Receiver

John Liebenau

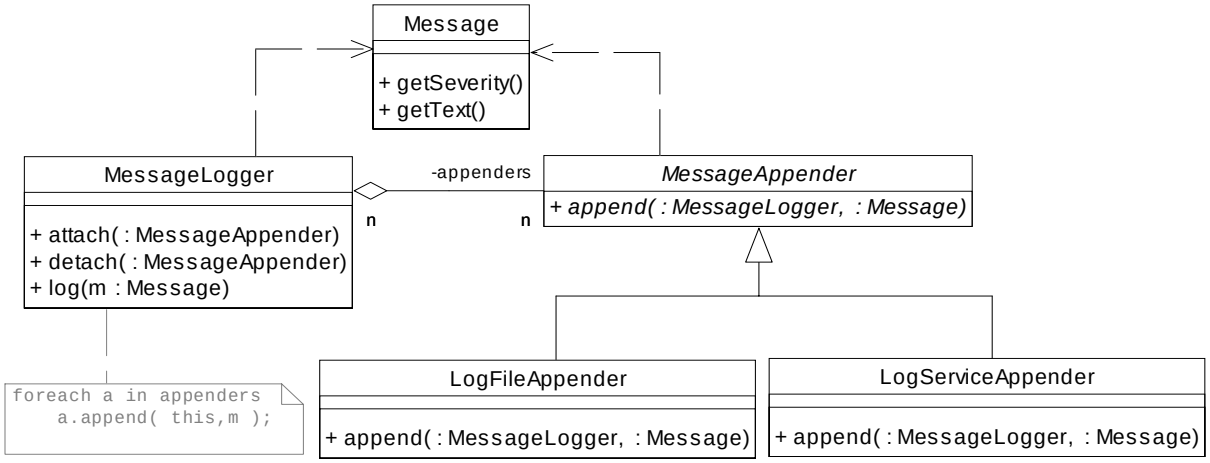
Intent

Decouple sending objects from receiving objects by: dynamically establishing connections between senders and receivers, encapsulating requests/data in content objects, and passing content from senders to receivers.

Motivation

Consider a generic logging facility that provides an API for applications to log messages pertaining to status and operational state. Once a message has been passed to the API it is processed. For most applications messages are processed by simply appending them to a log file that can be tailed or viewed at a later time (e.g. during a debugging session). However this may not be adequate for all applications. Enterprise-level server applications may send these log messages to a sophisticated logging/monitoring service to provide a feedback mechanism for production administrators to monitor and control multiple systems. Some applications may need to process log messages in multiple ways such as both appending to a log file and sending to a logging/monitoring service.

In order to make the logging facility reusable across multiple applications its message processing functionality has to be configurable. We can also make the logging facility API easy to use by separating the logging component from the message appending component. This separation allows the message appending component to vary independently of the logging API so applications can change the message appending behavior without having to change all the logging API calls.



The Receiver pattern illustrates how this separation is organized. Objects are categorized into three kinds: senders, receivers, and content. Senders transmit content to one or more associated receivers which uses the content to perform specific processing. From the above example, the message logger is a sender object that sends messages (content objects) to one or more message appenders which are receiver objects thus enabling applications to use a stable logging API defined by the MessageLogger class while varying the appenders. The appender API is specified by a MessageAppender interface which is implemented by several concrete classes (e.g. LogFileAppender, LogServiceAppender, ...).

