Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik

DANIEL WILHELM (21089384)
DIPLOMARBEIT

JDOWNLOADER PLUG-IN SYSTEM

DESIGN UND IMPLEMENTIERUNG

Eingereicht am 1. August 2014

Betreuer: Prof. Dr. Dirk Riehle, M.B.A.
Professur für Open-Source-Software
Department Informatik, Technische Fakultät
Friedrich-Alexander-Universität Erlangen-Nürnberg

Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, 1. August 2014

License

This work is licensed under the Creative Commons Attribute 3.0 Unported license (CC-BY 3.0 Unported), see `http://creativecommons.org/licenses/by/3.0/deed.en_US`

Erlangen, 1. August 2014

Abstract

JDownloader is a download manager that utilizes plug-ins to fulfill different needs of its users. Its lastest stable version dates back to the year 2009. Since then JDownloader 2 is in development. But the old plug-in system for JDownloader 1 was never designed for a growing numbers of plug-ins and multiple updates per day. Therefore a new plug-in system had to be developed, which was capable to deal with this situation and to allow additional new features like hot deployment of plug-ins. At the same time it should be able to reduce the memory usage and load times.

This work describes all the requirements for the new plug-in system for JDownloader 2, challenges during development as well as its final design and implementation. In addition, this work shows that the system successfully meets the requirements.

Keywords: JDownloader, plug-in system, weak references, hot deployment, java, classloader

Zusammenfassung

JDownloader ist ein Download-Manager, der Plug-ins verwendet, um die unterschiedlichen Bedürfnisse der Benutzer zu erfüllen. Die letzte stabile Version stammt aus dem Jahr 2009. Seitdem befindet sich dessen Nachfolger JDownloader 2 in Entwicklung. Das alte Plug-in System von JDownloader 1 war nie für eine wachsende Anzahl von Plug-ins und mehreren Aktualisierungen pro Tag ausgelegt. Deshalb musste ein neues Plug-in System entwickelt werden, das in der Lage war, mit dieser Situation umzugehen und zusätzlich neue Funktionen, wie dem Aktualisieren der Plug-ins zur Laufzeit, zu ermöglichen. Gleichzeitig sollte es in der Lage sein, den Speicherverbrauch und die Ladezeiten zu reduzieren.

Diese Arbeit beschreibt die Anforderungen an das neue Plug-in System für JDownloader 2, die Herausforderungen während der Entwicklung, als auch dessen endgültiges Design und Implementierung. Darüber hinaus zeigt diese Arbeit, dass das System die Anforderungen erfolgreich erfüllt.

Schlüsselwörter: JDownloader, Plug-in System, schwache Referenzen,
Hot Deployment, Java, Klassenlader

Inhaltsverzeichnis

1 Einführung **1**
1.1 Ursprüngliche Aufgabenbeschreibung 1

2 Forschung **2**
2.1 Einführung 2
2.1.1 JDownloader 1 3
2.1.2 JDownloader 2 BETA 3
2.2 Plug-in Systeme 4
2.2.1 Plug-in Systeme als Erweiterung von Anwendungen 5
2.2.2 Plug-in Systeme als Architektur von Anwendungen 8
2.2.3 Eigenentwicklung für die Anwendung 10
2.3 Anforderungen an das neue Plug-in System 12
2.3.1 Funktionale Anforderungen 13
2.3.2 Nicht-funktionale Anforderungen 14
2.4 Das Plug-in System von JDownloader 2 19
2.4.1 Systemarchitektur 19
2.4.2 Mehrere Klassenlader - Segen oder Fluch? 22
2.4.3 Verbesserungen im Vergleich zu JDownloader 1 25
2.5 Bewertung des neuen Plug-in Systems 27
2.6 Zusammenfassung und Ausblick 33

3 Ausarbeitung der Forschung **35**
3.1 Vergleich mit ähnlichen Plug-in Systemen 35
3.1.1 Weitere nicht-funktionale Anforderungen 35
3.1.2 Anforderungsbewertung 37
3.1.3 Bemerkungen 38

4 Anhänge **41**
Anhang A Plug-ins im Jahr 2013 41
Anhang B Aktualisierungen im Jahr 2013 42
Anhang C Plug-in Updates im März 2014 43
Anhang D Plug-ins in JDownloader 44

D.1 Crawler-Plug-in 44
D.2 Hoster-Plug-in 45
Anhang E Lazy-Plug-ins 47
Anhang F PluginClassLoader 49
F.1 PluginClassLoaderChild 51
Anhang G PluginController 52
G.1 HostPluginController 53
G.2 CrawlerPluginController 54
Anhang H schwache Referenzen 56
H.1 MinTimeWeakReference 57
Anhang I VisualVM-Analyse für 2.4.3 58
Anhang J VisualVM-Analyse für 2.5 59

Literaturverzeichnis **60**

1 Einführung

1.1 Ursprüngliche Aufgabenbeschreibung

This Diplomarbeit describes the design and implementation of the JDownloader plug-in system. JDownloader is a file downloading software with more than 20 million users world-wide. JDownloader has a plug-in system, where each plug-in adapts the core software to the specifics of a file hosting service. The plug-in system evolved from a simple first version to a more elaborate current version. This thesis describes the non-functional requirements for the current version and how the design and implementation of the current version fulfills these requirements. It then compares the current JDownloader plug-in system with other comparable (simple) plug-in systems and shows how and where it is superior or inferior.

2 Forschung

2.1 Einführung

JDownloader[1] ist eine in der Programmiersprache Java entwickelte, quelloffene Anwendung, die dem Anwender das Herunterladen von Dateien und Inhalten aus dem Internet ermöglicht. Für diese Hauptfunktion besitzt JDownloader eine Vielzahl an unterschiedlichen Plug-ins (siehe Anhang D). Anhand dieser können verschiedenste Anbieter und Quellen unterstützt und JDownloader so den jeweiligen Wünschen und Bedürfnissen eines Anwenders angepasst werden. Zusätzlich kann JDownloader dem Anwender durch automatisierbare Schritte, wie zum Beispiel der Eingabe eines CAPTCHA[2] oder dem Entpacken von Archiven, viel Arbeit und dadurch letztendlich auch Zeit einsparen.

Für die Wartung und Weiterentwicklung des Kerns von JDownloader sowie der Bereitstellung zusätzlicher technischer Voraussetzungen, wie zum Beispiel der Update-Server und des Entwickler-/Support-Forums, zeigt sich die Appwork GmbH[3] aus Fürth verantwortlich. Neben ihr existiert auch eine Gemeinschaft von Entwicklern[4] aus der ganzen Welt. Sie haben sich der Pflege und Aufnahme neuer Plug-ins verschrieben, damit JDownloader auch in Zukunft weiterhin die gewünschten Dateien und Inhalte des jeweiligen Anwenders herunterladen kann. Ich selbst bin seit über sechs Jahren Teil dieser Entwicklergemeinschaft und besitze dementsprechend tiefgreifendes Wissen über den JDownloader. Dabei habe ich mich auf Themen wie dem Plug-in System, der Netzwerkprogrammierung, der Optimierung des Ressourcenverbrauchs sowie der Synchronisierung von Nebenläufigkeiten spezialisiert.

Die Appwork GmbH stellt derzeit der Öffentlichkeit JDownloader in den zwei Versionen JDownloader 1 und JDownloader 2 BETA zum Herunterladen bereit.

[1] http://www.jdownloader.org

[2] Completely Automated Public Turing test to tell Computers and Humans Apart

[3] http://www.appwork.org

[4] http://svn.jdownloader.org/projects/jd

2.1.1 JDownloader 1

Als JDownloader 1 wird aktuell jene Version bezeichnet, welche im Dezember 2009 als stabil angesehen und der Öffentlichkeit zur Verfügung gestellt wurde.

Offiziell unterstützt sie Java nur in den heutzutage veralteten Versionen 1.5 und 1.6. Die aktuelle Version 1.7 sowie die gegenwärtig in Entwicklung befindliche Version 1.8, können nur mit Einschränkungen der Funktionalität verwendet werden. Aufgrund des stark veralteten Kerns von JDownloader 1 und die hierdurch verursachten Kompatibilitätsprobleme mit neueren Plug-ins sowie dem fehleranfälligen und zeitintensiven veralteten Update-System von JDownloader 1 (Rechenmacher, 2013), wird diese Version nur noch unregelmäßig alle ein bis drei Wochen mit Plug-in Aktualisierungen versorgt.

Eine Aktualisierung des Kerns oder anderer Komponenten des JDownloader wäre bei dieser Version aufgrund von technischen Gegebenheiten und Entscheidungen mit viel Aufwand und großen Risiken verbunden und wird deswegen nicht weiter angewandt.

2.1.2 JDownloader 2 BETA

Seit der Veröffentlichung des JDownloader 1 fand eine stetige Weiterentwicklung des Kerns sowie weiterer JDownloader betreffender Software-Projekte statt. Dies war nötig um den geänderten Anforderungen und neueren technischen Gegebenheiten gerecht zu werden zu.

Aus dieser Weiterentwicklung heraus wurde im April 2013 der Öffentlichkeit die JDownloader 2 Version zur Verfügung[5] gestellt. Um dem Benutzer zu signalisieren, dass es sich hierbei um eine noch in der Entwicklung befindliche Version handelt, tragen alle aktuellen Versionen derzeit noch den Zusatz BETA.

Zu den wichtigsten Neuerungen gegenüber JDownloader 1 zählen die offizielle Unterstüzung der Java Versionen 1.7 und 1.8 sowie der Kombination aus neuem Update- und Plug-in System. Letzteres ermöglicht eine deutliche Reduktion des Speicherverbrauches zur Laufzeit und der schnelleren Aktualisierung von JDownloader und dessen Plug-ins. Eine für viele Anwender besonders angenehme Neuerung dadurch ist, dass eine Aktualisierung von Plug-ins zur Laufzeit durchgeführt werden kann und somit der lästige und teils unpassende Neustart, wie er mit JDownloader 1 noch nötig war, entfällt.

Mittels kontinuierlicher Integration und Auslieferung konnte auch die Entwicklungsgeschwindigkeit erhöht und die Fehlerbehebung (Rechenmacher, 2014) deutlich verbessert werden. Aufgrund der unregelmäßigen Aktualisierung von

[5] `http://www.jdownloader.org/download/offline`

JDownloader 1 war ein direktes Feedback seitens des Anwenders zu neuen Funktionen oder Fehlerbereinigungen nur schwer oder stark verzögert möglich. Bei JDownloader 2 erhalten Anwender die Änderungen des Entwicklers nun innerhalb weniger Minuten und können dem Entwickler so ein direkteres Feedback geben. Vor allem im Bereich der Fehlerkorrektur und Plug-in Entwicklung ist dies ein deutlicher Vorteil gegenüber JDownloader 1.

2.2 Plug-in Systeme

Unter Plug-in Systeme versteht man im Allgemeinen die Möglichkeit bestehende Anwendungen mittels zusätzlicher Komponenten zu erweitern oder an besondere Bedürfnisse oder Umgebungen anzupassen. In seiner Dissertation „Konzeptionelle Modellierung von Plug-in Systemen mit Petrinetzen“ fasst Duvigneau (2010, S. 3) einige Gründe zusammen, die für den Einsatz von Plug-in Systemen sprechen.

So listet Duvigneau zum Beispiel vier Gründe für den Einsatz von Plug-in Systemen auf, die Chatley et al. (2004, S. 129ff.) bereits in seiner Dissertation identifiziert hat:

- „den Wunsch, die Funktionalität eines Systems nach seiner Fertigstellung noch erweitern zu können,“
- „die Dekomposition großer Softwaresysteme, so dass nur die in der jeweiligen Situation benötigten Teile installiert bzw. geladen werden,“
- „das Aktualisieren von lang laufenden Anwendungen ohne sie anhalten und neu starten zu müssen, oder“
- „der Wunsch, Erweiterungen von Drittanbietern ohne direkte Zusammenarbeit mit den Systementwicklern integrieren zu können.“

Weitere von ihm genannte Gründe anderer Autoren wären:

- „die Reduktion von Laufzeiten und Hardwareanforderungen von Softwaresystemen (Mayer et al., 2003, S. 89),“
- „die einfache Anpassbarkeit eines Softwaresystems an unterschiedliche Laufzeitumgebungen (Fowler, 2002, S. 499ff.),“
- „die einfache Integrierbarkeit nutzerspezifischer Wünsche (Birsan, 2005, S. 41).“

Neben den verschiedensten Gründen, die für den Einsatz eines Plug-in Systems sprechen, gibt es auch mehrere Möglichkeiten, wie ein solches Plug-in System realisiert werden kann. Rathlev (2011) fasst die Möglichkeiten hierbei in zwei

verschiedene Arten von Plug-in Systemen zusammen, die im folgenden vorgestellt werden.

2.2.1 Plug-in Systeme als Erweiterung von Anwendungen

Abbildung 2.1 beschreibt ein Plug-in System, wie es im allgemeinen Sprachgebrauch verstanden wird. Eine Anwendung kann mittels Plug-ins mit zusätzlichen Funktionen ausgestattet, an bestimmte Laufzeitumgebungen oder Anforderungen des Anwenders angepasst werden. Hierfür nutzen die Plug-ins eine in der Anwendung bereitgestellte Plug-in Schnittstelle, um ihre Funktionen oder Änderungen der Anwendung zur Verfügung zu stellen. Der Rahmen, inwieweit ein Plug-in Einfluss auf die Anwendung nehmen kann, hängt dabei sehr von dem verwendeten Plug-in System ab.

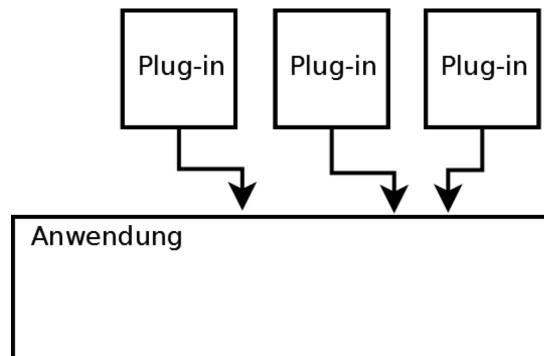


Abbildung 2.1: Erweiterung von Anwendungen

Findet lediglich ein Datenaustausch anhand fester Vorgaben zwischen der Anwendung und dem Plug-in statt, so ist dessen Einfluss bereits seitens der Anwendung fest vorgegeben. Der Nachteil einer solchen Lösung liegt in den eingeschänkten Möglichkeiten des Plug-ins. So bedarf es unter Umständen einer zusätzlichen Änderung des Plug-in Systems, beziehungsweise dessen Plug-in Schnittstelle, wenn ein Plug-in weiteren Einfluss auf die Anwendung erhalten soll. Trotz allem bietet ein solches Plug-in System erhöhte Sicherheit und Stabilität.

