The Command Dispatcher Pattern

Benoit Dupire and Eduardo B Fernandez
{bdupire@seatech.fau.edu, ed@cse.fau.edu}

Department of Computer Science and Engineering.
Florida Atlantic University
Boca Raton, FL 33431

Can also be called: Command Evaluator Pattern.

Intent

This pattern increases the flexibility of applications by enabling their services to be changed, by adding, replacing or removing any command handlers at any point in time without having to modify, recompile or statically relink the application. By simulating the command-evaluation feature common in interpreted languages, this pattern supports the need for continual, incremental evolution of applications.

Example

Autonomous Underwater Vehicles (AUVs) are small, unmanned, untethered submersibles. There are numerous applications for AUVs, such as oceanographic surveys, operations in hazardous environments, underwater structure inspection and military operations.
We implement the high level controller of this AUV as a Hierarchical Finite State Machine (Figure 1).
A state of the Finite State Machine (FSM) represents a mode of the system, in which some tasks have to be executed, as described in the mission plan. The system takes transitions based on the results of these actions and the progression of the AUV. The system is programmed with high-level commands of the style "set depth 3"

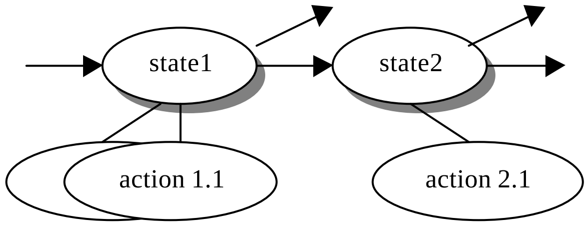

Figure 1: Simplified Finite State Machine for the AUV.

The FSM must be able to fire any type of actions, without knowing anything about them. This problem is addressed by the Command Pattern [GOF95], which encapsulates an action as an object, thereby letting you parameterize clients with different actions. However, there are two problems with this solution.

Memory-Cost: In the case where action 1.1 in Figure 1 is 'set depth 10' and action 2.1 is 'set depth 5', we end up with two different objects which only differ by a single parameter. This approach is non-scalable: for long mission-plans, we may end up with hundreds of 'heavy-weight' Action objects, which will consume lots of memory and may incur run-time overhead. We would therefore like to factorize the common functionalities, parameterize them, and share them, as in the Flyweight Pattern [GOF95].

Limited Flexibility: The command pattern addresses design problems encountered in user interface toolkits in which objects such as buttons carry out a request in response to user input. A button object has a reference to the Action object to fire. If you want to add a new button, or a new functionality for a button, you have to modify the program, and recompile everything. This is not convenient in an embedded real-time system: the design of the architecture has to be developed so that new functionality can be added and included into the software without modifying the code or recompiling it. We want to be able to add new payloads to the AUV, so the software needs to cope with change, and has to be able to evolve along with the needs. There should be no need to stop the mission to update, remove or add a new 'Action Handler' component and there should be no installation to do to benefit from the plug-ins downloaded through Radio Frequency communication.

The 'Command Dispatcher Pattern' describes how to satisfy all these requirements, and to increase the flexibility and the reuse of the system (Figure 2). An Action-Request object is an object that both encapsulates the identity of the action we want to fire and the parameters for this action, i.e. the extrinsic state of the action to undertake. In other words, an Action-Request object is a representation of the action to undertake, which is identified using a key, possibly a string such as 'set_depth'. An Action-Handler is the object that knows how to perform a particular action, and is passed the parameters at run-time. It is therefore a shared object that can be used in multiple contexts simultaneously. The Command-Dispatcher is the object that links the Action-Request to the appropriate Action Handler object. It has a dictionary that contains a reference to all the registered Action-Handlers. The Command-Dispatcher uses the Action-Request's key to find the right entry and dispatches the appropriate Action-Handler. The Action Handler can then perform the requested action.