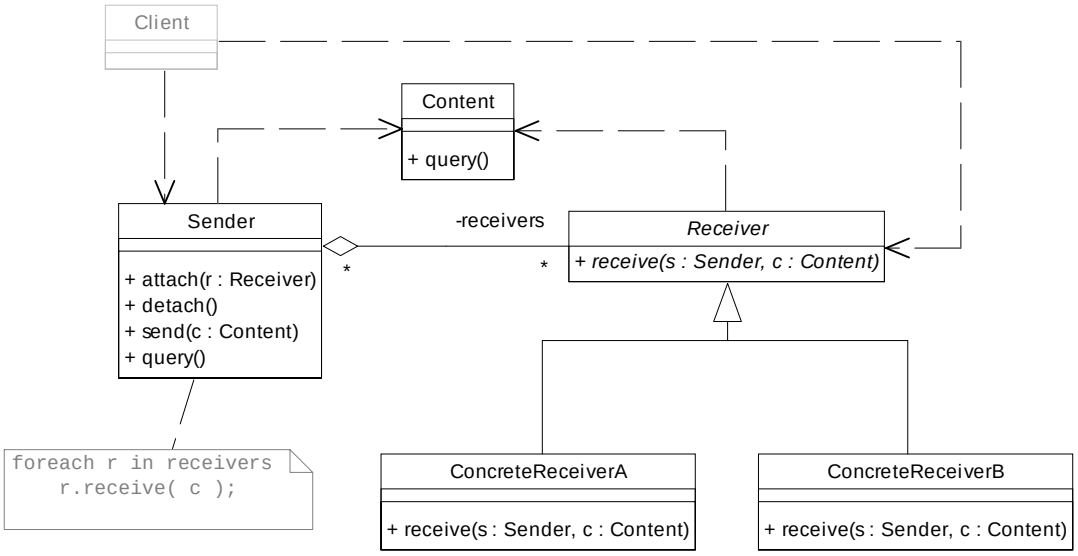
The Receiver pattern is a general design pattern that is usually part of more specialized design patterns such as Observer, Mediator, Chain of Responsibility, and Typed Message[GHJV95][Vlissides98]. Receiver focuses on decoupling senders from receivers which is one part of the more specific patterns. The mechanism for decoupling the "sending" objects from the "receiving" objects is practically the same for all four patterns. The differences in these patterns mostly occur in the behavior of the "receiving" objects once they have received something. For example, in Observer the ConcreteObserver receives an update notification from the Subject. The notification follows the Receiver pattern but once the ConcreteObserver receives the update, it obtains the data it is interested in from the ConcreteSubject. It is this extra action that specializes the Receiver pattern into the Observer pattern. Mediator, Chain of Responsibility, and Typed Message all specialize the Receiver pattern in specific ways to accomplish their intents. The specialized design patterns impose additional responsibilities and constraints on the Receiver pattern's participants, enabling them to solve more specific problems. The Related Patterns section provides more details.

Applicability

Use the Receiver pattern when:

- a class of objects (senders) needs to signal an event or send content to one or more other objects (receivers) without depending on the receivers specific types
- different clients need different sender-receiver connections
- the connections between senders and receivers change during the lifetime of these objects
- requests or data can be encapsulated (content) and used interchangeably by the receiver hierarchy

