The Emissary Design Pattern

Ramiro González Maciel
LIFIA – La Plata National University
Argentina
ramiro@lifia.info.unlp.edu.ar

EMISSARY

Object behavioral

Intent

Define an object responsible of handling the events that occur during the execution of an operation. This object decouples the objects implementing the operation from the one requesting it. This object (the Emissary) thus represents the client at the server objects, and acts as an intermediary in the communication between the servers and the client.

Motivation

An object can implement different kinds of methods, like accessors, initializers, private methods and methods that provide access to complex operations. In this paper an operation is regarded as complex if its execution involves more than one object, can be divided in sub-operations and can have different results. During the execution of such a method, different events can occur, particularly some that the object implementing the method can not handle.

As an example, consider the design of a game of fighting robots. Each robot is composed of different parts, like weapons, shields and a CPU. These CPUs can be programmed by the player in order to command the robot. These programs are written in a particular event-based language. Regarding the design of the game, it will have a subsystem responsible of accepting and translating the robots programs to an internal executable representation. This subsystem is composed of a scanner-parser and a compiler. During the compiling of the programs different kinds of events can occur. Each of these events could be handled in a different way depending on the context of the compilation. For example, if a syntax error is detected and the program is being edited on a GUI, the offending sentence can be highlighted and a dialog box can report the error to the user. If, on the other hand, the program is loaded from disk, the error could be logged in a file and the compilation would be interrupted. Also, some actions are independent of the source of the programs, for example the programmer can request all the errors and events to be logged on a file.

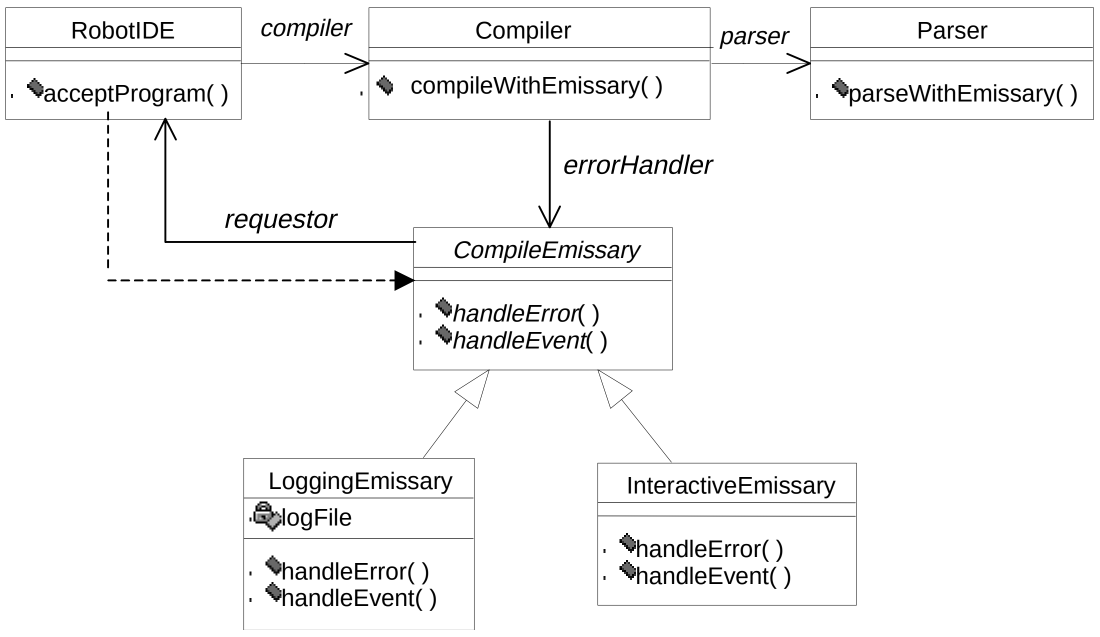
According to this design, there are two problems regarding the event handling mechanism. The first one is which object decides what actions to take when an event happens. The second problem is how to define those actions so different clients of the operation can share its implementation.

