AsyncHttpClient

*Ajayveer Aujla 6595863*

*Mridula Dubey 6319173*

*Mark Massoud 6599486*

*Pirasath Vallipuram 6631258*

*Wei Yee Yap 9220712*

# Summary of Project

The Async Http Client Java library is a project which allows applications to make HTTP requests while asynchronously processing the responses from the servers. Furthermore, it also provides support for the WebSocket Protocol.

# Class Diagram of Actual System

Below is the conceptual diagram from Milestone 2 for the provider, Netty, which represents the scoped down topic selected from the entire AsyncHttpClient project.

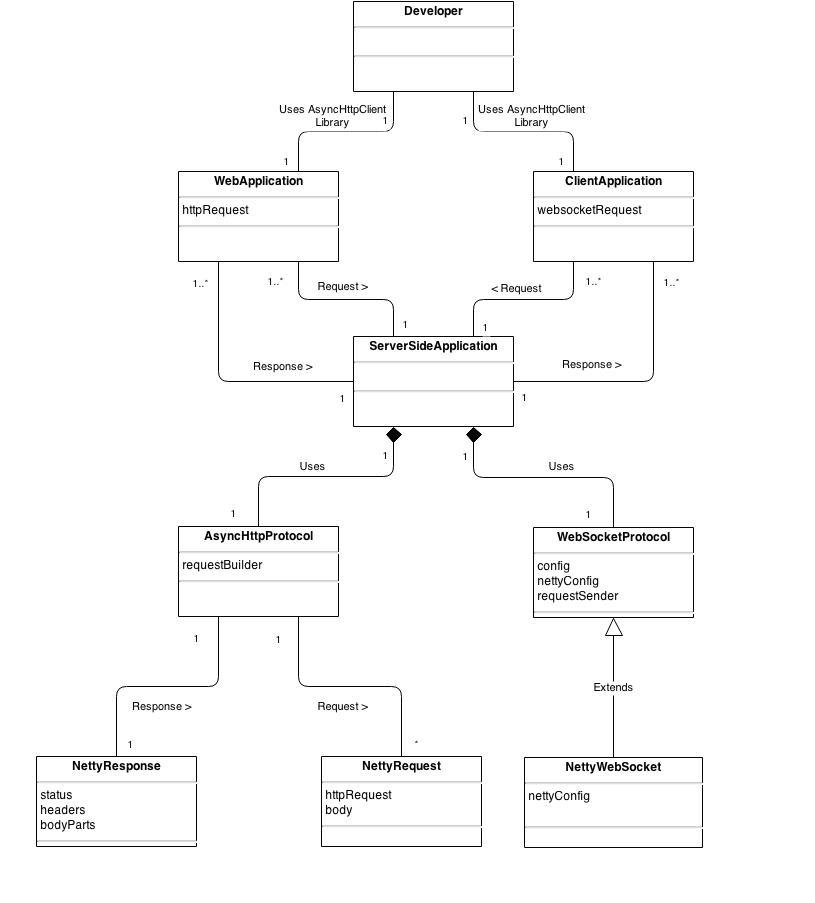


Figure : Conceptual Diagram of Netty from Milestone 2

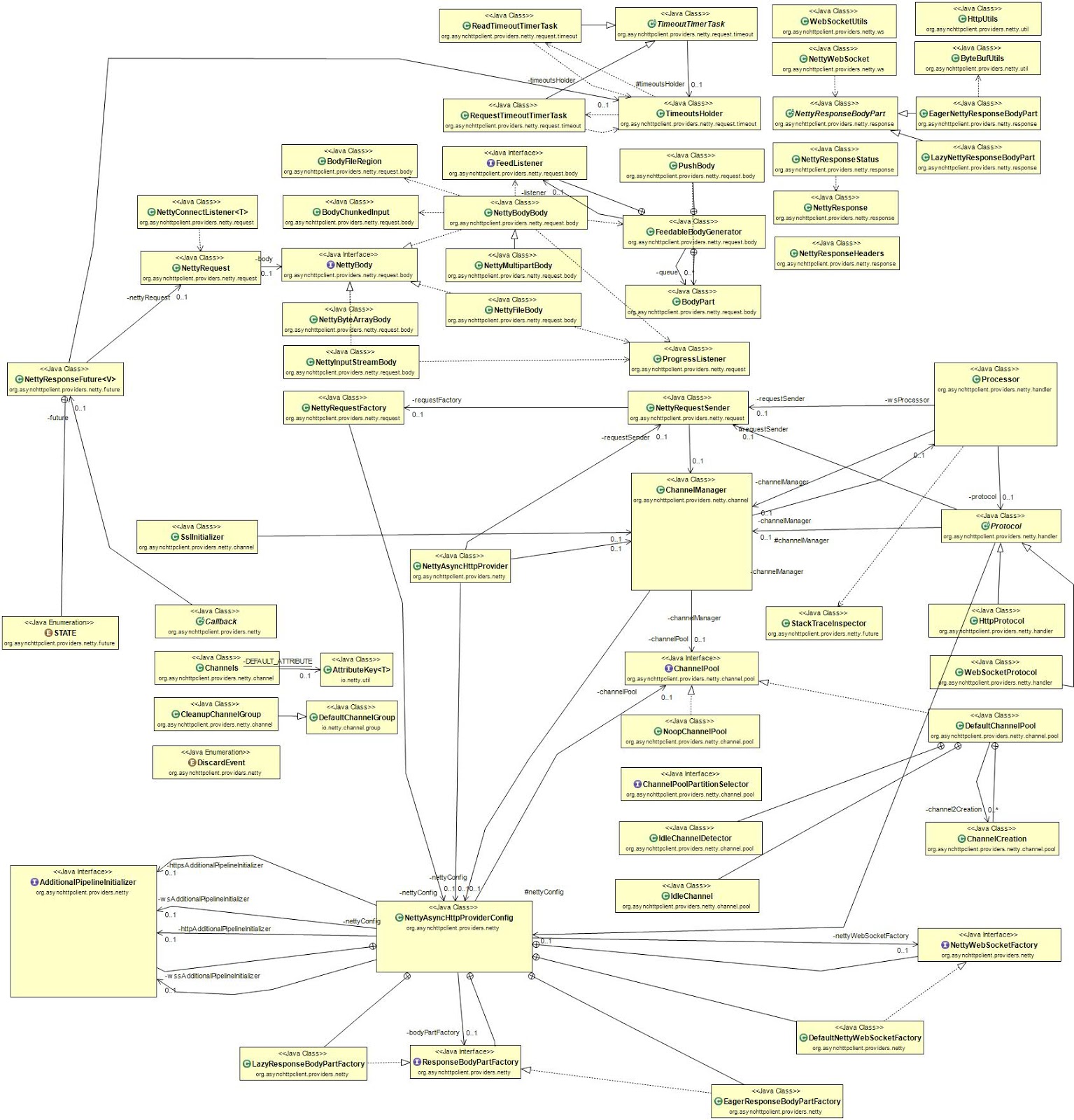
Below is a domain model consisting of the entire AsyncHttpClient project which was scoped down to focus on the elements concerned with the provider, Netty.