Structure *(Many-to-Many Receiver variant)*



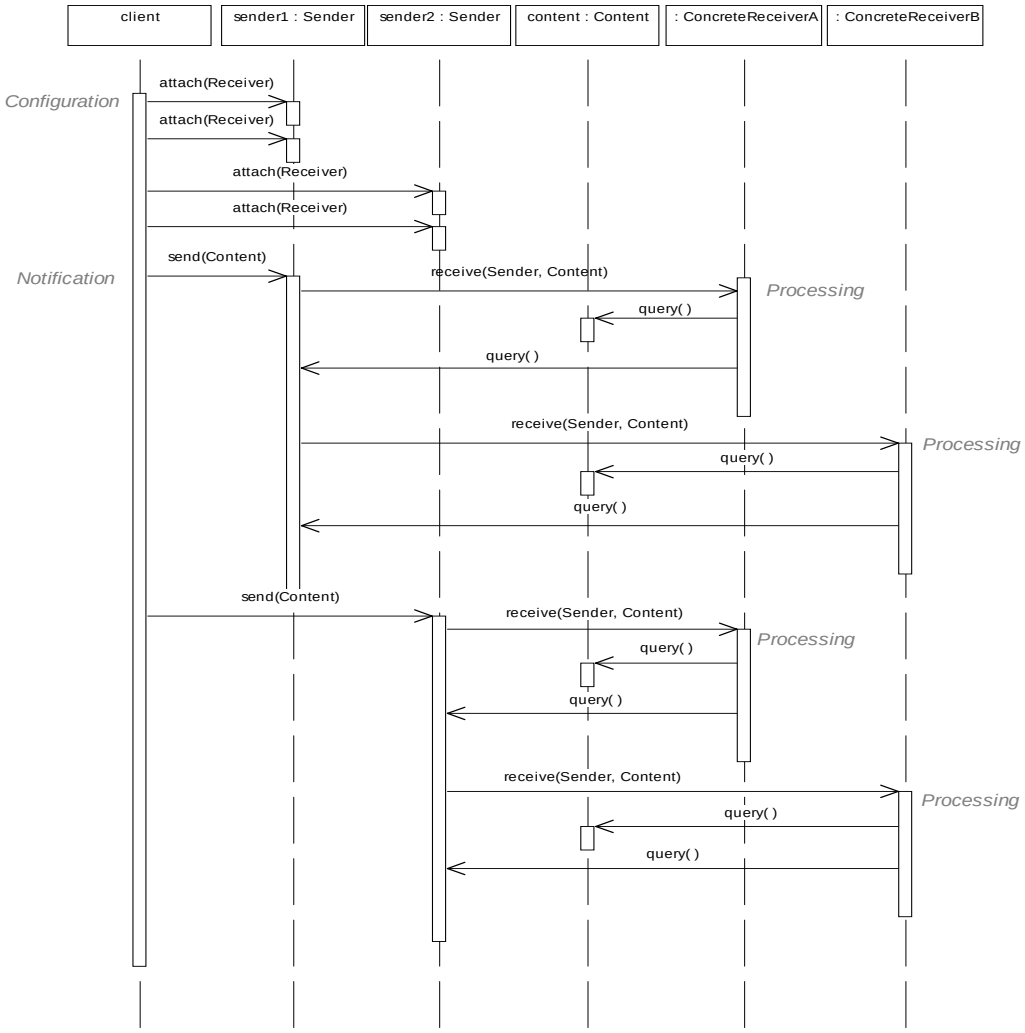
Participants

- SENDER (MessageLogger)
  - defines operations for `attaching` and `detaching` RECEIVER objects
  - defines a `send` operation for sending CONTENT to RECEIVER objects
  - defines one or more optional `query` operations for RECEIVERS to retreive information from the SENDER
- RECEIVER (MessageAppender)
  - declares a `receive` operation for receiving CONTENT from SENDER objects
- CONCRETERECEIVER (LogFileAppender, LogServiceAppender, ...)
  - implements the interface declared by RECEIVER
- CONTENT (Message)
  - represents information that is transferred from SENDERS to RECEIVERS
- CLIENT
  - establishes connections between SENDERS and RECEIVERS
  - directs SENDERS to send CONTENT to RECEIVERS

Collaborations

The Receiver pattern has the following collaborations:

- *Configuration* - CLIENT establishes the connections between SENDERS and RECEIVERS by attaching RECEIVERS to the appropriate SENDERS.
- *Notification* - SENDER notifies its associated RECEIVERS of significant events by passing CONTENT to its RECEIVERS when directed by CLIENT.
- *Processing* - Each CONCRETERECEIVER may query the CONTENT passed to it or may query the CONTENT's SENDER to obtain the appropriate data for use in executing the CONCRETERECEIVER's functionality.

