to retrieve all other extension interfaces.

Structure

Clients use functionality provided by components. Sometimes they also act as containers[3] where these components live. They access component functionality via interfaces. Initially, clients can only interact with the factory associated with a particular component type. Once they got the first extension interface from the factory, they may

---

2. For example, an object-oriented class defines a single role that all of its instances support. Another example is an interface which defines a role all implementations support. If a component needs to support a given role, its implementation class must provide an implementation of the interface defining the role. Components expose different roles by implementing different interfaces. Different components may expose the same role by implementing the same interface which allows clients to treat them polymorphically with respect to that particular role.

3. Typically, a component is loaded into the address space of a surrounding run-time environment that provides resources to all of its components. This run-time environment is often denoted as *container*. Containers shield components from the details of the underlying infrastructure. In a non-distributed scenario the client itself *contains* the component and therefore acts as a container.

© Douglas C. Schmidt 1998, 1999, all rights reserved, © Siemens AG 1998, 1999, all rights reserved

access the extension interface functionality to retrieve any other extension interface.

Class Client	**Collaborator**
Responsibility • Implements application-functionality • Accesses factories to create new components • Accesses component interfaces	• Extension Interface(s) • Root Interface • Factory

Class Component	**Collaborator**
Responsibility • Comprises different roles • Implements extension interfaces • Returns initial interface to factory	

Components aggregate different kinds of functionality. They present different roles to their clients. Each role, that is each semantically related group of functionality, is offered by a separate extension interface. In contrast to a pure object-oriented approach, a component may provide multiple extension interfaces. However, it must at least provide one extension interface. Note, that modern object-oriented languages such as Java or Delphi already support the concept of interfaces.

The *root interface* provides three different kinds of functionality:

- *Generic functionality* which all extension interfaces are required to support. For example, functionality that allows clients to retrieve the interfaces they request.
- *Domain-independent functionality* such as methods for the life-cycle management of components.
- *Domain-specific functionality* that should be provided by all components within the domain.

While the root interface must implement generic functionality, the provision of further Domain-independent and Domain-specific functionality is optional.

Each role a component presents to its clients is implemented by a separate *extension interface*. All extension interfaces must implement the root interface functionality. Therefore each of them can take the role of the root interface. Thus, it is guaranteed that each extension

© Douglas C. Schmidt 1998, 1999, all rights reserved, © Siemens AG 1998, 1999, all rights reserved

interface is capable of returning any other extension interface on behalf of a client request.

An extension interface specifies a contract between the client and the component, such that a component guarantees to provide functionality precisely as it is described by the extension interface. Clients must follow the rules for how the interface methods are used, for instance, the correct type of parameters and the order in which methods must be called.

Class Root Interface	***Collaborator***
Responsibility • Provides generic functionality each extension interface must provide	

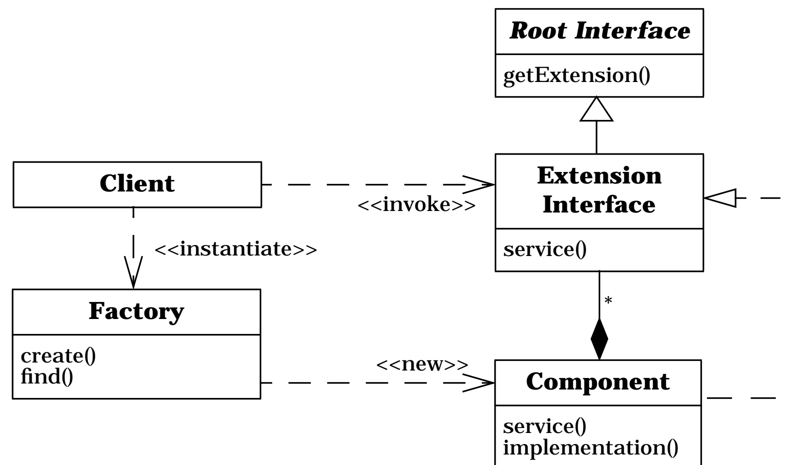
Class Extension Interface	***Collaborator***
Responsibility • Provides specific and possibly generic functionality	

To enable clients to create new components, a *factory* is associated with each component type. A factory is responsible for separating creation and initialization functionality of components from usage aspects. Whenever a client needs to create a new component, it asks the factory to perform this task. The factory might offer the client options, such as allowing it to specify the initial extension interface to be returned after the component has been created. In addition, factories could provide functionality for finding existing components.

