

Message Redirector

Michael Goedicke⁺ Gustaf Neumann* Uwe Zdun⁺

⁺ *Specification of Software Systems*
University of Essen, Germany
{goedicke|uzdun}@cs.uni-essen.de

* *Department of Information Systems*
Vienna University of Economics, Austria
gustaf.neumann@wu-wien.ac.at

Many object-oriented applications require explicit control over the message flow to support flexible wrapping, interceptions, modifications of messages, traces, and other message control tasks. In object systems this control can also be used to express architectural semantics across several objects or classes. But most programming languages do not support such techniques as native language constructs. As a solution, a MESSAGE REDIRECTOR controls the method calls to (and within) the affected subsystems by providing a symbolic indirection and a message dispatch mechanism. Callbacks can be invoked during redirection to modify or extend dispatch-related semantics.

Application Example: Employee Type Knowledge Level

Let us consider the example of TYPE OBJECT [7] that is a common pattern in many complex object systems. It resolves the problem that a certain type relationship has to be dynamic in a statically typed, object-oriented language. By building the type relationship with the objects of the language, instead of the static classes, dynamic typing is “simulated” using delegation.

Figure 1 shows a common example of TYPE OBJECTS in business systems (see [3]): An employee has an associated object to model the employee type. The employee types are created with common combinations of payment and retirement policies. Employee types indicate which policies they should use. Of course, the nature of the employee types may change. And the employee type of a particular employee may change, too.

Here, we cannot use static classes, as in Java or C++, to model the real world situation properly. Each employee object may change its type during its lifetime. An employee may have more than one type at a given point in time. Moreover, the properties of the type, like the payment or retirement policy, may dynamically change as well.

The solution in Figure 1 includes an additional layering into a group of objects (knowledge level) that describe how another group of objects (operational level) should behave. Such a design is known as KNOWLEDGE LEVEL [3] or “meta level.” In the KNOWLEDGE LEVEL additional objects and relationships are attached to the TYPE OBJECT.

The idea to introduce explicit TYPE OBJECTS is frequently used in object-oriented languages with static type systems to mimic a dynamic type concept. If types are known to change dynamically and if classes are composed by changing components, the type system of statically typed languages cannot be used to express the targeted concerns properly. TYPE OBJECTS avoid proliferation and combinatorial explosion of classes.

However, the approach based on explicit TYPE OBJECTS yields several problems:

- *Recurring Type Objects:* In complex object systems TYPE OBJECT related concerns are recurring; that is, different TYPE OBJECTS in the system, such as employee types, party types, organization types, etc. share common features. The implementation variant in Figure 1 requires implementing basic concerns, like issues of object creation and destruction, repeatedly for each occurrence of a TYPE OBJECT.

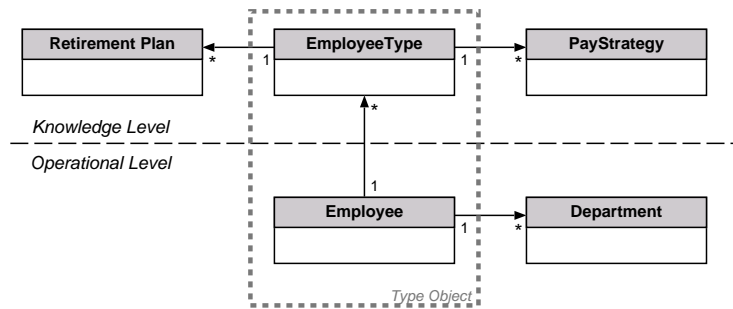


Figure 1. Employee Type Knowledge Level Using a Type Object

- *Message Dispatch-Related Semantics:* Often there are recurring dispatch-related semantics that should be ensured. For instance, in the employee type example it may be beneficial to be able to inherit from the employee TYPE OBJECT. But since the employee type is an object, the semantics of the inheritance relationship have to be built by hand. These tasks can be hard-coded into the employee type, but then the inheritance implementation cannot be reused. Besides implementing object-oriented relationships, there are many other examples of dispatch-related semantics one may want to ensure, as for instance assertions, logging, object sharing through FLYWEIGHTS, etc.
- *Customization Hooks:* When using dynamic types in the targeted language, as in the employee type example, we expect to benefit from the explicit dynamic relationship between “employee” and “employee type.” That is, customer- or domain-specific ADAPTERS or DECORATORS should be transparently and dynamically composable with the connection of employee and employee type .
- *Layered Knowledge Level:* If a KNOWLEDGE LEVEL is used, base level objects should not be able to access all knowledge level objects directly. Thus the LAYER property has to be ensured.

As a solution to these and similar problems a MESSAGE REDIRECTOR can be used to control the message flow between an object and its TYPE OBJECT. Thus it can ensure certain properties and introduce adaptations and decorations.

Context

The system and/or the domain in focus require high interpretational and/or architectural flexibility. In this context, there are numerous different situations and contextual patterns, such as the following.

When black-box components are glued together, one can only rely on the interfaces provided by the components. As systems evolve, interfaces of black-boxes tend to change. Thus clients have to cope with such changes. COMPONENT WRAPPERS [21] are a central access point to one component, where we can apply specific changes. However, changes that affect more than one component, more than one component version, or other subsystem objects than the COMPONENT WRAPPER, have to be propagated through the code. This is because dealing with such concerns, that are cutting across various objects or components, requires to have a control over the message flow to all objects and components affected by a change.