Figure : Domain Diagram of the entire AsyncHttpClient library

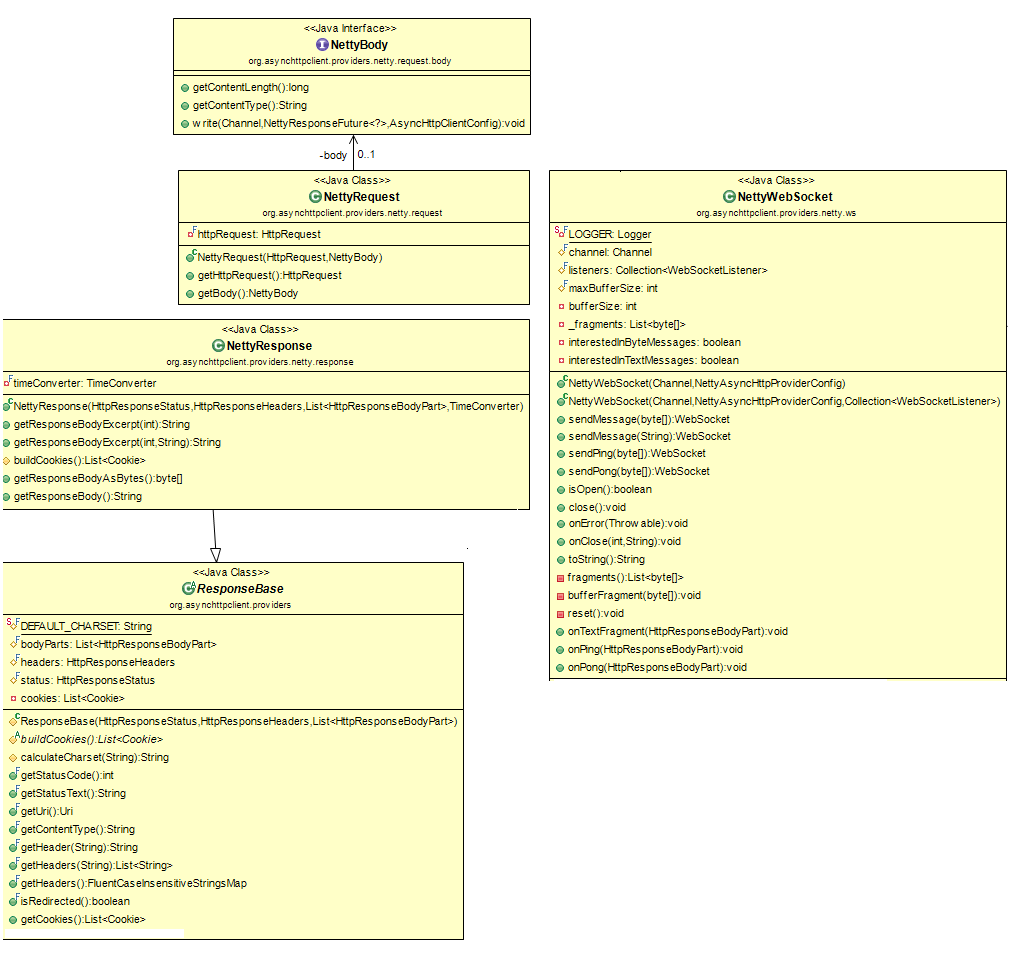
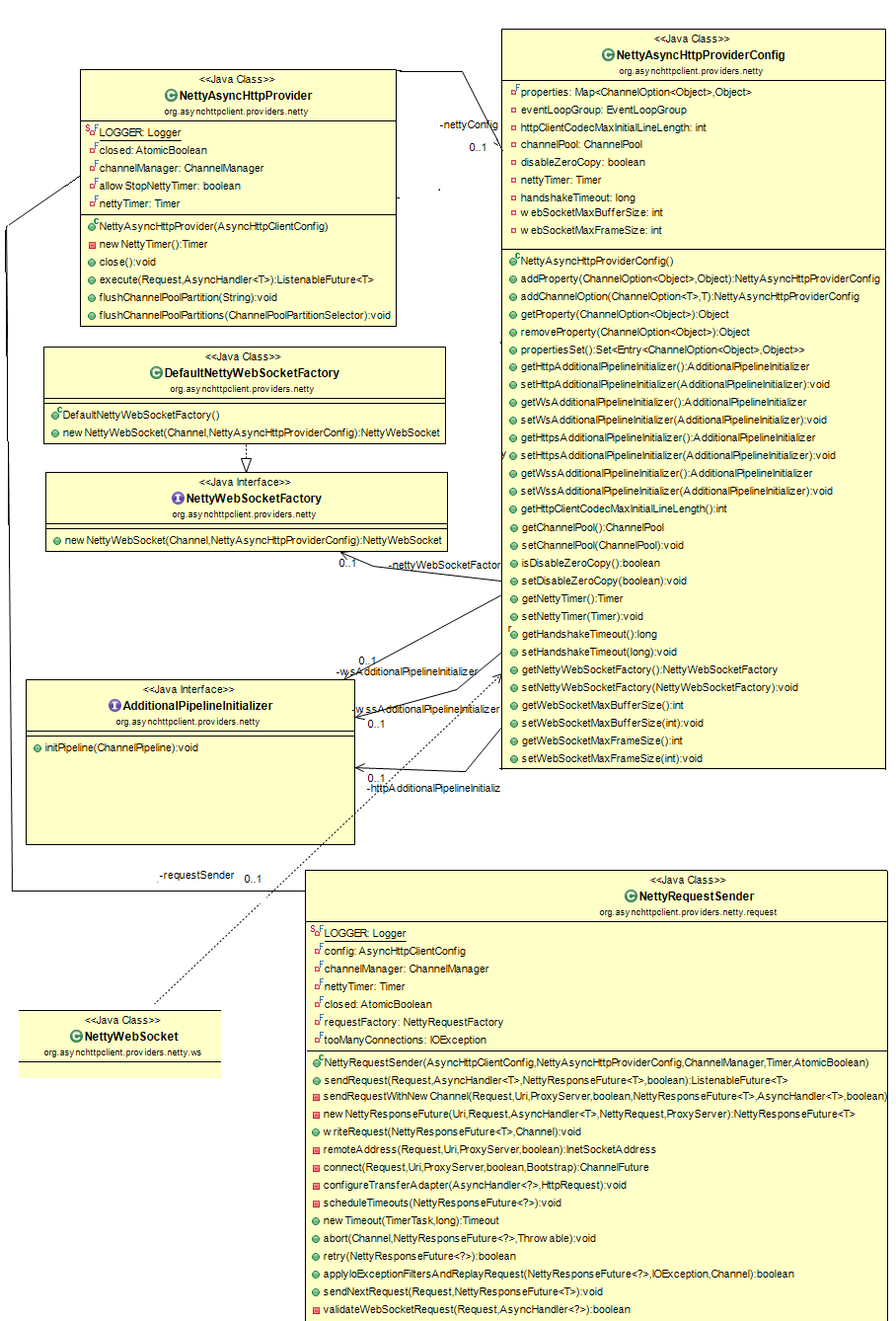
Below is the scoped down UML class diagram for Netty, representing the classes of interest for this milestone.