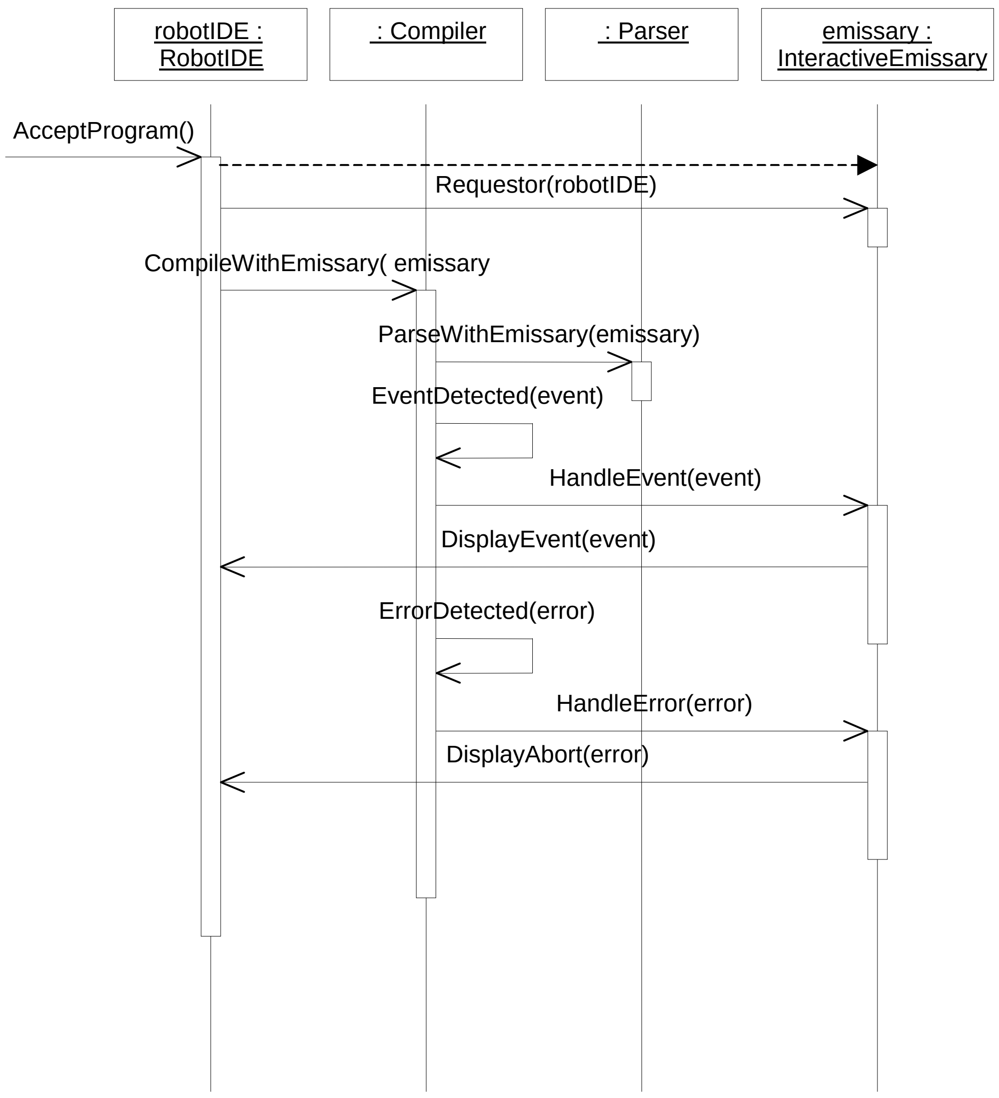
One solution could be that the compiler subsystem requires its clients to implement a fixed set of event handling messages. The compiler subsystem will use these messages to report the events to its clients. Depending on the language used in the implementation, this solution could force the client to be part of a "CompilerClient" hierarchy, which defines the event handling messages. This is not a good solution, since it forces the client objects protocol, thus coupling the client and server objects. It also provides inheritance as the only mechanism for sharing event response actions and does not provide the flexibility needed to change the clients' response at runtime.

Solution

The solution to this problem is to define an object that encapsulates the reactions to the events that can occur during the execution of the compilation. This object also implements the interface required by the compiler subsystem. This object acts as an emissary of the client at the server, during the execution of an operation. Each client chooses a particular emissary and configures it based on the context if necessary. When the client sends a message to the server requesting the operation, it also tells the server which object will be the responsible of handling the events (the emissary). Each emissary reacts to the events in its own way, probably interacting with the client.

In this way, the decision of how to react to the different events is on the client, but the implementation of these reactions is on the emissary. Since each emissary defines different reactions, different clients can share this implementation by creating and configuring a particular Emissary.