Class Factory	***Collaborator*** • Component
Responsibility • Defines functionality for creating components • (Optionally) contains further functionality, e.g., for finding components	

© Douglas C. Schmidt 1998, 1999, all rights reserved, © Siemens AG 1998, 1999, all rights reserved

The UML class diagram below presents the concept of the extension interface pattern:



Note, that this diagram shows the logical relationships among components, rather than the physical relationships. For example, extension interfaces could be implemented using multiple inheritance or nested classes. This should be considered as a mere implementation detail transparent to clients. More information will be provided in the implementation section.

Dynamics

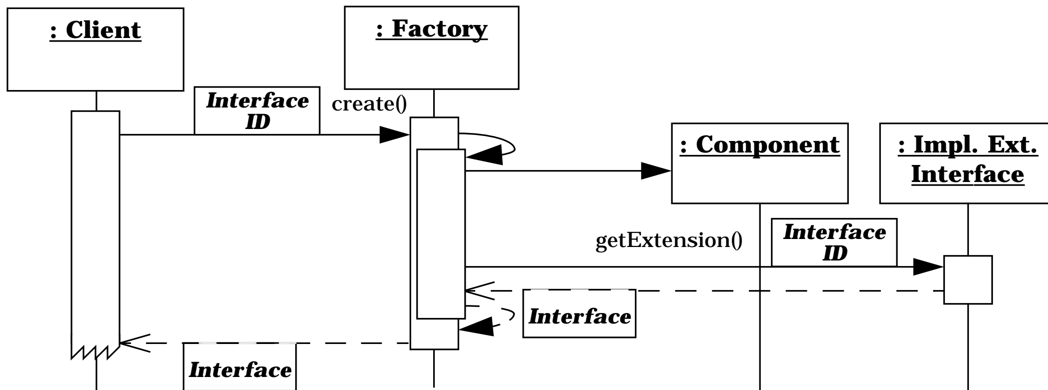
Two scenarios are important for the Extension Interface pattern.

Scenario I describes how clients create new components and retrieve an initial extension interface:

- The client asks the factory to create a new component and to return a particular extension interface.[4]
- The factory creates a new component and retrieves an extension interface as a result.
- The factory asks the root interface for the requested extension interface and then returns the extension interface to the client.

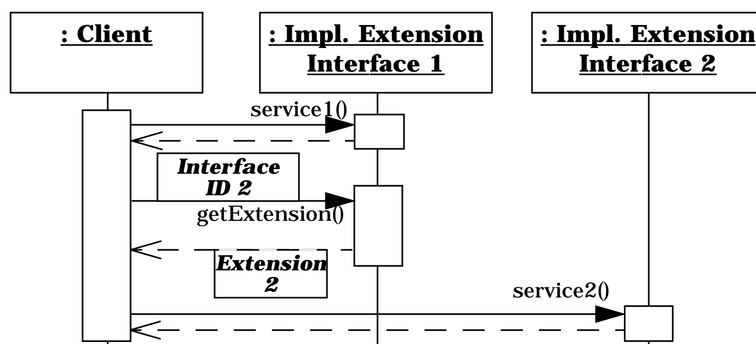
4. Instead of retrieving a specific extension interface, the factory could also return any extension interface to the client. This, however, leads to additional roundtrips in the distributed case.

© Douglas C. Schmidt 1998, 1999, all rights reserved, © Siemens AG 1998, 1999, all rights reserved



Scenario II demonstrates the collaboration between clients and extension interfaces. The component implementation itself is not visible to the client: it only deals with extension interfaces:

- The client calls a method on `Extension Interface 1`.
- Being called by the client, the implementation of `Extension Interface 1` within the component executes the requested method and returns results, if any, back to the client.
- The client calls the `getExtension()` method of `Extension Interface 1`. It passes a parameter specifying which extension interface it is interested in. The `getExtension()` denotes a generic method derived from the root interface, therefore it is implemented by all extension interfaces. The implementation of `Extension Interface 1` within the component locates the requested `Extension Interface 2` and returns it to the client.
- The client calls a method on `Extension Interface 2` which is then executed.



