PluggableComponent – A Pattern for Interactive System Configuration

(c) 1999 Markus Völter, voelter@acm.org

Intent

PluggableComponent is a pattern for runtime component exchange. It allows to change parts of a software system and lets the user configure these parts interactively. It also provides for registration and storage of these components. It is especially interesting for framework developers.

Problem and Motivation

In many applications, especially in frameworks, it is necessary to exchange components. The standard approach to achieve this is to use an abstract class as the interface against which the application is programmed, and several concrete subclasses to supply implementations for the abstract interface class. There are several GOF patterns that deal with this kind of component exchange (see Related Patterns section). There, the classes must be available at compile time and they are instantiated according to program state or user selection. However, there are situations when this degree of flexibility is not enough.

The following forces are adressed by this pattern:

- A system needs configured components for different hot spots.
- These components might not be available at compile time or they might not even be available when the system is launched.
- The components might need flexible configuration eventually by a non-programmer using a GUI application.
- Each component type might need different configuration.

Example

Imagine a webshop system[1]. This system runs many different shops at the same time in a single web server, with each shop servicing multiple customers concurrently. Each of these shops has special requirements as to what certain "hot spots"[2] of the webshop‘s workflow should do, for example

- how customers should be authenticated,
- how an order should be processed (send email/fax, insert a record into a database table, etc.),
- how search results should be formatted,
- etc.

To achieve this flexibility, each of these hot spots is filled with a separate component. These components must be exchangeable. Furthermore, it is not possible to shut down the entire shop system just because one of the shops has a component changed, because that would mean that all other shops running in the system would be down, too. It is therefore neccessary to change these components at runtime.

---

[1] A **webshop system** is an e-commerce tool that allows one to build virtual shops in the World Wide Web.

[2] According to the definition in [WP95], a **hot spot** in a framework is a point in the framework that will need application specific implementation when the framework is instantiated, i.e. an application is created based on the framework. It can therefore be „filled“ with application specific code e.g. by registering object with the framework.

An additional requirement is that each of those components needs individual configuration. For example, an order processing component which sends an email to process a customer's order needs to know the destination email adress, an SMTP host, and some other parameters. Another order processing component may want to insert the order into a database table, so it must be informed about database driver, connection URL and tablename, etc. The values for these parameters must be supplied by a non-programmer (called the shop administrator). The different components used and their parameters are illustrated in figure 1a.

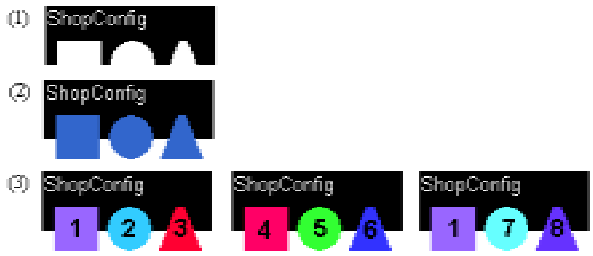
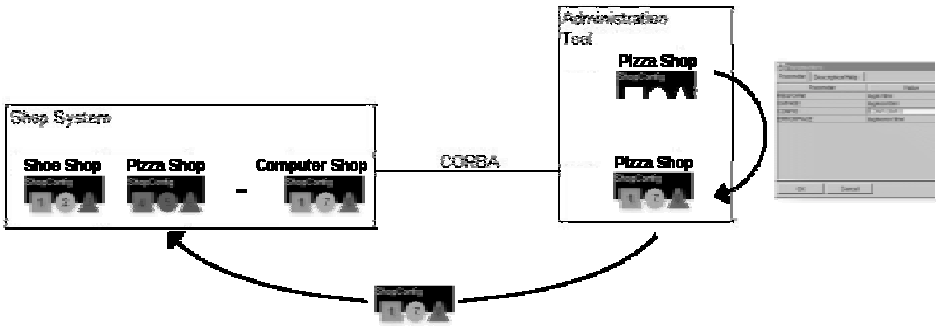


***Figure 1a**: Example shop configuration*

A shop configuration has a couple of hot spots, depicted by the cut-outs in the `ShopConfig` object (1). Each of these hot spots requires a special type of component to be filled in. For example, the first cut out might be the hot spot where the component which displays search results is installed, the second might be used for the component that displays a single product, and the third might be used for the order processing component. The `ShopConfig` object (2) in Figure 1a illustrates this. In (3) the configurations for multiple shops can be seen. Each shop has different classes used to implement the hot-spots, denoted by the color and the number of the shape. All classes of the same shape have in common that they all implement the same interface so that they fit into the hot spot. But each of the classes can implement the interfaces differently, and additionally, each class has different parameters defined (see order process handler example in previous paragraph). The values of these parameters are specific to one concrete instance of such a class, i.e. the component **1** which is used in the first and the third shop has the same parameters, but they have distinct values assigned in the two uses of the component.

