Katholieke
Universiteit
Leuven

**Department of
Computer Science**

PROJECT REPORT

DESIGN OF SOFTWARE SYSTEMS

Daniel Andrés Pérez Pérez
Jasper Marien
Robin Vanden Ecker
Romain Carlier

Contents

1 Design decisions

1.1 Iteration 1 & 2: Image Filtering

We concluded that the best way to add this new functionalities was creating a new instantiation of *FilterAdapter*.
Because we were constrained by the external configuration file, *FilterReplaceImage* operates as the coordinator between the *ConfigurationReader* and the concrete implementations of *SimpleImageFilter*.
ConfigurationReader is implemented as a factory class. It parses the configuration file and creates instance of *SimpleImageFilter* accordingly.
One of the important design decisions was that the configuration file contains the name of the concrete classes of *SimpleImageFilter* and the order in which the filter are applied is established by the same order of the configuration file.
For iteration 1 and 2, we added a zap filter by extending *FilterAdaptor*. The subclass is *FilterReplaceImage* and overrides the textitonHttpResponseReceive method. *FilterAdaptor*. *FilterReplaceImage* extracts the *BufferedImage* from the *HttpMessage*. It finds all the filter operations to apply and passes the *BufferedImage* on to each of them in turn to be modified. The actual filters are subclasses of *SimpleImageFilter*. The *ConfigurationReader* finds them, given the url of the config file.

1.2 Iteration 3: Content Filtering

We separated the distinct responsibilities among several (object) classes. *FilterHttpContent* is the link to the restof ZAP. It extends *FilterAdaptor* in the parosproxy filter package. Upon receiving an HTTP response, it only verifies the *HttpMessage* has content. Then it instantiates a *FilterApplyer* (concrete subclass) and calls one of its filtering methods with 2 parameters. Currently only one is available, but other methods can be added to offer various filtering algorithms. The 2 parameters are the *HttpMessage* and the url of the file containing the filter terms and additional info.
The *FilterApplyer* then instantiates 2 helpers. One, a *PageContent*, represents a given (upon creation) *HttpMessage*'s content. Subclasses provide the content with the desired type, *String* for the assignment. The other, a *FormatFileToFilterInfo*, parses the file at given url and returns the useful information. Currently only one parsing method is available, that supports the format of the assignment and returns a *Pair*. This pair consists of the weight threshold and a list of *InappropriateElement* instances. *InappropriateElement* models inappropriate content, of generic type, its weight and explanatory tags. To support different formats, other parsing methods can be added. Next, the *FilterApplyer* combines both result and filters the content using the found inappropriate elements. An alternative with lower coupling would be by sending a closure from one class to the next one and never returning results.

1.3 Iteration 4: Content Reporting

It was created a new package to be consistent with the naming that zaproxy has. The package *org.zaproxy.zap.extension.imgreport* encapsulates the classes used for the extension. The class *ExtensionImageReport* extends *ExtensionAdaptor* (creates, initializes and

hooks a new extension), *XmlReporterExtension* (gets our XML format which will be added to the zaproxy report extension) and *HttpSenderListener* (converts our new extension in an observer object which is able to catch all the HttpMessages).
ExtensionImageReport validates whether *HttpMessage* content is an image content, stores *HttpImage* objects and delegate the creation of specific statistics format to the *ImageStatistics* classes.
ImageStatisticsFactory instantiates new concrete classes of *ImageStatistics*.
HttpImage processes *HttpMessage* and returns the corresponding object.
ImageDimensionStatistics is an template class implementation of *ImageStatistics* used by *ImageHeightStatistics*, *ImageSizeStatistics* and *ImageWidthStatistics*. *ImageTypeStatistics* is an implementation of *ImageStatistics* that creates a unique XML format.
This *ExtensionImageReport* was implemented as core functionality since the given XML format to the ReportExtension relies on XSL style sheets to add the new information in HTML and MarkDown reports thus those corresponding XSL files were properly updated.

