Matcher-Handler

Frank Metayer

Intent

The Matcher-Handler pattern defines a generalized mechanism for delivering spontaneous data to one or more data handlers in a loosely coupled way. This pattern explicitly separates the responsibilities of data identification (matching) from data handling.

Motivation

Many industrial computer systems are equipped with a variety of input devices that are capable of accepting many different types of data from the environment. For example, point-of-sale devices like cash registers and lottery terminals are often equipped with one or more optical scanning devices. One such device is a vertical barcode reader attached to a cash register. This barcode reader is expected to read barcode data off of store products, coupons, customer cards, etc. Once a barcode has been scanned, the cash register application front-end must process the raw data enough to identify the type of data (e.g., a coupon) so that it may deliver it to the appropriate application object for processing. It is beneficial to develop a generic data routing framework to deliver such spontaneous data.

Such a framework should have the following characteristics. Provide consistent behavior for the input device regardless of the type of data being handled. Clearly define an application-programming interface (API) for introducing support for new input data formats. Minimize the risk of introducing a bug into the application when adding support for new types of data.

The Matcher-Handler pattern can be used as a basis to develop such a framework because of the way that it generalizes data recognition and data delivery. Figure 1 shows the classes involved in the barcode example described here.

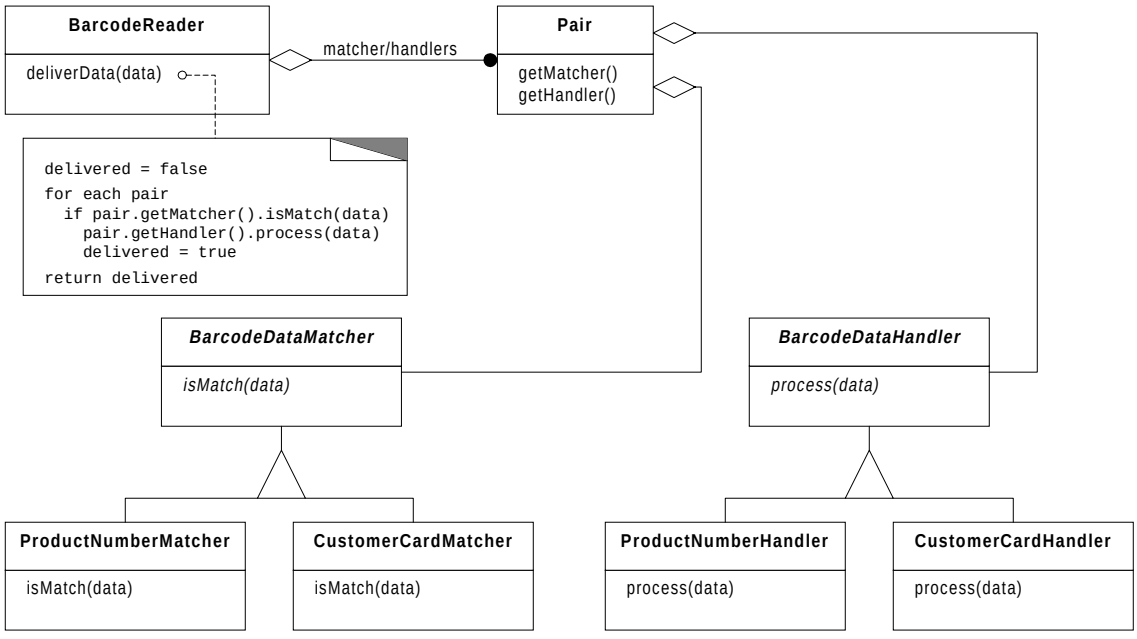

Figure 1 Barcode data handler class relationship

`BarcodeReader` is the class that implements the application front-end that is responsible for accepting data[1] from a device and delivering the data to the proper entity for processing. Its implementation is generic and extendible. It manages a list of matcher/handler pairs and defers all data recognition and processing to those objects respectively.

The abstract classes ***`BarcodeDataMatcher`*** and ***`BarcodeDataHandler`*** define the generalized interface for performing data matching and data handling. Concrete matchers such as **`CustomerCardMatcher`** interrogate the raw data to determine if the data matches a specific criteria such as data size, header content, etc., and return **`true`** from **`isMatch()`** if the data is recognized or **`false`** otherwise. Concrete handlers such as **`CustomerCardHandler`** are only notified when the data has been identified as being specifically for them.

