The Extension Objects Pattern

Erich Gamma
IFA Consulting
Ceresstrasse 27
CH-8034 Zurich
gamma@ifa.ch

Name

Extension Objects

Intent

Anticipate that an object's interface needs to be extended in the future. Additional interfaces are defined by extension objects.

Motivation

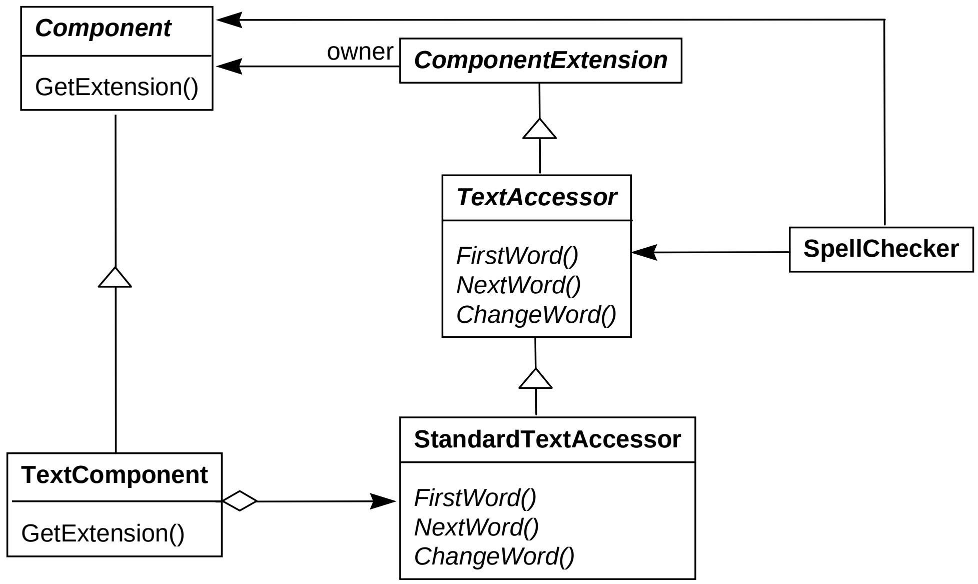
For some abstractions it is difficult to anticipate their complete interface since different clients can require a different view on the abstraction. Combining all the operations and state that the different clients need into a single interface results in a bloated interface. Such interfaces are difficult to maintain and understand. Moreover, a change to a client specific part of an interface can affect other clients that use the same abstraction.

As an example consider compound document architectures like OLE 2 or OpenDoc. A compound document is made of components like text, graphics, spreadsheets, or movies. Therefore the key abstraction of a compound document is something like a *Component*. To assemble components in various interesting ways there is a need for a common interface. Let's assume that this interface is defined by an abstract class Component. This interface provides operations to manage and arrange components in a document.

Now consider a spelling checker for a compound document. It requires an interface to enumerate the words of components that store text. One solution would be to add this interface to Component and have an empty implementation for components that have no text. However, this interface is client specific and would contribute to a bloated Component interface. It would be better if new or unforeseen interfaces could be added separately. Each component should be able to provide an interface for the spelling checker without having to extend the Component interface.

The idea of the Extension Objects pattern is to anticipate such extensions. It proposes to package the spell checker interface in a separate object. Clients that want to use this extended interface can query whether a component supports it.

ComponentExtension is the common base class for extensions. It provides only a minimal interface used to manage the extension itself. For example, it provides an operation to inform the extension that it is about to be deleted. The ComponentExtension subclass TextAccessor defines the interface for accessing text that is used by the spell checker client. Its operations are FirstWord, NextWord for enumerating the words and ChangeWord to replace a misspelled word. To enable different implementations of the spelling checker interface it is defined as an abstract class. For example, a FancyTextComponent can implement this interface with a custom FancyTextAccessor.



Extensions themselves aren't useful – there needs to be a way to find out whether a component supports a specific extension. For the purpose of this example let's assume that we name an extension with a simple string. To avoid conflicts there should be a registry for such extensions. The spell checker can query whether a component provides a certain interface by calling GetExtension(extensionName). If the component provides an extension with the given name it returns the corresponding extension object otherwise it returns nil.

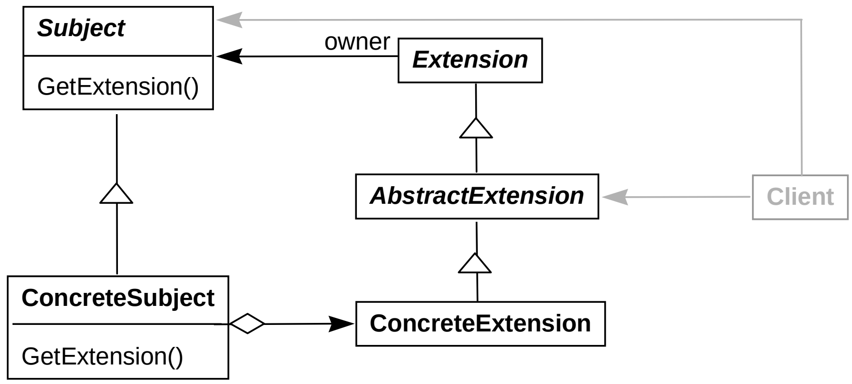
TextComponent overrides GetExtension to return a TextAccessor object when it is asked for an interface with the name TextAccessor. Based on this infrastructure a spell checker for a compound document is implemented as follows. Traverse the components in the document. Ask each component for its TextAccessor extension. If the component returns a corresponding TextAccessor extension object, use it (in C++ after down casting it to a TextAccessor) to spell check the component. Otherwise, skip the component and move on to the next.