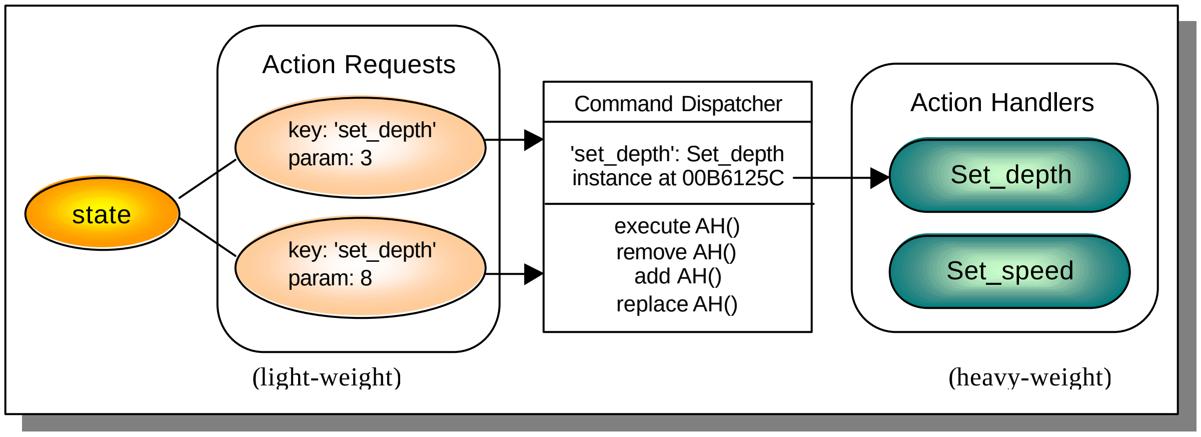

Figure 2: Execution of the Command Dispatcher

This approach allows us to parameterize a State object by an action to perform, but also allows us to parameterize the action to perform and to share it.
Moreover, as the request and the processing are decoupled, they are independent. We are able to add new Action-Handlers to the application, because we can register some new (key, Action Handler) pairs with the Command-Dispatcher, so that this latter one can dispatch new Action Handlers. We can also remove some keys, or just replace the Action Handler to which a key was associated with with another Action-Handler. We bind a name to a function and this binding can be changed dynamically.
The State object triggers an Action Request object, without knowing anything about the actions that will be performed by the customized-side of the application that handles the request. The request that is issued can even be not registered yet with the command dispatcher object. Not only the state does not know about the Action it requests, but the Action-request itself does not know how it will be interpreted by the command-dispatcher. This adds a lot of flexibility.

We just want to have the dynamics provided by the 'eval' or the 'exec' commands in the functional programming formalism. In an interpreted programming language with functional programming capabilities such as Python, this could be done in this way:

```
>>> def set_depth(a):                          # definition of the Action- Handler
        print 'Desired Depth:', a

>>> command='set_depth(3)'                     # definition of the request object (string)
>>> exec(command)                              # calling the interpreter and evaluate the command
Desired Depth: 3                               # processing
```

The first instruction just binds a name to a function object. This binding can be changed at any point in time: we can rebind the name to any other function object.

Context

Applications in which we want to parameterize objects with actions to be performed, parameterize the actions themselves, and share them. The request is independent of its processing and the application dynamically links the command to the command handlers. The application has to dynamically support new or improved command handlers.

Problem

As systems confront changing requirements, they must change as well. Successful systems face unrelenting pressure to change. This pressure comes from defect repair, hardware evolution, operation system evolution, market competition, increasing user sophistication, etc.. It is impossible to predict and cope with these forces in a front-loaded fashion. The system must be able to address these forces. It should be possible to tailor the system to meet the user's need and to integrate new components which add some new functions in the software, or replace old ones.
The Object Oriented paradigm becomes uncomfortable when dealing with problems that are more naturally seen as function requests than as operations on objects. These one-shot behaviors do not seem to belong in the interface of any class. On the other hand, functional programming allows for some very elegant solutions: it treats functions and lambdas as objects that could be assigned to variables. Interpreted languages allows the construction and the interpretation of such functions at run-time. Languages such as Python, Ruby, etc., that mix both paradigms and are interpreted at the same time, are able to combine both programming styles in an easy way. In Python, for example, functions and classes are first class-objets which can be constructed and evaluated at run-time. However, evoking the interpreter at run-time is expensive and may not be suitable to every application. Also, this option is not available for compiled OO languages.
Implementing problems elegantly solved using interpreted languages functionality can be rather difficult in more strict compiled languages that do not offer these nice features. The overall goal of this pattern is to find an equivalent to the command-evaluation feature offered by interpreted languages in the OO paradigm, to be able to cope with changing requirements and software evolution.