1.4 Iteration 5: Refactoring

1.4.1 org.parosproxy.paros.core.proxy.ExtensionLoader

We first tackled the massive code duplication in the *ExtensionLoader* class. Most of the time, we used Java 8's support for lambda expressions and closures. Next we split off 3 components from the *ExtensionLoader* that took over distinct responsibilities: *MenuHandler*, *ExtensionList* and *HookProxyLinkerManager*. All JMenu related methods moved from *ExtensionLoader* to *MenuHandler*. This relieves the *ExtensionLoader* from a responsibility unrelated to its main concerns, improbing cohesion. We applied the proxy pattern and extracted the extension list and the extension map from the *ExtensionLoader*. Instead, the *ExtensionLoader* has an *ExtensionList* field. *ExtensionList* encapsulate both the list and the map. It provides most methods to manipulate them. We also made a minor improvement to methods going over all extensions. They now use a for-each (over the private list) rather than an indexed for-loop and the public 'getExtension(i)'. The order is preserved so the behavior is the same. The *HookProxyLinkerManager* handles the linking of *ExtensionHooks* to the *Proxy*, or to the *SiteMapPanel* in one case, via the corresponding (proxy) listeners. It does so by managing *HookProxyLinkers* and providing methods to interact with them. A (subclass of a) *HookProxyLinker* encapsulates all operations for (un)linking an *ExtensionHooks* to the *Proxy*. Reflection greatly reduced the parameter passing and code duplication. It was better than closure passing in this case, due to the repetitive names of the methods called in *ExtensionHook* and *Proxy*. patterns: factory?

1.4.2 org.parosproxy.paros.core.proxy.ProxyThread

Response was created to handle the errors messages while*notification* package was created to delegate all the notification method used in *ProxyThread*. Due to the similarity in the algorithm, it was implemented using an abstract template class and the internal behavior was implemented in the concrete classes.

2 Strengths of the design

2.1 Iteration 1 & 2: Image Filtering

SimpleImageFilter was implemented over the strategy pattern, easily allowing new subclasses for new image filters. The *ConfigurationReader* is generic and could be used to extract instances of any (super) type from config files in other applications.

2.2 Iteration 3: Content Filtering

Splitting responsibilities increased cohesion of these classes. *FilterHttpContent* only handles messages now and does a basic check before sending the work to the *FilterApplyer*. This one coordinates the preparative tasks and does the actual filtering with the collected results. Then it returns the result to *FilterHttpContent*, but it could also set it itself. The only class with relatively high coupling is *FilterApplyer*. Due to the high cohesion, the classes are easy to understand and modify. Support for various extension is also provided with some generic typing and abstract super classes.

2.3 Iteration 4: Content Reporting

The extension is encapsulated in its package and relies in the interfaces provides by zaproxy. Due to the Strategy pattern applied for the image statistics, developers can create new concrete classes either using the template class or implementing a new one. Developers can select specific image statistics types via the *ImageStatisticsFactory*.

2.4 Iteration 5: Refactoring

2.4.1 org.parosproxy.paros.core.proxy.ProxyThread

Due to the template class *ProxyListenerNotifier*, developers can create new notification method using the concrete class without affecting the behavior of the others. The *notification* package can also be reused in other parts of the code since it does not depends on *ProxyThread*.

3 Weaknesses of the design

3.1 Iteration 1 & 2: Image Filtering

Concrete *SimpleImageFilter* is highly coupled to the configuration file because of the name classes and *ConfigurationReader* relies completely in the configuration fileâĂŹs correctness.
The *ConfigurationLoader* parses concrete *SimpleImageFilters* from the config file by using reflection. This strictly requires the config files to contain exactly the class names of the needed filters.

3.2 Iteration 3: Content Filtering

The *FilterApplyer* has a relatively high coupling: with *HttpMessage* as well as both helpers. This is due to its coordinating role in addition to the filtering, which also indicates cohesion can be improved.

3.3 Iteration 4: Content Reporting

The extension can add the new images statistics to the XML report in a straightforward way but it is not the case for HTML and MarkDown reports which are highly coupled to the XSL files. Whenever new XML image statistics format is created in the ImageExtension, the XSL files must be modified; the main issue is that those classes/files are not even directly related, making difficult to convert this *ImageExtension* into an add-on plugin.

3.4 Iteration 5: Refactoring

3.4.1 org.parosproxy.paros.core.proxy.ExtensionLoader

ExtensionList is does not cover all interactions with the wrapped list. It still provides a method getExtensions, which breaks its role as a proxy.