There are two applications that are interesting in the context of this pattern:

- The shop system. This is the application which runs the shops, it is implemented as a Java Servlet. It handles sessions, keeps track of a customer's shopping cart etc. In the context of the PluggableComponent Pattern this application plays the role of the **application**.
- The administration tool which the shop administrator uses to administrate the shop system. This is called the **configuration tool**.



***Firure. 1b:**The overall architecture of the example webshop system.*

The component configuration works in the following way (see figure 1b):

1. The administrator creates a new shop, the Pizza Shop in figure 1b.
2. For each hot spot, he selects which concrete class should be instantiated to fill the hot spot.
3. An object of this class is created and its parameters are filled with values. For this purpose a GUI based editor may be used, for example.

4. This process is repeated for all hot spots of the `ShopConfig` object.
5. When all hot spots are filled, the complete configuration, including the values for the components‘ parameters, is transferred to the shop system at runtime.

The PluggableComponent pattern provides an infrastructure for runtime component exchange. In detail, it provides:

- A basic exchangeable component that can be configured with parameters. This component is called a PluggableComponent.
- A registry to manage the different types of PluggableComponents.
- A way to store and transfer configured PluggableComponents.

Applicability

Use this pattern, if

- it is necessary to configure many different "hot spots" of a system at runtime,
- each concrete implementation of a PluggableComponent for a certain hot spot needs different configuration parameters,
- and each instance of a components requires different values for their parameters.

Because this pattern makes use of dynamic class loading and serialization, the language or system used to implement the pattern must support these features. In the given example, the Java programming language is used.

Solution

Provide a base class PluggableComponent which can be configured with parameters. For each hot spot, create an abstract base class that inherits from PluggableComponent, from which the concrete implementations for the hot spots inherit. Make these classes serializable and create facilities to configure, store, transfer and register them.

Structure

For each „hot spot“ of the system an abstract class is defined which inherits from the `PluggableComponent` class. This abstract class provides the interface for the hot spot. Several concrete sublasses are created to provide for different behaviours of a given hot spot. Each class defines, which configuration information (parameters) it needs. When an object has been configured by a user, for example with the help of an editor, it is serialized and stored together with its configuration. When the PluggableComponent should be used, it is deserialized. Figure 2 shows the class structure of a PluggableComponent.

The abstract base class `PluggableComponent` serves as the root for all replaceable components. It holds a collection of parameters and contains all code that takes care of parameter management. These parameters store the configuration information the component needs when it is used in the application. The parameters can be stored as name-value pairs, whereas the value could be of type `String` in a very simple implementation, alternatively some polymorphic type such as `Object` in Java or an Anything [ANY] could be used.

Parameters handling is a two-step process:

1. The name of the parameter must be defined. This happens in the constructor of the classes. Parameters can be defined in every point in the hierarchy, i.e. it is also possible to create parameters in the abstract category classes if all concrete classes have some parameters in common.
2. Later, when the PluggableComponent is configured, the values for the parameters are supplied. That’s why it is not possible to supply the parameter values in the constructor of the parameter. In other words: The parameters are a property of the class, the values for these parameters are specified when an object of the class is configured.

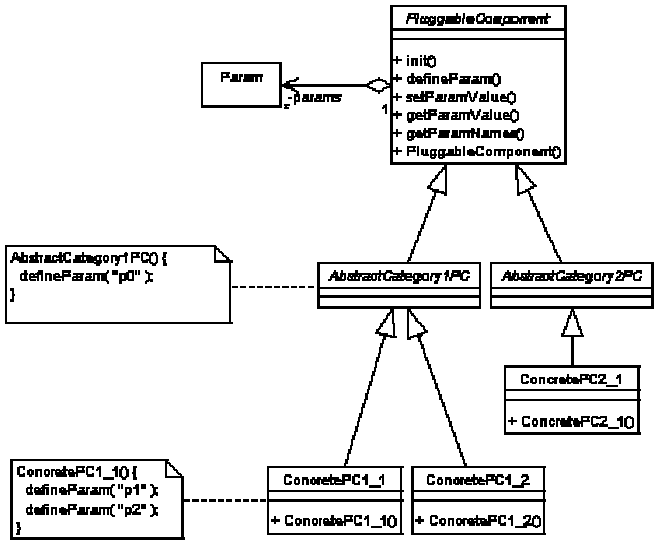


Figure 2: *The basic Structure of a PluggableComponent system*

In a system that uses PluggableComponents there will probably be many different hot spots. We call a PluggableComponent that fits the interface of one such hot spot a Category PluggableComponent. It is modeled as an abstract class, `AbstractCategoryPC`. For each hot spot there will be a corresponding `AbstractCategoryPC` class.