Applicability

Use the Extension Objects pattern when:

- you need to support the addition of new or unforeseen interfaces to existing classes and you don't want to impact clients that don't need this new interface. Extension Objects lets you keep related operations together by defining them in a separate class.
- a class representing a key abstraction plays different roles for different clients. The number of roles the class can play should be open-ended. There is a need to preserve the key abstraction itself. For example, a customer object is still a customer object even if different subsystems view it differently.
- a class should be extensible with new behavior without subclassing from it.

Structure



Participants

- **Subject** (Component)
  defines the identity of an abstraction. It declares the interface to query whether an object has a particular extension. In the simplest case an interface is identified by a string.
- **Extension** (ComponentExtension)
  the base class for all extensions. It defines some support for managing extensions themselves. Extension knows its owning subject.
- **ConcreteSubject** (StandardTextComponent)
  implement the GetExtension operation to return a corresponding extension object when the client asks for it.
- **AbstractExtension** (TextAccessor)
  declares the interface for a specific extension.
- **ConcreteExtension** (StandardTextAccessor)
  implement the extension interface for a particular component. Store the state associated with a specific extension.

Collaborations

- A client asks a Subject for a specific extension.
- When the extension exists the Subject returns a corresponding extension object. The client subsequently uses the extension object to access additional functionality.
- If the Subject doesn't support an extension it returns nil to signal that it doesn't support it.

Consequences

Some of the consequences and liabilities of the Extension Objects pattern are as follows:

1. *Extension Objects facilitates adding interfaces.* Adding a new interface to a subject is simplified since this doesn't require any change to the existing subject interface. Extension Objects provides this additional functionality while preserving the key abstraction itself.
2. *No bloated class interfaces for key abstractions.* A key abstraction doesn't become polluted with operations that are specific for a client. This could also be achieved by

subclassing a key abstraction. That is, client specific operations are defined in subclasses. However, this results in a class hierarchy that can be difficult to manage. Inheritance is static and requires creating a new class for each additional interface.

3. *Support for modeling different roles of a key abstraction in different subsystems.* When an abstraction is used across subsystems it often play different roles. Each role requires its own state and behavior. Extension Objects support this by modeling a role with an extension object. By keeping the roles separate one subsystem doesn't have to know the roles used in other subsystems. The abstraction can be passed from one subsystem to the other by passing around the subject.
4. *Clients become more complex.* An extended interface is more complicated to use than one which is provided by the subject itself. A client has to query for the interface and check whether it exists. This introduces additional tests and control paths in your program.
5. *Tension to abuse extensions for concepts that should be explicitly modeled.* It is important that extensions are only used for unanticipated extensions. Otherwise the understandability of a system suffers.

Implementation

A Subject class would be declared like this in C++:

```
class Subject {
public:
   //...
   virtual Extension* GetExtension(const char* name);
};
```

Subject::GetExtension is implemented as:

```
Extension* Subject::GetExtension(const char* name)
{
   return 0;
}
```

Here is a ConcreteSubject that provides a SpecificExtension:

```
class ConcreteSubject: public Subject {
public:
   //...
   virtual Extension* GetExtension(const char* name);
private:
   SpecificExtension* specificExtension;
};
```

The implementation of ConcreteSubject::GetExtension is defined like:

```
Extension* ConcreteSubject::GetExtension(const char* name)
{
   if (strcmp(name, "SpecificExtension" == 0) {
      if (specificExtension == 0)
         specificExtension = new SpecificExtension(this);
```

```
        return specificExtension;
    }
    return Subject::GetExtension(name);
}
```

Finally, to access an extension the client writes:

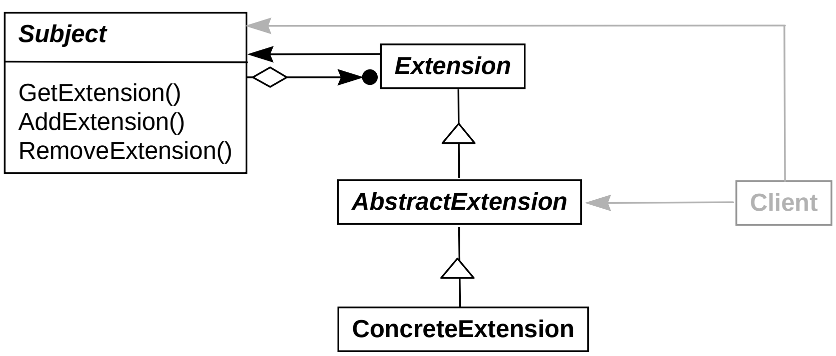
```
SpecificExtension* extension;
Subject* subject;

extension = dynamic_cast<SpecificExtension*>(
                    subject->GetExtension("SpecificExtension")
                        );
if (extension) {
    // use the extension interface
}
```

Before the extension can be used it has to be narrowed to a more specific type. In C++ this is achieved with the dynamic_cast operator.