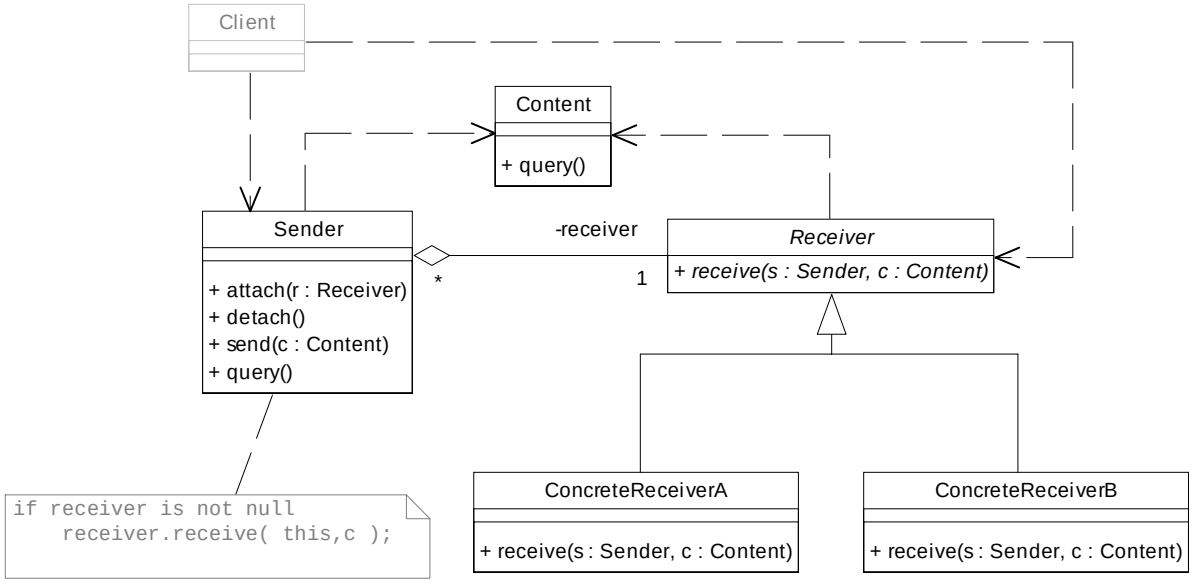


Variants

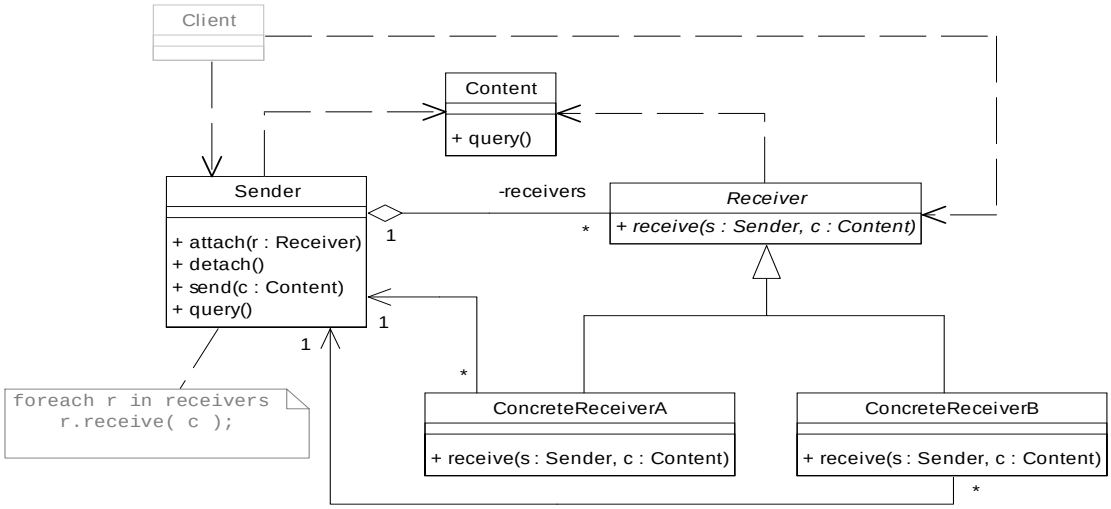
The Receiver pattern has several variants that impose key constraints on the pattern particpants, subtly altering their structures, interfaces, and interactions. The variants explicitly named here focus on cardinality constraints applied to the association between SENDERS and RECEIVERS because these variations in cardinality have greater prominence when dealing with implementation issues and when relating the Receiver pattern to other patterns than other variants.

The major Receiver variants are:

- *Many-to-Many Receiver* - A synonym for the default Receiver pattern. All other variants can be derived by adding constraints to the *Many-to-Many Receiver*.
- *Many-to-One Receiver* - This variant puts a constraint on the default pattern by restricting SENDERS to having only one RECEIVER (i.e. many SENDERS are associated with one RECEIVER).



- *One-to-Many Receiver* - This variant puts a constraint on the default pattern by restricting RECEIVERS to having only one SENDER (i.e. one SENDER is associated with many RECEIVERS). If a CONCRETERECEIVER needs access to the SENDER, the CONCRETERECEIVER must maintain its own reference to the SENDER.



Consequences

The Receiver pattern has the following benefits:

- *Decouples SENDERS from RECEIVERS*. SENDERS are not tightly coupled to their RECEIVERS, simplifying the implementation of the SENDERS and enabling the them to be configured with different functionality supplied by their RECEIVERS.
- *Dynamic configuration of SENDERS*. CLIENTS can configure SENDERS with RECEIVERS multiple times during program execution, effectively changing behavior to fit the needs of each specific scenario.
- *RECEIVERS are open to extension*. RECEIVERS form a class hierarchy that can be extended with specific functionality that can vary with the needs of each application.
- *Multicast communication to RECEIVERS*. SENDERS can transmit CONTENT to many RECEIVERS enabling a variety of processing combinations to be performed for each request.