Forces

- *Integrating new or improved command implementations at run-time*: The Command-Dispatcher Pattern does retain the dynamics that an interpreted language offers. It supports the dynamic integration of new or improved command implementations with minimal effort. In particular, it allows to bind a name to a new command implementation and thus make it callable. In other words, the system is extensible and mutable [Foo95]: existing behaviors can be changed, new behaviors can be added (such as 'set sonar'). The software is dynamically reconfigurable. The implementation of any command might be changed without recompilation or reinstallation of any part of the system.

- *Share command handlers to allow their use at fine granularities without prohibitive cost:* Adding some new commands to the system won't affect much the amount of memory resources needed if these commands can be replaced by relatively few shared objects once extrinsic state is removed.

Thus, the system can use an apparent large number of objects and effectively use far less because they are shared. The system is hence scalable and does not have high storage costs.

Solution

For each different type of command that can be invoked, introduce a separate command-handler that implements a specified interface. Command handlers can be registered and removed at run time just by providing a name for the registration, independently of the application.
In other words, we provide a way to change the commands-handler associated with a particular command. An interface is provided which allows to modify the linking between a command and a command-handler. This interface is different from the interface through which the system is normally used. It can be thought of as a way of getting under the hood, when necessary.

When the application now issues a command, it notifies the Command-Dispatcher with the name of the command, and this one dispatches the command handler(s) which was associated with this command, so that it can perform the requested command. This allows the decoupling of the dispatching mechanism from the application-specific processing of the command.

As in the 'exec' or 'eval' commands supported by interpreted languages, the system must be able to trigger a command without knowing anything about the requested actions nor about its processing. All requests are identified with a key, possibly a string, and also encapsulate some parameters. The application then evaluate the request and dynamically dispatches it to the corresponding command-handler at run-time, along with the parameters. We decouple here three functions: the client that triggers the command, the command, and the processing of the command.

Structure

An *invoker* (such as a state object of the FSM in the above example) has a list of commands that are to be executed at some point of the application. This list of commands is represented by an aggregation.

The objects of class *Command* represent requests that have to be processed. Each *command* object encapsulates a key (possibly a string). This key can be thought as an ID for this command. It identifies the command so that the appropriate command handler can be fired by the Command-Dispatcher when the latter evaluates the key. The command object also encapsulates the parameters to be passed to the command-handler object to perform the request. These parameters are the extrinsic state of the command-handler.

A *Command-handler* specifies an interface consisting of one or more hook methods [GOF95]. These methods represent the set of operations available to process application-specific command requests that may occur. Common operations include initializing and running the command-handler.

Concrete command-handlers specialize the command handler and implement a specific operation that the application offers. Each concrete command-handler is associated with some commands that identify the operation to perform within the application. In particular, concrete command handlers implement the hook method(s) responsible for processing the requests.
Application developers are thus responsible for implementing the concrete command handlers. These concrete command handlers can be registered or removed dynamically by just adding libraries or modules at run-time.

The *Command-dispatcher* defines an interface that allows dynamic registration or removal of command handlers. It acts as an administrative entity that manages the linking of commands to the appropriate Command Handlers. To register a new Command-Handler, the command-dispatcher is called by the client with a reference to this new command-handler.

The Command-dispatcher then requests to the new Command-Handler the key with which this new Command-Handler will be associated (get_key). This key (a string, most of the time) corresponds to the Id of the command handler. From now on, the command handler is activated, and will be fired if a command with the same name (key) is requested to the command-dispatcher.