© Douglas C. Schmidt 1998, 1999, all rights reserved, © Siemens AG 1998, 1999, all rights reserved

Implementation

For an instantiation of the Extension Interface pattern, iterate through the following steps:

1 *Analyze your domain.* If your applications and components are restricted to one single application domain or a small set of related domains, the first step consists of a requirements and domain analysis phase. As a result you will get a domain model that serves as a base for identifying generic functionality to be implemented by each component.

➥ For the management console every type of entity that can be controlled is implemented as a separate object. These objects, called managed objects, may represent hardware units such as routers, computers, bridges, or switches. They may also represent software elements such as applications, ports, or connections. They support functionality such as controlling and monitoring their state and behavior, displaying debug information, or visualizing them on the console. ❑

If your applications apply to multiple domains, consider to use an existing component technology such as Microsoft COM+ or OMG CORBA components before trying to invent a ubiquitous component model.

2 *Specify the functionality provided by the root interface.* Add only functionality to the root interface that makes sense to be provided by all extension interfaces. Thus, check carefully for each kind of functionality whether it should be part of the root interface or separated into an extension interface. Keeping this in mind, iterate through the following three substeps:

2.1 Decide which kind of *generic functionality* should be part of the root interface. The root interface must at least include a method for returning extension interfaces to clients. It depends on the programming language which representation suits best for the return type of the method. While Java clients expect to retrieve an object reference, in C++ interface pointers would be an appropriate choice. In this context, you also need to decide how to uniquely produce and identify extension interfaces. For example, you could use numbers or strings. Strings are easily readable by humans, but numbers are much faster to handle for machines. In large systems, it is necessary to prevent name clashes. Thus, interface identifiers could be

© Douglas C. Schmidt 1998, 1999, all rights reserved, © Siemens AG 1998, 1999, all rights reserved

generated by some kind of 'smart' algorithm. For instance, Microsoft COM/COM+ uses 128 bit wide bitfields as identifiers. A special generator takes the MAC address of the available ethernet card as well as date and time information to produce identifiers that are unique over space and time.

Error handling must be addressed as well. For instance, what is a component supposed to do when the client requests an extension interface that is not supported by the component. In this case, the method may either return an error value or raise an exception.

2.2 In addition to generic functionality, the root interface might also offer methods for housekeeping tasks or other commonly used *domain-independent services*. For example, in programming languages such as C++ that do not provide garbage collection, only the clients themselves can inform components when they no longer need particular extension interfaces. Components could support this by implementing some kind of reference counting methods. Use the Counted Pointer idiom [POSA1] for this purpose .

Another example for domain-independent services is the integration of run-time reflection mechanisms. Such mechanisms allow clients to ask components what concrete functionality they provide. Clients may query components at run-time for the methods they implement. Using this knowledge, clients might construct and send method invocations dynamically. This way, you can support builder tools and scripting environments to (visually) integrate components into existing client applications. In order to support reflection mechanisms you could instantiate the Reflection architectural pattern [POSA1].

2.3 *Domain-specific functionality* of your application domain could also become part of the root interface. For instance, in our management console example drawing functionality could be moved to the root interface. However, you can defer this decision to a later point. First, try to place all domain-specific functionality in separate extension interfaces. If it turns out that all components have to implement a particular extension interface, you may refactor your solution and move the methods of that particular extension interface to the root interface [Fow97], [Opd92].

© Douglas C. Schmidt 1998, 1999, all rights reserved, © Siemens AG 1998, 1999, all rights reserved

➥ In our example we are using Java as the implementation language. Java eases the burden of memory management issues. Thus, the only generic functionality required is a method that allows to retrieve any interface we need.

```
// File UnknownEx.java:
public class UnknownEx extends Exception {
     protected int ID;
     public UnknownEx(int ID) {
          this.ID = ID;
     }
     public int getID() {
          return ID;
     }
}
// File IRoot.java:
public interface IRoot {
     IRoot getExtension(int ID) throws UnknownEx;
}
```

The interface `IRoot` serves as a generic base interface all (extension) interfaces must derive from. Extension interfaces are uniquely identified by integer constants. If a component does not support a particular interface, it throws an exception of type `UnknownEx`. In the error case the identifier of the requested interface is passed as an argument to the `UnknownEx` constructor, so that the client can determine which interface caused the exception.