Notice in the pseudo-code for the **`BarcodeReader`** **`deliverData()`** method that more than one handler might receive the same data. This might be beneficial for certain types of data processing. If this is not the desired behavior, the **`for`** loop can be reconstructed to ensure that only one handler receives the data. (To create an even more flexible implementation of **`BarcodeReader`**, this **`for`** loop behavior can be extracted out of the **`BarcodeReader`** class and placed into a strategy [Gamma+95] object that is passed to the **`BarcodeReader`** instance during construction.)

Applicability

Use the Matcher-Handler pattern when

- You want to deliver data to objects without specifying the receivers explicitly.
- A data event may need to be processed simultaneously and independently by more than one object.
- The set of objects that handle spontaneous data events should be specified dynamically.
- A handler may need to process data from more than one data source.
- You expect that new data types are likely to be introduced into the application over time.

Structure

See Figure 2.

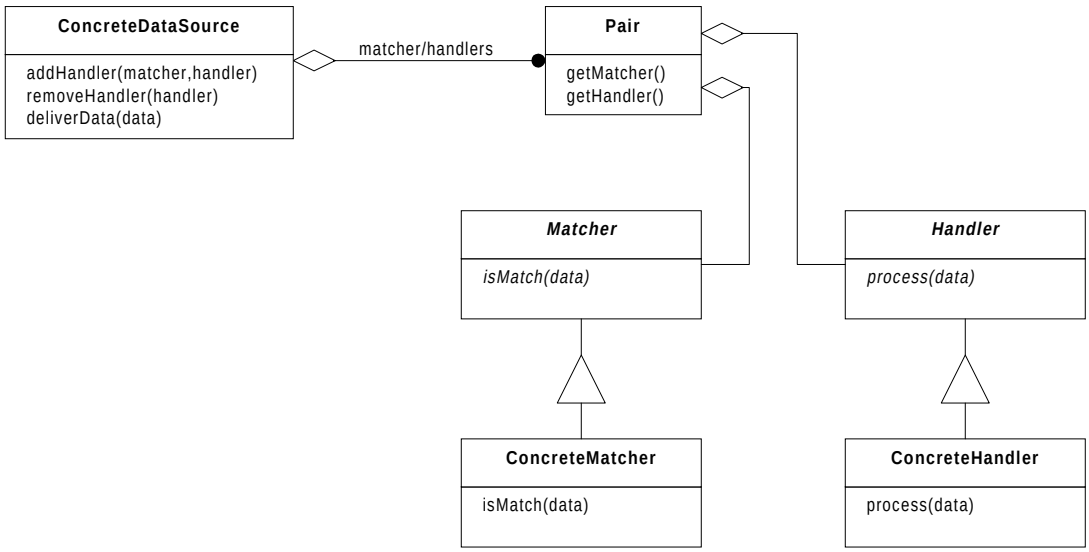

Figure 2 Class relationship

[1] Often, the data is not an object in its own right, but simply raw data that is yet to be identified.

Participants

- **`ConcreteDataSource`** (**`BarcodeReader`**):
  - Manages the data source (i.e., the device) and accepts or generates a data event. It invokes **`deliverData()`** on itself to initiate the delivery of the data to the data handlers.
  - Delivers data to the appropriate handler(s) after first querying a matcher to determine if the associated handler should receive the data.
  - Manages a list of matcher/handler pairs through the registry methods **`addHandler()`** and **`removeHandler()`**.
- ***`Matcher`*** (***`BarcodeDataMatcher`***):
  - Declares the interface that the data source uses to determine if a particular data event matches the criteria required by a particular handler.
- **`ConcreteMatcher`** (**`CustomerCardMatcher`**):
  - Implements the matcher interface for determining whether or not data is recognized.
- ***`Handler`*** (***`BarcodeDataHandler`***)
  - Declares the interface used by the data source to deliver data to a data handler.
- **`ConcreteHandler`** (**`CustomerCardHandler`**)
  - Processes the data that is passed to it by the data source in an application defined way.

Collaborations

1. A **`ConcreteDataSource`** asks each matcher if the data event is recognized as matching whatever criteria the matcher was designed to encapsulate.
2. If the data is matched (i.e., **`isMatch()`** returns **`true`**), the **`ConcreteDataSource`** will deliver the data to the handler associated with the matcher. The handler will process the data.
3. If the matcher does not recognize the data, the handler will not be notified at all. The next matcher in the list is queried.

Consequences

The Matcher-Handler pattern has these advantages.