The Command-dispatcher is a kind of repository to store all the (key, Command Handler) pairs. When the command-dispatcher is called with a command, it looks at the command's keys and performs a look-up in this repository. If there is a match, it dispatches the appropriate hook method on the handler to process the command. The command-dispatcher also passes parameters from the Command object to the Command-Handler. This ensures the sharing of the Command Handlers. Heavy-weight command-handlers can be therefore replaced by relatively few shared heavy-weight command-handlers once extrinsic state is removed. We end up with far fewer 'heavy-weight' objects than in the 'command pattern' approach because we store the extrinsic states in separate lightweight objects. In the AUV application, for one hundred 'set depth n' command, there will only be one shared command-handler object.

The *Client* imports a Concrete Command Handler object (for example, loads a DLL or a shared library). The client is the mechanism that registers and removes the newly created concrete command-handler object with the command dispatcher. When the client registers a new command-handler to the command dispatcher, this latter one asks the new command handler for its key. The command-dispatcher then adds the (key, command-Handler Ref) to its repository. This key will be used by the command dispatcher to know when to dispatch this particular new command handler.

The current pattern allows for concrete Command handlers to be independent and self-sufficient. Concrete command handlers are just simple functions called by the command-dispatcher, that implement the operation to perform, do some processing, and return a result, if any, to the command-dispatcher. The present pattern is therefore appropriate to execute lambda functions or firing primitives, and the concrete Command handlers do no need to know anything about the other objects of the system.

However, as in the Command pattern, you can notify a *receiver* for the action, and mix both paradigms. For example, set_depth is a command that modifies the depth of the AUV and pass the new depth requirement to the sternplane controller. In this case, the concrete command handler has to specify a receiver that has knowledge to carry out the request. This reference can be stored in the command object as a parameter and is then passed at run-time to the command handler.

The structure of the participants in the Command Dispatcher Pattern is illustrated in Figure 3.
The sequence diagram of Figure 4 shows the interaction between these objects. It illustrates how commands are decoupled from their processing, and how concrete command handlers can be dynamically added.

At run-time, when the invoker triggers a command, there are two possibilities:

- The invoker can pass a reference of the Command object to the command-dispatcher, as shown in Figure 4, and then the Command-dispatcher uses this object to get the key and the parameters of the action to undertake.
- Pass the data (the key and the parameters) to the command-dispatcher. In other words the invoker asks the Command object for the key and the parameters and passes everything to the Command-dispatcher, which can then dispatch the right Command Handler with the parameters. This solution might be better than the first one, for two reasons. The first one is that the first solution requires the Command object to define a more elaborate interface to its data, which couples tightly the Command-dispatcher and the Command. The second reason is in the case where Command only stores the key, or just the key and a few parameters, there is no need now to implement Command as a separate object, it can just be stored as an attribute of the invoker object to save space.

The extrinsic data requirements will determine the best technique, as long as consistency is maintained.