Often a project is faced with non-object-oriented languages or with object systems that are not powerful enough for the project’s purposes. But nevertheless the developers want to apply advanced object-oriented techniques in these languages. In such cases, the pattern OBJECT SYSTEM LAYER [5] tells us how to build an object system as a language extension on top of the target language. A vital part of OBJECT SYSTEM LAYER is a control mechanism over the messages in the system. Otherwise we cannot ensure the semantics of the language constructs and relationships implemented by the OBJECT SYSTEM LAYER.

Patterns, such as TYPE OBJECT [7] or KNOWLEDGE LEVEL [3], divide the object system into an implementational base level and a meta level, carrying meta information, such as types or other reflective information. If further semantics should be expressed and ensured with these patterns, such as for instance an automated meta level inheritance feature, we require control over the message flow as well. Otherwise we cannot ensure that the semantics of the feature are not violated.

Problem

How to gain control over the message flow in an object-oriented system, so that we can at least trace and modify the messages and their results? How to ensure that no messages bypass the control over the message flow and violate semantics we want to express with such a construction?

Forces

- *Controlling the Message Flow*: In object-oriented systems behavior is represented at runtime through messages. Therefore, a diverse set of cases exists in which we want to have a more fine-grained control over the message flow (most other forces, discussed below, are example cases).
- *Transparency of Adaptations*: Neither component nor component client necessarily needs to know of adaptations. Tasks such as logging in the background should be applicable in a transparent fashion.
- *Traceability of Messages*: Often, say, for debugging purposes, it is useful to trace all accesses to a component or subsystem.
- *Version Integration*: Often several different versions of one product have to be supported. If the messages sent to the product can be controlled, it is often possible to adapt them to the common denominator or to handle exceptional cases.
- *Flexible Wrapping of Components*: When black-box components should be glued together, we have the problem of (often minimal) changes in the interfaces. COMPONENT WRAPPERS [21] can be used for accessing black-box components with a central access point for each component. However, changes and extensions of the component wrapping process itself have to be propagated through the code.
- *Crossing Namespace Boundaries*: In environments with multiple namespaces, a mapping mechanism between identifiers, unique for each namespace, is required. One example of such a mapping mechanism is to map unique object identifiers in an object-oriented programming language to the identifiers in an object-oriented database management system. Very similar situations occur, when several object systems have to be integrated.
- *Dynamic and Extensible Message Resolution Schemes*: When the message flow to a component can be controlled, modifications or adaptations of these messages can be dynamically handled at runtime. Many programming languages offer only limited resources for dynamic message resolution. It is hard and/or inelegant to compose orthogonal concerns into the message precedence order and often it is impossible to customize message dispatch rules, such as the path taken through the class graph. However, in many situations it is beneficial to enhance the resolution scheme with more dynamics and customizability.
- *Central Access Point*: Often there are several places that are accessed from one client and that share some common property, such as being part of a black-box component or being of a certain type. Sometimes the way of client access tends to change. Then a bottleneck for client messages is required, where changes can be applied centrally. Otherwise all accesses, possibly scattered throughout the code, have to be searched to apply the change.
- *Shielding a Subsystem*: Often a component represents an opaque subsystem. Clients should be hindered to directly access the component.
- *Efficiency*: The reason why many object system do not provide dynamic message resolution is that static resolution is more efficient. A technique that resembles or implements dynamic message resolution schemes always has to deal with efficiency issues.
- *Constraining Message Semantics*: Constraints for the inputs and outputs of messages, as well as for the architectural structure before and after a message execution, have to be provided and checked in many circumstances.

Solution

Build an explicit MESSAGE REDIRECTOR instance that is called with a symbolic call every time a message should be sent to a certain (sub-)system. The message itself is given as an argument to a call of the MESSAGE REDIRECTOR. The MESSAGE REDIRECTOR maps the symbolic calls to actual implementation objects and methods. After this mapping is performed, the redirector invokes the message implementation and returns the result.

The MESSAGE REDIRECTOR may have callbacks registered that are invoked in certain situations. The criteria for invoking callbacks are application dependent. Thus, the callbacks may be invoked on basis of certain called objects, types, components, method names, or any other discriminating criteria.

Architecture Overview

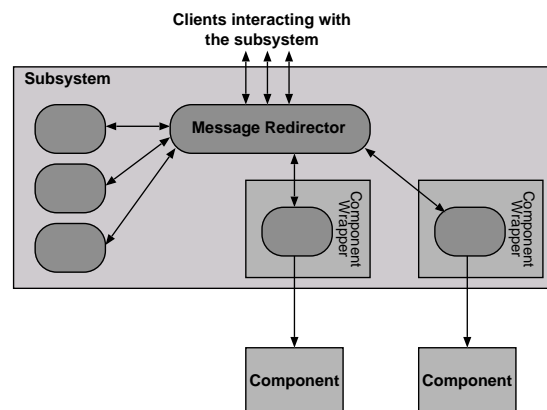


Figure 2. MESSAGE REDIRECTOR – Architecture

In Figure 2 we can see the architecture of a typical MESSAGE REDIRECTOR. A set of *interacting objects* form a *subsystem*. In general, none of these objects directly communicates with each other.