A `ConcretePC` is a concrete (non-abstract) implementation of an `AbstractCategoryPC`. For each `AbstractCategoryPC` there may be many different `ConcretePC`s which may have different behaviour and different sets of parameters. These will later be plugged into the corresponding hot spot with the help of the configuration tool.

To define its parameters, each of these PluggableComponents can call `defineParam(paramName)` in its constructor. The `AbstractCategoryPC`s thereby define all the parameters which all concrete PluggableComponents of the respective category have in common. The `ConcretePC`s define the parameters that are needed by this specific concrete PluggableComponent in excess of those defined in its category. As mentioned above, defining a parameter does not imply that a value for the parameter is specified. This happens later, at configuration time.

To supply values for the defined parameters, the two methods `setParamValue(name, value)` and `getParamValue(name)` are used by the configuration tool. This tool uses a `Configurator` object to do this.

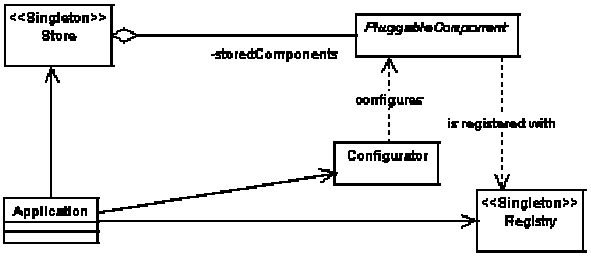


Figure 3: *The overall structure of a system using PluggableComponents*

The configuration tool must know which PluggableComponent categories are available in the system. As described above, these categories correspond to the abstract category classes (`AbstractCategory1PC` and `AbstractCategory2PC`). A special class called `Registry` is used to store this information.

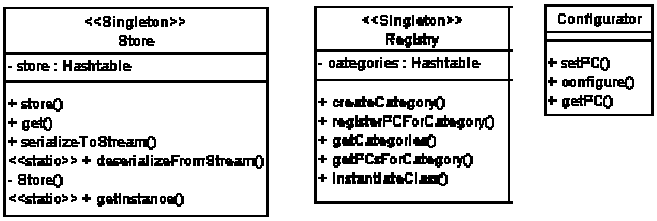


Figure 4: *Other classes in the system*

This class is a singleton, because in each application only one global `Registry` should be used. This `Registry` must store the names of the categories and the available concrete PluggableComponents for these categories persistently. To name the category, the class name of the corresponding `AbstractCategory` class may be used.

The method `createCategory(catName)` creates a new category. To register a concrete PluggableComponent for a category, the class provides the method `registerPCForCategory(catName, className)`. `getCategories()` returns an `Enumeration` of all the category names registered in the `Registry`, and `getPCsForCategory(catName)` returns the class names of all concrete PluggableComponents registered for the specified category `catName`. These two methods are used by the configuration tool to query the system about the PluggableComponent categories and the concrete PCs that are available for those categories. This information can be used for example to create choice boxes to allow the administrator to select concrete PluggableComponents for the system's hot spots.

After the administrator has configured (i.e. supplied parameter values for) a PluggableComponent, it must be stored. The proposed mechanism is to use a central object, the `Store`, which contains all the configured PluggableComponents. The `Store`-object (possibly also a singleton) is then serialized and can then either be saved or transferred.

It stores the PluggableComponents as name/PluggableComponent pairs, i.e. each PluggableComponent is identified in the `Store` by a unique name. It is not defined how this name should be constructed. In a typical application there will be one configured PluggableComponent per hot spot, i.e. per category. Then it may make sense to use the category name as the name for the configured PluggableComponent.

`store()` and `get()` are used to store PluggableComponents in and get PluggableComponents from the store, respectively. A method for removing a PluggableComponent from the store may also be added.