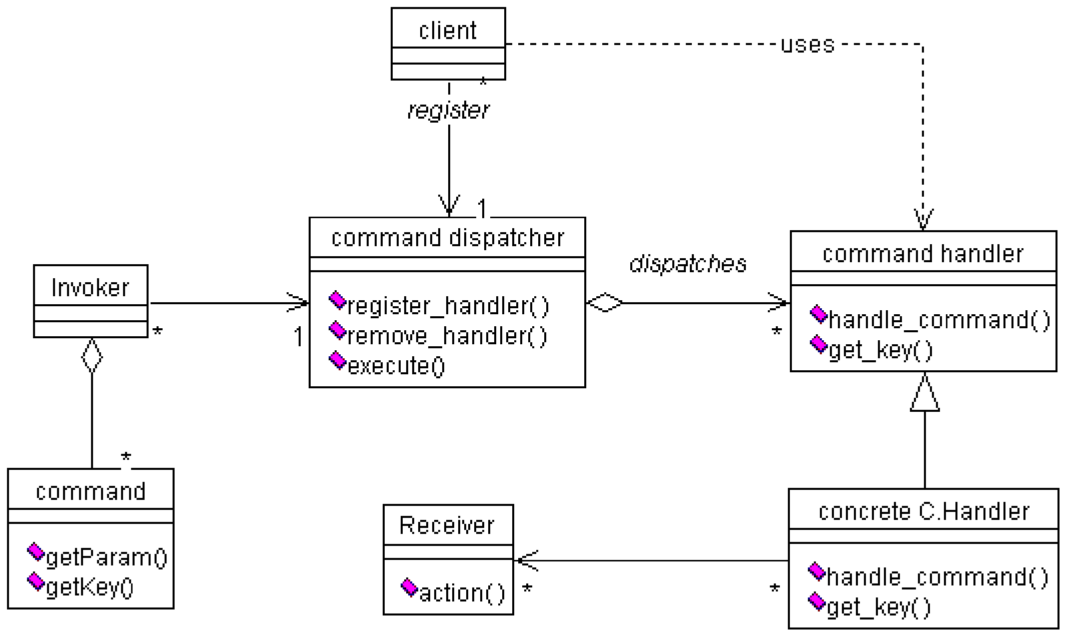

Figure 3: Class Diagram for Command Dispatcher Pattern.

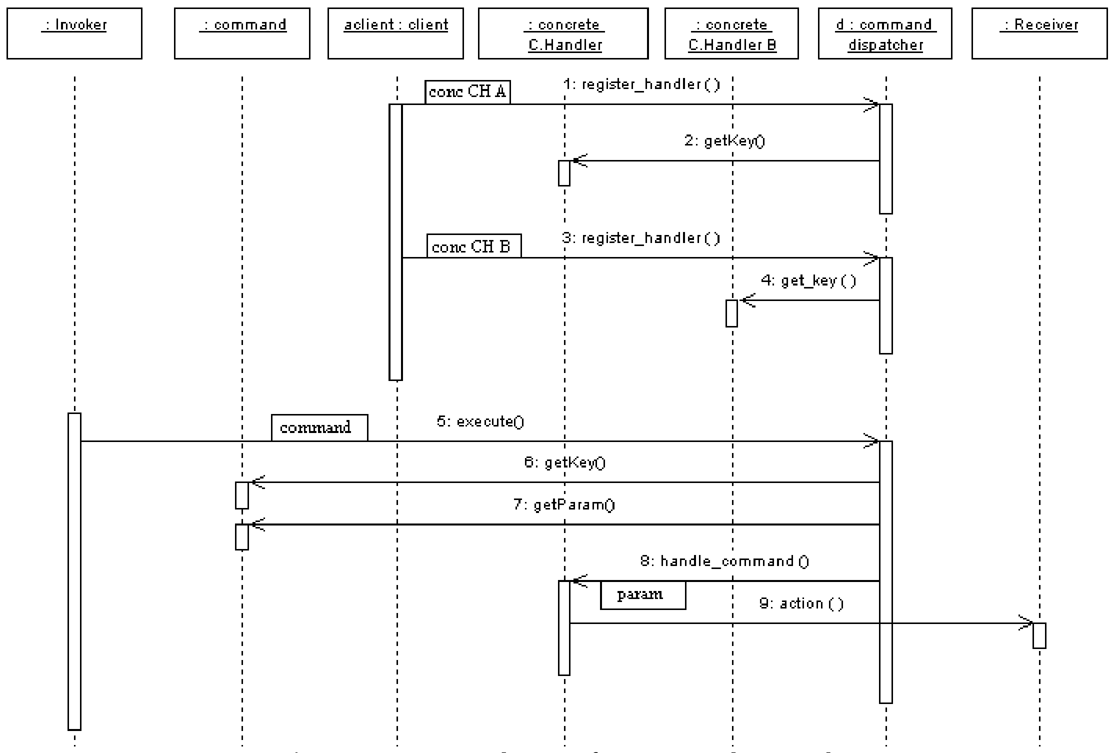

Figure 4: Sequence diagram for Command Dispatcher.