Figure : Scoped Class Diagram of the provider, Netty

Figure 3: Scoped Class Diagram of the provider, Netty

The diagram presented above is the scoped down AsyncHttpClient class diagram which focuses on the asynchronous HTTP provider, Netty. The diagram shows all the packages included in the Netty source code and it is composed of all the important relationship between the significant classes which make up Netty. Certain dependencies were omitted from the diagram as they were neither necessary nor as important as others, and also they would have clustered the diagram had they been included alongside the significant ones. Also it the config class helped to decide which portion of source code to emphasize for code smell and refactoring.

Taking a closer look at the entire diagram, it is evident that the NettyAsyncProviderConfig class in the package org.asynchttpclient.providers.netty, is associated to a number of other classes. Therefore, t was deemed significant to take a closer look at  this package and  describe further relationship between classes within this package.

The diagram above contains all the classes, interfaces, and factories that will be the focus for the analysis and refactoring of this project.

The majority of elements within the diagram are clearly depicted as being related to the **NettyAsyncHttpProviderConfig** class, whether by dependency (not shown in this UML) or by association. NettyAsyncHttpProviderConfig  is the class that is associated with assigning the channel pool, establishing  and configure channel pipelines, buffer size, and length. Furthermore, it is also in charge of establishing timing concerns related to receiving data. In terms of the handshake timeout and how long the channel remains active, the class maintains full control over the AsyncHttp communication.  NettyAsyncHttpProviderConfig hosts all the accessor and mutator methods required to properly configure the data members for a connection between a server and a client. It is associated with the interface, NettyWebSocketFactory for creating a NettyWebSocket instance.

**DefaultNettyWebSocketFactory** simply extends the interface, **NettyWebSocketFactory** and defines an implementation for its only method. This method takes an instance of NettyAsyncHttpProviderConfig as well as a Channel as parameters. This method yields a new NettyWebSocket instance upon execution.

The **NettyWebSocket** class is  dependent on the NettyAsyncHttpProviderConfig object, which is required to appropriately assign an active channel and listeners  to the web socket. This class defines the buffer size and has appropriate methods in charge of data transactions and  byte interpretation. This class’s methods help define a different communication paradigm from browser communication protocols which provides bi-directional, full duplex communications channels, over a given protocol socket.

NettyAsyncHttpProviderConfig is also associated with **NettyAsyncHttpProvider** by its private attribute, nettyConfig. NettyAsyncHttpProvider takes as parameter an AsyncHttpClientConfig object which, based on the client’s configurations, the NettyAsyncHttpProvider class will make the appropriate request by communicating with NettyRequestSender class.

The **NettyRequestSender** class offers an important method, sendRequest(...), which sends a request from the client to the server, taking into account the nettyTimer, a ChannelManager, and both the client’s and the provider’s configurations.

The **NettyRequest** sent is initialized with the HttpRequest as well as the NettyBody (the request itself). The **NettyBody** interface associated with the NettyRequest has accessor methods to this NettyBody.