A central *redirector object* dispatches the object calls through simple dispatch scheme *message* → *message implementation*. If *message* is not equal to *message implementation*, then the MESSAGE REDIRECTOR object acts as an ADAPTER. All object interactions in the subsystem are handled via the MESSAGE REDIRECTOR object. Sometimes, for instance when the dispatcher of an existing interpreter can be used as a redirector, the redirector is not an explicit instance of the object system. In the central instance implementation variant, the redirector is a MEDIATOR in the subsystem.

If the subsystem integrates components into the subsystem's object system, then COMPONENT WRAPPERS [21] can be used. The COMPONENT WRAPPER objects are handled like all other subsystem objects and are dispatched by the MESSAGE REDIRECTOR object, too.

Clients from other system parts, interacting with the subsystem, use the MESSAGE REDIRECTOR object as a FACADE to access the subsystem. This way it is ensured that the MESSAGE REDIRECTOR can trace all accesses to the subsystem.

Design and Implementation

Class-Based Design and Implementation

In Figure 3 a simple class-based design of a MESSAGE REDIRECTOR is shown. For more sophisticated redirection or dispatching tasks the design will get more complex, but the general design is more or less similar.

Clients only interact with a central MESSAGE REDIRECTOR. The MESSAGE REDIRECTOR allows objects of the subsystem to be registered with `addObject`. The objects, in turn, provide dynamic registration means for

method implementations. Both, objects and methods, are registered with symbolic names, as for instance strings. These symbolic names are stored with the implementations. This can, for instance, be done in a linear list, with hash tables, or with associative arrays. In any MESSAGE REDIRECTOR there is such a mapping based on tuples of the form (*message-key, method-impl*). Often strings are used as message keys.

Language support for method implementations varies in different languages. In C and C++ often function pointers are chosen to point to method implementations. In Java reflective method objects can be chosen. Many languages, such as Lisp, Smalltalk, or Tcl, let us directly access program code as data. If the MESSAGE REDIRECTOR is used in the interpreter of a whole programming language, the method implementation may cover several classes (e.g., to access an on-the-fly byte-code compiler).

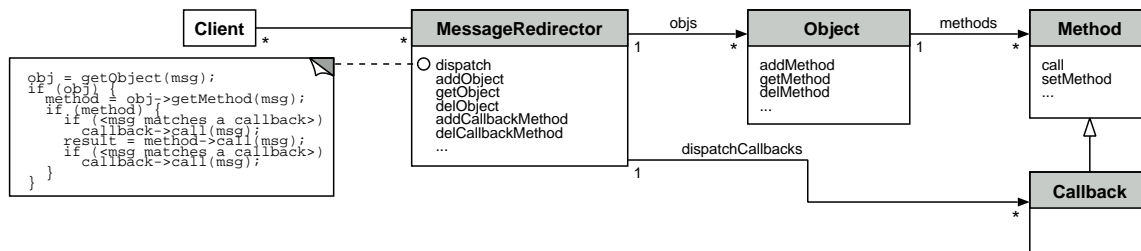


Figure 3. Class-Based Design of a MESSAGE REDIRECTOR

The MESSAGE REDIRECTOR is differently created in different variants. If it is a FACADE of a component, it is usually created by the component initialization code. In complete object-oriented languages often it is initialized during interpreter initialization. Responsibility of the current state in the object and method tables usually stays with the components implementing them. However, a reference counting mechanism can be used to deregister objects upon destruction.

Employee Type Example Resolved

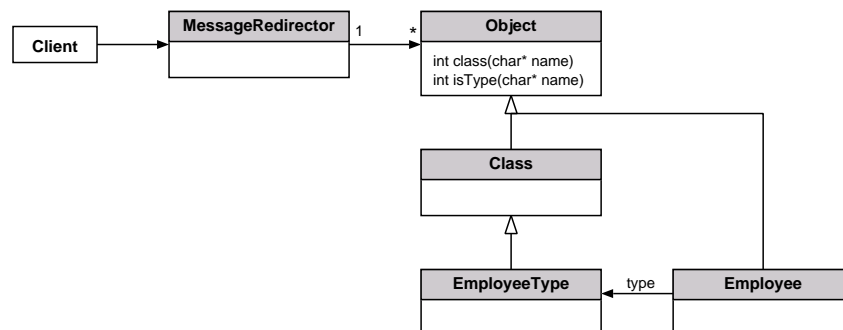


Figure 4. Employee Type Knowledge Level with a MESSAGE REDIRECTOR

In Figure 4 a resolved design for the employee type example is shown, hooked into the design from Figure 3. In addition to the implementation example, Figure 4 contains a `Class` abstraction. The class `Class` is used to provide the employee type with class-based features, like instance creation and inheritance in a dynamic fashion. All instances of `Class` and its subclasses are `Objects`, and contain a `type` property to their `Class` type. This property can be dynamically changed.

For this purpose a method `class` is added to the `Object` type. Each instance can change its class dynamically. The `class` method receives a string as argument. In Figure 5 the principal collaboration is shown. A client object request the MESSAGE REDIRECTOR to reclass the employee to a manager with a symbolic string command. First the MESSAGE REDIRECTOR resolves the object in its object table. Then, on success, it retrieves the method `class` from employee object which is found on `Object`.