Consequences

Benefits

Flexibility/ dynamics:
One of the advantages of this design is that it is very flexible. The invoker object has no idea of the Command it triggers, and the command is oblivious of how the request is processed. The way objects interact changes frequently so the objects have to be expressed such that, in order to minimize coupling, they do not reference each other explicitly, but by name. We bind a name to the command Handler. The command Handler to which this name refers can be changed at run-time. When you trigger function_foo, the command-dispatcher runs the precise object to which the name "function_foo" refers at the time.
Thus, this pattern increases the flexibility of applications by enabling their services to be added, modified or configured at any point in time. To create a new Command handler, one has just to conform to some specifications, and to implement operations defined in the Command Handler interface.

Sharing
As in the Flyweight pattern [GOF95], we make a distinction between intrinsic and extrinsic state of the command handler. Intrinsic state is stored in the Command Handler object: they are all the parameters that are independent of the Command Handler context, thus making it sharable. Extrinsic state (i.e.: '3' in "set depth 3') are the particular parameters for this command, which vary on a command basis, and therefore can't be shared. Extrinsic state can be stored in the Command Object, which has far smaller storage requirements. Storing this information externally to each Command Handler is far more efficient than storing it internally, which would require having multiple large Command Handler objects, in which only a few parameters may vary. Objects are shared, so they are not instantiated directly by the invoker.

Security
The extra level of indirection may be used to support security enforcement [Sum97]. The Command Dispatcher can check whether the invoker of a command has the right to execute this command. Often, dynamic addition of services and dynamic configuration is undesirable due to security restrictions. This pattern can be easily extended so that the command-dispatcher can check the rights of the invoker before undertaking the request.

Drawbacks

Invalid commands
Some invoked commands may not have been registered with the command dispatcher. The command dispatcher might have to return an error signal or just ignore the request. Another issue happens when a request cannot be satisfied due to violation of security constraints.

Naming
Object handlers should be registered with meaningful names. Good names provide insight into the purpose and design of a system, reveal its inner workings. The key with which a command handler is registered in the Command Dispatcher object should be an Intention Revealing Selector name, and should describe what the method is trying to accomplish.

Lack of determinism.
The pattern makes it hard to determine the behavior of a command handler until run-time. For real-time systems, such as the AUV application, we have to make sure that the command handlers are short enough, in order not to exceed their periodic processing constraints, or other command-handlers will miss their deadlines and the system will not behave predictably.

Reliability.
In the same way, we have to make sure that newly added command handlers do not jeopardize the reliability of the system. For example, a faulty command-handler may crash, thereby corrupting the system, the shared memory in our AUV system. This is particularly problematic with real time systems such as the AUV application.

Increased overhead
The pattern adds an extra level of indirection to execute a command, compared to the Command Pattern [GOF95]. This may be undesirable or an unnecessary overhead in time-critical applications.

Extensions to the basic pattern.

We consider three possible extensions to the basic Command Dispatcher Pattern. These act in the following way:

- The Macro Command extension supports the definition of a command that is used to sequentially execute multiple primitives (multiple command-handlers, to be accurate)
- The Component Integration extension supports the dynamic binding of a component that defines several command-handlers at the same time.
- The Command handler state extension just adds some attributes to the Command handler, so that it keeps track of the latest calls.

Macro Commands

Sometimes a particular command requires executing a sequence of operations. This is useful to structure a system around high-level operations built of primitives operations. Such a structure is common in information systems that support transactions.
High level operations can be described as macro commands (or composite commands) consisting of several commands. However, this approach is not the most efficient.
The problem can be modeled in a different way: we can associate a command to a composite Command-handler (macro command-handler), that executes some other command-handlers (Figure 5). Therefore, the system complexity remains on the customized side that processes the request, and not on the side of the generic application, which do not know if the request it issued was a simple or a macro command.