Receiver has the following tradeoff:

- *Increased number of objects*. By separating the RECEIVERS from the SENDER, each class individually becomes easier to manage but this can make tracing and debugging more difficult because functionality is now spread out over a (sometimes dynamically changing) set of objects

Implementation

The following implementation issues should be considered when using the Receiver pattern:

- *Maintaining references to RECEIVERS*. The **-to-Many Receiver* variants typically use a container of RECEIVER references while the *Many-to-One Receiver* variant has only a single RECEIVER reference.
- *Maintaining a reference to SENDER*. RECEIVERS commonly reference their corresponding SENDER in three ways: the SENDER reference is passed as a parameter to the `receive()` operation, the CONCRETERECEIVER contains the SENDER reference as a field, or the CONTENT contain the SENDER reference as a field. Usually the *Many-to-* Receiver* variants will pass the SENDER reference as a parameter to `receive()` while the *One-to-Many Receiver* variant often embeds the SENDER reference as a field in the CONCRETERECEIVER.
- *Implementing the `send()` operation*. The `send()` operation has to call the `receive()` on each of the SENDER's RECEIVERS. The Iterator pattern provides an excellent mechanism for accomplishing this task. In Java, `send()` would look like:

```
class Sender...
    public void send(Content content)
    {
        List temp = receivers.clone();

        for (Iterator r = temp.iterator();r.hasNext();)
            ((Receiver)r.next()).receive( this,content );
    }
```

In C++, `send()` could be implemented using STL iterators:

```
class Sender
{
private:
    list<Receiver*> receivers;
public:
    void send(const Content& content);
};

void Sender::send(const Content& content)
{
    for(list<Receiver*>::iterator r = receivers.begin();r != receivers.end();++r)
        r->receive( *this,content );
}
```

- *Thread Safety*. In multithreaded programming environments like Java, it is important to address synchronization issues to avoid race conditions and other concurrency related inconsistencies. At a minimum, CONCRETERECEIVERS should make their methods synchronized to avoid problems when receiving CONTENT from SENDERS in different threads.
- *Using Singleton for stateless ConcreteRECEIVERS*. The Singleton pattern can be used to reduce the overhead of creating separate CONCRETERECEIVER objects when the CONCRETERECEIVER class does not contain any internal state. All SENDERS would share the CONCRETERECEIVER singleton instance.
- *Sending and receiving multiple kinds of CONTENT*. There are three scenarios involving transmission of multiple event types or data between SENDERS and RECEIVERS. The first case is when a RECEIVER can accept CONTENT from different kinds of SENDERS. This kind of RECEIVER will provide a `receive()` operation for each CONTENT type.

```
interface Receiver
{
    public void receive(ContentA a);
    public void receive(ContentB b);
}
```

The Typed Message pattern specializes Receiver by having the CONCRETERECIEVER implement multiple RECIEVER interfaces, unifying all RECIEVERS into the CONCRETERECEIVER.

The second case is an expansion of the first: the RECEIVER remains the same but the SENDER is augmented to send multiple kinds of CONTENT. This kind of SENDER must provide a `send()` operation for each CONTENT type.

```
class Sender
{
    public void send(ContentA a);
    public void send(ContentB b);
}
```

The third case involves a hierarchy Content classes. Depending on the Content interface RECEIVERS can treat all Content the same or Receivers may need to cast to more specific Content types:

```
class Content...
class ContentA extends Content ...
class ContentB extends Content ...

class ConcreteReceiver ...
    public void receive(Content content)
    {
        if ( content instanceOf ContentA )
            // ...
        else if ( content instanceOf ContentB )
            // ...
        else
            // throw appropriate exception
    }
```

RECEIVERS may identify the specific CONTENT types by downcasting, by using some kind of type code, or by using double dispatch.