Kann ein Plug-in jedoch auf eine Vielzahl oder sogar alle Klassen und Methoden der Anwendung zugreifen, so ist es nahezu uneingeschänkt in der Lage Einfluss auf diese zu nehmen. Diese Offenheit der Anwendung birgt aber auch Gefahren. So können sicherheitsrelevante Bereich kompromittiert oder die gesamte Stabilität der Anwendung durch Fehler einzelner Plug-ins beeinträchtigt werden.

Es hängt somit stark von der Anwendung und der gewünschten Freiheit der Plug-ins ab, welches Plug-in System letztendlich zum Einsatz kommen wird. Im folgenden werden einige Beispiele für solche Plug-in Systeme in Java gegeben.

- **ServiceLoader**

  Seit der Version 1.6 verfügt Java über ein primitives Plug-in System, den *ServiceLoader*[6]. Ein Entwickler muss lediglich eine Schnittstelle (*Interface*) für die benötigten Methoden definieren, welche dann von den unterschiedlichen Plug-ins implementiert werden. Mittels dem *ServiceLoader* kann man im Klassenpfad oder in bestimmten JAR-Dateien nach allen Plug-ins suchen lassen, welche das gewünschte *Interface* implementieren. Damit ein Plug-in gefunden werden kann, muss es einen entsprechenden Eintrag in dem Ordner „META-INF/services/“ in derselben der JAR-Datei geben, in der sich auch die Klasse des Plug-ins befindet. Die entsprechenden Plug-ins können mittels eines *Iterators* verwendet werden. Sie werden dabei erst während des Zugriffs auf den *Iterator* geladen und instanziiert. Bei diesem Plug-in System verwendet der Entwickler lediglich das zuvor definierte *Interface* und kommt mit der eigentlichen Implementierung von anderen Plug-ins anschließend gar nicht in Berührung.

- **JSPF**

  Im Jahr 2008 begann Ralf Biedert mit der Entwicklung seines „Java Simple Plugin Framework[7]“ (JSPF). Das Plug-in System zeichnet sich durch seine einfache Verwendung aus. Wie auch der *ServiceLoader*, nutzt JSPF *Interfaces* um die jeweiligen Methoden der Plug-ins zu definieren. Dieses Plug-in System benötigt aber, im Gegensatz zu dem *ServiceLoader*, keine zusätzlichen Dateien für den Umgang mit den Plug-ins. Somit wird nicht nur die Entwicklung sondern auch die Wartung des Plug-ins erleichtert. JSPF verwendet stattdessen Annotationen (*Annotations*) um Informationen über das Plug-in bereitzustellen. So können zum Beispiel auch Abhängigkeiten zu weiteren Plug-ins definiert werden oder es lassen sich Aufrufe von Methoden zeitlich oder ereignisgesteuert regeln. Die Dokumentation des Plug-in Systems ist umfangreich. Mittels einfachen Beispielen wird ein Entwickler schnell und verständlich in die Verwendung des Plug-in Systems eingeführt. Leider scheint es so, als wird das Plug-in System nicht mehr weiterentwickelt oder gepflegt, da es seit September 2011 keine Änderungen mehr erfahren hat.

- **JPF**

  Das „Java Plug-in Framework[8]“ (JPF) wurde Mitte 2004 ins Leben gerufen. Bei diesem Plug-in System werden die Informationen über die Plug-ins, wie zum Beispiel deren Abhängigkeiten zu anderen Plug-ins, extern in sogenannten Plug-in Manifesten bereitgestellt. Hierfür wird standardmäßig eine

[6] http://docs.oracle.com/javase/6/docs/api/java/util/ServiceLoader.html
[7] https://code.google.com/p/jspf/
[8] http://jpf.sourceforge.net/

XML-Syntax zur Konfiguration der einzelnen Plug-ins verwendet. Dies kann aber auch vom Entwickler beliebig angepasst werden. Das Plug-in System besteht insgesamt aus den drei Komponenten *PluginRegistry*, *PathResolver* und *PluginManager*. Die *PluginRegistry* kümmert sich um die Verwaltung aller Plug-ins. Der *PathResolver* ist für das Auffinden und Laden von zusätzlichen Ressourcen zuständig. Der *PluginManager* dient dem Laden, Aktivieren und Entladen von Plug-ins. JFP überlässt dem Entwickler dabei die Wahl, ob er die Standardkomponenten verwenden möchte oder eine eigene Implementierung bestimmter Komponenten bevorzugt. Jede Komponente lässt sich mittels der Fabrikmethode (Factory pattern (Metsker, 2002)) individuell anpassen oder implementieren. Plug-ins können zur Laufzeit geladen/entladen oder aktiviert/deaktiviert werden, was den Speicherverbrauch des Plug-in Systems reduzieren kann. Das Plug-in System ist ausführlich dokumentiert. Durch seine Komplexität und die nötigen Plug-in Manifeste benötigt ein Entwickler jedoch eine gewisse Einarbeitungszeit bis er das Plug-in System verwenden kann. Die letzte stabile Version von JPF stammt aus dem Jahr 2007. Seit dem gibt es an dem Plug-in System keine Änderungen mehr.

- **PF4J**

  Das „Plugin Framework for Java[9]“ (PF4J) von Decebal Suiu befindet sich seit Oktober 2012 in Entwicklung. PF4J nutzt, ebenso wie JPF, Plug-in Manifeste um Informationen über die Plug-ins zur Verfügung zu stellen. Ebenso werden Abhängigkeiten zu weiteren Plug-ins unterstützt, die im Manifest des Plug-ins deklariert sein müssen. PF4J besteht aus den drei Komponenten *PluginManager*, *PluginDescriptorFinder* und *ExtensionFinder*. Der *PluginManager* kümmert sich um das Laden und Aktivieren der Plug-ins. Der *PluginDescriptorFinder* dient dem Auffinden der einzelnen Plug-ins. Der *ExtensionFinder* findet Erweiterungen, welche in den einzelnen Plug-ins zusätzlich zur Verfügung gestellt sein können. Wie es auch bei JPF der Fall, überlässt PF4J dem Entwickler ebenfalls die Wahl, ob er die Standardkomponenten verwenden möchte oder eine eigene Implementierung bestimmter Komponenten bevorzugt. Das System ermöglicht auch Plug-ins zur Laufzeit zu laden/entladen oder aktivieren/deaktivieren, was den Speicherverbrauch reduzieren kann. Die Dokumentation des Plug-in System ist jedoch nur zufriedenstellend. Sie bietet lediglich einen kleinen Überblick über die einzelnen Komponenten. Tiefergehende Informationen muss man sich anhand des Quellcodes und der vorhandenen Beispiele selber erarbeiten. Die Einstiegshürde zum Verwenden von PF4J ist somit sehr hoch. Das Plug-in System wird noch immer von Decebal Suiu gewartet und weiterentwickelt.

[9] `https://github.com/decebals/pf4j`

2.2.2 Plug-in Systeme als Architektur von Anwendungen

Die Abbildung 2.2 beschreibt eine eher weniger bekannte Möglichkeit, wie Plug-in Systeme genutzt werden können. Im Unterschied zu 2.2.1 erweitern die Plug-ins hierbei nicht die Anwendung. Erst durch ihren Zusammenschluß wird die letztendliche Anwendung erzeugt und ermöglicht. Die Plug-ins in einem solchen System bilden ein komplexes Netzwerk, indem sie selbst Funktionen bereitstellen, aber auch auf Funktionen anderer Plug-ins zurückgreifen können.

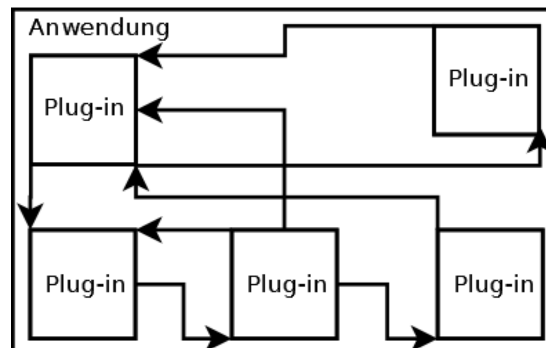


Abbildung 2.2: Architektur von Anwendungen

Die bekannteste Variante eines solchen Plug-in Systems spiegelt sich in dem OSGI-Framework[10] (Alliance, 2014) wieder. Es handelt sich dabei jedoch nicht um ein fertig verwendbares Plug-in System, sondern vielmehr um eine Spezifikation eines solchen Systems und seiner Komponenten. So besteht das OSGI-Framework aus einer Reihe von Schichten, wie sie in Abbildung 2.3 aufgezeigt werden.

- **Module (Modules)**

  Diese Schicht definiert eine Komponente (Bundle) als grundlegende Modularisierungseinheit. Komponenten sind JAR-Archive und enthalten alle Klassen, Ressourcen und das Bundle-Manifest, welches die Eigenschaften der Komponente festlegen. Das Manifest beschreibt dabei alle Abhängigkeiten und die Schnittstelle, über welche auf die Funktionen der Komponente zugegriffen werden können.

- **Lebenszyklus (Life cycle)**

  Diese Schicht definiert alle Zustände einer Komponente während ihres Lebenszyklus. So können Komponenten zur Laufzeit geladen, aber auch wieder entladen werden. Ebenso lassen sich Komponenten starten und stoppen.

[10] `http://www.osgi.org/Main/HomePage`

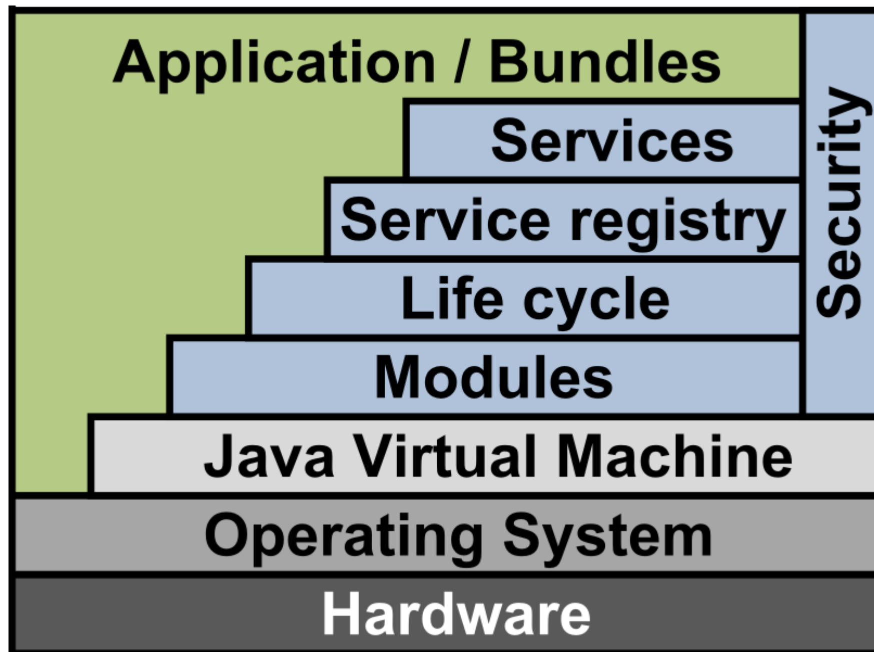


Abbildung 2.3: OSGi Framework
Quelle: `http://en.wikipedia.org/wiki/OSGi#mediaviewer/File:Osgi_framework.svg`

- **Dienstverwaltung (Service registry)**

  Diese Schicht definiert eine Dienstverwaltung, an der sich Komponenten in bestimmten Abschnitten ihres Lebenszyklus an- oder abmelden können. Dadurch können sie auf Dienste anderer Komponente zugreifen oder selbst welche bereitstellen.

- **Dienste (Services)**

  Diese Schicht definiert die Schnittstelle von Komponenten, um auf deren Dienste zugreifen zu können.

- **Sicherheit (Security)**

  Diese Schicht kümmert sich um die Rechteverwaltung der Komponenten und der Einhaltung von Sicherheitsvorschriften (zum Beispiel der Signaturprüfung der JAR-Archiven der Komponenten).

- **Anwendung/Komponenten (Application/Bundles)**

  Die Anwendung und ihre Komponenten agieren nun durch ihre untergeordneten Schichten mit dem Betriebssystem und der Hardware.

Die bekannteste Umsetzung des OSGI-Frameworks dürfte wohl Equinox[11] (McAffer, VanderLei & Archer, 2010) sein, vielen eher aus der Entwicklungsplattform Eclipse bekannt. Eclipse ist dabei keine einfache Anwendung, sondern vielmehr ein Zusammenschluss vieler verschiedener Komponenten, die mittels Equinox erst die gesamte Anwendung ausmachen. So stellt zum Beispiel die sogenannte „Rich Client Platform[12]“ alles zur Verfügung, was zum betreiben einer grafischen Oberfläche benötigt wird. Darauf aufbauend können weitere Komponenten die Unterstützung für verschiedene Programmiersprachen beisteuern. Neben Equinox existieren noch weitere Implementierungen, welche aber meist auf bestimmte Anwendungsszenarien angepasst sind.

Obwohl die Spezifikation des OSGI-Frameworks mittlerweile in der sechsten Version vorliegt und seit nunmehr über 14 Jahren weiterentwickelt wird, konnte es dennoch keine wirkliche Verbreitung finden. Sheikh Sajid fasst in seinem Blog[13] einige Gründe zusammen, warum das OSGI-Framework auch in Zukunft eher bei großen Projekten eingesetzt werden wird. Neben der Komplexität der Spezifikation, sind es auch die vielen Fachwörter, weshalb sich gerade Neulinge mit dem Thema OSGI schwer tun. Auch fehlt es an einer brauchbaren Referenzimplementierung. Viele Projekte „kochen ihr eigenes Süppchen“ und setzen nur jene Teile der Spezifikation um, die sie auch benötigen.

2.2.3 Eigenentwicklung für die Anwendung

Ein Plug-in System kann aber auch extra für den Einsatz in einer Anwendung entwickelt werden. Gegenüber den bereits beschriebenen Arten von Plug-in Systemen, bringt ein solches Vorteile, aber auch Nachteile mit sich. Wie das Sprichwort „Das Rad neu erfinden.“ bereits erahnen lässt, sollte man immer gut abwägen, ob sich eine Eigenentwicklung wirklich lohnt. In vielen Fällen ist der Einsatz eines existierendes Plug-in System eher zu empfehlen.

Die Vorteile von existierenden Plug-in Systemen liegen auf der Hand. So können sie zum Beispiel, je nach Verbreitung, über eine gute Dokumentation und reife, fehlerarme Codebasis verfügen. Je länger ein Plug-in System im Einsatz ist, desto stabiler und fehlerfreier arbeitet es in der Regel. Auch kann die Häufigkeit der Verwendung ein Indiz über dessen Qualität sein. Wird ein Plug-in System weiterhin durch Entwickler gepflegt, so hat man auch gleich Ansprechpartner, die sich mit dem System auskennen und bei Problemen oder Fragen weiterhelfen können.