In the `class` method implementation, we first have to find the `manager` object on the `MESSAGE REDIRECTOR`. Then it has to be checked, whether it is a class or not. This task is fulfilled by another additional method `isType` on `Object`. Finally, the information in `type` is changed and `class` returns.

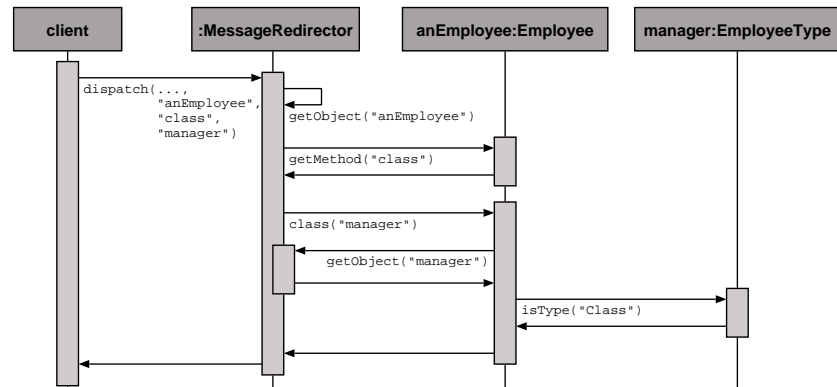


Figure 5. Collaboration during Reclassing of an Employee to a Manager

The client of an `Employee` does not see the `type` property. The client communicates only with the `MessageRedirector` using the symbolic name of the `Employee` object. The `MessageRedirector` redirects the message to the `Employee` object, and ensures typing semantics automatically and transparently.

Thus the `Class` abstraction plus the `MessageRedirector` implement a reusable language-support for `KNOWLEDGE LEVEL`. Similar patterns (or `KNOWLEDGE LEVEL` variants) can be implemented similarly. Moreover, variations can be explicitly treated by introducing callbacks. Therefore, even variations of the `KNOWLEDGE LEVEL` are reusable and composable. The `MESSAGE REDIRECTOR` ensures the `LAYER` property by letting clients only access those objects and methods of the employee type that should be exposed. All object and method combinations that are unknown to the `MESSAGE REDIRECTOR` cause a runtime error to be raised.

Note that in the example we have not implemented a `superclass` relationship for employee types. We can implement such a relationship with the same techniques as the `class` relation. However, in statically typed languages, such as `C++` or `Java`, it is hard to implement an one-to-one integration with a dynamic class concept because the static classes cannot follow dynamic class changes. A solution is to constrain those dynamic classes that have a static counterpart: a callback on the `MessageRedirector` prohibits superclass changes. Note that even in dynamically typed languages integration of two different class concepts is a considerable effort. Constraints of both class concepts have to be ensured in their respective counterparts. Such dispatch-related semantics can be ensured by callbacks on the `MessageRedirector`.

Implementation in Java

As an example, we present a simple implementation variant of `MESSAGE REDIRECTOR` in `Java` using the `Reflection API`. Here, a variant that just contains the notion of dynamic objects with dynamic method bindings, and a simple dispatch scheme is presented. It can be used as a reusable component in several different `TYPE OBJECT` situations. Many projects using `MESSAGE REDIRECTORS` implement it in a reusable form because the concerns that lead to the introduction of a `MESSAGE REDIRECTOR` are usually recurring in an application domain.

The implementation can be done in nearly any language. We have to perform the following steps:

- Find (and implement) a suitable method type abstraction which can be an object (as implemented by the `Java Reflection API`) or e.g. a function pointer in `C` and `C++`.
- Build a generic table or list for objects (used in the `MESSAGE REDIRECTOR`) and for the method implementations (used in the objects). The table contains keys (such as strings) and associated object or method implementations.
- Build a class for objects containing the key as a variable (here: a string) and a table of its methods. Moreover, add methods for registering, deregistering, and querying of methods in the method tables.

- Build a MESSAGE REDIRECTOR class with a table of its objects. Moreover, add methods for registering, deregistering, and querying of the objects in the object tables. Optionally callback methods which are executed upon certain criteria can be added.

For demonstration we will discuss a generic and reusable example implementation in Java. First, a simple abstraction for methods can be defined that uses the `Method` class from the Reflection API:

```
class MRMethod {
    protected Method method;
    MRConverter converter;
    ...
    MRMethod(MRConverter c) {
        converter = c;
        ...
    }
    void setMethod(Method m) {method = m;}
    Method getMethod() {return method;}

    public Object invoke(MRObject obj, String args[]) {
        Object result = null;
        try {
            Class[] parameterTypes = method.getParameterTypes();
            Object arglist[] = new Object[parameterTypes.length];
            if (parameterTypes.length + 2 != args.length) {
                ... // error handling
            }
            for (int k = 0; k < parameterTypes.length; k++) {
                arglist[k] = converter.toObject(parameterTypes[k], args[k+2]);
            }
            result = method.invoke(obj, arglist);
        } catch (Throwable e) {
            ... // error handling
        }
        return result;
    }
}
```

In the `MRMethod` class convenience methods for setting and getting the wrapped method implementation are provided, and a constructor that associates a handler for type conversions. In the method `invoke` the string-based arguments are converted to the proper Java types of the method to be called. These types are obtained using the reflection API. If the conversion is successful, and the number of given arguments is equal to the number of arguments in the called method, then we dynamically call the wrapped method with these arguments. The `Object` returned by the Reflection API's `invoke` is also returned by the method wrapper's `invoke` method as result.