When all the PluggableComponents are configured and stored in the `Store`, the complete store is serialized to a stream (which can be a file, byte array or network socket) using `serializeToStream(stream)`. The application can then deserialize a `Store` by calling the static method `deserializeFromStream(stream)`.

The most important aspect of the PluggableComponent Pattern is the possibility to configure arbitrary PluggableComponents, even interactively. Configuration here means that values for the defined parameters are supplied. For this purpose a `Configurator` is used. This `Configurator` can, for example, present the parameters in a table and let the administrator edit the parameters by entering values (or by using a special editor for each type of parameter, see Variants section). In figure 5, an example editor is shown.

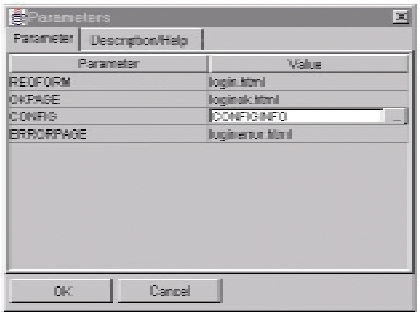


Figure 5: *An example for a PCEditor*

The `Configurator` has three methods. `setPC(pc)` sets the PluggableComponent instance to be configured. `configure()` configures the componet (e.g. by opening the above displayed the editor), and `getPC()` returns the configured PluggableComponent.

Participants

PluggableComponent
Abstract base class for all PluggableComponents. It is responsible for parameter handling and thereby provides the interface against which the `PCEditor` is implemented.

AbstractCategoryPC
The abstract base class for PluggableComponents of a certain category. It is used in the application as an interface against which the application is programmed. It supplies abstract methods for the business logic.

ConcretePC
A concrete implementation for a specific category. It extends the corresponding `AbstractCategoryPC` and fills its abstract business logic methods with useful code.

Param
This class contains information about a parameter for a PluggableComponent.

Registry
Contains information on the categories and their respective concrete PluggableComponents in a system.

Store
Stores the completely configured PluggableComponent objects.

Configurator
Used by the configuration application to set the parameters, may provide some kind of GUI.

Interactions

Creating a new Category

To create a new category, the programmer must derive a new abstract class from `PluggableComponent`. In the constructor of this class parameters can be defined with `defineParam(name)`. These are the parameters that are common to all concrete PluggableComponent classes of the newly created category. The programmer must then decide on a name for that category, one possibility is to use the class name of the abstract category class. The method `createCategory(name)` must then be called on the `Registry` singleton object. If the information on categories and their concrete PluggableComponents is stored in a file, this file can also be modified directly. The format could be like the following:

```
Category1/ConcretePC1
Category1/ConcretePC2
Category1/ConcretePC3
Category2/ConcretePC1
Category2/ConcretePC2
Category3/ConcretePC1
```

Adding concrete PluggableComponents

The concrete PluggableComponents are classes derived from the previously defined abstract category class. Again, these classes can call `defineParam(name)` to define the specific parameters which these classes need in excess of those defined by its abstract category superclass (Don't forget to call the `super()`-Constructor!).

Then, either `registerPCForCategory(name, PluggableComponent)` must be called on the `Registry`, or the file must be modified directly (see above).

Creating and configuring PluggableComponent objects

The configuration tool must allow the administrator to fill the hot spots of an application with a configured PluggableComponent. If the administrator has decided which hot spot should be filled (i.e., which PluggableComponent category is to be used) the system can e. g. prompt the administrator with a combobox where he/she can select which concrete PluggableComponent he/she wants to use. The system then loads the class dynamically and instantiates it. This process is illustrated in figure 6.

When this process has taken place we have an object of the intended PluggableComponent class with its parameters defined internally. Next comes the setting of those parameters.

- The application invokes `setPC(pc)` on the previously instantiated PluggableComponent `Configurator`.
- The `Configurator` then queries the PluggableComponent for all its parameters by calling `getParamNames()`.
- The application then calls `configure()` on the `Configurator`, which displays its windows and lets the administrator edit the parameters. For each parameter the administrator fills in, the `Configurator` calls `setParamValue(name, val)` on the PluggableComponent.
- When the administrator has clicked the OK button, the configuration tool calls `getPC()` and gets a reference to a completely configured PluggableComponent.

This process can be found in figure 7.