3.4.2 org.parosproxy.paros.core.proxy.ProxyThread

ProxyThread still has to create the concrete notification classes and stores them in a data structure. Hence the responsibilities were turned from calling internal methods to managing *ProxyListenerNotifier* classes.

4 Future improvements

4.1 Iteration 1 & 2: Image Filtering

A proposed improvement is adding input fields (potentially a tuple <priority, filterType>) in the *FilterReplaceImage* in order to do the configuration using the GUI provided by zaproxy. *ConfigurationReader* now is updated to a new factory class which creates and removes instances of *SimpleImageFilter* according with the final user configuration. The configuration file is no longer need, therefore the coupling in the name classes disappears. *Configurationreader* could be equipped with a richer parsing mechanism that would have an error margin on the parsed names, for example: case insensitivity and trimming.

4.2 Iteration 3: Content Filtering

4.3 Iteration 4: Content Reporting

The current implementation does not allow the final user to select the specific image statistic type in the report. Next improvement considers a GUI implementation using the *hook* abstract method provide by *ExtensionAdaptor*. *ImageStatisticsFactory* can be adapted to add those responsibilities: keep tracking the final user image statistic type selections and instantiating/removing the corresponding ImageStatistics classes in runtime.

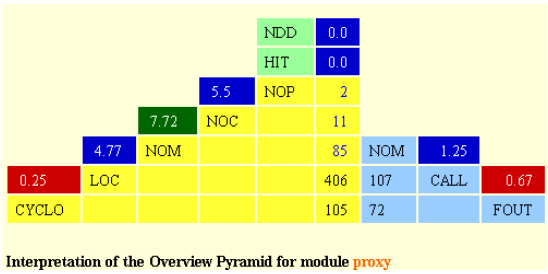
4.4 Iteration 5: Refactoring

4.4.1 org.parosproxy.paros.core.proxy.ExtensionLoader

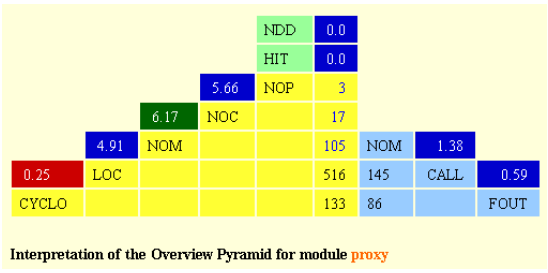
*ExtensionLists*hould take over more functionality for the encapsulated list and map. This will require some careful redesigning of the classes, other than *ExtensionLoader*, previously accessing the extensions.

5 Overview pyramid and test coverage

5.1 Pyramid

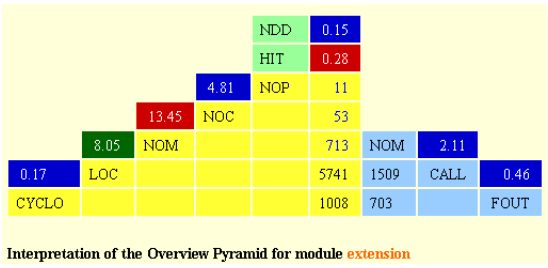


(a) Pyramid before refactoring

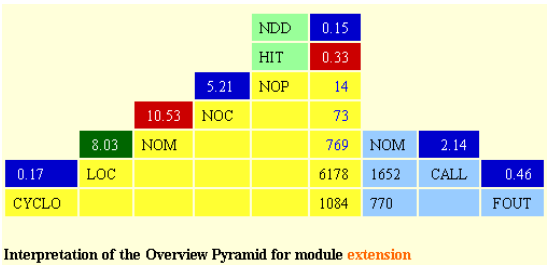


(b) Pyramid after refactoring

Figure 1: Pyramids of the org.parosproxy.paros.core.proxy package.



(a) Pyramid before refactoring



(b) Pyramid after refactoring

Figure 2: Pyramids of the org.parosproxy.paros.extension package.