The class `MRObjct` is defined as the most general class of the MESSAGE REDIRECTOR object system. It provides methods and variable slots (variables are not discussed here).

```
class MRObjct {
    protected String name;
    protected Hashtable methods;
    protected Hashtable variables;

    MRObjct () {
        methods = new Hashtable();
        variables = new Hashtable();
        ...
    }
    public String getName() {return name;}
    public void setName(String n) {name = n;}
    void addMethod(String methodName, MRMethod method) {
        methods.put(methodName, method);
    }
    MRMethod getMethod(String methodName) {
        return (MRMethod) methods.get(methodName);
    }
    void delMethod(String methodName) {
        methods.remove(methodName);
    }
    ...
}
```

In the code excerpt, methods and a property for defining and querying the object name are declared. Moreover, two hash tables for method and variable entries are provided. The methods `addMethod`, `getMethod`, and `delMethod` allow one to dynamically store, retrieve, and delete a method entry in the object's `methods` hash table.

Finally, a MESSAGE REDIRECTOR object has to be defined. On the MESSAGE REDIRECTOR methods for dynamically adding, getting, and deleting objects are declared. These handle the objects in the redirector's method table.

```
class MessageRedirector {
    Hashtable objects;
    Hashtable callbacks;

    MessageRedirector () {
        objects = new Hashtable();
        callbacks = new Hashtable();
        ...
    }

    void addObject(String name, MRObjct o) {
        objects.put(name,o);
    }
    MRObjct getObject(String name) {
        return (MRObjct) objects.get(name);
    }
    void delObject(String name) {
        objects.remove(name);
    }
    ...

    public Object eval (String command) {
        String args [];
        // splits command into String args[] and calls dispatch:
        ...
        return dispatch(args);
    }
    public Object dispatch (String args[]) {
        int argc = args.length;
        if (argc < 2) {
            ... // error handling
        }
        MRObjct obj = getObject(args[0]);
        if (obj == null) {
            ... // error handling
        }
        MRMethod m = obj.getMethod(args[1]);
        if (m == null) {
            ... // error handling
        }
        return m.invoke(obj);
    }
}
```

The `MessageRedirector` class has convenience methods for adding, getting, and deleting objects. Moreover, there are methods for adding and deleting callbacks on the `METHOD REDIRECTOR` class (not shown here). Clients usually call `eval` that splits a given string into a string array and calls `dispatch` with the split array. The `dispatch` method has to map the object name and method name, given in a string array, to method implementations. Therefore, the length of arguments has to be at least two. If object and method are found on the `METHOD REDIRECTOR`, we can invoke the wrapped method on the object and return the result.

For different application scenarios, different criteria for the callback are necessary. At least each callback has to define a method implementation if the callback applies:

```
class MRCallback extends MRMethod {...}
```

The callbacks may be used in specialized `METHOD REDIRECTORS`. Various criteria may serve as a condition to determine when the callback has to be applied. For instance the called object and method may be added as activation criteria:


```

class ObjMethodCallback extends MRCallback {
    ...
    String methodName;
    String objName;
}

```

In a specialized MESSAGE REDIRECTOR this callback type can be added to the dispatch algorithm. Such callbacks may be placed before or after the original receiver method, or they may override it. Callback methods may have a different signature than ordinary methods. For instance they may also receive the result of the computation in the original receiver method.

As a variant, the MESSAGE REDIRECTOR can itself be a subclass of MRObject. Then the MESSAGE REDIRECTOR is instantiable in its own object system. Classes can also be added as a special object type. Then each object has to associate the class object via a `type` property. The class objects implement class features, like instance creation, destruction, inheritance, etc.

Often a generic client data slot is associated with each method and each MESSAGE REDIRECTOR. In such a slot, for instance, a *propagating redirector* can be placed. Then the primary redirector propagates to a secondary redirector, if the client data slot is not void.

Simple Object and Method Example

In the previous section a simple, generic MESSAGE REDIRECTOR was implemented. Here, we briefly explain how an example object with a method can be registered and called. First, a special object type has to be derived:

```

class MyObject extends MRObject {
    MRMethod printHello, add;
    public void printHello() {System.out.println("Hello");}
    public int add(int a, int b) {return a + b;}
}

```

The methods `printHello` and `add` are examples for dynamically registered methods. For instance we can register them in the constructor:

```

Method meth = null;
try {
    Class cls = Class.forName("MyObject");
    meth = cls.getMethod("printHello", null);
} catch (Throwable e) {
    ... // error handling
}
if (meth == null) {
    printHello = new MRMethod(converter);
    printHello.setMethod((Method) meth);
    addMethod("printHello", printHello);
}

```

In the constructor, the method `printHello` is added as an `MRMethod` and then registered with the string-based name `printHello`. Now the method can be used:

```

MessageRedirector mr = new MessageRedirector();

MyObject o = new MyObject();
mr.addObject("o", o);
mr.eval("o printHello");
System.out.println("");

Object resultAdd = mr.eval("o add 3 4");
System.out.println("Result = " + resultAdd);

```

To call the methods `printHello` and `add` a `MessageRedirector` has to be instantiated. Afterwards an object is instantiated and registered on the `MessageRedirector`. Finally, the `eval` method is called with the symbolic commands for object and method. These are mapped to the correct implementation.