- *Omitting CONTENT for signalling simple events*. Sometimes it is not necessary to pass Content from the Sender to Receivers. The `receive()` method call may be sufficient to signal an event to a Sender's Receivers. This is often the case when the Receiver pattern is specialized by the Observer pattern or the Mediator pattern. In the Observer pattern the `receive()` operation is renamed `update()` and in the Mediator pattern it is renamed `changed(Colleague)`. In these patterns the `receive()` operation is triggered when the Sender changes state.
- *Extending SENDER with CLIENT*. It is useful to subclass the SENDER and make the subclass into a CLIENT when Receivers need to be notified of state changes in the SENDER. The Observer pattern specializes the Receiver pattern by introducing a subclass of SENDER, labeled CONCRETESUBJECT, that acts as the CLIENT by directing the SENDER to notify its RECEIVERS when the CONCRETESUBJECT's state changes.

- *Combining SENDER and RECEIVER*. It is useful to combine the Sender and Receiver into the same class when a chain of notifications is needed. The Chain of Responsibility pattern, which is a specialization of the *Many-to-One Receiver* pattern variant, is often used for these cases.
- *Using C++ templates*. In C++ templates can be used to parameterize simple Senders that only send Content to their Receivers and do nothing else. These simple Senders are good candidates for becoming mixin classes. The Sample Code section shows a template implementation of the message logger example from the Motivation.

Sample Code

The example described in the Motivation section can be illustrated with the following Java code:

```
interface MessageAppender // Receiver
{
    public void append(MessageLogger l,Message m);
}

class MessageLogger // Sender
{
    private List appenders;

    public void attach(MessageAppender a)
    {
        if ( !appenders.contains( a ) ) appenders.add( a );
    }
    public void detach(MessageAppender a)
    {
         appenders.remove( a );
    }
    public void log(Message m)
    {
        for(Iterator i = appenders.iterator();i.hasNext();)
        {
            MessageAppender a = (MessageAppender)i.next();
            a.append( this,m );
        }
    }
}

class LogFileAppender implements MessageAppender // ConcreteReceiver
{
    private PrintWriter   logFile;
    private MessageFormat formatter;

    public LogFileAppender(String path)
    {
        logFile   = new PrintWriter( new FileOutputStream( path,true ) );
        formatter = new MessageFormat( "{0,date} <<{1}>>\n{2}\n" );
    }
    public void append(Message m)
    {
        logFile.println(
            formatter.format( {m.getTimestamp(),m.getSeverity(),m.getText()} )
            );
    }
}
```

LogServiceAppender and other CONCRETERECEIVERS would have similar implementations.

The next code sample illustrates the Motivation's example implemented using C++ templates.

```
template<typename R>
class ReceiverTraits
{
public:
    typedef R*                                    ReceiverPtr;
    typedef typename R::Content                   Content;
    typedef vector<ReceiverPtr>                   ReceiverContainer;
    typedef typename ReceiverContainer::iterator ReceiverIterator;

    template<typename S>
    static void receive(S& sender,const Content& content,ReceiverPtr receiver)
    {
        if ( receiver )
            receiver->receive( sender,content );
    }
};

template<typename R,typename T = ReceiverTraits<R> >
class Sender
{
public:
    typedef typename R::Content           Content;
    typedef typename T::ReceiverPtr       ReceiverPtr;
    typedef typename T::ReceiverContainer ReceiverContainer;
    typedef typename T::ReceiverIterator  ReceiverIterator;
private:
    ReceiverContainer receivers;
public:
    Sender();

    void attach(ReceiverPtr r);
    void detach(ReceiverPtr r);
    void send(const Content& c)
    {
        for(ReceiverIterator r=receivers.begin();r != receivers.end();++r)
            T::receive( *this,c,*r );
    }
};

class MessageAppender // Receiver
{
public:
    typedef Message Content;

    virtual ~MessageAppender() {}
    virtual void append(MessageLogger& logger,const Message& message)=0;
};

template<>
template<>
void ReceiverTraits<MessageAppender>::receive(
    MessageLogger& logger,const Message& message,MessageAppender* appender)
{
    if ( appender )
        appender->append( logger,message );
}
```

```
class LogFileAppender:public MessageAppender // ConcreteReceiver
{
private:
    ofstream logFile;
public:
    LogFileAppender(const string& path): logFile( path.c_str(),true ) {}
    void append(MessageLogger& logger,const Message& message)
    {
        logFile << message.getTimestamp() << " "
                << message.getSeverity() << endl
                << message.getText() << endl;
    }
};

class MessageLogger:private Sender<MessageAppender> // Sender
{
public:
    void log(const Message& message)
    {
        Sender<MessageAppender>::send( *this,message );
    }
};
```

Known Uses

There are several logging frameworks that incorporate the Receiver pattern into their design. These include log4j [Gülcü03], log4cpp [Log4Cpp03], java.util.logging [Sun03], and the Foundation Package's Message Framework from which the Motivation example was derived. The Receiver pattern is also a low level pattern that provides the basic elements of sender-receiver decoupling found in other patterns such as Observer, Mediator, Chain of Responsibility, and Typed Message.

