ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing Group

Spamato Plug-in Architecture

Remo Meier

Semester Project

October 18, 2004 – April 30, 2005

Supervising Professor: Prof. Dr. Roger Wattenhofer
Supervising Assistant: Keno Albrecht

Abstract

Spamato is an extendable, collaborative spam filter system written in Java that combines different filter technologies to provide more accurate results. This thesis takes Spamato a big step further by introducing a plug-in architecture. Users and developers can now customize their system by adding, updating, and removing plug-ins.

The underlying plug-in architecture is completely independent of Spamato and can be reused in other applications. Compared to other projects, it is extremely small and easy-to-use and still provides advanced features such as security, dependencies, and updates at runtime. The plug-in container, responsible for managing the plug-ins, is itself implemented as a plug-in and thus can be updated and extended like every other part of the system.

Contents

1 Introduction

The goal of this thesis is to design and implement a plug-in architecture to manage filters, especially a possibility to update and install filters at runtime and a simple way to configure them. Since other parts of the system could also benefit from such features, the complete system has been refactored and is now based on a plug-in architecture.

The initial SPAMATO design was based on one large library containing all the classes. Most of the important components were implemented as singleton and thus everything could basically use everything else. This approach was simple and sufficient for a first version, but lacked a clear structure. For example, dependencies between different parts of the system were hard to track. Seemingly negligible changes could render the system unusable because of dependency problems (e.g. cycles). Simple tasks like starting the system became difficult because it was not known where to start.

The new plug-in architecture subdivides the system into smaller, easier manageable pieces. Except of a few bootstrap classes, everything is implemented as a plug-in and can be installed, updated, and deleted at runtime. The main component of SPAMATO is the *Base* plug-in. It features methods to check, report, and revoke messages. These methods are used by the mail client addons. Besides the *Base*, there is a plug-in for each filter. Five filters are currently implemented: *Domainator* [1], *Razor* [2], *Ruleminator*, *Earl Grey* [3], and *Bayesianato*. The *Base* combines the results from these filters into a global decision whether or not a message is spam. Moreover, there are additional plug-ins for the sound, the user interface, and the statistics, among others. By adding and removing plug-ins, the system can now be customized depending on the needs. For example, a server version of SPAMATO does not need the sound plug-in beeping in the server room.

2 Related Work

Spamato is not the first application using a plug-in architecture and therefore, there was the choice between using an existing plug-in framework or creating an own one. Since it provides the foundation for the complete application, there are some important prerequisites. First of all, the framework has to provide the possibility to allow a reasonable application design. It has to be stable, small, and easy-to-use and should not cause any substantial runtime overhead. An install, update, and delete mechanism is required to manage the Spamato filters, the main goal of this thesis. And last but not least, a security mechanism is needed to restrict potential dangerous third-party plug-ins.

It follows a brief overview over some plug-in frameworks:

1. **Apache containers:**
   Apache offers several projects in this area ([4], [5], [6], among others). The history and relations between the different projects are rather complex. For the moment, the long-term future of the different projects is not certain. Some of the projects already failed and new projects have been founded based on their source code. If Spamato would choose one of the projects, there is a good chance that it has to migrate to another one in the future. The projects are also more component than plug-in oriented, meaning that they do not provide an install and update mechanism, an essential requirement for Spamato. Moreover, most projects are to large for Spamato, which targets a download size of only one or two megabytes.

2. **PicoContainer:**
   The *PicoContainer* [7] consists of a container that automatically wires all the application components together. A component is a normal Java object. In the component's constructor, the component can list other, needed components. The developer only has to pass all component classes to the container. The container analyzes the dependencies by inspecting their constructors and accordingly instantiates the classes. This approach is one variant of the so called *Dependency Injection* pattern [8]. The downside of the *PicoContainer* is the lack of advanced features, needed by Spamato, such as class loading, security, install, and update mechanisms.