Persistence mechanisms seem also to be a potential candidate for inclusion into the root interface. However, there is a whole bunch of different strategies and policies for handling persistence issues such as managing component state in databases or files. We are not able to anticipate all possible usage scenarios. Thus, we decide to provide persistence mechanisms by separate extension interfaces. Components then have the choice to support whatever persistence mechanism they consider appropriate by implementing specific extension interfaces. ❑

3 *Introduce general purpose Extension Interfaces.* Group together all semantically related methods that implement general purpose functionality which is not included in the root interface. Each of these groups should become an extension interface of its own. For example, you could introduce extension interfaces for dealing with persistence aspects of components as we have discussed above. If you need to

© Douglas C. Schmidt 1998, 1999, all rights reserved, © Siemens AG 1998, 1999, all rights reserved

support different kind of persistence algorithms, consider the use of the Strategy pattern [GHJV95].

➥ The management console helps to control and monitor remote network entities, the managed objects. Managed objects send information to the management console and receive commands from it. Therefore, every managed object must implement the interface `IManagedObject`.

```
// File IManagedObject.java:
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);
}
```

It is important to mention 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. Clients can be oblivious of this fact when proxies are used. For the sake of clarity, we are assuming that all interfaces have local implementations. For the details of introducing proxies to support distributed environments refer to the *Distributed Extension Interface* variant.

We deal with managed objects that are visualized on a management console. Therefore we are introducing two additional extension interfaces, `IDump` and `IRender`. All components that need to print debug information on the screen or to draw themselves implement these interfaces.

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

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

❑

© Douglas C. Schmidt 1998, 1999, all rights reserved, © Siemens AG 1998, 1999, all rights reserved

4 *Define component-specific functionality.* Some of the extension interfaces necessary to cover this functionality were already specified in step 2. Now, you might define additional interfaces that are specific to the component under construction or that are only applicable to a small range of components.

➥ We specify the extension interfaces `IPort` and `IConnection`. Managed objects that represent ports on a specific host implement `IPort`. Objects that represent a physical connection between two ports implement `IConnection`:

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

// File IConnection.java:
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;
}
```
❑

5 *Implement the components.* For this purpose, the following substeps are required:

5.1 The first activity is to *decide how the extension interfaces should be linked* together in the implementation:

- You could provide a component class that inherits from all of its extension interfaces.
- Extension interfaces could be implemented as nested classes of your component class. The component class instantiates one instance of each nested class per extension interface. In addition, it implements the root interface. Whenever the client asks for a particular extension interface, the implementation of `getExtension()` returns the appropriate nested class object.

For the client there is no difference which strategy you are actually using because it only sees extension interfaces.

© Douglas C. Schmidt 1998, 1999, all rights reserved, © Siemens AG 1998, 1999, all rights reserved

5.2 *Implement the extension interfaces.* To do so, you first have to provide the functionality of the root interface. When you are going to implement the query method for the retrieval of extension interfaces be aware that the method implementation should conform to three rules. It must be *reflexive*: when clients ask the extension interface A for the very same extension interface A this should work as expected. The method must be *symmetric*. If you can get extension interface B from extension interface A, you should also be able to retrieve extension interface A from extension interface B. Last but not least, the implementation should be *transitive*. If you can get extension interface B from extension interface A and extension interface C from extension interface B, it must be possible to get extension interface C directly from extension interface A.

➥ To uniquely identify different extension interfaces we provide a class `interfaceID` where all interface identifiers are globally stored. In more sophisticated implementations specific repositories would be a better choice. In this case, unique identifiers could be automatically generated by tools so that name clashes are prevented when different providers provide different interfaces.

```
// File interfaceID.java:
public class interfaceID {
    public final static int ID_ROOT = 0;
    public final static int ID_MANAGEDOBJECT = 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_CONNECTION = 5;
}
```

In the management console a component type is available that represents a connection between two ports. The component supports the extension interfaces `IManagedObject`, `IRender`, `IConnection`, and `IDump`. We decide to implement all extension interfaces using interface inheritance:

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

© Douglas C. Schmidt 1998, 1999, all rights reserved, © Siemens AG 1998, 1999, all rights reserved

```
// <listener> contains event sinks:
private Hashtable listeners = new Hashtable();
long nListeners = 0;
private IPort port1, port2;
private String filterExpression;