- *Reduced coupling.* The pattern frees a data source from knowing which other objects process the data events. A data source only knows if the data was accepted or ignored by the application. The data source and the data handler need not have any explicit knowledge of each other.
- *Runtime flexibility of data handling.* Since the list of handlers is specified dynamically, the application can change the current handler set at any time. An application can add or removed handlers because of some mode change and later restore the previous handler configuration using the Memento [Gamma+95] pattern or some similar mechanism.
- *Allows an application to evolve over time.* Matcher-Handler provides the basis for a framework that allows the application to respond to spontaneous data events that are significant to it while allowing unrecognized or unsupported data to be gracefully ignored. This is particularly important during the early development of a new application when whole portions of the application do not yet exist. As new objects are created to support new types of input data, these objects are registered with the appropriate data sources and start receiving input.
- *Reduced risk when introducing new functionality.* The introduction of new code to support new types of input data does not require any code changes to the data source. This means that the Matcher-Handler pattern can be used to develop an extendible framework that can be distributed in binary form (not source code). More importantly, not needing to make

changes to the data source to introduce new functionality means that the risk of breaking existing code is greatly reduced as new handlers are introduced.

Matcher-Handler has this disadvantage.

- *Matchers may know too much about other Matchers.* If the data that is being processed is not an object in its own right, but is instead raw data, then concrete matchers will typically have some knowledge about how other matchers for a particular data source are implemented. This happens because a matcher must know what is unique about its data relative to any other data. Therefore, it must know something about the other types of data.

  For example, barcode data for a store product is typically a number that identifies the product in a database. Barcode data for a customer card may contain a customer's name, address, etc. If it is known that product numbers are always encoded into eight bytes of data, and customer information is always encoded into forty bytes of data, and these are the only two types of data that are processed, then a matcher can be implemented based solely on data size.

  This assumption might break down as new data types are introduced. As previous assumptions break down, matchers will need to be rewritten using new matching criteria. Notice that this problem does not affect the data processing since the processing is isolated in another class, the handler. This is the primary motivation for separating the matcher implementation from the handler implementation.

Implementation

The following issues should be kept in mind when implementing the Matcher-Handler pattern.

- The example outlined in the Motivation section declares the matcher interface to return a boolean value that indicates if the data should be passed to the handler. For non-trivial applications it is probably better to have the matcher return a high-level object that encapsulates the raw data when the data is recognized or **`null`** (or a Null Object [PloPD3]) when the data is not recognized. The object returned by the matcher is then passed by the data source to the Handler. This way the hander doesn't need to reinterpret (or even understand) the raw data. Also, if the handler manipulates high-level objects instead of raw data, it can operate on polymorphic types whose concrete type is known only by the matcher.

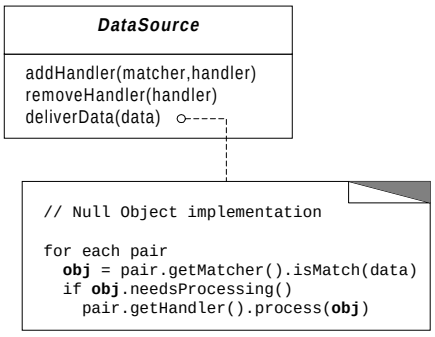

Figure 3 Matcher variation

- Implementing the matcher and handler interfaces in a single class allows one object to perform both the matching and handling operations. This will reduce the number of classes and objects in the application, and therefore the apparent complexity, but with significant risk. If one object is performing both the matching and handling operations, there is a tendency to implement the behaviors in such a way that the matcher behavior passes information to the handler behavior through instance variables. As new data types are introduced and the matching criteria changes (see Consequences), the data handling code will likely break as a side affect of some subtle dependency between the two methods.

- When the processing of data is expected to take a long time, it may be beneficial to restructure the **`deliverData()`** method to query all of the matchers at once and build a list of handlers that should receive the data. Then, pass the data and the list of handlers off to a lower priority background thread for actual processing. With this implementation, the call to the **`deliverData()`** method can return more quickly. This will allow the data source to immediately provide audio and/or visual feedback to the user indicating whether the data was accepted or rejected. (The behavior of the **`deliverData()`** method is subjective to the many factors. Consider implementing the **`deliverData()`** behavior using a Strategy [Gamma+95] pattern.)

- Introducing an abstract super-class to **`ConcreteDataSource`** is useful for an application that has several different but similar data sources. The new super-class will implement the matcher/handler registration and data delivery behavior. The concrete sub-class will focus on device control (which is probably what it should be doing anyway). Adding the abstract super-class makes it easy to implement data handlers to accept data from different data sources. For example, a single handler instance could accept and process data from a barcode reader, a magnetic stripe reader, and a smart card reader, and not make any special provisions for any of the devices.