3. **Eclipse:**
   The Eclipse development environment is based on a flexible plug-in architecture [9] that enables dozens of third-party companies and open-source projects to contribute new plug-ins. Especially the concepts of *Extensions* and *Extension Points* offers an easy way for plug-ins to extend and use each other. On the downside, Eclipse lacks a security mechanism and the update user interface can not be used because it is based on the platform-dependent SWT library [10]. Spamato uses a web-based user interface that can also be used with mail and proxy servers. In recent versions, Eclipse has migrated to *OSGi* [11]. *OSGi* has been introduced as a service platform for all types of networked devices in home, vehicle,

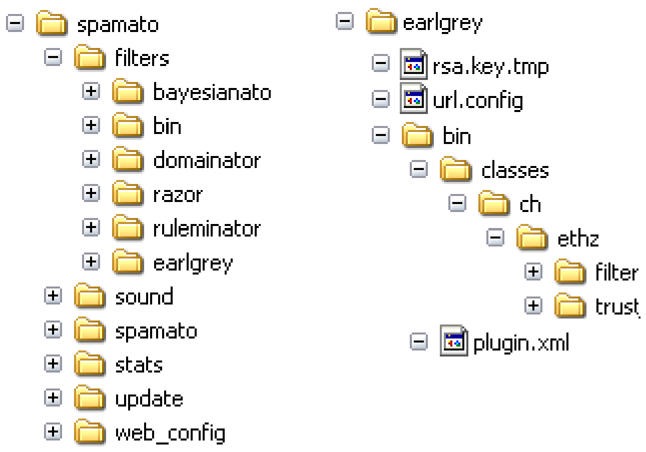
mobile, and other environment. Over time, *OSGi* has evolved to a more plug-in-based architecture, similar to Eclipse. But, especially with *OSGi*, writing plug-ins in Eclipse is not as easy as with other frameworks.

There are other implementations, mostly similar to one of the approaches above. Currently, none of the above frameworks fulfils all the mentioned prerequisites. And it seems that SPAMATO is not the only project that has come to this conclusion. For example, Nutch [12], JNode [13], and iMeMex [14] are all applications in a similar situation as SPAMATO and implemented their own simple solutions. But these implementation are tailored to their specific needs and not generally available without the remaining application. SPAMATO also follows this approach, however, the plug-in framework is completely independent of SPAMATO and powerful enough that it can be used in other applications. It also tries to fulfil all the requirements stated at the beginning of this chapter. The next chapter will give a more detailed overview of the SPAMATO container.

3 Spamato Plug-in Container

The plug-in architecture of SPAMATO uses a central container to manage the plug-ins. The container is completely independent of SPAMATO and can easily be reused in other projects. It is less than fifty kilobytes in size. To contribute a plug-in, developers have to know only a few classes and the plug-in configuration file.

Plug-ins are loaded from the file system. Each one has its own directory, containing the classes (in the *bin* directory), configuration files, and child plug-ins. In SPAMATO it looks like:



The *bin* directory contains all Java classes, libraries, resources (like images), and the *plugin.xml*. The *plugin.xml* provides all necessary information about a plug-in, like the version number, main class, dependencies, needed security permissions, and update information:

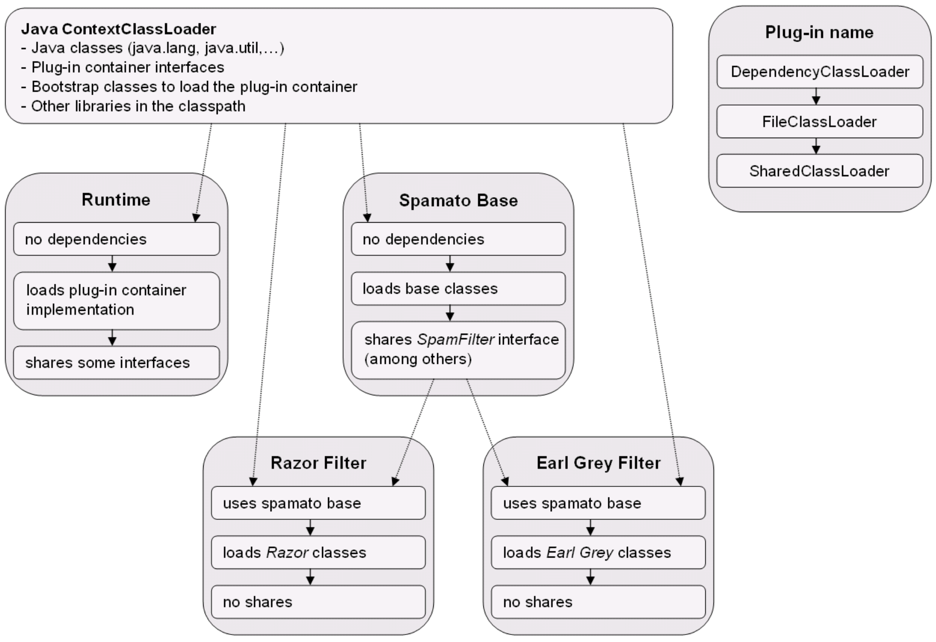
```
<plugin>
    <name>Earl Grey Filter</name>
    <class>ch.ethz.filter.earlgrey.client.EarlGreyFilter</class>
    <version>1.0</version>
    <update-url>http://spamato.ethz.ch/update</update-url>
    <description>
        A collaborative filter working on the domains in the messages.
    </description>
    <requires>
        <permission type="socket" host="*" actions="resolve,connect"/>
        <plugin key="spamato␣base">
            <extension point="filters" class="ch.ethz...EarlGreyFilter"/>
        </plugin>
        <plugin key="config" optional="true">
            <extension point="config.pages" handler="..." menu="Earl␣Grey␣Filter"/>
            <extension point="config.pages" handler="..." menu="Whitelist"/>
            ...
        </plugin>
    </requires>
</plugin>
```

It follows an overview over some of the most important features:

3.1 Dependencies

A simple plug-in has its own classloader and is completely independent from other plug-ins. It can be installed, updated, and deleted without disturbing the remaining system. To build a more sophisticated system where plug-ins interact with each other, the container enables plug-ins to declare dependencies. For this purpose, each plug-in can share classes and object instances. Shares and dependencies are usually defined in the *plugin.xml*. For example, the *Base* plug-in of Spamato shares the *SpamFilter* interface that is implemented by all filters.

The dependencies are internally represented as an acyclic, directed graph. The graph is used, for example, to determine a start-up and a shutdown order, something that was not possible with the old system because of the unknown dependencies. To isolate the plug-ins from each other, each plug-in has three class loaders: the *FileClassLoader*, the *SharedClassLoader*, and the *DependencyClassLoader*. The *FileClassLoader* loads the class files from the *classes* and the libraries from the *lib* directory. The *SharedClassLoader* uses the *FileClassLoader* as its parent, but it only allows access to shared classes as declared in the plug-in descriptor. And the last class loader, the *DependencyClassLoader*, enables plug-ins to access shared classes of other plug-ins by using their *SharedClassLoaders* (given that a plug-in dependency is declared). The *DependencyClassLoader* is the parent of the *FileClassLoader* in order that plug-in classes can access shared classes. Additionally, there is a single *ContextClassLoader* that is used as the parent of all *DependencyClassLoaders*:



More information about the class loaders is available in the JavaDoc of the *Runtime* plug-in (the *Runtime* is explained further below).

3.2 Configuration

An interface unifies the access to configuration settings from various sources. Currently, text, properties, and xml files are supported, and additional formats are feasible. For instance, a database implementation would be more appropriate for a SPAMATO instance running on a mail or a proxy server. With thousands of user accounts and personal settings, the database could better handle the resulting load and could also offer advanced features like replication.

3.3 Dependency Injection (IoC-Pattern)

Plug-ins usually depend on several other objects. They do not only use other plug-ins, but, for example, also configuration objects. The question arises how to get these instances. Common solutions are to let the plug-in create the objects, to access them via singletons, or to use a service locator [15]. To create an object, the plug-in has to know the implementation, and thus, switching to another one becomes more difficult. A server version of SPAMATO might use a SQL-based configuration, whereas normal versions still use the property files. Singletons should not be used because they complicate maintaining a clear structure. More sophisticated is the service locator pattern. However, a simpler and more transparent solution offers the dependency injection pattern [8], promoted by containers like *PicoContainer* [7] and *Spring* [16]. There are several variants, one of them using constructors to pass the objects. In doing so, the container and not the plug-in is responsible for creating and passing the appropriate objects. The SPAMATO container supports both the service locator and the constructor-based dependency injection. Therefore, a plug-in class looks as simple as:

```java
package ch.ethz.filter.earlgrey.client;

import ch.ethz.common.SpamFilter;

public class EarlGreyFilter implements SpamFilter, Disposable {

    public EarlGreyFilter(Spamato spamato,  Configuration config) {
        ...
    }

    public void dispose(){
        ...
    }
    ...
}
```