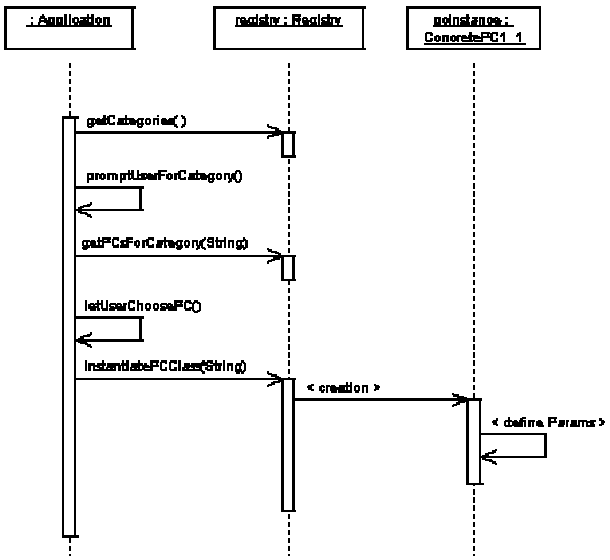


Figure 6: *Selecting and instantiating a PluggableComponent*

Now that we have a completely configured PluggableComponent we can store it in the `Store`. This is simple, we just have to find a name for this concrete PluggableComponent and store it by invoking `store(name, PluggableComponent)` on the `Store`-object. When all PluggableComponents are stored, the complete `Store`-object has to be serialized by calling `serializeToStream(stream)` on the store.

Using configured PluggableComponent - objects in the application

A application (e.g. the shop system) that wants to use the configured PluggableComponents must first load the store (deseriale it from a stream) by calling the static method `deserializeFromStream(stream)` on the `Store`-class which will return a `Store`-object that can be used in the application.

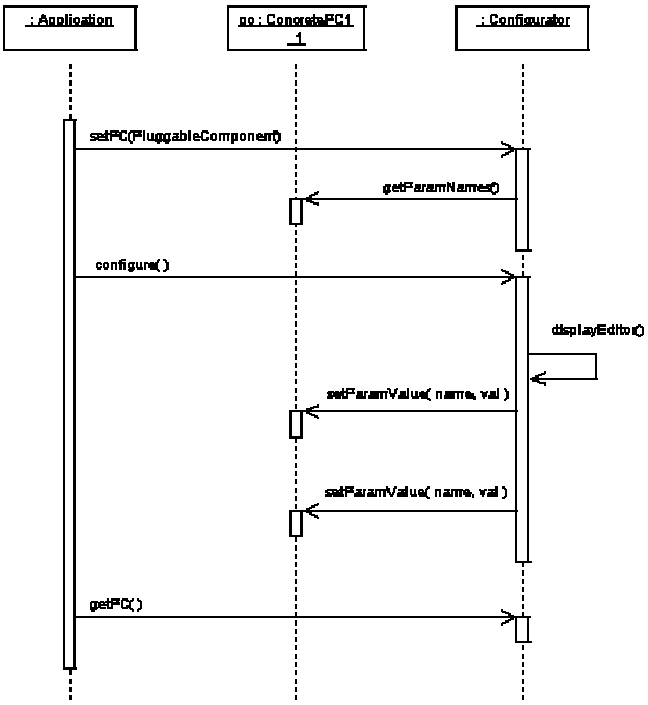


Figure 7: *Editing a PluggableComponent*

The application can then call `get(name)` to retrieve a completely configured PluggableComponent from the store, where `name` is the name that has been given to the configured PluggableComponent at configuration time. See figure 8. All business methods defined in the `AbstractCategoryPC` can now be invoked on the object.

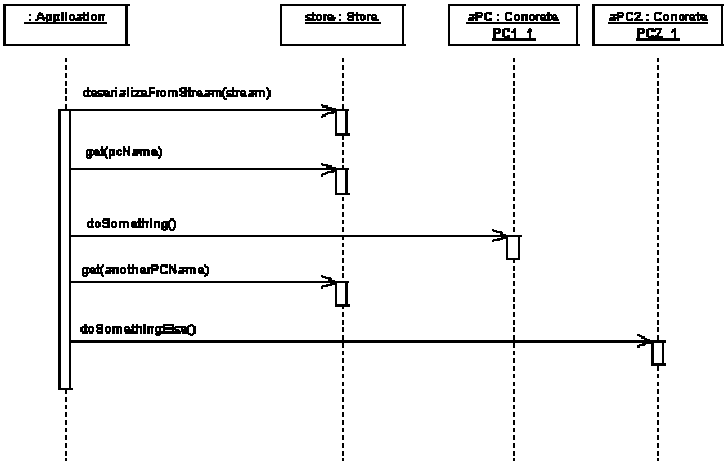


Figure 8: *Using a configured PluggableComponent*

Consequences

Using PluggableComponent has the follwing **benefits:**

- Components can be added and exchanged at runtime

- The configuration of components can be done interactively with a nice GUI or automatically. The `Configurator` does not mandate a specific solution.
- New concrete PluggableComponents can be developed and integrated into the system without recompilation or restart of the system

However, the pattern has the **drawback** that the PluggableComponents stored in the `Store` are real objects, not classes with configuration options. Therefore, if it is necessary to have multiple instances of the same PluggableComponent in the application, a (possibly expensive) copy operation has to be performed.