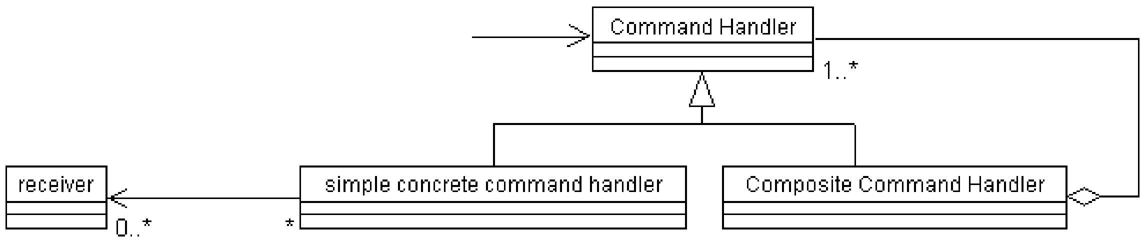

Figure 5: Macro Command Handler

Component Integration

The Command Dispatcher Pattern defines a fine granularity approach. Command handlers can be integrated dynamically one by one into the application. Often, it is more convenient to group several related command handlers in a component.

Command Handler State

We can add a state to the command_handler, to know for example how it performed, or to store state for reversing its effects ('undo'). This is of course not applicable to real time systems, such as the AUV application, where it makes no sense to undo an operation.

Implementation

Python is a 4th generation language, which is both very high level and object-oriented. As Java, it is interpreted but offers functional programming constructs and is highly dynamic. As shown below, this makes trivial the implementation of this pattern in Python.

```
my_cmd_handler =  "f= lambda: sys.stdout.write('Hello World!')"
exec (my_cmd_handler)

cmd = "f()"
exec(cmd)                    # Hello World!
```

The first two lines define a function (which outputs the useful 'Hello World!' message) from a string, and then the two other lines evaluate another string which would perform the function call.

A problem which occurs here is that the Python interpreter is called at run-time to interpret the string, making the program very slow, which is not very convenient in a real-time system.

Thus, it would be interesting to implement the pattern using Python's object oriented features. As Python is highly dynamic and supports late binding, all Python methods are virtual in the C++ terminology, and thus there is no need to define an abstract Command Handler class.

```
class CommandHandler:
       def __call__(self, parameters):
              # ...
       def __init__(self):
              # ...


class Command:
       def __init__(self, key, params = []):        # constructor
              self.key= key
              self.params = params

       def getKey(self):
              return self.key

       def getParams(self):
              return self.params

       # ...


class CommandDispatcher:
       cmdDict = {}

       def register_handler(self, key, cmdHandler):
              # add the new entry to the dictionary
              self.cmdDict[key]= cmdHandler
              cmdHandler.init()


       def remove_handler(self, key, cmdHandler):
              if self.cmdDict.has_key(key) and self.cmdDict[key]== cmdHandler:
                     del self.cmdDict[key]


       def execute(self, key, argument):
              if self.cmdDict.has_key(key):
                     # call the Command Handler
                     self.cmdDict[key](argument)
              else:
                     # no Command Handler registered for this command
                     pass
```

Moreover, classes are first-class objects in Python. It is therefore possible to implement the Command Dispatcher as a factory that instantiates a command-handler object when it is used.

<u>Determine the implementation of the Command handler.</u>
As in the Reactor pattern [POSA2], a Command-handler can be either implemented as an object or as a function.

1. *Concrete Command Handler object*. In object oriented applications a common way to associate a command handler with a command is to create a command handler object. Using an object as the dispatching target makes it convenient to subclass command handlers to reuse and extend existing components. Similarly, objects make it easy to integrate some intrinsic state and methods of a service into a single component.

2. *Command Handler functions*. Another strategy for associating a command handler with a command is to register a pointer to a function with the command-dispatcher rather than an object. Using a pointer to a function as the dispatching target makes it convenient to register callbacks without having to define a new subclass that inherits from an command handler base class.

As Python is dynamically typed, it allows us to do both at the same time. In the above implementation, variable cmdHandler can either be a reference to the Command Handler object (because we used the special __call__ method, so we can call the object like a function), or a reference to a function, or a reference to a class. As a matter of fact, functions and classes are first-class objects in Python. In the later case, the call to the Command Handler will instantiate the class, so if no Command targets a particular Command Handler, this one is not even instantiated.