The response to the request is the **ResponseBase**. This ReponseBase is initialized with a HttpResponseStatus, the HttpResponseHeaders, and the HttpResponseBodyPart, in the form of List.

**NettyResponse** extends the ResponseBase class by also adding a TimeConverter object as a parameter. This parameter compares the time between the communication of the client and the server in order to determine whether the connection should be closed or not, via the nettyTimer object.

There are discrepancies between the conceptual classes and actual classes because the actual classes possess detailed class definitions which describe how to use the class. To attain a high level understanding of the code, one could look at the conceptual classes and quickly understand what the architecture is attempting to achieve.

To take it one step further, one who plans on implementing an HTTP or web socket application would need the actual classes to fully understand the relationships and flow of data. Since lots of planning went into coming up with the classes, development should be much quicker. The developer can always reference the actual classes as needed. If they used the conceptual classes to design an application, they might succeed, however it would be missing many features and would most likely be arise in many bugs since many details would be missed. The impact is a properly working system versus an incomplete system.

The ObjectAid UML Explorer tool was used to generate class diagram and sequence diagram. It is a code visualization tool for the IDE, Eclipse. It illustrates Java source code and libraries in the form of UML class diagrams and sequence diagrams. These diagrams update automatically to reflect code changes made even after generating the diagrams. The tool gives the user the flexibility to choose what elements should appear in the chosen diagram (i.e. one can select different types of relationships to show or hide). The tool is very simplistic and offers a solution to the purpose of creating a class diagram for given source code, however it does not offer the user the ability to perform certain modifications. For example, user does not have the freedom to delete a select group of attributes or methods from class diagram view only (simply altering the class diagram without changing the code). It also does not offer any visualization options such as changing the color of classes or highlighting them (which would aid in making the diagrams both visually appealing and facilitate differentiation between packages by having unique colors for separate packages).

Directly from the source code (after removing uncessary code), the relationship between the classes, NettyAsyncHttpProvider and NettyRequestSender, is depicted below.

public class NettyAsyncHttpProvider implements AsyncHttpProvider {

private static final Logger LOGGER = LoggerFactory.getLogger(NettyAsyncHttpProvider.class);

private final NettyAsyncHttpProviderConfig nettyConfig;

private final AtomicBoolean closed = new AtomicBoolean(false);

private final ChannelManager channelManager;

private final NettyRequestSender requestSender;

private final boolean allowStopNettyTimer;

private final Timer nettyTimer;

public NettyAsyncHttpProvider(AsyncHttpClientConfig config);

public <T> ListenableFuture<T> execute(Request request, final AsyncHandler<T> asyncHandler);

}

public final class NettyRequestSender {

private static final Logger LOGGER = LoggerFactory.getLogger(NettyRequestSender.class);

private final AsyncHttpClientConfig config;

private final ChannelManager channelManager;

private final Timer nettyTimer;

private final NettyRequestFactory requestFactory;

public NettyRequestSender(AsyncHttpClientConfig config, NettyAsyncHttpProviderConfig nettyConfig, ChannelManager channelManager, Timer nettyTimer, AtomicBoolean closed);

public <T> ListenableFuture<T> sendRequest(final Request request, final AsyncHandler<T> asyncHandler, NettyResponseFuture<T> future, boolean reclaimCache);

private <T> NettyResponseFuture<T> newNettyResponseFuture(Uri uri, Request request, AsyncHandler<T> asyncHandler, NettyRequest nettyRequest, ProxyServer proxyServer);

private boolean validateWebSocketRequest(Request request, AsyncHandler<?> asyncHandler);

}

# Code Smells and System Level Refactorings

HttpProtocol.java:

A complicated, composite boolean statement is returned from handleHttpResponse function. This is a code smell that can be fixed with Decompose Conditional refactoring. The purpose for this refactoring is to simplify the complex conditional logic which aids in legibility and understandability. As it currently stands, the return statement obscures the intentions of the condition since there are a large number of atomic boolean operations joined with boolean OR statements. The composite conditional statement should be extracted into its own method with a descriptive and explicative name (exitAfterHandlingResponseComponents(...)), have the appropriate parameters being passed in (including an instance of Channel, Request, NettyResponseHeaders, integer representing statusCode, NettyResponseStatus, etc.) and applied to the rightful places, and then this new method should be called in that return statement from handleHttpResponse. This one function can replace a series of eight atomic booleans.