Consequences

The MESSAGE REDIRECTOR pattern has the following benefits:

- All incoming calls from a (set of) client(s) are bundled. Thus they can be changed and adapted easily.
- If dynamic resolution schemes are necessary, simple MESSAGE REDIRECTORS can be implemented very efficiently. For many tasks a simple search (or a sequence of searches) for the key in a hash table may suffice.
- MESSAGE REDIRECTOR implementations usually do not require more complicated client code.
- Adaptations and extensions in the MESSAGE REDIRECTOR are transparent to the client.
- As a FACADE, the MESSAGE REDIRECTOR shields a subsystem and provides a uniform way of access.
- The MESSAGE REDIRECTOR can be used to provide flexibility through highly programmable interfaces.

The MESSAGE REDIRECTOR pattern can also incur the following liabilities:

- All MESSAGE REDIRECTORS consume additional computation time. Even though most dynamic resolutions can be implemented with sufficient speed, for some tasks it might be a larger effort to find an efficient implementation.
- If a MESSAGE REDIRECTOR is used in parallel with ordinary method calls, two different styles of method calls are present in a system.
- Without language support it is hard to enforce that a MESSAGE REDIRECTOR is not bypassed by ordinary method calls. In many systems we can nevertheless avoid bypassing. For example, in distributed systems a MESSAGE REDIRECTOR on the server may prevent from direct calls to the subsystem.
- If the MESSAGE REDIRECTOR cannot be reused from an existing implementation, there are additional costs of design, implementation, and maintenance of the MESSAGE REDIRECTOR. However, this may cause less efforts and be less error-prone than many implementations of recurring MESSAGE REDIRECTOR concerns that are scattered throughout the code.

Pattern Variants

- *Dispatcher of an Interpreter*: Every interpreted (scripting) language must have some kind of dispatch mechanism in its interpreter. The dispatch mechanism redirects the scripting command to the implementation in a system language (as for instance C or C++). Often the design of such METHOD REDIRECTORS are much more complex than other variants, because such MESSAGE REDIRECTORS have to trigger several interpreter tasks, like command execution, on-the-fly byte-code compilation, garbage collection, etc.
- *Programmable Wrapper*: If the process of component wrapping has to be kept highly flexible, or if the several COMPONENT WRAPPERS have to be integrated, or if a mapping to a different interface has to be fulfilled during wrapping, a MESSAGE REDIRECTOR can be used as a highly programmable COMPONENT WRAPPER.
- *Partial Scripting Language Dispatch*: To make an application more flexible, many system language projects build a partial scripting language. For example, a small set of codes or strings is mapped to implementations. However, often the dispatch mechanism, necessary to map the symbolic codes to executable implementations and to evaluate the next command afterwards, are scattered throughout the code. An explicit MESSAGE REDIRECTOR instance is a more elegant design and enables changeability and maintainability of the dispatch mechanism. But often it is simply missing because the development team has not even realized that they have built a small, partial scripting language in their application.

- *Redirector for ID Conversion/Mapping*: Some – often very simple – MESSAGE REDIRECTORS are used for mapping of one namespace context into another. Here, the METHOD REDIRECTOR is rather used as a powerful ADAPTER. This variant is often used for integration of different object technologies or languages. For example, if a scripting language and a system language have to be integrated, or if different distributed object systems have to be integrated, or if unique object identifiers in an object-oriented programming language have to be mapped to identifiers in an object-oriented database management system, a MESSAGE REDIRECTOR can be used for ID conversion and mapping.
- *Hierarchical Message Redirector Extension*: An implementation variant is to build a MESSAGE REDIRECTOR split over a class hierarchy. For example, in C++ one can build such a hierarchical dispatcher with a virtual method, like:

```
class Top {
    // standard method redirector
    virtual int dispatch(int argc, const char*const* argv);
    ...
};
class Special : Top {
    int dispatch(int argc, const char*const* argv) {
        if (<can perform dispatch on Special>) {
            return <call on Special>
        } else
            return (Top::dispatch(argc, argv));
    }
};
```

- *Propagating Message Redirection*: Another implementation variant it to enable propagation to another MESSAGE REDIRECTOR. The dispatch scheme receives optional information which determines the responsible redirector. In the implementation example, we have discussed the variant to pass a void pointer, pointing to the responsible redirector, if the primary redirector should not handle the message.