In Java, you can compile a new Command Handler separately, which will gives you a new .class file. Then, to install the new Command Handler, you can copy this file in a predetermined directory. From this time, the Client object can determine that some new Command Handlers are available, and register them with the Command Dispatcher. The application will now be able to use these new command handlers, just using new Command objects built from a string.

Related patterns

<u>Command Pattern [GOF95]</u>
A major difference with the command pattern is the dynamics allowed by the Command Dispatcher pattern. The Command pattern is rigid: the invoker has a direct reference to a service procedure. This constraint is inflexible or costly for some applications. The dynamic reconfiguration aspect is therefore not addressed by the Command Pattern. Conversely, the Command Dispatcher pattern separates a request for a service from its execution. New command handler objects can therefore be added on the fly and an invoker can fire them, because there is no direct link. This adds a lot of flexibility: exactly the same flexibility that we have with the APPLY command in interpreted languages. Commands Handlers can be added at run time and we can have new invocations without recompiling anything.

<u>Component Configurator [POSA2]</u>
The Component Configurator pattern allows an application to link and unlink its component implementations at run-time without having to modify, recompile, or statically relink the application. The context is the one of an application in which components must be initiated, suspended, resumed and terminated as flexibly and transparently as possible. Most of these forces are also addressed in the present pattern.

However, the Component Configurator pattern is more about replacing components of an application. It takes for granted the kind of services the application has to support and assumes it is set. It is concerned with modifying their associated implementations to handle them in a more efficient way. It therefore only addresses the maintenance issue of an application's lifecycle, not the expandability issue.
In the current pattern, we do not assume anything about the functionality the application deals with, we can add and remove components dynamically.
The command Configurator pattern and the command Dispatcher pattern do not address the same kind of problems, but they can be easily combined to implement configurable command interpreters.

Strategy Pattern[GOF95]
The Strategy pattern defines a family of algorithms, encapsulates them in some Strategy objects with a common interface, and, depending on the context in which they operate, make them interchangeable. Strategy lets the algorithm vary independently from clients that use it [GOF95]. However all possible implementations of the different algorithms must be included at compile-time in order to support the selection of different strategies at run-time. This pattern does not address the maintainability issue.

Command Processor [Som95]
The command Processor design pattern also separates a request for a service from its execution. However, the intent is not the same as the one of the Command Dispatcher. The intent of this pattern is to increase the flexibility of the command pattern mostly to support additional functionalities related to service execution (such as undo, redo,..) for application based on GUIs. Our pattern applies to GUIs, but also to event-driven systems, and any kind of applications, and its intent is more about coping with change.

Known Uses:

Dynamic evaluation is mandatory for systems with high availability requirements, such as:
- Real time embedded systems (e.g. FAU's AUV)
- Software where you can plug-in some new functionality. There are many examples of these.

References

[Del00] Delarue A., "*Modular application server design for AUV data network access and distributed processing*", Thesis, Florida Atlantic University, Department of Ocean Engineering, August 2000.

[Dup00] Dupire B., "*Pattern Oriented Design of a Dynamically Reconfigurable Software Architecture for AUV High Level Control*", Thesis, Florida Atlantic University, Department of Ocean Engineering, August 2001.

[Foo95] B. Foote and J. W. Yoder, "Evolution, Architecture, and Metamorphosis", *Procs. of Second Conference on Patterns Languages of Programs (PLoP '95*), Monticello, Illinois, September 1995

[GOF95] Gamma E., Helm R., Johnson R., Vlissides J., *"Design Patterns – Elements of Reusable Object-Oriented Software*", Addison-Wesley, 1995.

[POSA2] Schmidt D., Michael S., Rohnert H., Bushmann F., "*Pattern Oriented Software Architecture, vol.2: Patterns for Concurrent and Networked Objects*", J.Wiley & Sons, 2000.

[Som95] Peter Sommerlad, "The Command Processor pattern", in *Pattern Languages of Program Design 2,* edited by J. M. Vlissides, James O. Coplien, and Norman L. Kerth, Addison-Wesley, 1996

[Sum97] R.C.Summers, *Secure Computing: Threats and Safeguards*, McGraw-Hill, 1997.