Neben ihren Vorteilen, können solche Plug-in Systeme auch Nachteile mit sich bringen. Mangelhafte oder gar fehlende Dokumentationen bedeuten oftmals eine

[11] http://www.eclipse.org/equinox/
[12] http://wiki.eclipse.org/index.php/Rich_Client_Platform
[13] http://myjavaexp.blogspot.in/2013/10/osgi-road-ahead.html

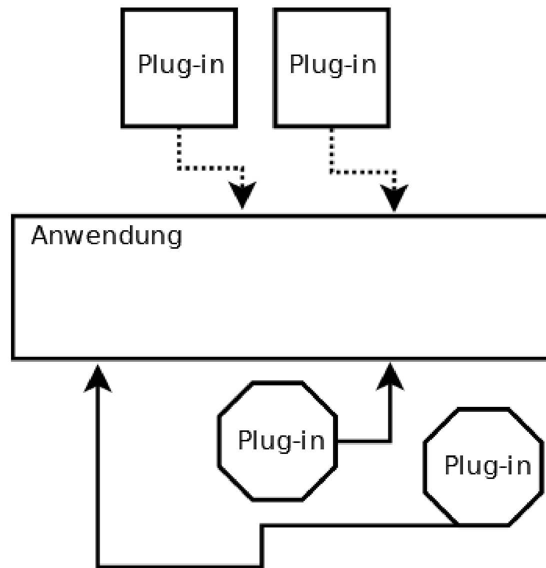


Abbildung 2.4: Eigenentwicklung für die Anwendung

längere Einarbeitungszeit und unter Umständen sogar eine falsche Anwendung des Plug-in Systems. Nicht immer lässt sich eine Aussage über die Qualität der Codebasis machen oder das gesamte Plug-in System bewerten. Dies ist zum Beispiel der Fall, wenn kaum Erfahrungsberichte vorliegen oder wenig realitätsnahe Beispiele zu dessen Verwendung existieren. Auch wenn ein Plug-in System in seiner Art für den Einsatz geeignet wäre, so macht es, je nach Projektgröße, wenig Sinn dieses zu verwenden, wenn die Unterstützung seitens des Entwicklers nicht ausreichend vorhanden ist. Ein Plug-in System ohne aktiven Entwickler kann im schlimmsten Fall sogar einen kompletten Neustart des Projektes mit einem anderen Plug-in System bedeuten. Um bereits existierende Plug-in Systeme zu nutzen, bedarf es bestenfalls nur geringer Anpassungen an der Anwendung. Aber genauso könnte die Integration des Plug-in Systems auch komplexere Anpassungen mit sich bringen, die nicht nur zeitaufwendig sind, sondern unter Umständen auch gar nicht umgesetzt werden können.

Genau hier liegt die Stärke einer Eigenentwicklung. Die Anwendung muss nicht explizit für ein Plug-in System angepasst werden, sondern das neue Plug-in System kann genau passend für die Anwendung entwickelt werden, wie in Abbildung 2.4 zu sehen. So kann ein System geschaffen werden, dass im Idealfall alle Anforderungen seitens der Anwendung erfüllt und jederzeit an weitere neue Anforderungen nachträglich angepasst werden kann.

Doch die Stärke der Eigenentwicklung ist zugleich ihr Nachteil. Da ein solches Plug-in System meist genau für eine bestimmte Anwendung entwickelt wurde, lässt es sich nur schwer in anderen Anwendungen einsetzen. Somit erlangt das Plug-in System keine große Verbreitung und damit auch oftmals weniger An-

wender, die eventuelle Fehler und Nachteile des Systems melden können. Auch birgt eine Eigenentwicklung ein größeres Fehlerpotential, da während der Design- und Entwicklungsphase nicht immer auf die Erfahrungen anderer Plug-in Systemen zurückgegriffen wird und somit bereits bekannte und gelöste Fehler erneut passieren.

Eigenentwicklungen können die richtige Entscheidung sein, gehen aber oftmals mit erhöhtem Lern- und Entwicklungsaufwand einher.

2.3 Anforderungen an das neue Plug-in System

Seit der Veröffentlichung von JDownloader 1 hat sich einiges getan. So bedient sich JDownloader 2 beispielsweise der Möglichkeiten kontinuierlicher Integration und Auslieferung. Hierfür wurde gegen Ende 2012 ein von Grund auf neu entwickeltes Update- und Build-System[14] entwickelt. Dadurch sind die Entwickler nun in der Lage JDownloader 2, wie in Abbildung 4.2 in Anhang B zu sehen, mehrmals täglich mit Aktualisierungen zu versorgen. Zudem muss JDownloader 2 die stetig wachsende Anzahl an Plug-ins bewältigen. Die Abbildung 4.1 aus dem Anhang A zeigt, dass alleine im Jahr 2013 die Anzahl der Plug-ins um circa zehn Prozent zunahm.

Anhand dieser und weiterer Entwicklungen konnten Anforderungen an das Plug-in System von JDownloader 2 gestellt werden, die während der Entwicklung von Bedeutung waren. Man unterscheidet zwei Arte von Anforderungen, nämlich funktionale und nicht-funktionale. Mit funktionalen Anforderungen[15] sind in dieser Arbeit jene Funktionen oder Forderungen an eine Anwendung oder Entwicklung gemeint, welche zwingend für ihre korrekte Ausführung oder Umsetzung nötig sind. Im Gegensatz dazu stehen die nicht-funktionalen Anforderungen[16], welche nicht an eine bestimmte Anwendung oder Entwicklung gekoppelt sind, sondern eher allgemeine Eigenschaften/Merkmale beschreiben. In dem ISO Standard 9126 wurden zum Beispiel Merkmale wie Performanz, Portabilität oder Sicherheit definiert (Abran, Khelifi, Suryn & Seffah, 2003), welche ebenfalls als nicht-funktionale Anforderungen angesehen werden können (Chung & do Prado Leite, 2009).

Im folgenden werden nun die Anforderungen an das neue Plug-in System von JDownloader 2 erläutert.

[14] automatisierte Übersetung (Kompilierung) der Quellen und Plug-ins mit Rechteverwaltung
[15] Sortieren von Zahlen
[16] effizientes Sortieren von Zahlen

2.3.1 Funktionale Anforderungen

Die Entwicklungs- und Planungsphasen des Plug-in Systems von JDownloader orientierten sich stark an folgenden zwei funktionalen Anforderungen.

- **Abwärtskompatibilität der Plug-ins zu JDownloader 1 (Interoperabilität, Stabilität)**

  Nach der Veröffentlichung von JDownloader 1 und mit dem Beginn der Weiterentwicklung wurde damals die Entscheidung getroffen, dass alle JDownloader Versionen für die Kompilierung der jeweiligen Plug-ins den gleichen Quellcode als Grundlage verwenden sollten.

  Diese Entscheidung wurde hauptsächlich aufgrund von zwei Tatsachen gefällt. Einerseits fehlte dem Projekt schlichtweg die nötige Arbeitskraft (Man-Power) um zwei oder mehrere getrennte Plug-in Quellen, für unterschiedliche JDownloader Versionen zu warten. Dies lag primär daran, dass JDownloader schon zum Zeitpunkt der Veröffentlichung von Version 1 über tausend verschiedene Plug-ins beinhaltete, welche zum Teil mehrmals am Tage oder in der Woche Änderungen erhielten. Andererseits dachte man damals noch nicht an die Notwendigkeit von zukünftigen Änderungen von Plug-in Schnittstellen oder gar des gesamten Plug-in Systems von JDownloader, da sich das Plug-in System von JDownloader 1 bis dahin als gut bewährt hatte.

  Es ist somit auch weiterhin zwingend notwendig, das bei allen Änderungen an der Plug-in Schnittstelle oder dem Plug-in System selbst, eine fehlerfreie Übersetzung (Kompilierung) der Plug-ins für JDownloader 1 und die korrekte Funktionsweise dieser ermöglicht wird. Dies bedeutet aber auch, dass Plug-ins keinerlei neue Methoden oder Klassen nutzen dürfen, welche nicht in JDownloader 1 existieren.

- **Erweiterbarkeit von Plug-ins und der Plug-in Schnittstelle (Modifizierbarkeit, Stabilität)**

  Mit der Weiterentwicklung des JDownloader waren nachträgliche Erweiterungen an der Plug-in Schnittstelle von JDownloader 1 unvermeidlich. Eine Änderungen und Fehlerbereinigung bestehender oder die Einführung neuerer Funktionen, wie zum Beispiel die Unterstützung von sogenannten Multi-Hostern[17], wären sonst nicht möglich gewesen.

  Wegen Rücksicht auf die Abwärtskompatibilität zu JDownloader 1 dürfen sich diese Änderungen nicht auf bestehende Teile der Plug-in Schnittstelle auswirken. Auch sollte die Verwendung neuerer Funktionen im Idealfall die Funktionalität eines Plug-ins in JDownloader 1 nicht negativ beeinträchtigen.

Ohne die genannten Anforderungen wäre die weitere Versorgung des JDownloader 1 mit Plug-in Aktualisierungen und die zeitgleich fortlaufende Weiterentwicklung des JDownloader zur jetzigen Version 2 nur mit erhöhtem zusätzlichen Aufwand möglich gewesen. Solange aber JDownloader 1 weiterhin mit Plug-in Aktualisierungen aus den selben Quellen wie JDownloader 2 versorgt werden soll, müssen die Entwickler auch in Zukunft diese zwei Anforderungen bei all ihren Änderungen oder Neuerungen beachten.

2.3.2 Nicht-funktionale Anforderungen

Zusätzlich zu den bereits genannten zwei funktionalen Anforderungen, kristallisierten sich während der fortlaufenden Entwicklung von JDownloader weitere nicht-funktionale Anforderungen heraus. Teilweise entstanden nachfolgende Anforderungen während der Verwendung von JDownloader selbst. So zum Beispiel der Wunsch die Startzeit von JDownloader zu reduzieren. Auch kamen durch die Weiterentwicklung und dem wachsendem Wissen seitens der Entwickler neue Möglichkeiten zum Vorschein, wie zum Beispiel das Laden und Entladen von Klassen zur Laufzeit.

Folgende nicht-funktionale Anforderungen spielten somit ebenfalls während der gesamten Entwicklungs- und Planungsphasen des Plug-in Systems von JDownloader 2 eine Rolle.

[17]Dienstleister, welche das Herunterladen von Dateien und Inhalten weiterer Anbieter/Portalen ermöglichen.

- **Reduktion der Ladezeit der Plug-ins (Effizienz, Attraktivität)**

  Mit zunehmender Anzahl an Plug-ins steigt auch die Startzeit des JDownloader. Sehr deutlich ist dies beim veralteten JDownloader 1 zu sehen. Nach dessen Start (zum Beispiel Doppelklick in Windows) dauert es gut 20 Sekunden[18] oder gar länger, bis JDownloader die ersten Befehle des Anwenders entgegennehmen kann.

  Die Startzeit ist, neben der Anzahl der Plug-ins, natürlich auch von weiteren Komponenten des JDownloader und der jeweiligen Ausstattung des Computers abhängig. Vergeht eine zu lange Zeit, wobei dies eine sehr subjektive Betrachtung ist, werten viele Anwender dies bereits als negatives Zeichen für die Qualität von JDownloader.

  Eine Reduzierung der Ladezeit der Plug-ins und somit der gesamten Startzeit ist daher wünschenswert und sorgt auch bei dem Anwender für einen positiveren (Erst-)Eindruck über JDownloader.

- **Reduktion des Speicherverbrauchs zur Laufzeit (Effizienz, Attraktivität)**

  Neben der Reduzierung der Startzeit von JDownloader sollte auch der Speicherverbrauch zur Laufzeit in einem angemessenen Verhältnis bleiben. Denn viele Anwender urteilen schnell anhand des Speicherverbrauchs und dem eigentlichen Verwendungszweck der Anwendung über deren Qualität. Auch werden schnell ähnliche Anwendungen[19] mit JDownloader verglichen, die für viele Anwender subjektiv mit gleicher Funktionsvielfallt daherkommen und dennoch deutlich weniger Speicher belegen. Das dies nicht immer allein der in Java entwickelten Anwendung angelastet werden darf und auch wenig über deren Qualität aussagt, sondern oftmals Java, mit seiner JVM und deren Speicherverwaltung[20], geschuldet ist, interessiert dabei die wenigsten.

  Neben dieser subjektiven Betrachtung gibt es aber auch objektive Gründe den Speicherverbrauch zu reduzieren. Dank der plattformübergreifenden Programmiersprache Java ist der Einsatz von JDownloader auf sogenannten Mini-Computern durchaus denkbar und aufgrund des geringeren Stromverbrauchs eine realisierbare Alternative zum heimischen Computer. Während heutzutage Computer jedoch bereits ab Werk mit mehreren Gigabyte an Hauptspeicher ausgestattet sind, müssen viele Mini-Computer mit deutlich weniger Hauptspeicher auskommen. So gelingt beispielsweise auf einem Raspberry Pi die Ausführung der veralteten Version JDownloader 1 nur mit

[18] Nach einem Neustart von Windows 7, 128 GB SSD, 8GB Ram und 2.8 GHz DualCore-CPU
[19] Beispiel: InternetDownloadManager, siehe `http://www.idm.com`
[20] automatisierte Garbage Collection statt manueller Speicherverwaltung

viel Mühe und einer spezieller Java Konfiguration. Ein Raspberry Pi verfügt je nach Model über 256 oder 512 Megabyte Hauptspeicher. Neben dem Betriebssystem und sonstiger Komponenten bleibt somit nicht viel Speicher für JDownloader übrig. Je geringer der Speicherverbrauch ist, desto besser läuft JDownloader auf solchen Systemen.

Ein weiterer Grund liegt in der Speicherverwaltung von Java selbst. Die beiden veralteten Java Versionen 1.5 und 1.6 sowie die derzeit aktuelle Version 1.7 legen alle geladenen Klassen und bestimmte statische Objekte[21] in einem separaten Speicherbereich ab. Dieser sogenannte Perm-Gen ist standardmäßig auf circa 64 Megabyte limitiert. Auch wenn 64 Megabyte zunächst nach reichlich Platz für die vielen Plug-ins von JDownloader klingt, so wurde dieses Limit in der Vergangenheit bereits mehrmals erreicht. Als Folge verweigerte JDownloader 1 unvorhersehbar mit OutOfMemory-Fehlern den Dienst. Auch heute existiert noch diese Einschränkung und muss bei der Wartung und Entwicklung von Plug-ins beachtet werden. Mit der Verschiebung des Perm-Gen in den flexibleren Heap-Speicherbereich in der kommenden Java Version 1.8 ist zwar diese Einschränkung beseitigt, der hohe Speicherverbrauch aber bleibt.

Um auch mit zukünftig steigender Anzahl von Plug-ins den JDownloader möglichst vielen Plattformen und Computern zugänglich zu machen, ist eine Reduzierung des Speicherverbrauches notwendig.