Example resolved and sample code

As mentioned above, the PluggableComponent Pattern can be used to achieve the necessary flexibility for a webshop system. In this system, PluggableComponent is used to implement many of the key features of a shop, among those is order processing. In the example below we will build the order processing PluggableComponents. We do not implement a `Registry` class, because it is simple to do and we don't need one in the example (only one Category!) . Note, that for brevity, exception handling and error checking is ommitted.

The first step is to build the base class `PluggableComponent`. The following piece of code shows a straight-forward implementation. The parameters are name-value pairs (both `String`s) stored in a hashtable for easy access via their names. Note that this class implements the `java.io.Serializable` interface, so that is can be serialized together with the `Store` in which it will be contained.

```
package de.voelter.pc;

import java.util.*;

import java.io.Serializable;

public abstract class PluggableComponent implements Serializable {
    private Hashtable params;

    public void init() {
    }

    public void defineParam(String paramName) {
            // create new parameter and store it in the parameter list
        Param p = new Param(paramName);
        params.put(paramName, p);
    }

    public void setParamValue(String paramName, String paramValue) {
            // try to get parameter object
        Param p = (Param)params.get(paramName);
        if (p == null) defineParam(paramName);
            // get again, must be present now!
        p = (Param)params.get(paramName);
            // set value
        p.value = paramValue;
    }

    public String getParamValue(String paramName) {
            // try to get parameter object
        Param p = (Param)params.get(paramName);
        if (p == null) return null;
        return p.value;
    }

    public Enumeration getParamNames() {
        return params.keys();
    }

    public PluggableComponent() {
        params = new Hashtable();
    }
}
```

The `Param` class which is used to store the parameters is also quite simple.

```
package de.voelter.pc;

import java.io.Serializable;

public class Param implements Serializable {
    public String value;
    public String name;

    public Param(String paramName) {
        name = paramName;
    }
}
```

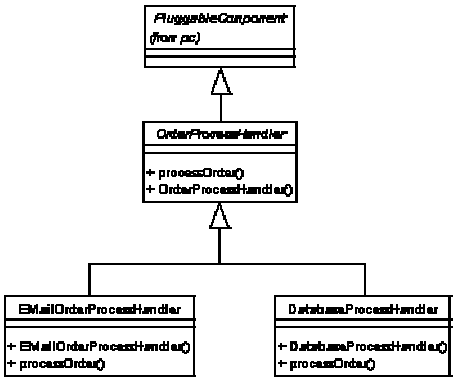


Figure 9: *The PluggableComponents in the example*

In our example we want to implement the `OrderProcessHandler`-PluggableComponents. Two concrete handlers shall be implemented, one that processes an order by sending an email, the other one will insert a record into a table in a database. Figure 9 shows the class hierarchy we have to create. As we only use one category of PluggableComponents, we only have to define one abstract category class, `OrderProcessHandler`. This class defines one abstract business method, `processOrder(Order)` which processes an order. In its constructor, `OrderProcessHandler` defines one parameter, `logFileName`, that is common to all kinds of `OrderProcessHandler`s.

```
package de.voelter.pc.example;

import de.voelter.pc.PluggableComponent;

public abstract class OrderProcessHandler extends PluggableComponent {

    public abstract void processOrder(Order o);

    public OrderProcessHandler() {
        super();
            // Common parameter, which denoting the log file name
        defineParam("logFileName");
    }
}
```

The two concrete PluggableComponent must now be implemented. In their constructors they define the parameters they need. The `EmailOrderProcessHandler` defines `smtpServer`, `receiver` and `sender`, whereas the `DatabaseOrderProcessHandler` defines `DBDriverClassName`, `DBUrl`, `tableName`. Here is the constructor for the `EmailOrderProcessHandler`:

```
public EMailOrderProcessHandler() {
   super();
      // create the necessary params
   defineParam("smtpServer");
   defineParam("receiver");
   defineParam("sender");
}
```

In addition, these classes implement the `processOrder()` method to send an email or to insert a record into a database table, respectively.