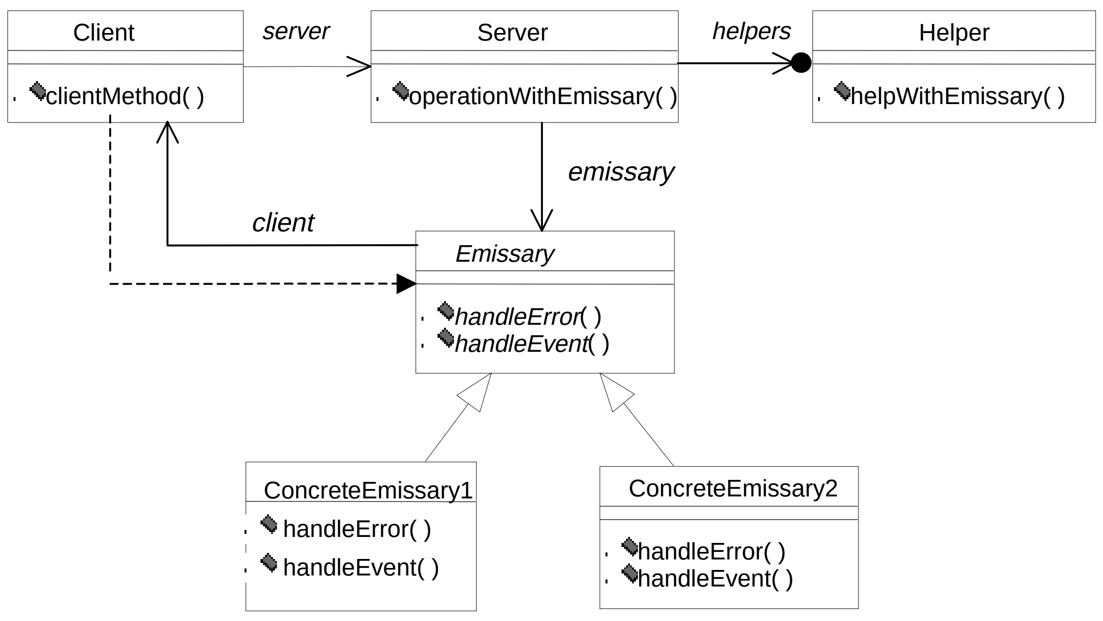




Applicability

- An object defines a complex operation. During the process of the operation different events can occur, which could be handled in different ways. The object defining the operation should not be responsible of deciding how to react to each event.
- Different objects requesting the operation react to some events in the same way.
- The clients have the knowledge needed to decide how to react to each particular event

Structure



Participants

Emissary

- Defines an interface to handle the different events that occur during the execution of a method that implements a complex operation.
- The Server and its Helpers use this interface.
- Its life is rather short. Usually, the Client creates an Emissary before requesting the operation, and is not used after the operation is finished.

Server

- Defines a complex operation. May need to collaborate with a Helper in order to fulfil the requests
- Communicates an Emissary the events that occur during the operation

Helpers

- Each Helper can implement part of the operation.
- Help the Server to fulfil a request.
- Tell the Emissary about the events that happen during an operation.

Client

- Creates and configures the appropriate Emissary for an operation, and passes it as a parameter when requesting the operation from the Server.
- Provides information to the emissary when it needs to take a decision in response to an event.

ConcreteEmissary

- Defines the actions to perform in response to different events, implementing the interface defined by the Emissary

Collaborations

- The Client creates, configures and passes the Emissary as a parameter when requesting an operation from the server.
- When the Server or one of its Helpers detects an event, they send a message to the Emissary.
- The Emissary can request information from the Client in order to take the appropriate response to an event

Consequences

- Representing the reactions on an Emissary class isolates in one place the code for handling some events in a particular way, therefore eliminating code duplication. Therefore different clients can have the same response to the server-generated events by creating and configuring new Emissaries.
- Decouples the client and the servers. The server objects do not restrict their clients to implement a particular set of messages to handle the events. They use the interface defined by the Emissaries.
- The Client needs to implement an interface to support the communication with the Emissary. Since the Emissary class is generic, and we do not want to impose a specific interface to every Client, in some cases the Emissary may need to be adapted to a particular Client. This can be done in different ways:
  - One is to implement a specific subclass that adapts a ConcreteEmissary to a Client. This solution doesn't scale if the Client can use many Emissaries, since this can lead to a class proliferation.
  - The Emissary can be designed as a pluggable adapter, an object that has built in interface adaptation [Adapter DP].
  - The use of other kind of adapters is too expensive since the traffic between the Emissary and the Client is usually small.

Implementation

Where is the Emissary referenced?

Since the Server implements a complex operation, the operation will probably be implemented in various methods. In any of these methods could be necessary to send a message to the Emissary. Thus, the Emissary can be passed as a parameter among the different methods that make the operation (as in the Motivation example), or be kept as an instance variable in the Server.

Events of interest and exceptional events

The main difference between events of interest and exceptional events is reflected in the communication between the Servers and the Emissary. In the case of events of interest the communication is unidirectional, from the Server to the Emissary. This reflects the fact that the Server is *reporting* an event, and it does not needs any information to continue with its task. An example of this would be the events that are logged in the compilation example, like the processing of one program in a batch compilation. In the case of the exceptional events the communication is bi-directional, since the Server does not have the necessary information to make a decision about that event, and so it needs to *consult* the Emissary about what to do. An example of this kind of communication would be the report of a syntax error to the programmer. If the syntax error is a typo, for example, the programmer can correct the error and continue with the compilation, or he can cancel the compilation.