- **Aktualisierung einzelner Plug-ins zur Laufzeit (Attraktivität, Zuverlässigkeit)**

  Benötigt ein Anwender von JDownloader 1 eine Aktualisierung eines von ihm verwendeten Plug-ins, so bedarf es im Regelfall hierfür zwei komplette Neustarts von JDownloader. Nach dem ersten Neustart entdeckt JDownloader 1 zunächst die zur Verfügung stehende Aktualisierung des Plug-ins und lädt diese herunter. Da zu diesem Zeitpunkt das veraltete Plug-in jedoch bereits geladen wurde, bedarf es nochmals einen weiteren Neustart von JDownloader, damit das neue Plug-in endlich geladen wird und zum Einsatz kommen kann.

  Allein diese Notwendigkeit des zweifachen Neustarts kann schnell zu Frust und einer negativen Einstellung gegenüber JDownloader bei Anwendern führen, wenn sie auf häufige Aktualisierungen von Plug-ins angewiesen sind. Manch ein Anbieter führt Änderungen mehrmals täglich durch oder ein Entwickler behebt Fehler in einem Plug-in, was aufgrund fehlender oder mangelhafter Dokumentation seitens des Anbieters oft auch mehrere Aktualisierungen benötigen kann. Auch darf man die lange Startzeit von JDownloader 1 nicht außer Acht lassen. Ein Neustart ist zudem auch nicht

[21] zum Beispiel Strings, welche zur Laufzeit nicht mehr geändert werden

jederzeit möglich. Eventuell entpackt der JDownloader gerade ein Archiv, oder lädt derzeit eine Datei herunter, dessen Fortschritt nach einem Neustart wohlmöglich verloren ginge. Es sprechen viele Gründe dafür, Neustarts einer Anwendung zu vermeiden, wenn es denn möglich ist.

Deshalb ist das Aktualisieren einzelner Plug-ins zur Laufzeit eine Anforderung, welche dem Anwender einen großen Mehrwert bietet und so für einen positiveren Gesamteindruck über JDownloader sorgen kann.

- **Einfache Integration von Plug-ins Dritter (Anpassbarkeit, Installierbarkeit)**

  Da die Quellen von JDownloader jedem zur Verfügung stehen, kann auch jeder Entwickler, der ein wenig Erfahrung mit der Programmiersprache Java mitbringt, sich ein bestimmtes Plug-in anschauen und bei Bedarf den eigenen Wünschen entsprechend anpassen. Auch steht es jedem Entwickler frei weitere Plug-ins für neue oder nicht unterstützte Anbieter zu entwickeln.

  Nicht jedes Plug-in soll oder kann dabei in die offizielle JDownloader Version der Appwork GmbH aufgenommen werden. Vielleicht hat der Entwickler lediglich ein Plug-in für einen von ihm persönlich genutzten Anbieter entwickelt. Oder er hat ein vorhandenes Plug-in mit Modifikationen versehen, welche nicht für die Allgemeinheit bestimmt sind. Auch können rechtliche Gründe gegen die Aufnahme des Plug-ins in die offizielle Version sprechen.

  Ein Entwickler sollte nicht in der Entwicklung von Plug-ins, ob für den privaten Gebrauch oder die offizielle JDownloader Version, eingeschränkt werden. Ebenso sollten Anwender in der Lage sein, ohne größere Umstände Plug-ins von Dritten zu nutzen.

  Jedoch sind genau diese Freiheiten in der JDownloader 1 Version nicht gegeben. Das Plug-in System ist nicht in der Lage zuverlässig für einen Anbieter ein entsprechendes Plug-in zu bestimmen, falls hierfür mehrere Plug-ins zur Auswahl stehen. Dies ist zum Beispiel der Fall, wenn ein Anwender ein inoffizielles Plug-in für einen Anbieter nutzen möchte, dieser aber bereits von einem offiziellen Plug-in unterstützt wird. Das Plug-in System ist hier nicht deterministisch, da es je nach verwendetem Betriebssystem, Dateisystem und den Zeitstempel der Plug-in Dateien ein anderes Plug-in für den Anbieter auswählen wird. Der Anwender kann sich also nicht darauf verlassen, dass das inoffizielle Plug-in verwendet wird.

  Die Funktionsweise eines Plug-in Systems sollte jederzeit deterministisch sein, denn nur so lässt sich die einfache Integration von externen Plug-ins umsetzen.

- **Geringe oder keine Änderungen an vorhandenen Plug-ins nötig (Wartbarkeit, Benutzbarkeit)**

  Aufgrund der vielen Plug-ins und der geforderten Abwärtskompatibilität zu JDownloader 1 ist es verständlich, dass ein Entwickler seine Zeit sinnvoll in die Weiterentwicklung und Fehlerbehebung des neuen Plug-in Systems investieren können sollte, statt dauernd auf die Kompatibilitäten aller Plug-ins achten zu müssen.

  Es wäre also von Vorteil, wenn das neue Plug-in System möglichst keine direkte oder nur wenige Bindungen zu den eigentlichen Plug-ins hätte. Änderungen an der Plug-in Schnittstelle könnten somit jederzeit durchgeführt werden und hätten im Idealfall keinen Einfluß auf das vorhandene Plug-in System. Ebenso kann dieses weiterentwickelt werden, ohne das dabei viel Rücksicht auf die Plug-ins genommen werden muss.

2.4 Das Plug-in System von JDownloader 2

In diesem Abschnitt wird das neue Plug-in System von JDownloader 2 besprochen. Zunächst wird in 2.4.1 ein Überlick über dessen Architektur und einzelnen Komponenten gegeben. Anschließend werden in 2.4.2 Probleme erläutert, welche während der Entwicklung und dem Einsatz des neuen Plug-in Systems auftraten. Am Ende dieses Abschnittes wird unter 2.4.3 ein Vergleich der beider Plug-in Systeme durchgeführt, um die Verbesserungen der Weiterentwicklung aufzuzeigen.

2.4.1 Systemarchitektur

Abbildung 2.5 verschafft einen Überblick über die Architektur und aller involvierten Komponenten des Plug-in Systems von JDownloader 2. Sie lässt sich in zwei Hälften einteilen. Die Komponenten auf der linken Seite der Abbildung dienen dem eigentlichen Laden der Plug-ins und deren Klassen, während die Komponenten auf der rechten Seite für ihre Verwaltung und Verwendung zuständig sind.

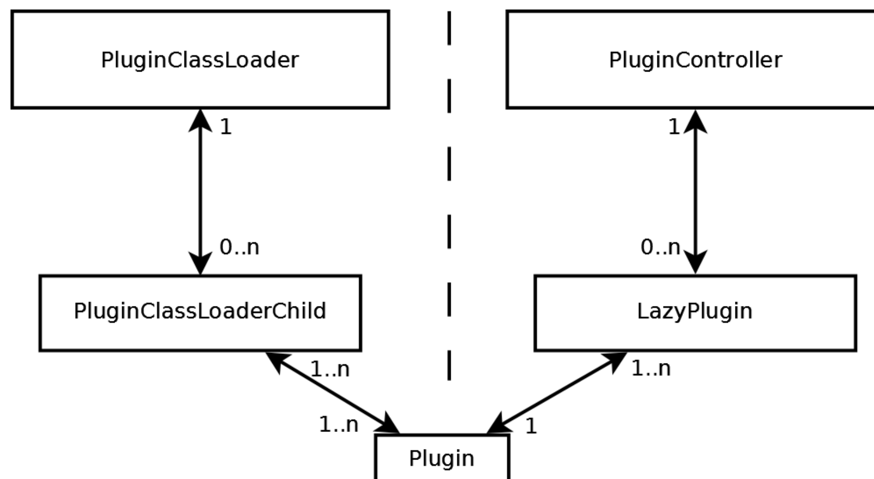


Abbildung 2.5: Plug-in System von JDownloader 2

- **PluginController und Lazy-Plug-in**

  Der *PluginController* (siehe Anhang G) agiert in JDownloader 2 als zentrale Anlaufstelle für die Verwaltung, den Zugriff und die Aktualisierung der Plug-ins. Hierfür wird ein Verzeichnis aller installierter Plug-ins gepflegt und aktuell gehalten, falls sich durch das Update-System oder seitens des Anwenders Änderungen an ihnen ergeben sollten. Um den Speicherverbrauch hierbei so gering wie möglich zu halten, wird jedoch nicht die eigentliche Implementation des Plug-ins referenziert, sondern lediglich dessen am

häufigsten verwendeten Informationen seitens JDownloader. Hierzu werden im *PluginController* $0..n$ installierte Plug-ins durch $1..n$ (ein Plug-in kann mehrere Anbieter implementieren) entsprechende Lazy-Plug-ins (siehe Anhang E) repräsentiert.

Im Umgang mit den Plug-ins benötigt JDownloader normalerweiße nur ein paar wenige statische Informationen. Deshalb bietet es sich an, diese Informationen in einem Lazy-Plug-in auszulagern und so die Zugriffe auf dessen eigentliches Plug-in zu minimieren. Trotz des zusätzlichen Lazy-Plug-ins kann der Speicherverbrauch im Umgang mit den Plug-ins weiter reduziert werden. Zum einen benötigten die Lazy-Plug-in Pendants deutlich weniger Speicher und zum anderen muss das Plug-in nun erst bei Bedarf geladen werden. Auch kann es nach der Verwendung im Anschluss wieder freigegeben werden. Nicht oder nur selten verwendete Plug-ins belegen somit nur einen Bruchteil ihres eigentlichen Speicherbedarfs. Das Lazy-Plugin dient JDownloader als Informationsquelle und übernimmt, sobald auf das Plug-in zugegriffen werden muss, alle nötigen Schritte zum Laden, Instanziieren und Entladen des Plug-ins.

- **PluginClassLoader und PluginClassLoaderChild**

  Der *PluginClassLoader* (siehe Anhang F) ermöglicht den korrekten Umgang mit mehreren Klassenlader. Dies ist nötig, da erst der Einsatz mehrerer voneinander unabhängiger Klassenlader das Entladen von Plug-ins und sonstigen Klassen zur Laufzeit ermöglicht. Ebenfalls werden sie benötigt, um Plug-ins und Klassen zur Laufzeit zu aktualisieren. Um Probleme zu vermeiden, die hierbei auftreten können (siehe Abschnitt 2.4.2), bedarf es einer Verwaltung der Klassenlader. Der *PluginClassLoader* stellt deswegen alle nötigen Methoden bereit, damit der Entwickler einfach und fehlerfrei mit den Klassenlader arbeiten kann. Zusätzlich bedarf es aber noch einem modifizierten Klassenlader, um das gewünschte Verhalten überhaupt zu ermöglichen. In JDownloader geschieht dies in der Klasse *PluginClassLoaderChild* (siehe Anhang F.1).

  Ein *PluginClassLoaderChild* ist ein Klassenlader, bei dem das Laden von Klassen dahingehend abgeändert wurde, dass er unabhängig von dem sonst üblichen Delegationsmodell von Java arbeiten kann. Das heißt, dass er das Laden von Plug-ins und Klassen nicht erst an seinen übergeordneten Klassenlader deligiert, sondern sich selbst darum kümmert.

  Der *PluginClassLoader* verwaltet somit $0..n$ *PluginClassLoaderChild*, welche jeweils $1..n$ Plug-ins geladen haben. Durch den Einsatz mehrerer Klassenlader ist es ebenfalls möglich, dass das selbe Plug-in von $1..n$ Klassenlader geladen wurde.

- **Plugin**

  Zum Laden und Instanziieren eines Plug-ins sind 2 Komponenten nötig. Zum einen ein entsprechendes Lazy-Plug-in und zum anderen ein *PluginClassLoaderChild*, welcher für das Laden des Plug-ins und seiner Klassen zusändig ist. Ebenso kann das Plug-in nach der Verwendung wieder entladen oder bei Berdarf bereits zur Laufzeit aktualisiert werden. Der Zugriff auf beide benötigte Komponenten wird durch den *PluginController* und dem *PluginClassLoader* ermöglicht und erleichtert.

Die Aufteilung des Plug-in Systems in die vier vorgestellten Komponenten bietet zwei große Vorteile. So können zum einen Komponenten weiterentwickelt oder ersetzt werden, ohne dass dabei die Funktionalität der anderen Komponenten oder des gesamten Plug-in Systems beeinträchtigt wird. Das dies der Fall ist, zeigt bereits die Tatsache, dass das neue Plug-in System bereits während seiner Entwicklung in der öffentlich zugänglichen Version von JDownloader 2 zum Einsatz kam. Zum anderen gestaltet sich die Fehlersuche und Wartung einzelner, leicht überschaubarer, Komponenten einfacher, als bei einem großen komplexen System.

Folgender Pseudocode demonstriert, wie einfach der Umgang und die Nutzung des neuen Plug-in Systems von JDownloader 2 ist.

```
//Zugriff auf das LazyPlugin
LazyPlugin lazyPlugin = PluginController.get("github.com");

//ohne Zugriff auf das Plugin, rein nur LazyPlugin
String host=lazyPlugin.getDisplayName();

//mit Zugriff auf das Plugin, geteilte Instanz
Plugin a = lazyPlugin.getPrototype(null);
String hosta = a.getHost();

//mit Zugriff auf das Plugin, neue Instanz
Plugin b = lazyPlugin.newInstance(null);
String hostb = b.getHost();

//mit Zugriff auf das Plugin, neue Instanz und neuer Klassenlader
Plugin c = lazyPlugin.newInstance(PluginClassLoader.getInstance().getChild());
String hostc = c.getHost();

//mehrere Klassenlader
System.out.println("Instance:" + a + "|Class:" + a.getClass() + "|CL=" + a.getClass().getClassLoader());
System.out.println("Instance:" + b + "|Class:" + b.getClass() + "|CL=" + b.getClass().getClassLoader());
System.out.println("Instance:" + c + "|Class:" + c.getClass() + "|CL=" + c.getClass().getClassLoader());
//Ausgabe
Instance:GitHubCom@574516d4|Class:class GitHubCom|CL=PluginClassLoader$PluginClassLoaderChild@2c59e61a
Instance:GitHubCom@78dde50e|Class:class GitHubCom|CL=PluginClassLoader$PluginClassLoaderChild@2c59e61a
Instance:GitHubCom@2fe93820|Class:class GitHubCom|CL=PluginClassLoader$PluginClassLoaderChild@2db36a91
```

2.4.2 Mehrere Klassenlader - Segen oder Fluch?

Die Idee, mehrere Klassenlader zum Laden und Entladen von Plug-ins zu verwenden, war der Grundstein für das Plug-in System, wie es jetzt in JDownloader 2 existiert. Schnell war eine erste lauffähige Version des neuen Plug-in Systems entwickelt und diese zeigte auch die erhofften Resultate. Das Entladen von Plug-ins war wie gewünscht möglich und so wurde das System stetig weiterentwickelt. Doch die Ernüchterung kehrte schnell ein, als erste komplexere Plug-ins getestet wurden und sie ihren Dienst mit bis dato unbekannten Fehlern oder Fehlverhalten quitierten. Wie sich schließlich herausstellte, lag die Ursache hierfür in dem falschen Umgang mit mehreren Klassenlader und deren Auswirkung auf die Plug-ins.