Known Uses

- XOTCL [11] is a TCL extension and implements powerful message interception techniques, dynamic object aggregations, nested classes, assertions, and several other high-level language constructs. The callbacks for these functionalities and for language extensibility are implemented in the XOTCL message dispatcher. The dispatcher is a MESSAGE REDIRECTOR with a similar architecture to the OTCL [19] MESSAGE REDIRECTOR. The Otcl dispatcher is a more simple variant that does only support a dynamic object-call-to-implementation mapping.
- There are several other object-oriented scripting languages that provide a similar (mostly static) mapping of object-call-to-implementation, including [incr Tcl], Perl, and Python.
- TclCL [18] is an integration of OTcl and C++ that provides a secondary redirector. The TclCL MESSAGE REDIRECTOR is hierarchically ordered in C++ virtual methods. TclCL is used in:
 - Mash [14] is a streaming media toolkit for distributed collaboration applications based on the Internet Mbone tools and protocols.
 - The Network Simulator (NS) [17] supports network simulation including TCP, routing, multicast, network emulation, and animation.
- *ActiWeb Web Object URL Redirection*: If web objects should accessed with URLs, then the URL string has to be mapped to object-oriented methods. In ActiWeb [12] a MESSAGE REDIRECTOR on the HTTP server's respond methods fulfills this task. It also handles security issues, like protecting resources or executing in safe environments. The mapping can be based on the URL, the protocol header, or the calling IP address.
- *Access Control in xoComm*: In the web server implementation xoComm [10] access control mechanisms can be composed as an extension architecture. As a MESSAGE REDIRECTOR, these access control extensions intercept the messages and redirect them to the implementation, if one of the available access control mechanisms challenges the message. When more than one access control mechanism is used at once (like HTTP basic and digest access control), the call is propagated through a CHAIN OF RESPONSIBILITY.

- *Generic DBMS Access*: In [6] we present a design for generic DBMS access that uses a MESSAGE REDIRECTOR for mapping of object-oriented message calls to relational DBMS access implementations. The main aim of this architecture is to cope with different DBMS products and product versions. Moreover, a generic object-oriented abstraction is provided to allow for general object persistence on top of the different relational databases.
- *Code Fragments with XML*: There are several projects, like [2], that express certain flexible code fragments in XML. The XML parser is then used to provide events that are dispatched afterwards and redirected to message implementations. Such projects implement the partial scripting language dispatch variant on top of the XML parser.

Note that all named variants and most of the known uses implement MESSAGE REDIRECTORS that are useful in different applications and thus can be reused. In a reusable variant the pattern often is in part language supported; that is, the design and implementations repository of the developer offers the MESSAGE REDIRECTOR as a reusable component. However, as the diversity of known uses shows, none of these implementations is useful for all targeted situations. For instance in XOTCL, where a MESSAGE REDIRECTOR is language supported, we often implement MESSAGE REDIRECTORS on top of the language. For example, in ActiWeb we use a MESSAGE REDIRECTOR to map URLs to message implementations and to handle security issues. This MESSAGE REDIRECTOR is also reusable, but yet different in implementation and semantics to the C-based MESSAGE REDIRECTOR for the XOTcl programming language.

Related Patterns

Objects interacting through a MESSAGE REDIRECTOR are frequently placeholders for a component wrapped by a COMPONENT WRAPPER [21]. The criteria for the message control often depend on whether the call is targeted to an external component or not. For example, for distributed components and event-based systems the MESSAGE REDIRECTOR may allow for registration of asynchronous calls and implement the message queue.

If the MESSAGE REDIRECTOR object has to change a message call to a different interface, then it can perform this mapping in the style of an ADAPTER [4]. When a set of objects solely interact among each other using a MESSAGE REDIRECTOR then they form one or more subsystems. Usually, direct access to the subsystem's object from clients that are outside of the subsystem is prohibited. Therefore, the MESSAGE REDIRECTOR implicitly plays the role of a FACADE to the subsystem. In the subsystem it often is a MEDIATOR for the subsystem's objects.

Often a MESSAGE REDIRECTOR is explicitly or implicitly part of an OBJECT SYSTEM LAYER [5]. It then handles dispatching of object-oriented messages in the OBJECT SYSTEM LAYER. OBJECT SYSTEM LAYER, MESSAGE REDIRECTOR, and COMPONENT WRAPPER are part of a larger, architectural pattern language that is sketched in [21].

As MESSAGE REDIRECTOR contains an object and method abstraction in the target language, it can be used to implement the message-to-implementation-mapping in patterns introducing a foreign, more flexible object system. Besides OBJECT SYSTEM LAYER, foreign object system patterns are TYPE OBJECT [7], PROTOTYPE-BASED OBJECT SYSTEM PATTERN [13], and KNOWLEDGE LEVEL [3].

In some contexts, the patterns COMMAND [4] and COMMAND PROCESSOR [1] may be considered as alternatives to the MESSAGE REDIRECTOR. If only a message-to-implementation indirection is necessary, these patterns may yield a better performance than a MESSAGE REDIRECTOR. In such cases, MESSAGE REDIRECTOR is rather an implementation overhead. COMMAND lets the client perform the indirection by means of subclassing. Thus it is not transparent and does not shield a subsystem. COMMAND PROCESSOR additionally contains a simple indirection to a processor that allows for undo and redo functionality on the commands. Both patterns do not contain object and method abstractions and they are not MEDIATOR or FACADE for the subsystem.

REACTOR [15] dispatches calls using a synchronously demultiplexed event queue. Thus in event-based systems, like distributed object systems or GUI toolkits, REACTOR is an alternative to the combination of MESSAGE REDIRECTOR with an event queue. REACTOR primarily implements reactive event handling. It does not handle the additional responsibilities of a MESSAGE REDIRECTOR. MESSAGE REDIRECTOR, in turn, does not contain itself abstractions for event demultiplexer, event handle, or event handlers. However, in certain situations the

two patterns may be combined. In this combination, the event abstraction is usually implemented with the MESSAGE REDIRECTOR's object abstraction. Reasons for such an integration are flexibility in event typing or callback registration on the REACTOR.