Here are some implementation issues that arise when you apply Extension Object:

1. *Internal vs. External extensions.* A key issue is how the extensions are created and managed. One solution is to store an extension object in an instance variable of the ConcreteSubject. When the extension is requested it can be returned to the client. This solution assumes that the ConcreteSubject knows its extensions and they are therefore considered as internal. In this solution it is not possible to attach new extensions from the outside. However, this implementation only adds a GetExtension accessor operation and doesn't require any state inSubject. As a consequence the cost of adding support for extensions is small.

   An alternative that enables clients to attach extensions is shown below.

   In this approach the Subject maintains a dictionary that maps extension names to its extensions. Clients can register an extension in Subject by calling AddExtension(extensionName, extension). The dictionary approach enables clients to add new external extensions on demand and doesn't require that the ConcreteSubject knows all its extensions beforehand. An object is populated with different extensions as needed.

   These two approaches don't exclude each other and it is possible to build the dictionary variation on top of the GetExtension accessor implementation.

2. *Identifying extensions*. Extensions need a unique way to identify them. A simple approach is the use of strings. Strings require some infrastructure to avoid duplicates. One way to enforce uniqueness in C++ is to use RTTI and the *typeid* operator to uniquely identify an interface. In this case the GetExtension takes a *type_info* as a parameter. To ask for a particular extension the client passes the type_info of the AbstractExtension class. The typeid operator is used to retrieve a corresponding type_info.

   ```
   AbstractExtension* extension;
   Subject* subject;

   extension = subject->GetExtension(typeid(AbstractExtension));
   ```

   Another language independent way to identify an interface is the use of an interface identifier mechanism.

3. *Demand loading of extensions.* Subject defines a well defined access point to an extension. As a consequence it becomes easy to dynamically load the implementation of an extension when it is needed. The same is true for loading the persistent state associated with an extension. It can be activated on demand once a client requests it.

   A consequence of demand loading is a situation where an existing extension is requested but its dynamic link library is missing. In this case one has to be careful to preserve the persistent state of the extension object. Otherwise the state is lost and cannot be retrieved when the library becomes available in the future.

4. *Defining Abstract Extension*. In C++ all members of AbstractExtension are declared as pure virtuals. In Java an AbstractExtension is defined as an interface.

5. *Freeing Extension Objects.* Another implementation issue in non garbage collected environments is the freeing of extension objects. The Subject hands out a reference to an extension object and therefore has no control over the lifetime of the extension. One approach is to consider the extensions to be owned by the Subject and to destroy them when the Subject is destroyed. An alternative is the use of reference counting. Clients have to increment a reference count when they access an extension and to decrement it once they are done with it.

Known Uses

Support for extensible interfaces is common in Compound Document architectures. Both OpenDoc and OLE provide a corresponding mechanism. In OpenDoc the common base ODObject provides the interface for accessing extensions. OLE builds on top of the Component Object Model (COM). In COM all interfaces of an object are accessed by the *QueryInterface* mechanism.

In the user interface framework of Taligent's CommonPoint the class TView is responsible for managing a visual portion of screen real estate. TView is a key abstraction of the graphical user interface framework. It can be extended with additional interfaces and behaviour without subclassing. TView provides so called *Attributes*. An attribute can be attached to a view and a client can query a view for a specific attribute. To extend a view with additional behavior a client implements an Attribute subclass and attaches an instance to a view. For example, a TNameAttribute can be used to attach an interface for assigning a name to a view. TView uses the dictionary approach to manage the set of extensions.

Related Patterns

Visitor centralizes behavior and enables to add new behavior to a class hierarchy without having to change it. Visitor has similar benefits as the Extension Objects pattern. In contrast to Visitor the Extension Objects pattern doesn't require a stable class hierarchy and doesn't introduce a dependency cycle.

Decorator is another pattern to extend the behaviour of an object. For the client the use of decorated objects is more transparent than extension objects. Decorators work best in situations when the interface is narrow and some existing operations should be augmented.

Adapter supports to adapt an existing interface. The Extension Objects pattern supports additional interfaces. Extension Objects and Adapter can work together in situations where an object needs to be adapted to an extension interface.

Evolution

A first but incomplete version of this pattern appeared in a column that I authored together with Richard Helm [GH95].

References

[GH95] Erich Gamma and Richard Helm. Designing Objects for Extensions, In Dr. Dobbs Sourcebook, #236 pages 56-59, May/June 95