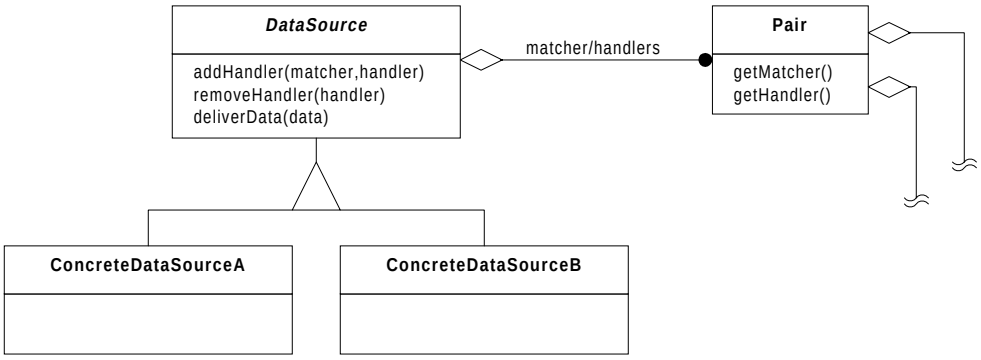

Figure 4 Abstract data source

- The **`Pair`** object that the data source uses to maintain the association between matchers and handlers is just one of several ways to link the two objects together. This variation provides significant flexibility because it allows a single handler instance to be registered with several different matchers, and also allows a single matcher instance to be registered with several different handlers. If this many-to-many relationship is not necessary, consider linking the matchers and handlers together by giving one an explicit reference to the other. The following figure illustrates an implementation where each matcher has an explicit reference to its associated handler.

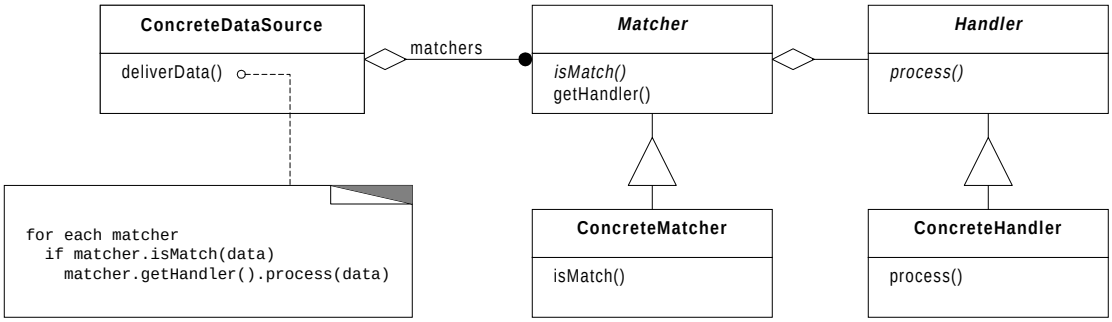

Figure 5 Matchers reference handler

Sample Code

Here is a Java implementation of the barcode reader example discussed in the Motivation section (refer to Figure 1). The sample code starts with the **`BarcodeReader`** class. Because it is not relevant here, all of the code that controls the device is left out.

```
public class BarcodeReader extends Thread
{
  private Vector _handlers;

  // Matcher/Handler registration.
  public void addHandler( BarcodeDataMatcher matcher,
                          BarcodeDataHandler handler )
  {
    _handlers.addElement( new Pair(matcher,handler) );
  }

  // Deliver data to the appropriate handler.
  protected boolean deliverData( byte[] data )
  {
    Enumeration enum  = _handlers.elements();
    boolean delivered = false;

    while( enum.hasMoreElements() )
    {
      Pair pair = (Pair) enum.nextElement();
      if( pair.getMatcher().isMatch(data) )
      {
        pair.getHandler().process( data );
        delivered = true;
      }
    }

    return delivered;
  }

  // Device Thread.  Accept data and deliver it, forever.
  public void run()
  {
    while( true )
    {
      byte[] data = _waitForInput();

      boolean delivered = deliverData( data );

      if( delivered )
        _flashGreen();
      else
        _flashRed();
    }
  }

  ...
}
```

The next two classes are the matcher interface and a concrete matcher implementation that matches reader data for a product number based on data size.

```
public interface BarcodeDataMatcher
{
  public boolean isMatch( byte[] data );
}
```

```
public class ProductNumberMatcher implements BarcodeDataMatcher
{
  public boolean isMatch( byte[] data )
  {
    return data.length == 8;
  }
}
```