Consequently, plug-ins are plain Java objects that can also be used without the container. They do not have to extend or implement special container classes. For example, this is an important prerequisite for unit tests where each class is separately tested. Unit tests do not have to use the container. Instead, they can manually instantiate the plug-in classes and pass mock objects as arguments to test all possible cases [17].

3.4 Extension Points and Extensions

Extension Points and *Extensions* are concepts borrowed from Eclipse. They are used between plug-ins to extend each other and are one of the keys for Eclipse's flexibility. For example, the SPAMATO *Base* plug-in offers an extension point *"filters"* that is used by filters to register themselves to the *Base*:

```
<plugin>
    <name>Spamato Base</name>
    <share>
        <extension-point id="filters"/>
        ...
    </share>
    ...
</plugin>

<plugin>
    <name>Earl Grey Filter</name>
    ...
    <requires>
        <plugin key="spamato_base">
            <extension point="filters" class="ch.ethz...EarlGreyFilter"/>
        </plugin>
    </requires>
</plugin>
```

To check, report, and revoke messages, the *Base* plug-in accesses these extensions to obtain the filter instances. More information is available in the JavaDoc (*Extension* and *ExtensionPoint* interfaces).

3.5 Updates

As already mentioned, the container is able to install, update, and delete plug-ins at runtime without restarting the system. Only the plug-in itself and dependent plug-ins are restarted, the remaining system is left untouched. Updates are currently downloaded from the Internet or from the local file system. Every common web server can be used, they only have to obey a simple directory structure. Other methods, like Bittorrent-based downloads, are also feasible.

The container distinguishes between install and update servers. The first ones are used to find new plug-ins and the latter to update them. A Plug-in declares its update server in the *plugin.xml*. Users can add additional update and install servers in the container configuration.

Additionally, it is possible to select a profile server. A profile server is a regular install server, but the container automatically installs all hosted plug-ins. SPAMATO uses this mechanism to install plug-ins into the user's empty profile directory during the first start-up. The initial plug-ins are bundled with the mail client add-on and saved in the application directory. Using the profile mechanism, these plug-ins are copied into the user's profile directory, where the user can install, update, and delete them without interfering with other users. If new plug-ins are available in the application directory (by updating the add-on), they are also updated in each profile. Besides SPAMATO, the profile mechanism could be applied in larger companies to distribute, install, and update plug-ins.

More information is available in the JavaDoc (*PluginContainer* and *Update-Handler* interfaces).

3.6 Security

The container provides the means to restrict the security permissions of individual plug-ins, like the file system or the network access. The implementation is based on the Java security mechanisms. Each plug-in has to list required permissions in the *plugin.xml*. A Java *CodeSource* is created for each plug-in to tag all its classes. The *Java Security Manager* can recognize the different *CodeSources*, checks their permissions, and permits or prohibits the current statements.

So far, all containers lack a similar feature or at least it is not directly included. The vast majority of Java projects usually disables the security manager. During the development phase, it seems to be a restriction and enabling it afterwards can cause problems if it is not sufficiently tested. But applications could greatly benefit from such a mechanism. Not only are server side application better protected against intruders and clients against malicious plug-ins, it also prevents programming errors like working with the wrong files or connecting to the wrong servers.

Another approach, not relying on the Java security mechanisms, would be to use a trust system to rate the plug-ins. Such a trust system is already implemented as part of the *Earl Grey* filter. The two approaches could also be combined.

3.7 Plug-in Handlers

Each plug-in has a handler to manage its life cycle. The handler decides, among others, how to load a plug-in, which class loader to use, granted dependencies, and available constructor parameters. Basically, the container is only responsible for finding plug-ins and instantiating their handlers, everything else is controlled by the handlers. It is possible to add new handlers and thus alter the behavior of plug-ins. For example, it would be possible to write a modified handler that could also load Eclipse plug-ins. Or another handler could generate a proxy for each plug-in, implementing the same interfaces and dispatching method calls to the currently active plug-in version. This way it is possible to update a plug-in without restarting dependent ones.