5.2 Test Coverage

We only wrote JUnit tests for the 3rd iteration since it wasn't really possible to write tests for the other. First off all we have a test *testApplyBasicStringFilter*, which tests the basic functionality off our code with some standard examples. Then we also wrote some tests that may be usefull for future upgrades on our code. The first future test *testApplyBasicStringFilterCapitalLetters* tests whether certain words are also filtered out when they are partly in capital letters. The test *testApplyBasicStringFilterWithWordsInBetween* checks if the filter still works if it needs to filter a combination of words and these words are not directly next to each other in a sentence. *testApplyBasicStringFilterWithChangedOrder* also applies to cases where it's a combination of words that need to be filtered out, here we check if it's also filtered when the words are in a different ordering. the test *testApplyBasicStringFilterWithConjugatedVerbs* tests if the filter also works when the verb that needs to be filtered is also detected when it is conjugated. Finally we have the test *testApplyBasicStringFilterPartialCensoring* that checks if the filter detects partial cencoring. What we mean by this is when for example you need to filter the word "nigger" it also detects the word n*gger.

6 Design Diagrams

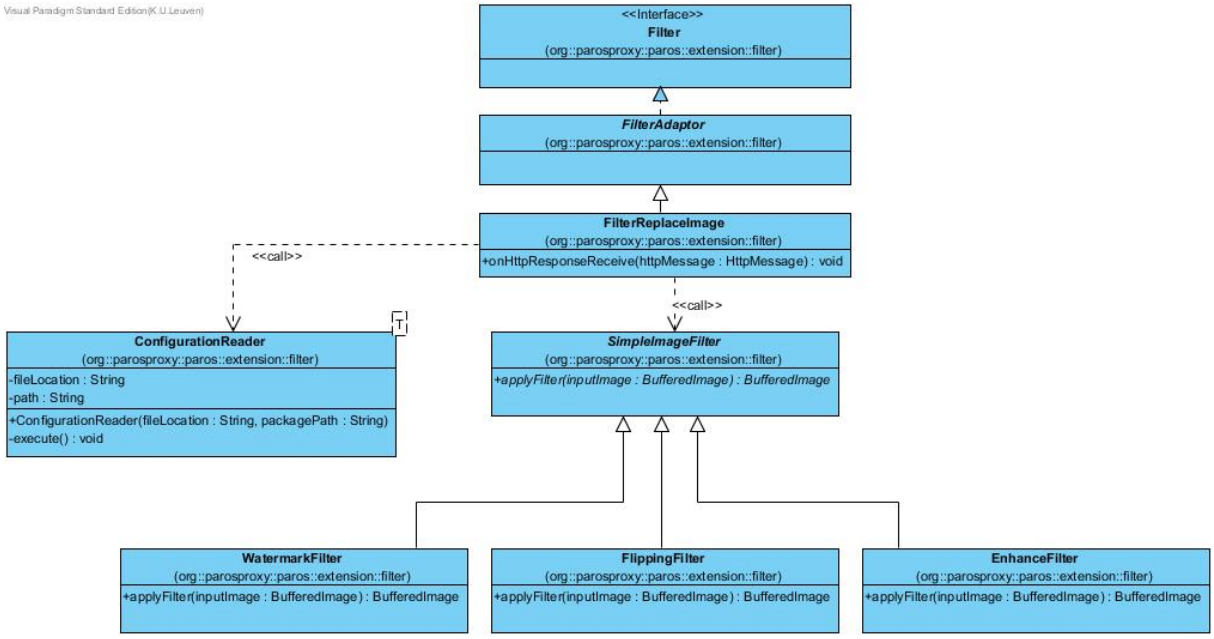


Figure 3: Class diagram iteration 1 & 2

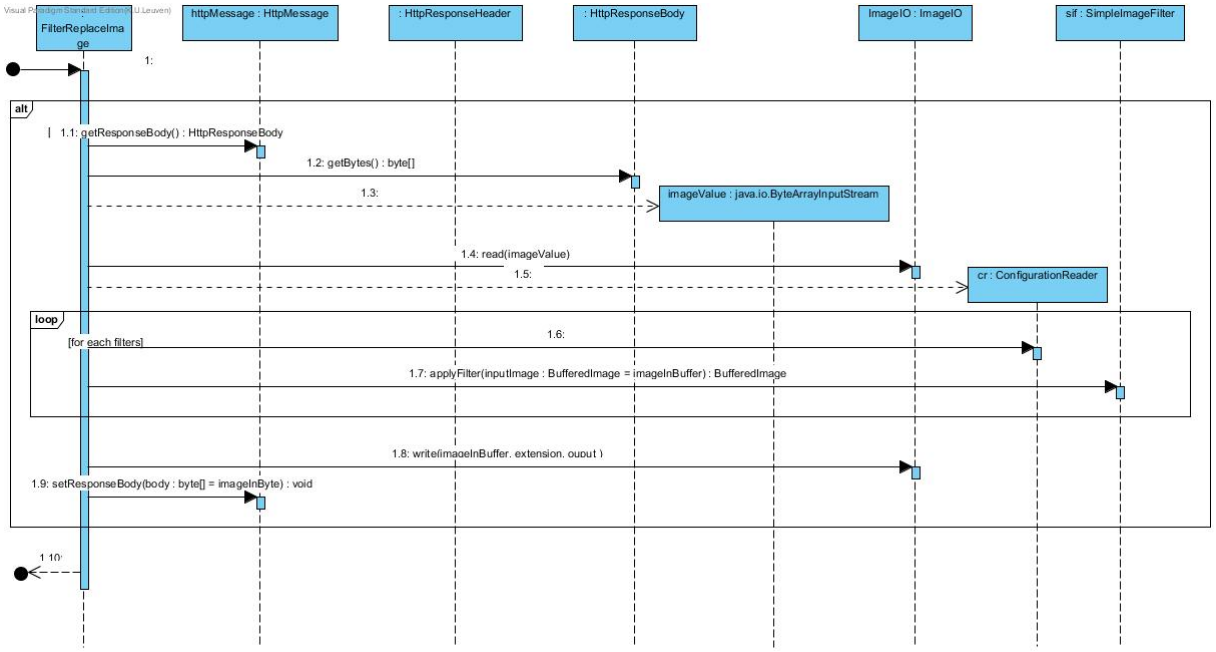