How is the communication between the Emissary and the servers?

There are three kinds of communication between the Emissary and the server objects (Server and Helpers).

- In the case of events that can be interesting to the client, the server establishes a unidirectional communication with the Emissary, reporting the event. The server does not need any additional information to continue its processing. It can continue with its processing after (or concurrently with) the event notification.
- In the case of events that signal the end of an operation (that can be successful or not), the server also establish a unidirectional communication to the Emissary.
- When the server detects an event that it can not handle, it will establish a bi-directional communication with the Emissary. The Emissary will then tell the server if it is ok to proceed or not, or if it needs to restart the operation. The server must be carefully designed since the Emissary can change the context of the operation.

Known uses

- Progress indicators.

In SmallWalker, an HTML browser for VisualWorks, some messages follow the pattern "`doSomethingReportTo: aReporter`" (as in "`loadUrl:reportTo`:"). Each time a progress is made on one of those operations (like loading part of an HTML document), the server object sends a message to the Reporter. This progress report can be displayed in a "Progress widget", be used to determine the speed of the connection, etc. The Emissaries in this example are located on the hierarchy TransferProgressStatus.

- VisualSmalltalk 3.0 defines the following hierarchy:

```
CompilerErrorHandler
        NonInteractiveEH
        SilentEH
                PARTSSilentEH
                TraditionalCompilerEH
        TextPaneEH
                PARTSTextPaneEH
                WorkspaceEH
```

These ErrorHandlers (Emissaries) implement different reactions to the compiling errors. For example, when the compiler detects a variable that begins with a capital letter that is not declared, it tells so to its ErrorHandler. The TraditionalCompilerEH prints a message on the Transcript and aborts the compilation. The WorkspaceEH ask the user whether or not he wants to declare the variable as a global one, and the retries the operation. In this hierarchy the SilentEH and TextPaneEH class represent different reactions to the events, and each of their concrete subclasses adapt this behavior to a specific client.

In VisualSmalltalk the classes implementing the compiler subsystem are hidden from the user, so there is a CompilerInterface class, which is a Façade. The hidden classes (helpers) are the ones that send messages to the ErrorHandler.

- VisualWorks compiler.

VisualWorks 2.5 compiler error handlers are organized in the following hierarchy:

```
CompilerErrorHandler
        InteractiveCompilerErrorHandler
        LoggingCompilerErrorHandler
        NonInteractiveCompilerErrorHandler
```

QueryCompilerErrorHandler
SilentCompilerErrorHandler

The interface between the compiler subsystem and the Handlers are the following two messages:
`error: type with: message at: sourcePosition` and
`error: type with: message in: node from: codeStream`

These messages analyze the kind of error reported, can modify the compilation context (the parameters "`message`", "`codeStream`" and "`node`") and they return:
- `true` for retry
- `false` for proceed
- an error string or `nil` for abort

The ErrorHandlers reference the client by the name Requestor.

- Blocks as Emissaries

  Sometimes is too expensive to create a separate hierarchy to implement the Emissaries, but however it is desirable to keep the decoupling between the server and the clients. In this case one solution would be to use a standard protocol to report the events to the clients. In Smalltalk the use of blocks allows the programmer to define "immediate" Emissaries that define the reaction to the server events. In this case the server uses the message "`value: anEvent`" to report the events to the Emissary. If the Emissary is a block, the block is evaluated in the context of the client with the event as a parameter. Since the `value`: message is also used by other objects, for example ValueModels in VisualWorks, it use allows the programmer to use different Emissaries. A ValueModel emissary is based on the Observer pattern, thus providing a dependency relation between the client and the server.

Related Patterns

NullObject: If the client wants to ignore the events that occur during the operation, but the server requests an Emissary, the NullObject pattern can be used to implement a do-nothing Emissary

Strategy: Since the Emissaries collaborate with the server in order to fulfil some operation, there seems to be a relation between the Emissary and the Strategy pattern, in particular the Emissary being a Strategy and the Server a Context. However, there are some fundamental differences between the Emissary and the Strategy pattern, as follows. First, many objects, the Server and its Helpers, can share an emissary. In the Strategy pattern each strategy belongs to only one Context. This leads to the second difference, that the Emissary does not represent an algorithm or part of it, as the Strategy does. The Emissary handles the behavior that it is not part of the algorithm, so it can adapt an algorithm to a particular client by handling the exceptional aspects. The third difference is that in the Emissary there is some communication between the Emissary and the Client, since the Emissary may need some information to implement its behavior. In the Strategy the Client only responsibility is to create a ConcreteStrategy.

Façade: The Server and the Helpers often form a Façade.