For the purpose of component wrapping, an ADAPTER [4] on a WRAPPER FACADE [15] provides a much less flexible alternative to MESSAGE REDIRECTOR, but frequently the flexibility is not required.

See Also

The primary purpose of a MESSAGE REDIRECTOR is to obtain a control over the message flow in object systems. Such a form of control can be used for purposes of limited granularity, such as message interceptions, modifications of messages, or traces. However, the MESSAGE REDIRECTOR can also be used to express and ensure architectural semantics that cut across several objects or classes. In [8] and [9] we discuss how the interceptors implemented with the MESSAGE REDIRECTOR of the XOTCL language can be used to implement design pattern fragments. In the same style we can also implement several other types of architectural fragments, like architectural constraints (see [21] for a detailed discussion).

MESSAGE REDIRECTOR can be used to implement and ensure the semantics of behavioral abstractions. In [20] user interface components are decomposed into smaller behavioral abstraction units, similar to mixins as in [16]. However, the descriptive language for behavioral abstractions can be composed with different programming languages, like Sather and C++. A MESSAGE REDIRECTOR can perform the mapping at runtime to avoid the necessity of a descriptive language for behavioral abstractions.

Acknowledgements

We would like to thank Gustavo Rossi for his great comments during the shepherding process. Additionally we would like to thank the members of the EuroPlop 2001 writers workshop for their constructive feedback.

References

- [1] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-oriented Software Architecture - A System of Patterns*. J. Wiley and Sons Ltd., 1996.
- [2] W. Emmerich, C. Mascolo, and A. Finkelstein. Implementing incremental code migration with XML. In *Proc. of ICSE'2000*, Limerik, Ireland, June 2000.
- [3] M. Fowler. Analysis patterns 2 - work in progress. <http://www.martinfowler.com/ap2/index.html>, 2000.
- [4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
- [5] M. Goedicke, G. Neumann, and U. Zdun. Object system layer. In *Proceeding of EuroPlop 2000*, Irsee, Germany, July 2000.
- [6] M. Goedicke and U. Zdun. Piecemeal migration of a document archive system with an architectural pattern language. In *Proceeding of 5th European Conference on Software Maintenance and Reengineering (CSMR 2001)*, Lisbon, Portugal, March 2001.
- [7] R. Johnson and B. Woolf. Type object. In R. C. Martin, D. Riehle, and F. Buschmann, editors, *Pattern Languages of Program Design 3*. Addison-Wesley, 1998.
- [8] G. Neumann and U. Zdun. Filters as a language support for design patterns in object-oriented scripting languages. In *Proceedings of COOTS'99, 5th Conference on Object-Oriented Technologies and Systems*, San Diego, May 1999.
- [9] G. Neumann and U. Zdun. Implementing object-specific design patterns using per-object mixins. In *Proc. of NOSA '99, Second Nordic Workshop on Software Architecture*, Ronneby, Sweden, August 1999.

- [10] G. Neumann and U. Zdun. High-level design and architecture of an http-based infrastructure for web applications. *World Wide Web Journal*, 3(1), 2000.
- [11] G. Neumann and U. Zdun. XOTCL, an object-oriented scripting language. In *Proceedings of Tcl2k: The 7th USENIX Tcl/Tk Conference*, Austin, Texas, February 2000.
- [12] G. Neumann and U. Zdun. Distributed web application development with active web objects. In *Proceedings of The 2nd International Conference on Internet Computing (IC'2001)*, Las Vegas, Nevada, USA, June 2001.
- [13] J. Noble. Prototype-based object system. In N. Harrison, B. Foote, and H. Rohnert, editors, *Pattern Languages of Program Design 4*. Addison-Wesley, 2000.
- [14] Open Mash Consortium. The open mash consortium. <http://www.openmash.org>, 2000.
- [15] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Patterns for Concurrent and Distributed Objects*. Pattern-Oriented Software Architecture. J. Wiley and Sons Ltd., 2000.
- [16] P. Steyaert, W. Codenie, T. D'Hondt, K. De Hondt, C. Lucas, and M. Van Limberghen. Nested mixin-methods in Agora. In *Proc. of ECOOP '93*, LNCS 707. Springer-Verlag, 1993.
- [17] UCB Multicast Network Research Group. Network simulator - ns (version 2). <http://www.isi.edu/nsnam/ns/>, 2000.
- [18] UCB Multicast Network Research Group. Tclcl. <http://www.isi.edu/nsnam/tclcl/>, 2000.
- [19] D. Wetherall and C. J. Lindblad. Extending TCL for dynamic object-oriented programming. In *Proc. of the Tcl/Tk Workshop '95*, Toronto, July 1995.
- [20] M. H. Wilkinson. *Behavioural Abstraction and Composition for User Interface Management*. PhD thesis, University of York, November 1998.
- [21] U. Zdun. *Language Support for Dynamic and Evolving Software Architectures*. PhD thesis, University of Essen, Germany, January 2002.