// IRoot methods:
public IRoot getExtension(int ID) throws UnknownEx {
     switch(ID) {
          case interfaceID.ID_ROOT:
          case interfaceID.ID_MANAGEDOBJECT:
          case interfaceID.ID_DUMP:
          case interfaceID.ID_RENDER:
          case interfaceID.ID_CONNECTION:
               return this;
          default:
               throw new UnknownEx(ID);
     }
}

// IManagedObject methods:
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);
}

public void setMultipleValues
     (Vector keys, Vector values) {
     // assure keys.size() == values.size()
     for (int i = 0; i < keys.size(); i++) {
          table.put(keys.elementAt(i),
                         values.elementAt(i));
     }
}


public Vector getMultipleValues(Vector keys)
     throws WrongKeyEx {
     boolean wrongKeyDetected = false;
     Vector result = new Vector();
     WrongKeyEx wkEx = new WrongKeyEx();
```

© Douglas C. Schmidt 1998, 1999, all rights reserved, © Siemens AG 1998, 1999, all rights reserved

```
        for (int i = 0; i < keys.size(); i++) {
            if(!table.containsKey(keys.elementAt(i))) {
                wkEx.addKey((String)keys.elementAt(i));
                wrongKeyDetected = true;
            }
            result.addElement(
                table.get(keys.elementAt(i))
            );
        }
        if (wrongKeyDetected) throw wkEx;
        return result;
    }

    public long addNotificationListener
            (INotificationSink sink) {
        listeners.put(new Long(++nListeners), sink);
        return nListeners;
    }

    public void removeNotificationListener(long handle) {
        listeners.remove(new Long(handle));
    }

    public void setFilter(String expr) {
        this.filterExpression = expr;
    }

    // IDump methods:
    public String dump() {
        return "Connection between " + port1.getHost()
            + " on port " + port1.getPort() + " and "
            + port2.getHost() + " on port "
            + port2.getPort();
    }

    // IRenderer methods:
    public void render() {
        System.out.println(" connection ");
        /* ... */
    }

    // 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 {
        System.out.println("Connecting"); }
    public void closeConnection() throws CommErrorEx {
        System.out.println("Disconnecting"); }
}
```
❑

© Douglas C. Schmidt 1998, 1999, all rights reserved, © Siemens AG 1998, 1999, all rights reserved

6 *Provide factories.* Iterate through the following substeps:

6.1 *Define the factory interface.* Each component type might provide its own factory interface. In this case, however, clients need to cope with lots of different factory interfaces. For instance, one component type could offer a factory interface with a single method `create()`, while another component type could offer a broad selection of different methods for creating components. Clients must cope with all of these different factory interfaces. Thus, consider to provide only one generic factory interface which all component factories must implement. This will significantly help clients to handle different components in a uniform way. Whenever a client is going to create a new component, the only thing it will need to know is how to deal with the generic factory interface. The Abstract Factory design pattern [GHJV95] describes how to handle these issues.

6.2 *Decide which functionality the factory is expected to comprise*:

- There could be one or more different methods for creating new components.
- In addition, methods could be available for finding existing components.
- You might supply functionality to specify policies for component usage. For instance, the policy to provide a singleton implementation for a particular component type. Another example for a policy is whether a specific component is expected to keep its state persistent or not.
- Life-cycle management support for components is also a candidate for the factory interface.

It is up to you and your domain what functionality a factory interface will support. There is one factory for each component type, therefore use the Singleton pattern for implementing component factories [GHJV95].

➥ For every managed object a separate factory is provided, which is implemented as a singleton. The interface `IFactory` is introduced as a generic interface to be supported by all concrete factory implementations. It contains the method `create()` that is used to instantiate a new component and to return the `IRoot` interface to the caller:

© Douglas C. Schmidt 1998, 1999, all rights reserved, © Siemens AG 1998, 1999, all rights reserved

```
// File Factory.java:
public interface Factory {
    IRoot create();
}
```

Every concrete factory must implement the factory interface. For instance, in the connection factory:

```
// File ConnectionFactory.java:
public class ConnectionFactory implements Factory {
    // code for using 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();
    }
}
```

❑

6.3 *Introduce a factory finder*. If the number of component types increases, you will have to tackle another problem: how to find the associated factories. For this purpose, you could provide a global factory finder that maintains associations between component types and their factories. In this context, components have to be uniquely identified. To ease the burden of clients use the same identifier type as you did for the extension interfaces (see step 2). When you want to instantiate a particular component type, ask the global factory finder for the component type. You will then get the factory interface of the associated component factory. With this interface you are able to instantiate all the components you need. There is only one global factory finder in the system, therefore use the Singleton pattern [GHJV95] for its implementation.

The factory finder might optionally provide some kind of trading mechanism. In this case, the client does not pass a concrete component type. Instead, it specifies conditions used by the factory finder to retrieve an appropriate component factory. For example, the client might define a set of extension interfaces it is interested in. It is then the responsibility of the factory finder to locate a component type that implements all of the requested interfaces.

© Douglas C. Schmidt 1998, 1999, all rights reserved, © Siemens AG 1998, 1999, all rights reserved

➥ The client should not need to know all component factories. Therefore a factory finder is introduced that is responsible for managing a hash table with component-to-factory associations. Thus, clients only must know where the single factory finder is located. To uniquely identify components we use the same strategy as we did for interfaces. A class `componentID` is introduced that contains integer values, each associated with a single component factory:

```
// File componentID.java:
public class componentID {
     public final static int CID_PORT = 0;
     public final static int CID_CONNECTION = 1;
}
```

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

```
// File FactoryFinder.java:
import java.util.*;

public class FactoryFinder {
     // ID/factory associations are stored in a hash table:
     Hashtable table = null;

     // we are using 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);
     }
```

© Douglas C. Schmidt 1998, 1999, all rights reserved, © Siemens AG 1998, 1999, all rights reserved

```
        // 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;
        }
    }
```

❑

7 *Implement the clients.* Analyze which functionality is necessary to develop the component-based clients. Use a top-down approach for this task: Are there any components available that cover some of the functionality you are expected to provide? Which components should be composed together? Is there any subsystem within your client application that is reusable throughout other applications and should be separated into a new component type? Note, that components themselves might use other components. When you have finished separating concerns plug the client application together in a bottom-up approach.

➥ In the example we provide a class `ComponentInstaller` within the client code that is responsible for creating all necessary component factories and registering them with the factory finder:

```
    class ComponentInstaller {
        static public void install() {
            // first, get the global factory finder instance:
            FactoryFinder finder =
                FactoryFinder.getInstance();
            // ask the factory finder for the port factory:
            PortFactory pFactory = PortFactory.getInstance();
            // ask the factor finder for the conn. factory:
            ConnectionFactory cFactory =
                ConnectionFactory.getInstance();
            // register both component factories:
            finder.registerFactory
                (componentID.CID_PORT, pFactory);
            finder.registerFactory
                (componentID.CID_CONNECTION, cFactory);
        }
    }
```

The main class of the client application provides the methods `dump-All()` and `drawAll()`. Both of them take an array of components as a parameter. Both iterate through the array asking each component for the extension interface `IDump` and `IRender`, respectively, and then call the methods `dump()` and `render()`. This approach shows

© Douglas C. Schmidt 1998, 1999, all rights reserved, © Siemens AG 1998, 1999, all rights reserved

that polymorphism can be supported without requiring implementation inheritance, but using interface inheritance instead.

```
// 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();
          }
     }

     public static void main(String args[]) {
          Factory pFactory = null;
          Factory cFactory = null;
          // register the components to 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_CONNECTION);
          }
          catch (UnknownEx ex) {
             System.out.println("ex.getID() +
                                   " not found!");
             System.exit(1);
          }

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

© Douglas C. Schmidt 1998, 1999, all rights reserved, © Siemens AG 1998, 1999, all rights reserved

```
            try {
               // initialize ports 1 and 2:
               IPort p1 = (IPort) port1Root.getExtension
                                   (interfaceID.ID_PORT);
               p1.setHost("Machine A");
               p1.setPort(1111L);
               IPort p2 = (IPort) port2Root.getExtension
                                   (interfaceID.ID_PORT);
               p2.setHost("Machine B");
               p2.setPort(2222L);
               // initialize connection:
               IConnection c = (IConnection) connectionRoot.
                 getExtension(interfaceID.ID_CONNECTION);
               c.setPort1(p1); // connecting p1 and p2
               c.setPort2(p2);
               // build array of components:
               IRoot components[] = {c, p1, p2};
               // dump all components:
               dumpAll(components);
               // draw all components:
               drawAll(components);
               // open and close the connection:
               c.openConnection();
               c.closeConnection();
            }
            catch (UnknownEx error) {
               System.out.println("Interface "+error.getID()
                                       + " not supported!");
            }
            catch (CommErrorEx commError) {
               System.out.println("Connection problem");
            }
         }
      }
```