3.8 Runtime Plug-in

An interesting aspect is that the plug-in container itself is implemented as plug-in, called the *Runtime* plug-in. This has several benefits. Like any other plug-in, it is possible to update the container at runtime and to restrict its security permissions. The container provides *Extension Points* to enable other plug-in to add new features, like new update protocols (e.g. P2P) or other file structures (e.g. support *Eclipse* plug-ins). Thus additional features can be added as plug-ins instead of including them directly and developers can choose what to use. This design prevents a bloated architecture where most of the functionality is never used. It is maybe possible to slim down and remove unwanted features from other containers, like the Apache ones, but it is certainly not the responsibility of the user to do so.

The implementation uses, besides the *Runtime* plug-in, a few bootstrap classes to initialize the system. Some interfaces are defined to access the plug-in system (*PluginContainer*, *Extension*, *Configuration*, and *PluginContext*, among others). A factory is used to instantiate the container. The factory loads the *Runtime* classes and instantiates the *Runtime Plug-in Handler*, a slightly modified version of the default handler, with the *Java Reflection API*. This handler creates the container as if it is a normal plug-in. The container finally loads all the other plug-ins. To mask container updates, a proxy is used. The proxy also implements the container interface and dispatches all method calls to the currently active container.

4 Conclusion & Future Work

By introducing a plug-in architecture, this thesis substantially improves Spamato. The plug-in container facilitates a simple design and provides the means for future extensions such as new filter and statistic plug-ins.

The project itself was interesting. Once the first plug-ins were implemented, plenty of new ideas for potential improvements emerged. Breaking the system into smaller pieces allowed a much clearer design. Because dependencies have explicitly to be declared, more time was generally spent designing the system and reducing dependencies.

While most of the important features have been implemented, there is a lot of space for potential improvements. Currently, only the client applications are based on the plug-in architecture, although, the statistic and *Earl Grey* servers could similarly benefit from it. It would also be possible to further subdivide the client plug-ins. The trust system and the white list, currently part of the *Earl Grey* filter, could be implemented as independent plug-ins, enabling other plug-ins to use their functionality. By doing so, the trust system could be used to rate third-party plug-ins and the *Domainator* filter could access the white list. The *Runtime* plug-in currently offers only a limited number of *Extension Points* and a lot more could be implemented.

References

[1] Christan Wassmers. Spamato statistics, a statistical approach towards spam filtering. Dimploma thesis, Swiss Federal Institute of Technology Zurich, 2005.

[2] Simon Schlachter. Spamato reloaded: Trust, authentication and more in a collaborative spam filter system. Master thesis, Swiss Federal Institute of Technology Zurich, 2004.

[3] Nicolas Burri. Spamato: A collaborative spam filter system. Diploma thesis, Swiss Federal Institute of Technology Zurich, 2004.

[4] Apache Avalon. `http://avalon.apache.org`.

[5] Apache Excalibur. `http://excalibur.apache.org`.

[6] DPML Metro - Composite Component Management. `http://www.dpml.net/metro/latest/index.html`.

[7] Picocontainer. `http://www.picocontainer.org`, 2004.

[8] Martin Fowler. Inversion of control containers and the dependency injection pattern. `http://martinfowler.com/articles/injection.html`, 2004.

[9] Eclipse Platform Project. `http://www.eclipse.org/platform/index.html`.

[10] Eclipse SWT Project. `http://www.eclipse.org/swt/`.

[11] The OSGi Service Platform - Dynamic services for networked devices. `http://www.osgi.org`.

[12] Nutch search engine. `http://www.nutch.org`.

[13] JNode - Java New Operating System Design Effort. `http://www.jnode.org`.

[14] iMeMex: Personal Information Management. `http://www.imemex.org`.

[15] Sun Microsystems. Service locator pattern. `http://java.sun.com/blueprints/patterns/ServiceLocator.html`, 2004.

[16] Spring's inversion of control. `http://www.springframework.org/docs/reference/beans.html`, 2004.

[17] Mock object. `http://www.picocontainer.org/Mock+Objects`, 2004.