The next important class is the `Editor`, the class that serves as the `Configurator` in the example. In our example, the editor works as follows: It displays a table with two columns, parameter name and parameter value. In `setPC(PluggableComponent)` the editor queries the PluggableComponent for its parameter names; `configure()` uses `setParamValue()` and `getParamValue()` to set and get parameter values.

To store the configured PluggableComponents we use a `Store`-object. This class uses a `java.util.Hashtable` to store name/PluggableComponent-pairs and provides the code to (de-) serialize a `Store` object (and all contained PluggableComponents) to (from) a stream. To make this possible, this class is also `Serializable`.

Last but not least there are two simple applications to demostrate the usage of the previously created classes. `AdminApp` creates a PluggableComponent, configures it and stores it in the `Store` which is then serialized to a file.

```
package de.voelter.pc.example;

import de.voelter.pc.*;
import java.io.*;


public class AdminApp {

    public AdminApp() throws Exception {
            // create new concrete PluggableComponent
        EMailOrderProcessHandler eoph = new EMailOrderProcessHandler();
            // create editor
        DlgPCEditor pce = new DlgPCEditor(null);
            // set PluggableComponent in editor, edit it and get it back
        pce.setPC(eoph);
        pce.edit();
        eoph = (EMailOrderProcessHandler)pce.getPC();

            // store PluggableComponent in store...
        Store.getInstance().store("cust1_OPH", eoph);

            // save store to file
        FileOutputStream fos = new FileOutputStream("filename");
        Store.getInstance().serializeToStream(fos);

        System.exit(0);
    }

    // main as usual, creates AdminApp object

}
```

The application restores the `Store` and gets the configured PluggableComponent from it. It then calls `processOrder()` on the `OrderProcessHandler` object. For verification purposes the parameters are printed.

```
package de.voelter.pc.example;

import de.voelter.pc.*;
import java.io.*;

public class UseApp {
```

```
    public UseApp() throws Exception {
            // create input Stream from some file, Store serialized by AdminApp
        FileInputStream fis = new FileInputStream("filename");
        Store.deserializeFromStream(fis);

            // get Store instance and retrieve a PluggableComponent called "cust1_OPH"
        OrderProcessHandler oph = (OrderProcessHandler)
        Store.getInstance().get("cust1_OPH");
            // create Order
        Order order = new Order();
            // process an order with that instance
        oph.processOrder(order);
    }

    // main as usual, creates UseApp object
}
```

Variants and improvements

There are many ways to enhance PluggableComponent, some are described below.

Additional methods

The basic `PluggableComponent` class can be improved by implemeting additional methods. For example, a method `undefParam(paramName)` can be added to the `PluggableComponent` class. This method could be used in the constrctor of a concrete PluggableComponent class to hide a parameter defined in the abstract category class.

Custom parameter editors

A very useful extension is the use of custom parameter editors. Upon definition of a parameter, a class name can be specified. This class has to implement an interface, `ParameterEditor`, which declares one method `editParameter( Param )`.

When the `Configurator` is displayed, it can add an ellipsis button to the parameter's edit field. When the administrator clicks the ellipsis button, the class specified at definition time will be dynamically loaded. An object of this class will be created and the `Configurator` calls the `editParameter()` method of the parameter editor with the current parameter as actual argument. The parameter editor is displayed, and the administrator can edit the value in a more comfortable way (e.g. choosing a value from a list, entering and testing a regular expression). The new parameter value is returned. The parameter editor's class name can be stored in an additional attribute of the `Param` class, and could be accessed by a method `getParamEditor(parName)` in the `PluggableComponent` class.

Reflection

Instead of using name-value pairs of `String` type, `Objects` or Anythings [ANY] can be used. Another possibility is to use regular class attributes if the language supports reflection. To enable the reflection tool to distinguish parameters from ordinary attributes, the parameter-attributes should start with a special prefix, e.g. with `par...`. The `Configurator` can then inspect the PluggableComponent class at runtime and show all `par...` parameters in the PluggableComponent `Configurator`. For each of the parameters a method `setPar...` and `getPar...` must be defined. The `Configurator` can then dynamically create invocations for these methods to set/get a parameter value.

Description, help text etc.