Note, that in our example the client could also apply type casting instead of calling the `getExtension()` method, because the components use interface inheritance for implementing the extension interfaces. This, however, introduces a tight connection between the client implementation and the component implementation. If we later restructure the components to leverage the concept of inner classes instead of multiple interface inheritance, all client code would inevitably break. ❑

Variants

The following variants of the Extension Interface design pattern change the indirection layer between components and clients:

In the *Extension Object variant* [PLoP96] the component itself is responsible for returning interface references to the client. This

© Douglas C. Schmidt 1998, 1999, all rights reserved, © Siemens AG 1998, 1999, all rights reserved

variant suits well for components that are built using only one single object-oriented programming language such as C++ or Java. Here, components derive from all interfaces they are going to implement. Type casting may be used as a convenient way to retrieve component interfaces. In language-specific implementations there is no need for implementing factories, because component classes themselves are responsible for instance creation.

In the *Distributed Extension Interface* variant there is an additional type of participants: *Servers* host the implementations of components. They contain the factory as well as the implementation of all supported extension interfaces. A single server can implement more than one component type.

In distributed systems, clients and servers do not share the same address space. Thus, it is the task of the server to register and unregister its components to a central lookup service, so that clients or factory finders are capable of retrieving remote components using the lookup database.

There is a physical separation of interface and implementation in these systems. Therefore client proxies [POSA1] are introduced to transparently attach clients to remote extension interfaces. Client proxies implement exactly the same interfaces as the components they represent. They hide all communication details from the client by transparently forwarding method invocations over the network to the remote component. The proxy could even behave in such a way that clients are able to leverage the Extension Object variant (see above). For optimization reasons, client proxies sometimes provide local implementations of general-purpose extension interfaces to reduce network traffic.

On the server side, proxies shield components and extension interfaces from distribution issues. They represent the clients within the address space of the server.

Proxies may also help to tightly integrate the component model with the object model of the programming language being used to implement components and their clients.

If components and clients are implemented in different programming languages, a high-level definition language might be introduced. This

© Douglas C. Schmidt 1998, 1999, all rights reserved, © Siemens AG 1998, 1999, all rights reserved

language is mapped to the corresponding constructs of the used programming language.

In this context, the question arises who is in charge to create proxies and to translate high-level definition language specifications. Doing this manually, is tedious and error-prone. Thus, compilers are provided which automatically generate the proxies as well as additional plumbing code. The compiler parses through the component declaration and identifies all available component interfaces. From this information, it is able to generate the necessary proxy classes.

To guarantee this kind of distribution and location transparency the underlying component infrastructure often instantiates the Broker architectural pattern [POSA1].

In the *Extension Interface with Access Control* variant the client has to authenticate itself to the extension interface. Access to an extension interface might then be restricted to particular clients. For instance, an administrator might access all interfaces of a client while a regular client should only be allowed to call interfaces that provide the business logic.

In the *Asymmetric Extension Interface* variant one distinguished interface is responsible for providing access to all other interfaces. This interface may be provided by the component itself, thus leading to the *Extension Object* variant.

Example Resolved

A few months after the company delivered the component-based management console to its customers, two update requests have been issued. Each component is expected to store and load its state using a database system. In addition, a new component with a star-like connection configuration is required.

- To solve the first problem, we introduce a new extension interface `IPersistence` that contains methods for storing and retrieving component state. Every existing component is enhanced to implement this interface:

```
public interface IPersistence extends IRoot {
    public long store();
    public load(long persistenceId);
}
```

© Douglas C. Schmidt 1998, 1999, all rights reserved, © Siemens AG 1998, 1999, all rights reserved

- To support star-like connections we first add the new interface `IConnectionStar`. The new component must implement the interfaces `IConnectionStar`, `IManagedObject`, `IRender`, `IDump`, and `IPersistence`:

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

In addition, the class `interfaceID` is extended with identifiers for the new interfaces.

Known Uses

Microsoft's COM/COM+ technology is based upon Extension Interfaces [Box97]. In COM each COM class implementation must provide a factory interface called `IClassFactory` that defines the functionality for instantiating new instances of that class. When the COM runtime activates the component implementation it receives a pointer to the associated factory interface. With this interface clients are able to create new component instances.