Figure 4: Sequence diagram of iteration 1 & 2

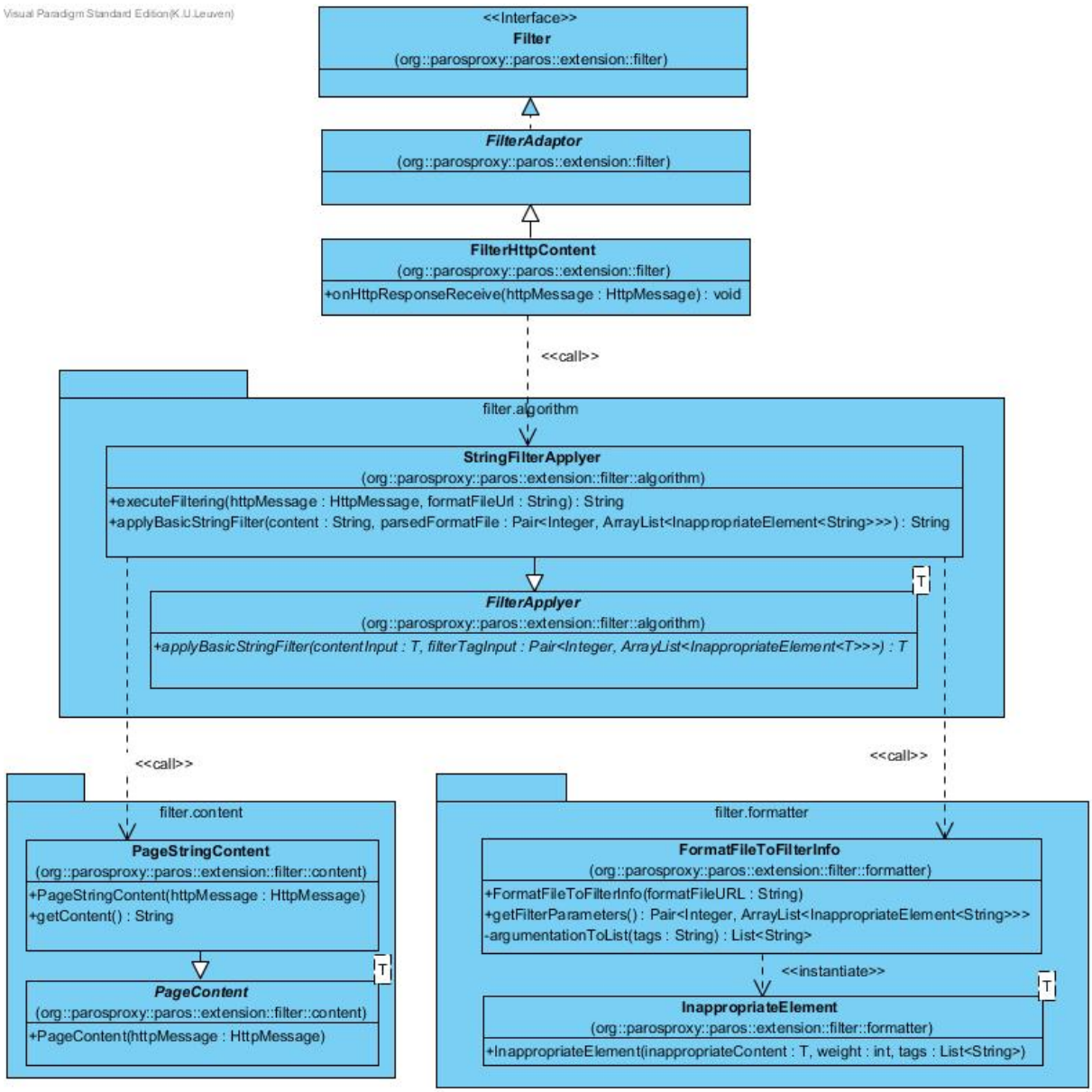


Figure 5: Class diagram of iteration 3

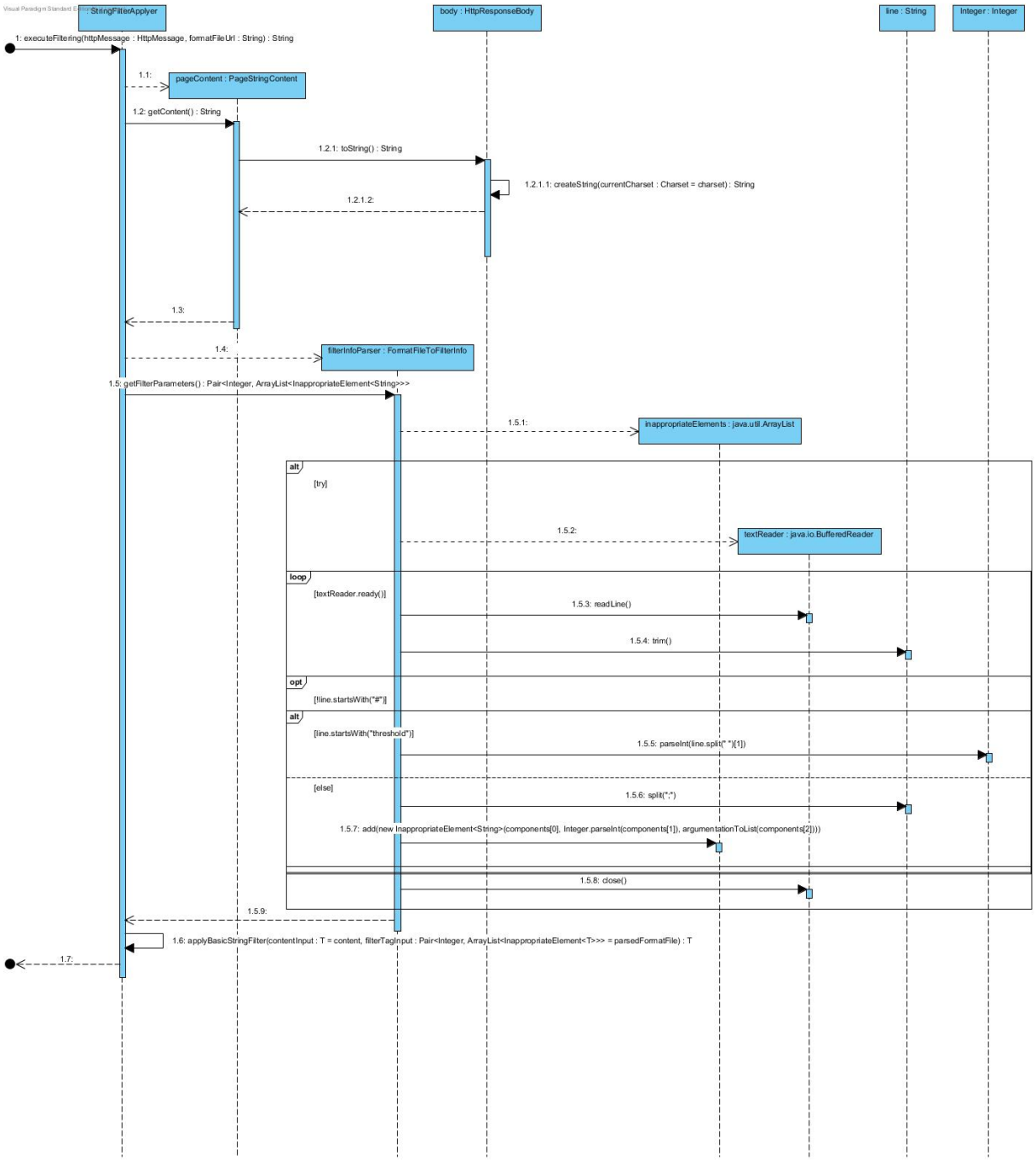


Figure 6: Sequence diagram of iteration 3

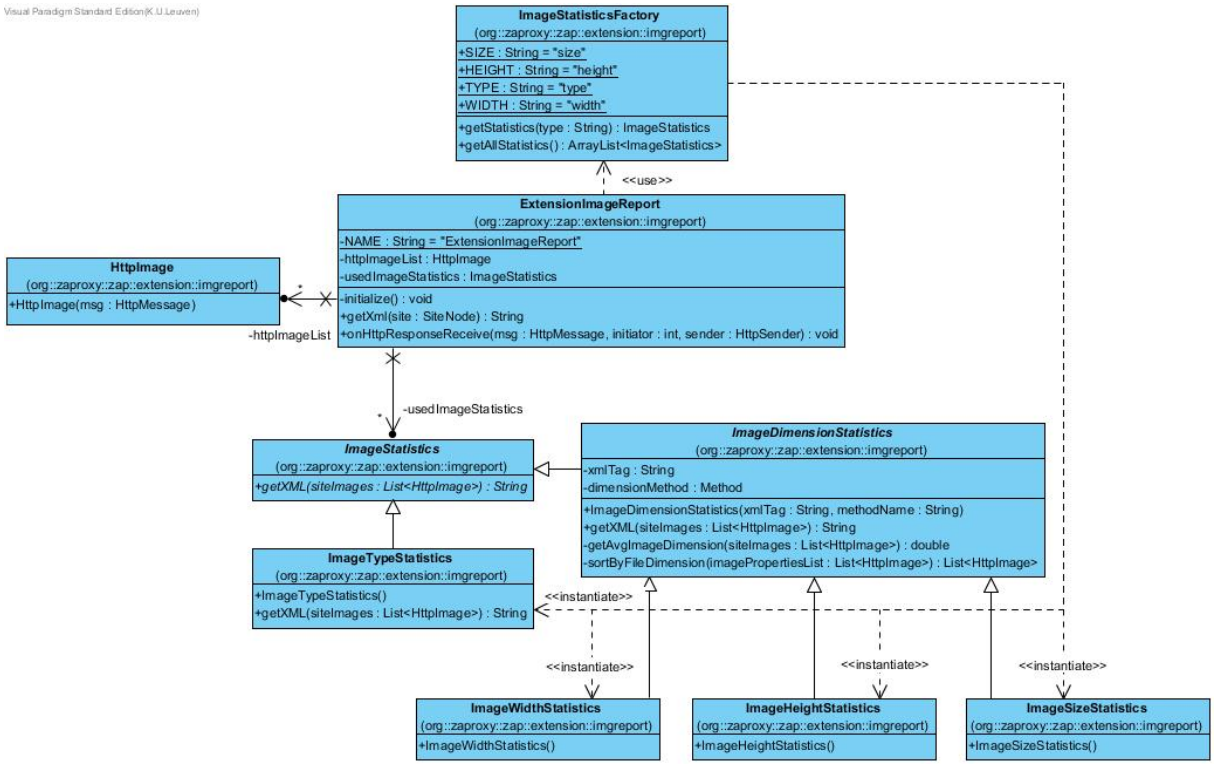


Figure 7: Class diagram of iteration 4

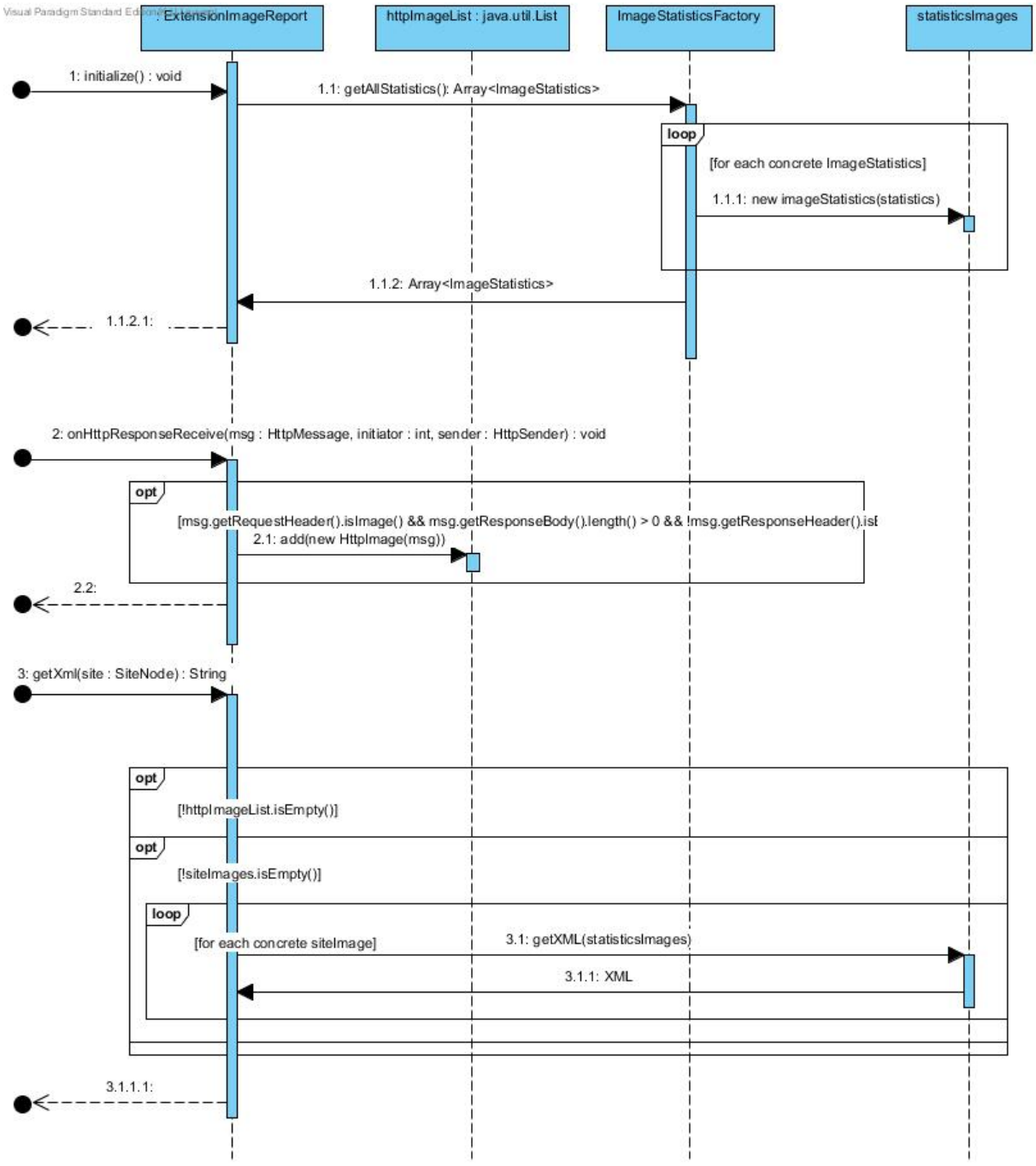


Figure 8: Sequence diagram of iteration 4