Specifically, the code segment is as seen below.

return exitAfterProcessingFilters(channel, future, handler, status, responseHeaders)

|| exitAfterHandling401(channel, future, response, request, statusCode, realm, proxyServer) || //

exitAfterHandling407(channel, future, response, request, statusCode, realm, proxyServer) || //

exitAfterHandling100(channel, future, statusCode) || //

exitAfterHandlingRedirect(channel, future, response, request, statusCode) || //

exitAfterHandlingConnect(channel, future, request, proxyServer, statusCode, httpRequest) || //

exitAfterHandlingStatus(channel, future, response, handler, status) || //

exitAfterHandlingHeaders(channel, future, response, handler, responseHeaders);

It can be refactored with a method as below.

public boolean exitAfterHandlingResponseComponents(channel, future, handler, status, responseHeaders, statusCode, realm, proxyServer){

return exitAfterProcessingFilters(channel, future, handler, status, responseHeaders)

|| exitAfterHandling401(channel, future, response, request, statusCode, realm, proxyServer) || //

exitAfterHandling407(channel, future, response, request, statusCode, realm, proxyServer) || //

exitAfterHandling100(channel, future, statusCode) || //

exitAfterHandlingRedirect(channel, future, response, request, statusCode) || //

exitAfterHandlingConnect(channel, future, request, proxyServer, statusCode, httpRequest) || //

exitAfterHandlingStatus(channel, future, response, handler, status) || //

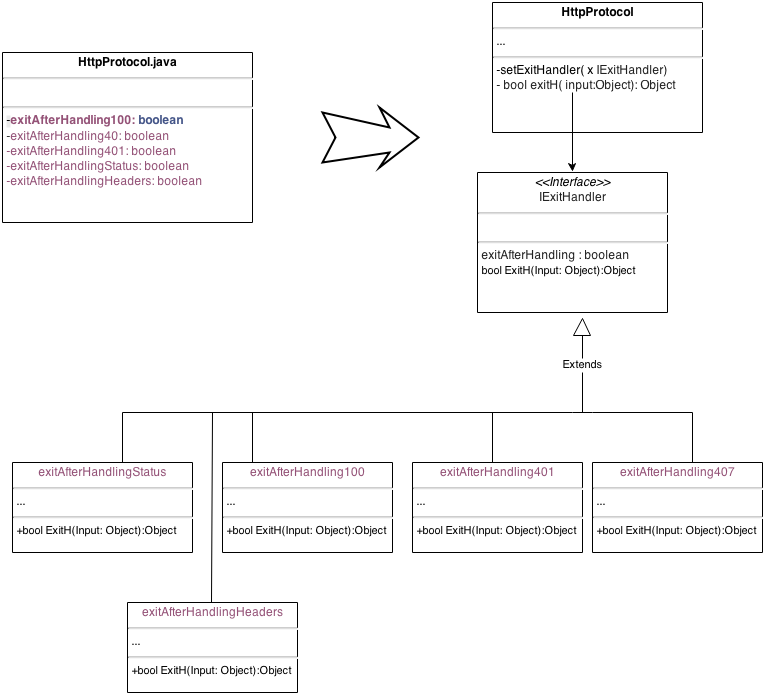
exitAfterHandlingHeaders(channel, future, response, handler, responseHeaders);

}

Then in the initial method will then have this code as its return statement.

return exitAfterHandlingResponseComponents(channel, future, handler, status, responseHeaders, statusCode, realm, proxyServer);

Refactoring similar code using Extract Methods would move responsibility into a separate class for two methods similar in implementation found in the HttpProtocol class. The methods exitAfterHandling401(...) and exitAfterHandling407(...), they can be extracted to another class to eliminate redundancy in the HttpProtocol class itself since a lot of the code is similar. With addition, methods that would share similar logic we thought it would be smart to incorporate the strategy pattern. In which the ExitHandler class will will contain a strategy bool ExitH(...) that is either of Strategy401 or Strategy407.



DefaultAsyncHttpClient.java

In the one of the overloaded executeRequest(...) methods of this class, a message chain code smell is present, which can be fixed by using refactoring with Hide Delegate. The message chain arises when the FilterContext asks FilterContextBuilder for an object who asks asyncHandler for an object, and so on, etc. This chain entails high coupling. The purpose of this is to reduce the length of the long method chain by creating a simple delegating method for the initial object (client) to call from the subsequent objects.