The usability of the pattern can be improved by adding a string that should be displayed in the PluggableComponent `Configurator` instead of the parameter name, or a help message can be specified etc. For each new attribute of `Param`, a corresponding setter/getter method must be added to `PluggableComponent`.

Changes to the Store class

It may be better to not directly serialize the `Store` object, but to have it contain another object that contains the actual PluggableComponents which is serialized.
It is also possible to have the `Store` contain multiple such objects, so that more than one PluggableComponent collection can be loaded.

Known uses

PluggableComponent is used extensively in the n-technology framework [ZBS99] which is the basis for the aforementioned webshop system (n-sell). All replaceable components of the shop are implemented using this pattern.

Many graphical GUI builders use variations of this pattern. For example Borland's (or Inprise's) Delphi. It allows the programmer to drag components onto a form. These components can be customized in the properties editor. The form is then serialized to a .DFM file, which is later linked into the executable. Upon form creation at runtime, the form is deserialized.

IBM's Aglet API is a specification for agent programming in Java. Key to agent programming is the ability to move an agent's code and state from one host to another. The Aglet API uses serialization to transfer the agent's state. Upon creation of an agent, it is initialized with configuration data. On each hop the agent is serialized at the source host and deserialized at its destination host, where it continues its life at the point at which it had been suspended. Like PluggableComponent, the Aglet API uses serialized objects that have been configured earlier. However, it does not use PluggableComponent's parameters. The initialization is not achieved with a GUI configuration tool. Aglets use an array of `java.lang.Objects` to initialize the agent's state. Information on Aglets can be found at [AGLET].

Related Patterns

It is interesting to see how some of the GOF patterns relate to PluggableComponent. They are all intended to change a part of a system, and they all achieve this by defining an abstract class against which the application can be programmed and providing different concrete classes which extend the abstract class. State changes the behaviour of a component depending on its state. Bridge separates the implementation of a component from its interface allowing to change the implementation at runtime. Strategy comes closest to PluggableComponent, because it allows to use different algorithms for a given purpose. PluggableComponent extends Strategy by providing a way to configure these components at runtime and by providing an infrastructure to manage the components in a system.

Peter Sommerlad and Marcel Rüedi presented the Do-it-yourself Reflection pattern collection at EuroPLOP'98. One of the patterns in this collection called Property List describes a way to „attach a flexible set of attributes to an object at run-time" [SR98]. This pattern might be used to implement PluggableComponent's parameters.

A pattern language that deals with Plug-Ins is Klaus Marquardt's „Patterns for Plug-Ins" [KM99]. It focuses on the infrastructure needed to use Plug-Ins in an applications as well as on how Plug-Ins can be distributed and packaged.

To implement the serialization features used in this pattern, the Serializer pattern by Riehle, Siberski, Bäumer, Megert and Züllighoven published in [PLOPD3, chapter 17] will help.

The pattern uses the Singleton Pattern from [GOF94].

If PluggableComponent is implemented using reflection (see Variants section), the Reflection pattern in [BMRSS96] may be interesting.

Although it is not directly related to patterns, the following web site about dynamic architectures might be interesting: `http://www.ics.uci.edu/~peymano/dynamic-arch/index.html`

Acknowledgements

Many thanks to my shepherd Andreas Rueping. His comments and criticism helped me to improve the paper significantly! I'd also like to thank the reviewers at EuroPLOP'99 who gave many useful comments and hints on how to improve this paper, most of them are included in this version.

References

[GOF95] Gamma, Helm, Johnson, Vlissides
Design Patterns
Addison-Wesley, 1995

[BMRSS96] Buschmann, Meunier, Rohnert, Sommerlad, Stal
A System of Patterns
Wiley, 1996

[MRB98] Martin, Riehle, Buschmann (Editors)
Pattern Languages of Program Design
Addison-Wesley 1998

[SR98] Peter Sommerlad, Marcel Rüedi
Do-it-yourself Reflection
Proceedings of EuroPLOP'98

[WP95] Wolfgang Pree
Design Patterns for Object Oriented Software Development
Addison-Wesley 1995

[ZBS99] Zimmermann Business Solutions
The n-technology Home Page
http://www.n-technology.de

[AGLET] IBM Corporation
Aglets Home Page
http://www.trl.ibm.co.jp/aglets/

[ANY] *The Java Any Framework*
http://www.ubs.com/e/index/about/ubilab/ext/publications/e_mar98.htm

[KM99] Klaus Marquardt
Patterns for Plug-Ins
EuroPLOP'99 Proceedings