Im Folgenden wird auf die hierdurch entstandenen Probleme, ihre Ursachen und Lösungen eingegangen.

- **Warum ist es nicht die gleiche Klasse?**

  Folgender Pseudocode und dessen Ausgabe soll das Problem verdeutlichen:

```
Class<?> c1 = new ClassLoader().loadClass("org.appwork.Test");
Class<?> c2 = new ClassLoader().loadClass("org.appwork.Test");
Object o1 = c1.newInstance();
Object o2 = c2.newInstance();
System.out.println("c1.equals(c2): " + c1.equals(c2));
System.out.println("o1 instanceof org.appwork.Test: " + (o1 instanceof org.appwork.Test));
System.out.println("o2 instanceof org.appwork.Test: " + (o2 instanceof org.appwork.Test));

c1.equals(c2): false
o1 instanceof org.appwork.Test: false
o2 instanceof org.appwork.Test: false
```

  Normalerweiße, so könnte ein Entwickler denken, sei eine Klasse durch ihren voll qualifizierten Namen (Package- und Klassennamen) identifiziert. Durch den Einsatz mehrerer Klassenlader wird er aber schnell eines besseren belehrt, denn seit der Java Version 1.2 wird, aus Sicherheitsgründen, eine Klasse zusätzlich durch dessen Klassenlader eindeutig identifiziert. Ohne der Beachtung des entsprechenden Klassenlader könnte die Integrität einer Anwendung untergraben werden, indem man der Anwendung fremde (manipulierte) Implementationen von Klassen als dessen eigene unterschiebt.

  Im JDownloader 2 führte dies zu fehlerhaftem Verhalten, da „plötzlich" Methoden wie *instanceof*, *equals* oder *isAssignableFrom* nicht mehr, wie gewohnt, funktionierten. Auch kam es zu Casting-Fehlern[22] innerhalb der Plug-ins.

  Daher war es nötig alle entsprechenden Stellen im Quellcode für den Umgang mit mehreren Klassenlader anzupassen. Zur Vermeidung der Casting-

[22] java.lang.ClassCastException

Fehler in den Plug-ins wurde eine zusätzlichen Verwaltung der verwendeten Klassenlader (siehe Anhang F) eingeführt.

- **Wieso läuft die synchronisierte Methode parallel?**

  Java stellt mittels dem Schlüsselwort *synchronized* eine einfache Möglichkeit zur exklusiven Synchronisierung von Abläufen zur Verfügung. Hierzu kann wahlweise auf die Klasse selbst[23] oder ein Objekt[24] synchronisiert werden.

  Diese Möglichkeit wurde auch von den Entwicklern der Plug-ins verwendet. Meist wurde dabei auf die Klasse selbst synchronisiert um Zugriffe über alle Instanzen des Plug-ins hinweg zu koordinieren. Da in JDownloader 2 jetzt aber mehrere Klassenlader zum Einsatz kommen und somit auch mehrere Klassen eines Plug-ins parallel existieren, funktioniert diese einfache Art der Synchronisierung nicht mehr.

  Deswegen war es nötig alle Plug-ins entsprechend so anzupassen, dass sie nicht mehr auf die Klasse selbst, sondern auf ein gemeinsames/geteiltes Objekt aller Plug-ins synchronisiert wird. Doch für die erfolgreiche Anpassung musste erst noch ein weiteres Problem gelöst werden, welches im nachfolgenden Punkt erläutert wird.

- **Warum zeigt die statische Referenz auf ein anderes Objekt?**

  Statische Referenzen auf gemeinsame Objekte werden in Plug-ins zum Austausch von Informationen zwischen mehreren Instanzen verwendet. In JDownloader 1 funktioniert dieses Vorgehen auch weiterhin ohne Probleme. Da sich aber der Klassenlader um alle statischen Referenzen und deren Zuweisung innerhalb der erzeugten Instanzen kümmert, führte dies nun zu Problemen in JDownloader 2. Um eine große, wohlmöglich fehleranfällige, Anpassung aller Plug-ins zu vermeiden und weiterhin abwärtskompatibel zu JDownloader 1 zu bleiben, musste also ein Ersatz für statische Referenzen gefunden werden.

  Um das zu erreichen, wurde die Methode zum Instanziieren von Plug-ins (siehe Anhang E) angepasst. Nachdem eine neue Instanz eines Plug-ins erzeugt wurde, werden per Reflexion (Shams & Edwards, 2013) alle statischen Referenzen innerhalb des Plug-ins durchlaufen und mit einem Zwischenspeicher abgeglichen. Danach besitzen alle erzeugten Instanzen dieselben statischen Referenzen auf gemeinsame Objekte und können diese wieder für den Austausch von Informationen verwenden.

  Dennoch waren Änderungen der Plug-ins unvermeidbar, da nicht alle Objekt-Typen hierfür möglich waren und deren Nutzung nun auch anders

[23]public static synchronized void do(){...

[24]...synchronized(Object){do...

erfolgen muss. Mit dem neuen Plug-in System von JDownloader 2 ist für statische Referenzen nur noch der Einsatz von „veränderlichen[25]“ Objekten erlaubt. „Unveränderliche[26]“ oder „primitive[27]“ Objekte können nicht mehr verwendet werden. Der Grund hierfür liegt darin, dass das Ändern einer statischen Referenz nicht in allen Instanzen sichtbar ist, da sie jeweils eigene Klassenlader haben. Eine Änderung innerhalb des referenzierten Objektes selbst bekommen jedoch alle Plug-in Instanzen mit, da sie weiterhin das gleiche Objekt statisch referenziert haben.

Anhand von Pseudocode soll diese Änderung veranschaulicht werden.

vorher:

```
public static int x=1;
public static Sring y="Test";

x=2;
y="Test"+x;
```

nachher:

```
public static AtomicInteger x=new AtomicInteger(1);
public static AtomicReference<String> y=new AtomicReference<String>("Test");

x.getAndSet(2);
y.set("Test"+x.get());
```

- **Woher kommt das „Speicherleck“?**

  Bevor es eine „saubere“ Verwaltung der Klassenlader (siehe Anhang F) gab, wurde der Einfachheit halber, der aktuell gültige Klassenlader im jeweiligen Prozessfaden (*Thread*) für die weitere Verwendung abgelegt[28]. Dennoch kam es, trotz sorgfältiger Verwendung der Prozessfäden, immer wieder zu „Speicherlecks“, bei denen ein Klassenlader und dessen geladene Klassen nicht mehr durch den Garbage Collector freigegeben wurde.

  Nach einer genaueren Analyse mittels VisualVM konnte die Ursache hierfür gefunden werden. Wird innerhalb eines Prozessfadens ein weiterer Prozessfaden erzeugt, so erbt dieser bestimmte Eigenschaften. Dazu gehören neben der Prozessgruppe eben auch der aktuell gesetzte Klassenlader. So konnten bestimmte Situationen, wie zum Beispiel das Entpacken eines Archives, langlebige Referenzen auf den Klassenlader erzeugen, die den Garbage Collector an der Freigabe des Speichers hinderten. Aber auch Referenzen auf instanziierte Plug-ins selbst konnten die Ursache sein, warum der Speicher nicht freigegeben wurde.

[25] mutable objects, wie *Map*, *List*, *Set*, *AtomicReference*, *AtomicBoolean...*

[26] immutable objects, wie *String*, *Integer*, *Boolean...*

[27] char,int,boolean...

[28] Thread.setContextClassLoader und Thread.getContextClassLoader

Damit das Entladen von Klassen und die Freigabe des Speichers durch den Garbage Collector funktionieren kann, sollten „starke Referenzen“ deshalb nur an solchen Stellen eingesetzt werden, wo sie zwingend erforderlich sind und garantiert auch wieder verworfen werden. In allen anderen Fällen sind „schwache Referenzen“, welche in Anhang H weiter erläutert werden, zu bevorzugen.

Insgesamt lässt sich aber feststellen, dass eine Umsetzung des neuen Plug-in Systems mittels mehrerer Klassenlader die richtige Entscheidung war. Die hieraus resultierenden Vorteile überwiegen bei weitem und rechtfertigen die nötige Zeit, sich mit dem Thema Klassenlader in Java näher auseinanderzusetzen.

2.4.3 Verbesserungen im Vergleich zu JDownloader 1

Bevor im nachfolgenden Abschnitt 2.5 das neue Plug-in System von JDownloader 2 anhand der in Abschnitt 2.3 genannten Anforderungen bewertet wird, soll zunächst ein direkter Vergleich beider Versionen einen Überblick über dessen Vorteile aufzeigen.

Hierzu wurden die zwei JDownloader Versionen, anhand frischer Installationen, in folgenden Punkten miteinander verglichen[29]:

- **Startzeit von JDownloader**

  Um die Startzeiten beider Versionen fair vergleichen zu können, muss dabei zwischen dem „kalten Zustand[30]“ und dem „warmen Zustand[31]“ unterschieden werden. Zum Löschen des Dateisystem-Caches von Windows dient dabei die Anwendung ATM[32].

  Im „kalten Zustand“ benötigt JDownloader 1 im Durchschnitt 20 Sekunden bis Befehle des Anwenders verarbeitet werden können. JDownloader 2 steht dem Anwender jedoch schon nach 7 Sekunden zur Verfügung. Für JDownloader 1 reduziert sich die Startzeit im „warmen Zustand“ auf zehn Sekunden, während JDownloader 2 nur noch vier Sekunden benötigt.

  Die Startzeit von JDownloader 2 konnte im Vergleich zu JDownloader 1 um mehr als die Hälfte verkürzt werden. Natürlich ist diese Verbesserung nicht alleine der Weiterentwicklung des Plug-in Systems zuzuschreiben, denn auch andere Komponenten des JDownloader wurden verbessert. Jedoch trägt das neue Plug-in System von JDownloader 2 maßgebend (siehe Anhang G) zur Reduktion der gefühlten Startzeit bei.

[29]Windows 7, 128 GB SSD, 8GB Ram und 2.8 GHz DualCore-CPU
[30]Keine Dateien von JDownloader befinden sich im Dateisystem-Cache von Windows
[31]Es befinden sich Dateien von JDownloader im Dateisystem-Cache von Windows
[32]`http://tmurgent.com/Tools/ATM/Default.aspx`

- **Speicherverbrauch von JDownloader**

  Der Taskmanager von Windows zeigt lediglich, wieviel Speicher der Java-Prozess derzeit belegt hat. Für genauere Aussagen oder für Vergleiche genügt diese Angabe jedoch nicht. Zur besseren Analyse des Speicherverbrauchs von JDownloader wird daher die Anwendung VisualVM[33] verwendet, die genauere Einblicke in den Speicher eines Java-Prozesses ermöglicht.

  Die Analyse mittels VisualVM (siehe Anhang I) zeigt, dass der Speicherverbrauch von JDownloder 2 um circa 30 % gegenüber JDownloader 1 reduziert werden konnte, obwohl dieser über mehr Plug-ins (Abbildung 4.15) verfügt. Während die Unterschiede des Heap-Speichers (Abbildung 4.13) primär der Weiterentwicklung weiterer Komponenten angerechnet werden kann, lässt sich die Reduktion des Perm-Gen-Speichers (Abbildung 4.14) direkt mit dem neuen Plug-in System in Verbindung (siehe Anhang E) bringen. Denn JDownloader 2 lädt ein Plug-in nur noch bei Bedarf, während das alte Plug-in System von JDownloader 1 alle Plug-ins bereits zu Beginn geladen hat.

- **Laden/Entladen von Plug-ins zur Laufzeit**

  Während zum Aktualisieren eines Plug-ins in JDownloader 1 mehrere Neustarts vonnöten sind, kann dies in JDownloader 2 nun zur Laufzeit geschehen. Neben dem Wegfall des sonst nötigen Neustarts, können Plug-ins nach Ihrer Verwendung und wenn sie nicht weiter gebraucht werden, auch wieder entladen werden.

  Anhand der VisualVM-Analyse (Abbildung 4.16) lässt sich das Laden und Entladen von Plug-ins demonstrieren. Zunächst ist kein Plug-in für den Anbieter Uploaded geladen (Abbildung 4.16.a). Nun fügt der Anwender dem JDownloader eine Datei des Anbieters hinzu (Abbildung 4.16.b), worauf dieser das entsprechende Plug-in lädt. Nachdem die Datei vom Anwender wieder aus dem JDownloader entfernt wurde, wird auch das Plug-in entladen (Abbildung 4.16.c) und dessen Speicher kann wieder freigegeben werden. Die in Abbildung 4.16.c noch immer geladene Klasse dient dem Datenaustausch zwischen mehreren Plug-ins und darf deswegen nicht entladen werden.

  Plug-ins belegen also nur noch Speicher, wenn sie auch in Verwendung sind. Somit ist der gesamte Speicherverbrauch von JDownloader 2 viel stärker vom Nutzungsverhalten des Anwenders abhängig und nicht mehr, wie bei JDownloader 1, von der Anzahl der Plug-ins.

[33] `http://visualvm.java.net/`

- **Unterstützung externer Plug-ins**

  Wie bereits in Abschnitt 2.3.2 unter dem Punkt „Einfache Integration von Plug-ins Dritter“ erläutert wurde, bietet JDownloader 1 keine brauchbare Unterstützung externer Plug-ins. Um dem Entwickler und dem Anwender hier eine saubere Lösung anzubieten, wurde genau für diesen Fall, wenn mehrere Plug-ins pro Anbieter zur Auswahl stehen, ein deterministisches und leicht verständliches Verhalten in den *HostPluginController* (siehe Abschnitt G.1) integriert. Für jeden Anbieter wird nun jeweils das Hoster-Plug-in (siehe Abschnitt D.2) mit der höchsten „interfaceVersion“ (siehe HostPlugin-Annotation[34]) ausgewählt. Ursprünglich sollte diese der Unterscheidung mehrerer Plug-in Schnittstellen dienen. Da es jedoch nie soweit kam und die Annotation wegen Abwärtskompatibilität zu JDownloader 1 nicht verändert werden konnte (siehe Abschnitt 2.3.1), wurde sie hierzu zweckentfremdet und dient seitdem der korrekten Auswahl des Plug-ins.

Allein anhand der oben genannten Punkte lässt sich sagen, dass das neue Plug-in System einen deutlichen Zugewinn für JDownloader 2 bedeutet. Der Anwender kann sich neben der beschleunigten Startzeit, ebenfalls über Aktualisierungen der Plug-ins zur Laufzeit erfreuen.

2.5 Bewertung des neuen Plug-in Systems

Im folgenden findet nun eine Bewertung des weiterentwickelten Plug-in System von JDownloader 2 anhand der in 2.3 definierten Anforderungen statt.