The next two classes are the handler interface and an example concrete handler. The product number handler creates a new product instance based on the product number using a factory method [Gamma+95], then posts this new product instance to a sales list singleton [Gamma+95].

```
public interface BarcodeDataHandler
{
  public void process( byte[] data );
}

public class ProductNumberHandler implements BarcodeDataHandler
{
  public void process( byte[] data )
  {
    // 'data' is an ASCII encoded integer
    String ascii = new String( data );
    Integer pid  = Integer.decode( ascii );
    Product prod = Product.newInstance( pid );

    Sales.getInstance().add( prod );
  }
}
```

The main application code might look something like this.

```
public class Application
{
  public static void main( String[] args )
  {
    BarcodeReader reader = new BarcodeReader();

    BarcodeDataMatcher matcher;
    BarcodeDataHandler handler;

    matcher = new ProductNumberMatcher();
    handler = new ProductNumberHandler();
    reader.addHandler( matcher, handler );

    matcher = new CustomerCardMatcher();
    handler = new CustomerCardHandler();
    reader.addHandler( matcher, handler );

    // add more handlers...

    reader.start();
  }
}
```

Known Uses

- **Altura[2] Image Reader.** An image reader scans in documents such as receipts, play slips, player registration cards, etc. The data obtained from the scan is passed along to the application by traversing a list of matcher-handler pairs. The data is delivered only to the handlers that want it. The handlers are called from a separate background thread so that the image reader can quickly provide accept/reject feedback to the user.
- **Altura Communications Framework.** While a lottery terminal is running, it receives many unsolicited messages from the central computer system. These messages range from news messages, to game closing notifications, to winning numbers announcements. The several types of messages have handlers that are very different from one another. For example, a game closing notification is delivered to a game prototype [Gamma+95] running in the terminal and tells the game that it can no longer be played. A winning numbers announcement is delivered to a display object so that the numbers can be displayed to the customers. A winning numbers announcement may also be delivered to a game prototype, signaling to the game that it may once again be played for the next drawing. In this implementation, there is only one matcher class for unsolicited message matching. The matcher reads the message header and compares fields within the header to values passed to its constructor.
- **Windows Registry.** When a Windows 95/NT application installs, it typically identifies itself to the registry. The application informs the registry that for particular file types, it can handle requests like Open, Print, etc. When the user issues a request to Open a specific file, Windows determines the file type (from the file extension) and matches that file type against the file types recorded in the registry. If a match is found, the corresponding application that is registered to support the Open request is invoked to process the Open.
- **Smalltalk Method Selection.** When a message is sent to an object, Smalltalk uses a method-lookup mechanism to determine which method to invoke to process the message. The signature of the message is compared to the method signatures that the object's class defines. If a match is found, the corresponding method is invoked to handle the message.
- **RPC Service Dispatch.** When a server receives a Remote Procedure Call, a dispatcher identifies the local procedure that should service the request and invokes that procedure to handle the request.

Related Patterns

Like the Observer [Gamma+95] pattern, Matcher-Handler addresses the problem of distributing information *as it happens* to the objects throughout the application that are interested.

Observer focuses on notifying application objects when a subject changes state. Observers of a subject are notified for any change in state, not necessarily the state change that they are interested in. In contrast, Matcher-Handler notifies handlers (observers) only when the information that they are specifically waiting for has arrived at the data source (subject).

The matcher object in the Matcher-Handler pattern is an example of the Strategy [Gamma+95] pattern, though the motivation for it is somewhat different. The purpose of the Strategy pattern is to provide selectable behavior, particularly at runtime. The matcher object however was motivated by the observation that matching criteria changes over the lifetime of the application and it makes sense to isolate that behavior to reduce the impact of the changes to the rest of the system.

The Matcher-Handler pattern is similar to the Chain of Responsibility [Gamma+95] pattern in that it avoids coupling the data source (client) to its handlers, it allows more than one object to handle a request, and it allows the handlers to be dynamically specified. The structure of Matcher-Handler differs from that of Chain of Responsibility in that handler sequencing is delegated to the data source in Matcher-Handler while data sequencing is established and maintained by linking handlers together in Chain of Responsibility.

---

[2] Altura™ is a lottery terminal developed by GTECH Corporation. The application that runs the terminal is a framework-based object-oriented implementation written in Java.

REFERENCES

[Gamma+95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[PloPD3] R. Martin, D. Riehle, F. Buschmann. *Pattern Languages of Program Design*. Reading, MA: Addison-Wesley, 1998.