Related Patterns

As stated earlier, Receiver is a general design pattern but what does this mean? One way to answer this question is to organize design patterns in the canonical form of complex systems [Booch93]. Entities in complex systems participate in two kinds of relationships: hierarchy (generalization/specialization) relationships and composition (using/containment) relationships. Applying this form to pattern relationships we can identify many patterns that exhibit the composition relationship. This is what most traditional pattern forms record in their "related patterns" sections [Alexander79][Zimmer95][Noble98]. More recently pattern researchers have shown that some patterns participate in hierarchy relationships [Zimmer95][Noble98]. Some patterns form hierarchies in which general patterns provide the basic elements from which specialized patterns extend, similar in the way an abstract base class is extended by its subclasses.

Receiver is the base of a pattern hierarchy. It is specialized by the Observer, Mediator, Typed Message, and Chain of Responsibility patterns [GHJV95][Vlissides98]. Receiver provides the basic elements of sender-receiver decoupling which the other patterns specialize in the following ways:

- Observer
  - renames Receiver participants:

Receiver Pattern	Observer Pattern
SENDER	SUBJECT
RECEIVER	OBSERVER
CLIENT + SUBJECT	CONCRETESUBJECT
CONCRETERECEIVER	CONCRETEOBSERVER

- usually omits CONTENT but some implementations may send CONTENT describing which aspect of the Receiver.Sender/Observer.SUBJECT has changed.
- constraints Receiver.SENDER/Observer.SUBJECT by only sending when Receiver.SENDER/Observer.SUBJECT state has changed.
- Receiver.CONCRETERECEIVER/Observer.CONCRETEOBSERVER updates itself by querying Observer.CONCRETESUBJECT when it receives a change notification.

- Mediator
  - renames Receiver participants:

Receiver Pattern	Mediator Pattern
SENDER	COLLEAGUE
RECEIVER	MEDIATOR
CLIENT + SUBJECT	CONCRETECOLLEAGUE
CONCRETERECEIVER	CONCRETEMEDIATOR

  - usually omits CONTENT because MEDIATORS only require change notification.
  - constraints Receiver.SENDER/Mediator.COLLEAGUE by only sending when Receiver.SENDER/Mediator.COLLEAGUE state has changed.
  - Receiver.CONCRETERECEIVER/Mediator.CONCRETEMEDIATOR determines which Mediator.COLLEAGUE has changed and then performs the appropriate operations on the other Mediator.COLLEAGUES corresponding to the change.
- Typed Message
  - renames Receiver participants:

Receiver Pattern	Typed Message Pattern
SENDER	SENDERA SENDERB
RECEIVER	ABSTRACTRECEIVERA ABSTRACTRECEIVERB
CONTENT	MESSAGEA MESSAGEB
CONCRETERECEIVER	RECEIVER

  - organizes the pattern around CONTENT/MESSAGE types: SENDERA sends MESSAGEA to ABSTRACTRECEIVERA...
  - Receiver.CONCRETERECEIVER/TypedMessage.RECEIVER provides the unifying implementation for all Receiver.RECEIVERS/TypedMessage.ABSTRACTRECEIVERS
- Chain of Responsibility
  - renames Receiver participants:

Receiver Pattern	Chain Of Resp. Pattern
SENDER + RECEIVER	HANDLER
CONCRETERECEIVER	CONCRETEHANDLER
CONTENT	REQUEST