- **Abwärtskompatibilität der Plug-ins zu JDownloader 1**

  Trotz des mittlerweile stark veralteten Kerns von JDownloader 1 kann diese Version noch immer mit Aktualisierungen von Plug-ins versorgt werden. Da JDownloader 1 aber weder Fehlerbereinigungen, noch neue Funktionen erhält, wird es für bestimmte Plug-ins immer aufwendiger diese weiterhin kompatibel und lauffähig zu halten. So müssen Entwickler immer häufiger *try/catch* Konstruktionen verwenden oder Funktionen, welche bereits in JDownloader 2 korrigiert sind, umständlich nachbauen. Beispielsweise führen HTTP-POST Anfragen in JDownloader 1 ab Java Version 1.7 zu Problemen[35], weshalb Entwickler hier Umwege über HTTP-GET Anfragen versuchen müssen.

  Da bis heute die Plug-ins beider Versionen aus den gleichen Quellen übersetzt (kompiliert) werden, gilt diese Anforderung somit als erfüllt.

[34] jd.plugins.HostPlugin

[35] Aufgrund der falschen Entscheidung Klassen aus dem „sun“ Package zu verwenden

- **Erweiterbarkeit von Plug-ins und der Plug-in Schnittstelle**

  Seit der Veröffentlichung von JDownloader 1 erfuhren die Crawler- und Hoster-Plug-in Schnittstellen (siehe Anhang D) viele Änderungen und Neuerungen.

  So konnte die Unterstützung von sogenannten Multi-Hostern in die Hoster-Plug-in Schnittstelle integriert[36] werden, ohne dass es zu Problemen mit JDownloader 1 gekommen ist. Dies wird durch das Entfernen der *@Override* Annotation erreicht, da die entsprechende Methode in JDownloader 1 nicht vorhanden ist und es sonst zu Fehlern bei der Übersetzung der Plug-ins für JDownloader 1 gekommen wäre. Ebenso konnte der Ablauf von Crawler-Plug-ins beschleunigt[37] werden. Ein *try/catch* um die entsprechende Methode genügt um deren Vorteile in JDownloader 2 nutzen zu können und dennoch weiterhin kompatibel zu JDownloader 1 zu bleiben.

  Neben diesen zwei Beispielen gibt es noch viele weitere und somit gilt die Anforderung als erfüllt.

- **Reduktion der Ladezeit der Plug-ins**

  Neben der spürbar schnelleren Startzeit von JDownloader 2, lässt sich die Reduktion der Ladezeit der Plug-ins auch anhand einer Auswertung[38] der Log-Dateien beider Versionen bestätigen.

  JDownloader 2

  ```
  @HostPluginController: init took 1576ms for 3489
  @HostPluginController: init took 1641ms for 3489
  @HostPluginController: init took 1652ms for 3489
  ```

  JDownloader 1

  ```
  09.07.14 16:57:29-...->Try to load class jd.plugins.hoster.AaVg
  09.07.14 16:57:32-...->Try to load class jd.plugins.hoster.ZuzuFileCom
  09.07.14 17:02:03-...->Try to load class jd.plugins.hoster.AaVg
  09.07.14 17:02:07-...->Try to load class jd.plugins.hoster.ZuzuFileCom
  09.07.14 17:05:47-...->Try to load class jd.plugins.hoster.AaVg
  09.07.14 17:05:50-...->Try to load class jd.plugins.hoster.ZuzuFileCom
  ```

  Der Vorteil des neuen Plug-in Systems lässt sich auch mathematisch beschreiben. Die Ladezeit der Plug-ins in JDownloader 1 t_{jd1} lässt sich wie folgt definieren.

[36] http://svn.jdownloader.org/projects/jd/repository/revisions/16318
[37] http://svn.jdownloader.org/projects/jd/repository/revisions/14904
[38] siehe Punkt „Startzeit von JDownloader“ im Abschnitt 2.4.3

$$
\begin{aligned}
t_{jd1} &= exists_{jd1} \cdot t_{scan} + loaded_{jd1} \cdot t_{load} \\
\\
exists_{jd1} &= \text{Anzahl aller vorhandener Plug-ins} \\
loaded_{jd1} &= \text{Anzahl der geladenen Plug-ins mit } loaded_{jd1} \leq exists_{jd1} \\
t_{scan} &= \text{Zeit zum finden eines Plug-ins} \\
t_{load} &= \text{Zeit zum laden eines Plug-ins}
\end{aligned} \tag{2.1}
$$

Aufgrund der Bedingung $loaded_{jd1} \leq exists_{jd1}$ und der Tatsache, dass JDownloader 1 alle Plug-ins lädt, kann eine untere Grenze der Ladezeit $t_{jd1_{min}}$ definiert werden.

$$
\begin{aligned}
t_{jd1_{min}} &= exists_{jd1} \cdot (t_{scan} + t_{load}) \\
\\
exists_{jd1} &= \text{Anzahl aller vorhandener Plug-ins mit } exists_{jd1} \geq 0 \\
t_{scan} &= \text{Zeit zum Finden eines Plug-ins} \\
t_{load} &= \text{Zeit zum Laden eines Plug-ins}
\end{aligned} \tag{2.2}
$$

Für JDownloader 2 kann die Ladezeit t_{jd2} der Plug-ins wie folgt definiert werden.

$$
\begin{aligned}
t_{jd2} &= known_{jd2} \cdot t_{cache} + exists_{jd2} \cdot t_{scan} + loaded_{jd2} \cdot t_{load} \\
\\
known_{jd2} &= \text{Anzahl aller bekannter Lazy-Plug-ins mit } known_{jd2} \geq 0 \\
exists_{jd2} &= \text{Anzahl aller vorhandener Plug-ins} \\
loaded_{jd2} &= \text{Anzahl der geladenen Plug-ins mit } loaded_{jd2} \leq exists_{jd2} \\
t_{cache} &= \text{Zeit zum Laden eines Lazy-Plug-ins} \\
t_{scan} &= \text{Zeit zum Finden eines Plug-ins} \\
t_{load} &= \text{Zeit zum Laden eines Plug-ins}
\end{aligned} \tag{2.3}
$$

Falls noch keine Lazy-Plug-ins bekannt sind ($known_{jd2} = 0$) und die Anzahl der Plug-ins identisch ($exists_{jd1} = exists_{jd2}$) ist, ergeben sich auch keine Unterschiede in der Ladezeit.

$$
\begin{aligned}
t_{jd1_{min}} &= t_{jd2} \\
\\
exists_{jd1} \cdot (t_{scan} + t_{load}) &= known_{jd2} \cdot t_{cache} + exists_{jd2} \cdot t_{scan} + loaded_{jd2} \cdot t_{load} \\
&\quad \text{mit } known_{jd2} = 0 \text{ ergibt sich} \\
exists_{jd1} \cdot (t_{scan} + t_{load}) &= exists_{jd2} \cdot t_{scan} + loaded_{jd2} \cdot t_{load} \\
&\quad \text{aus } known_{jd2} = 0 \text{ folgt } loaded_{jd2} = exists_{jd2} \\
exists_{jd1} \cdot (t_{scan} + t_{load}) &= exists_{jd2} \cdot (t_{scan} + t_{load}) \\
&\quad \text{mit } exists_{jd1} = exists_{jd2} \text{ ergibt sich} \\
exists_{jd2} \cdot (t_{scan} + t_{load}) &= exists_{jd2} \cdot (t_{scan} + t_{load}) \\
&\quad \text{daraus folgt} \\
t_{jd1_{min}} &= t_{jd2}
\end{aligned} \tag{2.4}
$$

Falls nun alle Lazy-Plug-ins bekannt sind ($known_{jd2} = exists_{jd2}$), hängt die Ladezeit der Plug-ins von JDownloader 2 nur noch von der Anzahl der geänderten Plug-ins $loaded_{jd2}$ ab.

$$t_{jd2_{min}} = exists_{jd2} \cdot (t_{cache} + t_{scan}) + loaded_{jd2} \cdot t_{load} \tag{2.5}$$

$$\begin{aligned} t_{jd1_{min}} &> t_{jd2_{min}} \\ exists_{jd1} \cdot (t_{scan} + t_{load}) &> exists_{jd2} \cdot (t_{cache} + t_{scan}) + loaded_{jd2} \cdot t_{load} \\ &\text{mit } exists_{jd1} = exists_{jd2} \text{ ergibt sich} \\ exists_{jd2} \cdot t_{load} &> exists_{jd2} \cdot t_{cache} + loaded_{jd2} \cdot t_{load} \\ &\text{mit } t_{load} = t_{diff} + t_{cache} \text{ ergibt sich} \\ \frac{exists_{jd2} \cdot t_{diff}}{t_{load}} &> loaded_{jd2} \end{aligned} \tag{2.6}$$

Somit verringert sich für $0 \leq loaded_{jd2} < \frac{exists_{jd2} \cdot t_{diff}}{t_{load}}$ die Ladezeit der Plug-ins von JDownloader 2 gegenüber JDownloader 1. Die reduzierte Ladezeit wird primär durch $t_{cache} < t_{load}$ und dass $loaded_{jd2}$ im Vergleich zu $exists_{jd2}$ normalerweiße sehr klein ist, erreicht.

Auch diese Anforderung wird somit vom neuen Plug-in System erfüllt.

- **Reduktion des Speicherverbrauchs zur Laufzeit**

  Anhand der Abbildung 4.14 der VisualVM-Analyse für den Punkt „Speicherverbrauch von JDownloader“ im Abschnitt 2.4.3 ist zu sehen, dass JDownloader 2 bereits circa 30 % weniger PermGen-Speicher als JDownloader 1 benötigt. Ziel der Anforderung ist jedoch nicht nur ein geringerer Speicherverbrauch gegenüber JDownloader 1, sondern den Verbrauch auch während der gesamten Laufzeit gering zu halten. Aus diesem Grund ist es nötig, das der Speicher von nicht verwendeten Plug-ins wieder freigegeben werden kann.

  Die VisualVM-Analyse in Abbildung 4.17 aus dem Abschnitt J bestätigt, dass JDownloader 2 dazu in der Lage ist. Die Analyse zeigt einen JDownloader 2, der ohne bekannte Lazy-Plug-ins (siehe Anhang G) gestartet wird. Zu Beginn müssen deshalb alle gefundenen Plug-ins neu geladen werden. In der Abbildung ist dies am Anstieg der Klassenanzahl und des Speicherverbrauchs zu erkennen. Im Anschluss daran werden deren Lazy-Plug-in Pendants auf der Festplatte gespeichert und die geladenen Plug-ins wieder freigegeben. Wie ebenfalls in der Abbildung zu erkennen ist, hat sich neben der Klassenanzahl auch der Speicherverbrauch wieder reduziert, was beweist, dass die Plug-ins freigegeben wurden. JDownloader 1 gibt während der gesamten Laufzeit keine Plug-ins mehr frei.

Auch mathematisch lässt sich der geringere Speicherverbrauch von JDownloader 2 beweisen. Der Speicherverbrauch der Plug-ins in JDownloader 1 mem_{jd1} lässt sich wie folgt definieren.

$$\begin{aligned} mem_{jd1} &= loaded_{jd1} \cdot mem_{load} + used_{jd1} \cdot mem_{use} \\ \\ loaded_{jd1} &= \text{Anzahl aller \underline{jemals} geladener Plug-ins} \\ used_{jd1} &= \text{Anzahl aller \underline{jemals} verwendeter Plug-ins mit } used_{jd1} \leq loaded_{jd1} \\ mem_{load} &= \text{Speicherverbrauch zum Laden eines Plug-ins} \\ mem_{use} &= \text{Speicherverbrauch um ein Plug-in zu verwenden} \end{aligned} \tag{2.7}$$

Aufgrund der Bedingung $used_{jd1} \leq loaded_{jd1}$ und der Tatsache, dass JDownloader 1 zum Start alle Plug-ins lädt, kann eine untere Grenze des Speicherverbrauchs $mem_{jd1_{min}}$ definiert werden.

$$\begin{aligned} mem_{jd1_{min}} &= loaded_{jd1} \cdot mem_{load} \\ \\ loaded_{jd1} &= \text{Anzahl aller Plug-ins (konstant) mit } loaded_{jd1} > 0 \\ mem_{load} &= \text{Speicherverbrauch zum Laden eines Plug-ins} \end{aligned} \tag{2.8}$$

Der Speicherverbrauch der Plug-ins in JDownloader 2 mem_{jd2} lässt sich wie folgt definieren.

$$\begin{aligned} mem_{jd2} &= used_{jd2} \cdot (mem_{load} + mem_{use}) \\ \\ used_{jd2} &= \text{Anzahl aller \underline{derzeit} verwendeter Plug-ins mit } used_{jd2} \geq 0 \\ mem_{load} &= \text{Speicherverbrauch zum Laden eines Plug-ins} \\ mem_{use} &= \text{Speicherverbrauch um ein Plug-in zu verwenden} \end{aligned} \tag{2.9}$$

Anhand dieser Definitionen lässt sich zeigen, dass JDownloader 2 bereits zum Start weniger Speicher benötigt.

$$\begin{aligned} mem_{jd2} &\leq mem_{jd1_{min}} \\ \\ used_{jd2} \cdot (mem_{load} + mem_{use}) &\leq loaded_{jd1} \cdot mem_{load} \\ &\text{mit } used_{jd2} = 0 \text{ ergibt sich} \\ 0 &\leq loaded_{jd1} \cdot mem_{load} \\ &\text{wegen } loaded_{jd1} > 0 \text{ folgt} \\ mem_{jd2} &< mem_{jd1_{min}} \end{aligned} \tag{2.10}$$

Solange $used_{jd2} \leq used_{jd1}$ und $used_{jd2} < loaded_{jd1}$ gilt, wovon man normalerweiß ausgehen darf, benötigt JDownloader 2 auch während der Laufzeit weniger Speicher.

$$
\begin{aligned}
mem_{jd2} &\leq mem_{jd1} \\
\\
used_{jd2} \cdot (mem_{load} + mem_{use}) &\leq loaded_{jd1} \cdot mem_{load} + used_{jd1} \cdot mem_{use} \\
&used_{jd1} = used_{jd2} + y \\
&loaded_{jd1} = used_{jd2} + x \\
used_{jd2} \cdot (mem_{load} + mem_{use}) &\leq used_{jd2} \cdot (mem_{load} + mem_{use}) + x \cdot mem_{load} + y \cdot mem_{use} \\
0 &\leq x \cdot mem_{load} + y \cdot mem_{use} \\
&\text{aus } used_{jd2} < loaded_{jd1} \text{ und } loaded_{jd1} > 0 \text{ folgt } x > 0 \\
&\text{aus } used_{jd2} \leq used_{jd1} \text{ folgt } y \geq 0 \\
&\text{daraus folgt} \\
mem_{jd2} &< mem_{jd1}
\end{aligned}
\tag{2.11}
$$

Die Anforderung ist daher erfüllt.

- **Aktualisierung einzelner Plug-ins zur Laufzeit**

  Der Punkt „Laden/Entladen von Plug-ins zur Laufzeit“ aus dem Abschnitt 2.4.3 bestätigt, dass das Entladen eines Plug-in in JDownloader 2 möglich ist. Zusammen mit dem *HostPluginController* aus Anhang G.1, welcher sich um den Austausch von Plug-ins kümmert, ist somit eine Aktualisierung von Plug-ins zur Laufzeit möglich. Mittels der Kombination aus kontinuierlicher Integration und Auslieferung in JDownloader 2 findet diese Möglichkeit auch bei den Entwicklern großen Anklang. Sie können den Anwendern nun innerhalb weniger Minuten neue oder fehlerbereinigte Plug-ins zur Verfügung stellen, ohne dass diese hierzu JDownloader neustarten müssen.