Specifically, the code is initially as seen below.

FilterContext<T> fc = new FilterContext.FilterContextBuilder<T>().asyncHandler(handler).request(request).build();

It can be rewritten as below after following the Hide Delegate refactoring.

FilterContext<T> fc = new FilterContext.FilterContextBuilder<T>().buildRequest(handler, request);

public buildRequest (handler, request) {

return asyncHandler(handler).request(request);

}

# Specific Refactorings that you will implement in Milestone 4

Two refactorings were deemed of interest to pursue in Milestone 4.

AsyncHttpClientConfig.java

This file has many arguments in the constructor. This code smell is Long Parameter List. The suggested remedy is to use the Replace Parameter with Method approach. Another approach is to use the Builder pattern which creates an object from the parameters. The object is then used as a parameter for the constructor.

NettyRequestFactory.java

This file contains two areas where refactoring is possible. The first is the swithc-case statement. We would use the Strategy design pattern to rewrite the code. Each case scenario would be their own class that implements adapter class which represents authorization schemas. The second is an extremely large if-clause which we will solve with the Strategy pattern as well.

public final class NettyRequestFactory {

… <snippet>

public String firstRequestOnlyAuthorizationHeader(Request request, Uri uri, ProxyServer proxyServer, Realm realm) throws IOException {

String authorizationHeader = null;

if (realm != null && realm.getUsePreemptiveAuth()) {

switch (realm.getAuthScheme()) {

case NTLM:

String msg = NTLMEngine.INSTANCE.generateType1Msg();

authorizationHeader = "NTLM " + msg;

break;

case KERBEROS:

case SPNEGO:

String host;

if (proxyServer != null)

host = proxyServer.getHost();

else if (request.getVirtualHost() != null)

host = request.getVirtualHost();

else

host = uri.getHost();

try {

authorizationHeader = "Negotiate " + SpnegoEngine.instance().generateToken(host);

} catch (Throwable e) {

throw new IOException(e);

}

break;

default:

break;

}

}

return authorizationHeader;

}

private String systematicAuthorizationHeader(Request request, Uri uri, ProxyServer proxyServer, Realm realm) {

… <snippet>

Much of the same code from firstRequestOnlyAuthorizationHeader

}

private NettyBody body(Request request, HttpMethod method) throws IOException {

NettyBody nettyBody = null;

if (method != HttpMethod.CONNECT) {

Charset bodyCharset = request.getBodyEncoding() == null ? DEFAULT\_CHARSET : Charset.forName(request.getBodyEncoding());

if (request.getByteData() != null) {

nettyBody = new NettyByteArrayBody(request.getByteData());

} else if (request.getStringData() != null) {

nettyBody = new NettyByteArrayBody(request.getStringData().getBytes(bodyCharset));

} else if (request.getStreamData() != null) {

nettyBody = new NettyInputStreamBody(request.getStreamData());

} else if (isNonEmpty(request.getFormParams())) {

String contentType = null;

if (!request.getHeaders().containsKey(HttpHeaders.Names.CONTENT\_TYPE))

contentType = HttpHeaders.Values.APPLICATION\_X\_WWW\_FORM\_URLENCODED;

nettyBody = new NettyByteArrayBody(computeBodyFromParams(request.getFormParams(), bodyCharset), contentType);

} else if (isNonEmpty(request.getParts())) {

nettyBody = new NettyMultipartBody(request.getParts(), request.getHeaders(), nettyConfig);

} else if (request.getFile() != null) {

nettyBody = new NettyFileBody(request.getFile(), nettyConfig);

} else if (request.getBodyGenerator() instanceof FileBodyGenerator) {

FileBodyGenerator fileBodyGenerator = (FileBodyGenerator) request.getBodyGenerator();

nettyBody = new NettyFileBody(fileBodyGenerator.getFile(), fileBodyGenerator.getRegionSeek(), fileBodyGenerator.getRegionLength(), nettyConfig);

} else if (request.getBodyGenerator() instanceof InputStreamBodyGenerator) {

nettyBody = new NettyInputStreamBody(InputStreamBodyGenerator.class.cast(request.getBodyGenerator()).getInputStream());

} else if (request.getBodyGenerator() != null) {

nettyBody = new NettyBodyBody(request.getBodyGenerator().createBody(), nettyConfig);

}

}

return nettyBody;

}

… <snippet>

}