DESIGN PATTERN - INTERCEPTING FILTER PATTERN

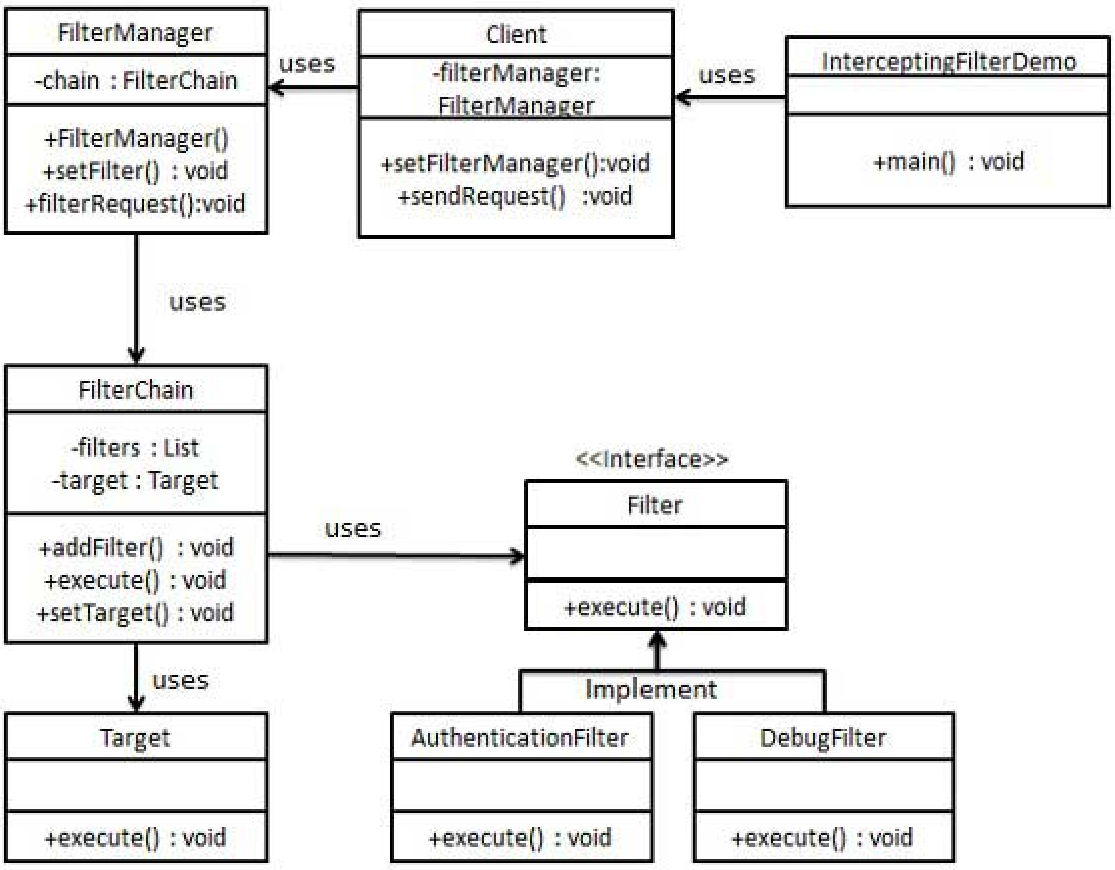
The intercepting filter design pattern is used when we want to do some pre-processing / post-processing with request or response of the application. Filters are defined and applied on the request before passing the request to actual target application. Filters can do the authentication/ authorization/ logging or tracking of request and then pass the requests to corresponding handlers. Following are the entities of this type of design pattern.

- **Filter** - Filter which will performs certain task prior or after execution of request by request handler.
- **Filter Chain** - Filter Chain carries multiple filters and help to execute them in defined order on target.
- **Target** - Target object is the request handler
- **Filter Manager** - Filter Manager manages the filters and Filter Chain.
- **Client** - Client is the object who sends request to the Target object.

Implementation

We are going to create a *FilterChain*,*FilterManager*, *Target*, *Client* as various objects representing our entities.*AuthenticationFilter* and *DebugFilter* represent concrete filters.

InterceptingFilterDemo, our demo class, will use *Client* to demonstrate Intercepting Filter Design Pattern.



Step 1

Create Filter interface.

Filter.java

```java
public interface Filter {
   public void execute(String request);
}
```

Step 2

Create concrete filters.

AuthenticationFilter.java

```java
public class AuthenticationFilter implements Filter {
   public void execute(String request){
      System.out.println("Authenticating request: " + request);
   }
}
```

DebugFilter.java

```java
public class DebugFilter implements Filter {
   public void execute(String request){
      System.out.println("request log: " + request);
   }
}
```

Step 3

Create Target

Target.java

```java
public class Target {
   public void execute(String request){
      System.out.println("Executing request: " + request);
   }
}
```

Step 4

Create Filter Chain

FilterChain.java

```java
import java.util.ArrayList;
import java.util.List;

public class FilterChain {
   private List<Filter> filters = new ArrayList<Filter>();
   private Target target;

   public void addFilter(Filter filter){
      filters.add(filter);
   }

   public void execute(String request){
      for (Filter filter : filters) {
         filter.execute(request);
      }
      target.execute(request);
   }

   public void setTarget(Target target){
      this.target = target;
   }
}
```

Step 5

Create Filter Manager

FilterManager.java

```
public class FilterManager {
   FilterChain filterChain;

   public FilterManager(Target target){
      filterChain = new FilterChain();
      filterChain.setTarget(target);
   }
   public void setFilter(Filter filter){
      filterChain.addFilter(filter);
   }

   public void filterRequest(String request){
      filterChain.execute(request);
   }
}
```

Step 6

Create Client

Client.java

```
public class Client {
   FilterManager filterManager;

   public void setFilterManager(FilterManager filterManager){
      this.filterManager = filterManager;
   }

   public void sendRequest(String request){
      filterManager.filterRequest(request);
   }
}
```

Step 7

Use the *Client* to demonstrate Intercepting Filter Design Pattern.

InterceptingFilterDemo.java

```
public class InterceptingFilterDemo {
   public static void main(String[] args) {
      FilterManager filterManager = new FilterManager(new Target());
      filterManager.setFilter(new AuthenticationFilter());
      filterManager.setFilter(new DebugFilter());

      Client client = new Client();
      client.setFilterManager(filterManager);
      client.sendRequest("HOME");
   }
}
```

Step 8

Verify the output.

```
Authenticating request: HOME
request log: HOME
Executing request: HOME
```