  Von der Möglichkeit Plug-ins zur Laufzeit zu aktualisieren machen Entwickler bereits regen Gebrauch. Eine Auswertung aller Plug-in Aktualisierungen vom März 2014 (siehe Abbildung 4.3) zeigt, dass JDownloader täglich mit Aktualisierungen versorgt wird. Die Anforderung ist somit erfüllt.

- **Einfache Integration von Plug-ins Dritter**

  Noch gibt es keine offiziellen Listen von Plug-ins von Dritten, doch sind derartige Plug-ins bekannt und bereits im Umlauf. So gibt es zum Beispiel Entwickler, die diese Möglichkeit nutzen, um Plug-ins an ihre eigenen Bedürfnisse anzupassen. Auch findet man in einschlägigen Foren modifizierte Plug-ins, die Änderungen beinhalten, welche nicht in die offizielle Version von JDownloader aufgenommen werden können.

  Auch wenn die einfache Integration von Plug-ins von Dritten noch nicht groß beworben wird, so ist sie bereits jetzt schon möglich, womit die Anforderung als erfüllt gilt.

- **Geringe oder keine Änderungen an vorhandenen Plug-ins nötig**

  Zusätzlich zu den Änderungen, welche sich aus den funktionalen Anforderung von 2.3.1 ergaben, waren außerdem noch die Anpassungen aus dem Abschnitt 2.4.2 notwendig. So musste zum Beispiel in allen Plug-ins der Zugriff auf statische Referenzen[39] dem neuen Plug-in System von JDownloader 2 angepasst werden.

  Der Aufwand pro Plug-in hielt sich dennoch in Grenzen. Nach einer kurzen Umgewöhungsphase beziehungsweise Einarbeitungszeit hatten sich die Entwickler an das neue Plug-in System angepasst und die Änderungen stellten sie vor keine weiteren Probleme. Somit wurde diese Anforderung ebenfalls erfolgreich umgesetzt.

2.6 Zusammenfassung und Ausblick

In seiner Gesamtheit betrachtet ist das neue Plug-in System von JDownloader 2 durchwegs positiv. Alle an das Plug-in System gestellten Anforderungen aus Abschnitt 2.3 konnten, wie in 2.5 aufgezeigt, erfüllt werden. Vor allem das schnellere Laden der Plug-ins, deren Austausch zur Laufzeit und der reduzierte Speicherverbrauch, bieten dem Anwender einen deutlichen Zugewinn gegenüber dem veralteten Plug-in System von JDownloader 1.

Die Entscheidung, dass alte Plug-in System von JDownloader 1 stetig weiterzuentwickeln und den neuen Anforderungen entspreched anzupassen und nicht auf ein bereits existierendes Plug-in System zurückzugreifen, kann ich rückblickend als richtig einstufen. Trotz oder vielleicht gerade wegen der in Abschnitt 2.4.2 beschriebenen Hindernisse, waren die Zugewinne an Wissen und Erfahrungen über Klassenlader, Referenzen und dem Garbage Collector meine Hauptgründe an der Eigenentwicklung weiter festzuhalten. Das erlangte Wissen half nicht nur bei der Entwicklung des Plug-in Systems, sondern kam auch in weiteren Komponenten des JDownloader zum Einsatz, wo es ebenfalls zur Reduzierung des Speicherverbrauchs beitragen konnte.

Auch wenn das neue Plug-in System alle Anforderungen erfüllt, bestehen trotzdem noch Möglichkeiten, dieses weiter zu verbessern. Beispielsweise hängt die Ladezeit der Plug-ins, wie in Abschnitt 2.5 unter dem Punkt „Reduktion der Ladezeit der Plug-ins“ erläutert wurde, maßgebend von den beiden Variablen t_{cache} und t_{scan} ab. Derzeit wird noch JSON (Crockford, 2006) zum Speichern und Laden der Lazy-Plug-ins verwendet. Eine entsprechende Auswertung[40] der Log-Datei zeigt, dass dieser Abschnitt fast die Hälfte der Zeit, wie das Auffinden

[39] `http://svn.jdownloader.org/projects/jd/repository/revisions/18484`

[40] @HostPluginController: „cache took 255ms“ im Vergleich zu „scan took 666ms“

aller Plug-ins, benötigt. Der Einsatz einer effektiveren Form der Serialisierung (Riehle, Siberski, Bäumer, Megert & Züllighoven, 1997) für die Lazy-Plug-ins, könnte diesen Abschnitt nochmals deutlich beschleunigen und damit t_{cache} entsprechend noch weiter verringern. Zusätzlich ließe sich t_{scan} verringern. So könnte man alle offiziellen Plug-ins in einem Archiv (JAR-Datei) zusammenfassen und speichern, statt wie bisher in einzelnen Dateien auf der Festplatte. Der Zugriff auf die Metainformationen einzelner Plug-ins kann in Archiven noch stärker optimiert werden, womit sich t_{scan} von JDownloader 2 gegenüber JDownloader 1 noch weiter beschleunigen ließe. Während das Laden und Speichern der Lazy-Plug-ins jederzeit verändert werden kann, bedarf es für Veränderungen an den Plug-ins zusätzlicher Anpassungen an der Klasse *PluginClassLoaderChild* (siehe Anhang F). Auch muss das Update-System von JDownloader 2 entsprechend angepasst werden, um auch weiterhin Plug-ins zur Laufzeit aktualisieren zu können.

Neben den bereits erwähnten Verbesserungsmöglichkeiten werden in das Plug-in System von JDownloader 2 sicherlich noch viele weitere Entwicklungen und Erfahrungen einfließen, damit es auch für zukünftige Anforderungen gerüstet sein wird.

3 Ausarbeitung der Forschung

3.1 Vergleich mit ähnlichen Plug-in Systemen

In diesem Abschnitt wird gepüft, wie das neue Plug-in System von JDownloader 2 im Vergleich seiner Anforderungen aus 2.3 zu anderen Plug-in Systemen abschneidet. Da es in seiner Art jenem Plug-in System ähnelt, wie es in Abschnitt 2.2.1 vorgestellt wurde, dienen die Plug-in Systeme JSPF, JPF und PF4J als Vergleichskandidaten.

Bevor nun in 3.1.2 die genannten Plug-in Systeme mit dem aus JDownloader 2 verglichen werden, sind vorab noch weitere Anforderungen zu definieren, welche sich durch einen möglichen Einsatz ihrerseits in JDownloader ergeben könnten.

3.1.1 Weitere nicht-funktionale Anforderungen

Anhand der Vergleichskandidaten JSPF, JPF und PF4J können noch weitere nicht-funktionale Anforderungen definiert werden, die bei einem Einsatz für JDownloader 2 von Relevanz sein könnten.

- **Unterstützung von Abhängigkeiten (Funktionalität, Fehlertoleranz)**

  Plug-ins in JDownloader sollten unter normalen Umständen nur auf Methoden und Komponenten von JDownloader selbst zugreifen. Aufgrund der Tatsache das die veralte Version von JDownloader 1 lediglich Aktualisierung der Plug-ins erfährt, jedoch keine Fehlerkorrekturen oder neue Funktionen, werden bestimmte Methoden und Komponenten in Plug-ins ausgelagert, damit sie in beiden JDownloader Versionen verwendet werden können. Dazu gehören zum Beispiel Klassen für die Unterstützung verschiedener Arten von CAPTCHAs, welche in Plug-ins ausgelagert[1] wurden.

[1] zum Beispiel Recaptcha in jd.plugins.hoster.DirectHTTP oder KeyCaptcha in jd.plugins.decrypter.LnkCrptWs

Das Plug-in System sollte einem Plug-in eine solche Nutzung weiterer Plug-ins ermöglichen.

- **Dokumentation des Plug-in Systems (Benutzbarkeit, Erlernbarkeit)**

  Eine gute und ausführliche Dokumentation erleichtert nicht nur neuen Entwicklern den Einstieg in die Plug-in Entwicklung, sondern unterstützt ihn auch im Fehlerfall bei der Fehlersuche und dessen Lösung. Auch ist es für all diejenigen hilfreich, die sich einen Überblick über die Funktionsweise von JDownloader und seiner Plug-ins verschaffen wollen.

  Gerade für JDownloader, mit seinen vielen Plug-ins und deren häufigen Aktualisierungen, ist es besonders wichtig, dass neue Entwickler soviel Unterstützung wie möglich bei der Plug-in Entwicklung und Wartung erhalten. Das Plug-in System sollte verständlich und einfach überschaubar bleiben.

- **Entwicklungsaufwand/Komplexität eines Plug-ins (Wartbarkeit, Erlernbarkeit)**

  Der Entwicklungs- und Wartungsaufwand für ein Plug-in sollte überschaubar sein. Damit ist gemeint, dass ein Entwickler möglichst schnell einen Überblick über die Funktionsweise eines Plug-ins erlangen können sollte und nicht erst komplexe Zusammenhänge innerhalb eines Plug-ins oder mehrerer Klassen/Dateien zu verstehen lernen muss. Je gebündelter alle Bestandteile eines Plug-ins sind, desto weniger Zeit bedarf es für eine Neuentwicklung oder der Wartung. Ein weiterer Vorteil hieraus ist, dass sich die einzelnen Änderungen eines Plug-ins einfacher nachvollziehen lassen, wenn sie nicht über mehrere Klassen hinweg verstreut sind.

  Bei der Vielzahl an Plug-ins in JDownloader ist es wichtig, dass das Plug-in System den Entwickler durch seine Einfachheit unterstützt und nicht vor seiner Komplexität zurückschrecken lässt. Je mehr Zeit ein Entwickler in die eigentliche Funktionalität investieren kann, desto besser.

3.1.2 Anforderungsbewertung

Um die Plug-in Systeme nun miteinander zu vergleichen, werden sie alle anhand der nicht-funktionalen Anforderungen aus den Abschnitten 2.3 und 3.1.1 bewertet. Zur besseren Vergleichbarkeit der Plug-in Systeme, erfolgt die Auswertung tabellarisch. Bemerkungen zu den jeweiligen Wertungen können dem Abschnitt 3.1.3 entnommen werden. Für die Bewertung der einzelnen Anforderungen kommt ein einfaches Punkte-System zum Einsatz, bei dem $-1/0/+1$ jeweils für „nicht erfüllt“,“neutral“ oder „erfüllt“ steht.

Anforderung	JD2	JSPF	JPF	PF4J
Reduktion der Ladezeit der Plug-ins	+1	+1	+1	0
Reduktion des Speicherverbrauchs zur Laufzeit	+1	-1	0	0
Aktualisierung einzelner Plug-ins zur Laufzeit	+1	-1	0	0
Einfache Integration von Plug-ins Dritter	+1	+1	+1	+1
Geringe/keine Änderungen an vorhandenen Plug-ins nötig	+1	-1	-1	-1
Unterstützung von Abhängigkeiten	0	+1	+1	+1
Dokumentation des Plug-in Systems	0	+1	+1	0
Entwicklungsaufwand/Komplexität eines Plug-ins	+1	+1	0	0
Gesamtbewertung	**6**	**2**	**3**	**1**

Die Auswertung zeigt, dass das neue Plug-in System für JDownloader 2 die geforderten Anforderung noch immer am besten erfüllen kann. Dies ist aber nicht weiter verwunderlich, wurde es doch genau den Anforderungen entsprechend entwickelt. JSPF bietet eine solide Ausgangsbasis für die Entwicklung eines Plug-in Systems, wie es in JDownloader 2 zum Einsatz kommen könnte. JSPF, JPF und PF4J „leiden“ aber unter den selben Problemen, wie sie schon in Abschnitt 2.4.2 bei JDownloader 2 beschrieben wurden. Die nötigen Änderungen an den Plug-ins würde die Abwärtskompatibilität zu JDownloader 1 zunichte machen. Ebenso ist eine Aktualisierung der Plug-ins zur Laufzeit mit ihnen nicht möglich.

Insgesamt lässt sich feststellen, dass das neu entwickelte Plug-in System explizit auf den Anwendungsfall JDownloader 2 zugeschnitten ist, während JSPF, JPF und PF4J für allgemein taugliche Anwendungen gedacht sind.

3.1.3 Bemerkungen

- **Reduktion der Ladezeit der Plug-ins**

  JSPF kann die Ladezeit reduzieren, indem ein optionaler Cache aktiviert wird. Durch den Cache wird ein erneutes Auffinden, Laden und Auswerten der Annotationen der Plug-ins verhindert. Dieser bedient sich zusätzlicher Manifeste der Plug-ins, wie Dateigröße oder MD5-Hash von Dateien. JSPF ist mit **+1** bewertet, da explizit ein Cache integriert wurde, um die Ladezeit zu verringern.

  JPF nutzt zur Verwaltung der Plug-ins Manifeste, die neben den eigentlichen Plug-ins verfügbar sind. Sie enthalten Informationen über das Plug-in, dessen Abhängigkeiten und alles Sonstige, welche für die Verwaltung benötigt werden. In Kombination mit dem verzögerten Laden der Plug-ins[2] und dem Aktivieren/Deaktivieren des Plug-ins zur Laufzeit, ist JPF ebenfalls in der Lage, die Ladezeit zu reduzieren. Deshalb wird es mit **+1** bewertet.

  Auch PF4J nutzt für die Verwaltung ebenfalls Manifeste. Nicht verwendete Plug-ins stehen somit zwar zur Verfügung, werden aber erst bei Bedarf geladen. PF4J ist lediglich mit **0** bewertet, da keine zusätzlichen Maßnahmen ergriffen wurden.

- **Reduktion des Speicherverbrauchs zur Laufzeit**

  JSPF ist nicht in der Lage ein bereits geladenes oder verwendetes Plug-in wieder freizugeben und wird deshalb mit **-1** bewertet.

  Auch JPF und PF4J können unter normalen Umständen ein solches Plug-in nicht mehr freigeben. Im Gegensatz zu JSPF unterstützen sie aber das Entfernen von Plug-ins, worauf ein erneutes Laden des Plug-ins möglich wäre. Bis zur nächsten Verwendung des Plug-ins kann somit der Speicherverbrauch reduziert werden. Da die Plug-ins jedoch erst aus dem Plug-in System entfernt werden müssen um ihren Speicher freizugeben, erhalten JPF und PF4J lediglich eine Wertung von **0**.

- **Aktualisierung einzelner Plug-ins zur Laufzeit**

  JSPF ermöglicht keine Aktualisierung von Plug-ins zur Laufzeit und erhält deswegen eine Wertung von **-1**. Dies liegt an Abläufen und Abhängigkeiten innerhalb von JSPF sowie den möglichen Abhängigkeiten zu weiteren Plug-ins, welche nicht einfach aufgelöst oder ausgetauscht werden können.