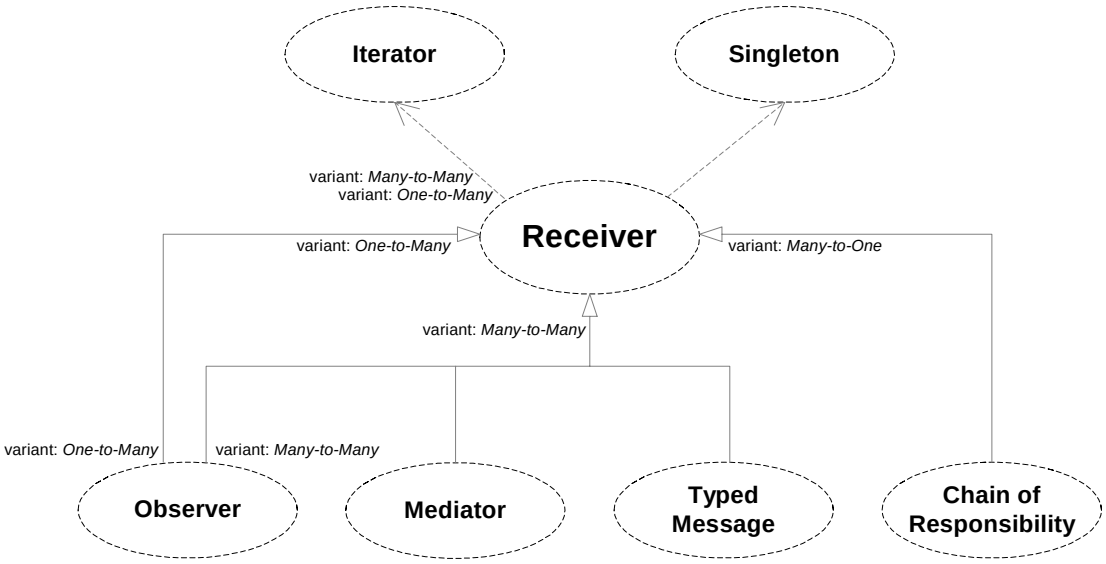
  - combines SENDER and RECEIVER into the HANDLER enabling HANDLERS to both receive REQUESTS and send them to successor HANDLERS if necessary.

- may or may not include Receiver.CONTENT/ChainOfResp.REQUEST depending on how requests are represented.

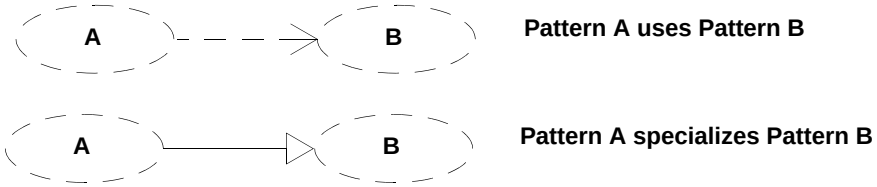
Receiver also participates in composition relationships by using other patterns to implement aspects of itself. The main pattern Receiver uses include:

- Singleton - the Receiver pattern can sometimes use the Singleton pattern to implement ConcreteReceivers that have no intrinsic state.
- Iterator - the Receiver pattern often uses the Iterator pattern to iterate though the RECEIVERS attached to a SENDER so that the SENDER can send them all the CONTENT.

The diagram below illustrates the pattern relationships in which the Receiver participates. The notation is based on Noble's pattern relationship notation [Noble98][NW01] with additional annotations to indicate variants.



Notation Key



References

Alexander79 Alexander, Christopher. The Timeless Way of Building. pp 311-324, Oxford University Press, 1979.

Booch93 Booch, Grady. Object-Oriented Analysis and Design with Applications, 2nd Edition. pp 12-14, Addison-Wesley, 1993.

GHJV95 Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

Gülcü03 Gülcü, Ceki. The Complete Log4j Manual. QOS.ch, 2003.

Log4Cpp03 http://log4cpp.sourceforge.net

Noble98 Noble, James. Classifying Relationships between Object-Oriented Design Patterns . In *Proceedings of the Australian Software Engineering Conference* (ASWEC), Adelaide, IEEE Computer Society Press, 1998.

NW01 Noble, James and Charles Weir. Small Memory Software: Patterns for Systems with Limited Memory . pp 16-17. Addison-Wesley, 2001.

Sun03 http://java.sun.com/j2se/1.4.2/docs/api/java/util/logging/package-summary.html

Vlissides98 Vlissides, John. Pattern Hatching: Design Patterns Applied. pp 123-144. Addison-Wesley, 1998.

Zimmer95 Zimmer, Walter. Relationships between Design Patterns. Pattern Languages of Program Design 1, pp 345-360. Addison-Wesley, 1995.