Each COM class implements one or more interfaces that are derived from a common base interface called `IUnknown`. `IUnknown` contains the method `QueryInterface(REFIID, void**)` that allows to retrieve particular extension interfaces of the component. In the first parameter clients pass an identifier that uniquely determines which extension interface is to be returned. If the component implements the requested interface, it returns an interface pointer in the second parameter. Otherwise, an error is returned. This is called *Interface Negotiation*, because clients are able to interrogate components whether they support particular extension interfaces.

COM/COM+ implements the Distributed Extension Interface variant and allows clients as well as components to be developed in any programming language of choice.

CORBA 3 [OMG98c] introduces a CORBA based component model where each component may provide more than one interface. A client may retrieve a particular interface either using a specific provide-method or by navigating through all interfaces of a component using one distinguished interface. Thus, CORBA components use the Asymmetric Extension Interface variant.

© Douglas C. Schmidt 1998, 1999, all rights reserved, © Siemens AG 1998, 1999, all rights reserved

OpenDoc [OHE96] introduces the concept of adding functionality to objects using extensions. In the root interface functionality is provided for retrieving extensions as well as for reference counting. In OpenDoc the Extension Object variant is implemented.

Consequences

Whenever the Extension Interface pattern is instantiated you will obtain the following **benefits**:

Extensibility of components: extending the functionality of a component only requires to add new extension interfaces while existing interfaces remain unchanged. Thus, there will be no impact on existing clients, because they will not recognize any difference. Moreover, developers can *prevent interface bloating* by using multiple extension interfaces instead of placing all methods in one single interface.

Separation of concerns with respect to a component's functionality is supported, because semantically related functionality can be grouped together in separate extension interfaces. A component may play different roles to the same client. For each role there is a separate extension interface.

Polymorphism is supported without requiring subclassing. If two components implement the same extension interface, it will be transparent to a client of that particular extension interface which component does actually provide the functionality. Due to the same reason multiple components may implement the same set of interfaces, thus allowing to exchange component implementations.

Decoupling of components and their clients: clients use extension interfaces but not the component implementation directly. Hence, there is no (tight) coupling between the component implementation and its clients. New implementations of extension interfaces might be provided without any impact on existing client code. It is even possible to separate the implementation of a component from its interfaces using proxies.

Support for interface aggregation and delegation. Components can aggregate other components in such a way that they can offer the aggregated interfaces as their own. The aggregate delegates all client requests to the aggregated component that implements the interface. This allows the aggregate to take over the identity of every aggregated component, as well as to re-use their code. Pre-condition for such a design is, however, that the aggregate component and its constituent

© Douglas C. Schmidt 1998, 1999, all rights reserved, © Siemens AG 1998, 1999, all rights reserved

aggregated components collaborate with respect to the `getExtension()` method.

On the other hand you should be aware of the following **liabilities**:

Performance might be restricted. Clients can never access components directly, thus there will be a slightly reduced run-time efficiency which can be neglected in most cases.

The *complexity* of developing and deploying components as well as implementing clients is much more difficult to cope with. This especially holds when the extension interface paradigm can not be tightly integrated into the object model of the programming language used. For example, it is very easy to instantiate the pattern using Java or C++, while implementing it in C is a complex task.

See Also

Sometimes, components and clients do not share the same address space or are provided in binary form. Thus, the problem arises how to connect clients and components that are developed using different programming languages or where components are delivered as binary entities. In this context, the Proxy pattern [POSA1] might be instantiated to separate a component's interface from its implementation. For even more sophisticated and flexible solutions, the Broker pattern [POSA1] might be used. Components act as servers and the broker represents a globally available factory finder.

In [PLoP96] the Extension Object variant of the Extension Interface pattern is introduced.

Credits

We were pleased to cooperate with Erich Gamma for this pattern description. Erich published the Extension Object variant in the PLoP 3 proceedings [PLoP96] which we were able to use as a base for specifying this more general Extension Interface pattern. In addition, we like to give credits to Don Box also known as the 'COM guy'. He helped us to gain some interesting insights into the paradigm behind Microsoft COM.

© Douglas C. Schmidt 1998, 1999, all rights reserved, © Siemens AG 1998, 1999, all rights reserved