  JFP unterstützt das Laden und Entladen von Plug-ins und somit auch die

[2] Plug-in wird erst beim ersten Gebrauch geladen

Aktualisierung zur Laufzeit. Da hierfür das Plug-in jedoch erst entfernt werden muss, bevor im Anschluss die aktualisierte Version geladen werden kann, erhält es nur eine Bewertung von **0**.

Auch PF4J unterstützt das Laden und Entladen von Plug-ins und somit auch die Aktualisierung zur Laufzeit. Aus den gleichen Gründen wie bei JPF, wird es ebenfalls lediglich mit **0** bewertet.

- **Einfache Integration von Plug-ins Dritter**

  Alle verglichenen Systeme können einfach mit Plug-ins von Dritten erweitert werden und erhalten jeweils eine Wertung von **+1**.

- **Geringe/keine Änderungen an vorhandenen Plug-ins nötig**

  JSPF, JFP und PF4J haben dieselben Probleme mit mehreren Klassenlader, wie sie auch bei der Weiterentwicklung für JDownloader 2 vorgekommen sind. Ebenso wären größere Umbauten oder die Einführung weiterer Annotationen in die Plug-ins nötig, was aufgrund der Abwärtskompatibilität zu JDownloader 1 nicht möglich ist. Die Plug-in Systeme können nicht einfach für JDownloader 2 eingesetzt werden. Deshalb sind sie jeweils mit **-1** bewertet worden.

- **Unterstützung von Abhängigkeiten**

  Das Plug-in System von JDownloader 2 erlaubt zwar die Nutzung weiterer Plug-ins, bietet hierfür aber keinerlei Verwaltung an. So werden fehlende Abhängigkeiten nicht sauber aufgelöst oder kontrolliert und würden einfach zu Fehlern in den jeweilige Plug-ins führen. Da zumindest eine Nutzung weiterer Plug-ins möglich ist, wird es mit **0** bewertet,

  JSPF, JFP und PF4J bieten eine saubere Unterstützung von Abhängigkeiten an, weshalb sie jeweils mit **+1** bewertet sind.

- **Dokumentation des Plug-in Systems**

  JDownloader stellt dem Entwickler keine wirkliche Dokumentation zur Seite. Lediglich Kommentare innerhalb des Quellcodes sind vorhanden. So muss er sich in den Quellcode „einlesen“ oder weitere Entwickler befragen, um Funktionsweisen und Zusammenhänge zu erlernen und zu verstehen. Der Vorteil an dem Plug-in System von JDownloader 2 jedoch ist, dass es explizit für JDownloader entwickelt wurde. Das heißt, dass ein Entwickler kaum Wissen über das Plug-in System selbst oder dessen Zusammenhänge benötigt, um Plug-ins entwickeln oder warten zu können. Und da es eine große Anzahl bereits existierender Plug-ins gibt, die dem Entwickler als Vorlage und zum besseren Verständnis dienen können, wird das Plug-in System trotz fehlender Dokumentation mit **0** berwertet.

JSPF und JPF stellen dem Entwickler gute Dokumentationen und Diagramme zur Verfügung, die den Einstieg in die jeweiligen Plug-in Systeme deutlich erleichtern. Deshalb sind beide mit **+1** bewertet worden.

PF4J erklärt lediglich das Nötigste, um mit der Entwicklung zu beginnen, weshalb es mit **0** bewertet wurde.

- **Entwicklungsaufwand/Komplexität eines Plug-ins**

  JSPF verwendet, wie JDownloader auch, Annotationen um dem Plug-in System Informationen über die jeweiligen Plug-ins bereitzustellen. Der Entwickler kann das Plug-in System einfach verwenden und behält weiterhin einen guten Überblick über die Funktionsweise und die Komponenten des Plug-ins. Aus diesem Grund erhält JSPF eine Wertung von **+1**.

  JPF und PF4J arbeiten mit Manifesten für die jeweiligen Plug-ins. Diese müssen zusätzlich vom Entwickler erstellt und gepflegt werden. Auch ist der Entwicklungsaufwand etwas komplexer, da der Entwickler viel mehr mit dem Plug-in System in Kontakt kommt. Aus diesen Gründen sind JPF und PF4J jeweils mit **0** bewertet worden.

4 Anhänge

Anhang A Plug-ins im Jahr 2013

Die Abbildung 4.1 zeigt die absolute Anzahl aller Plug-ins in JDownloader 2 über das gesamte Jahr 2013. Sie wurde anhand einer Auswertung der Log-Dateien des Update-Servers von JDownloader 2 erstellt. Die Anzahl der Plug-ins in JDownloader 2 stieg im Verlauf des Jahres 2013 um circa zehn Prozent.

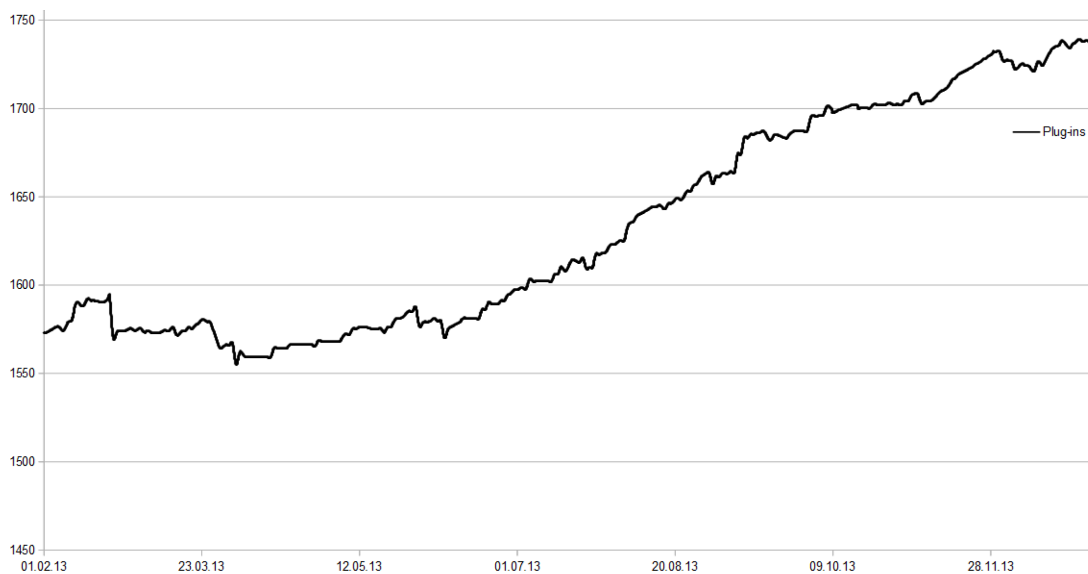


Abbildung 4.1: Anzahl aller Plug-ins in JDownloader 2 im Jahr 2013

Anhang B Aktualisierungen im Jahr 2013

Die Abbildung 4.2 zeigt anhand einer Auswertung der Log-Dateien aus dem Jahr 2013 des Update-Servers von JDownloader 2, wie häufig pro Tag Aktualisierungen für die Plug-ins zur Verfügung gestellt wurden. Im Mittel (arithmetisch) erhält der Anwender pro Tag fast fünf Aktualisierungen.

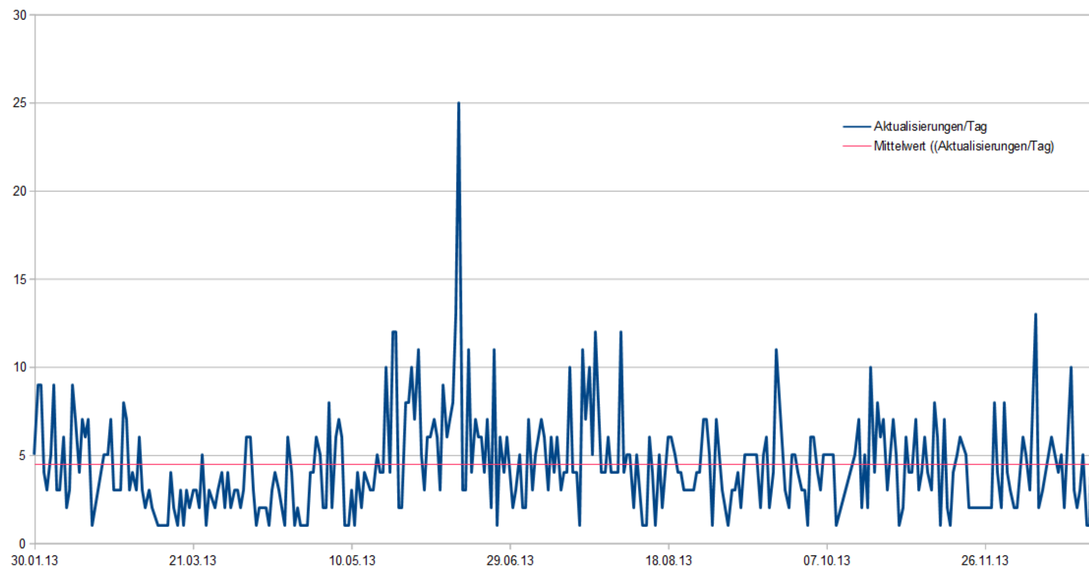


Abbildung 4.2: Aktualisierungen pro Tag im Jahr 2013

Anhang C Plug-in Updates im März 2014

Die gezeigte Abbildung 4.3 stellt eine Auswertung der Log-Dateien des Update-Servers von JDownloader 2 dar. Hierzu wurden alle neuen, geänderten oder entfernten Plug-ins für den jeweiligen Tag aufsummiert.

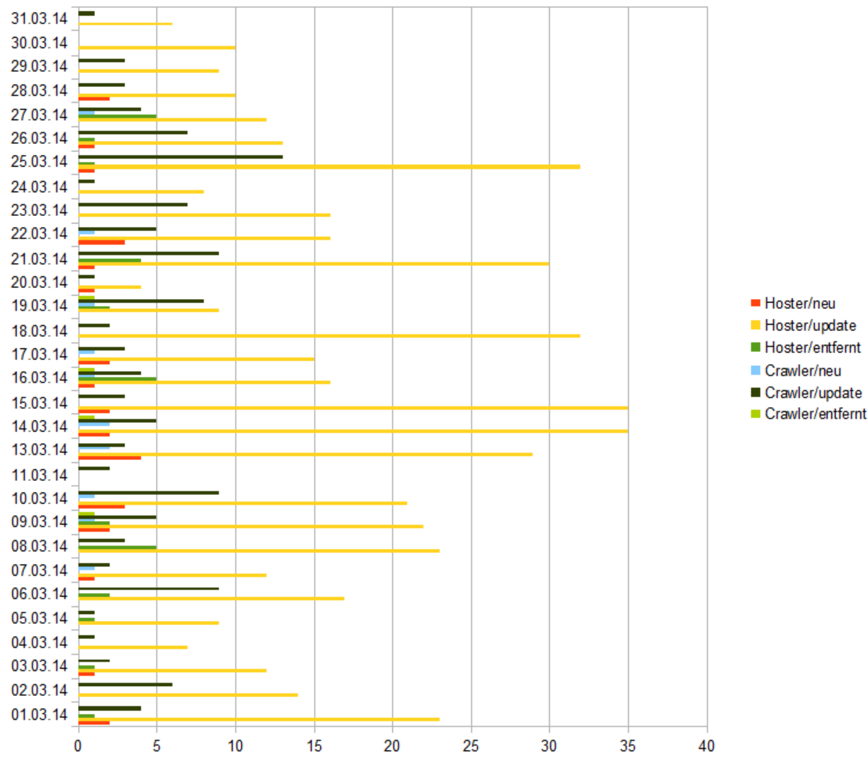


Abbildung 4.3: neue, geänderte oder entfernte Plug-ins pro Tag

Anhang D Plug-ins in JDownloader

Die Funktionalität zum Auffinden und Herunterladen von Dateien und Inhalten verschiedenster Anbieter wurde in JDownloader in zwei unterschiedliche Plug-in Arten ausgelagert. Für jede Plug-in Art existiert ein vorgefertigtes Grundgerüst in Form einer abstrakten Klasse. Diese enthalten bereits alle nötigen Basisfunktionen, wie zum Beispiel die Abfrage von Passwörtern vom Anwender oder der Eingabe eines CAPTCHA zur Verifikation für den Zugriff auf die Dateien und Inhalte eines Anbieters. Dies dient der Minimierung des Entwicklungsaufwandes neuer Plug-ins und der Vermeidung von Code-Duplizierung.

Der Entwickler muss das Plug-in somit lediglich um jene Teile erweitern, welche spezifisch für einen bestimmten Anbieter sind. In vielen Fällen können dabei bereits existierende Plug-ins als Vorlage dienen oder sie ähnlen sich in ihrem Ablauf, was den Entwicklungsaufwand nochmals deutlich reduziert. Das versetzt auch Java-Neulinge in die Lage, ohne viel Wissen über Internas von JDownloader, weitere Plug-ins zu entwickeln oder vorhandene zu pflegen. Einer der Hauptentwickler aus der Gemeinschaft, der sich mittlerweile ganz der Pflege und Entwicklung von Plug-ins verschrieben hat, besaß vor seiner Arbeit bei JDownloader keinerlei Kentnisse über Java.

D.1 Crawler-Plug-in

Crawler-Plug-ins (Sammler-Plug-ins) stellen Implementationen der abstrakten Klasse *PluginForDecrypt*[1]“ dar. Diese Art von Plug-ins kümmern sich um alle Schritte, welche für das Auffinden von Dateien und Inhalten der Anbieter nötig sind.

Die Komplexität eines Crawler-Plug-ins hängt dabei sehr vom jeweiligen Anbieter ab. Einfach gestrickte Plug-ins durchsuchen lediglich den Anbieter und leiten dessen gefundenen Dateien und Inhalte an den JDownloader weiter. Komplexere Plug-ins können dem Anwender auch eine Auswahlmöglichkeit bieten oder arbeiten sich selbstständig durch verzweigte Strukturen seitens des Anbieters. Crawler-Plug-ins helfen dem Anwender beim Auffinden und Zusammensuchen der gewünschten Dateien und Inhalte, die dafür nötige Zeit einzusparen und den Aufwand zu minimieren.

Im Idealfall sollte dabei der gesamte Ablauf so automatisiert wie möglich stattfinden.

[1]jd.plugins.PluginForDecrypt

Ein Beispiel für ein Crawler-Plugin ist das recht einfach gehaltene *TinyCc*. Es implementiert den Anbieter Tiny und löst dessen gekürzte URLs[2] auf und leidet die ursprüngliche URL an JDownloader weiter.

```
@DecrypterPlugin(revision = "$Revision: -1 $", interfaceVersion = 2,
 names = { "tiny.cc" },
 urls = { "http://(www\\.)?tiny\\.cc/[0-9a-zA-Z]+" },
 flags = { 0 })
public class TinyCc extends PluginForDecrypt {

    public TinyCc(PluginWrapper wrapper) {
        super(wrapper);
    }

    private String INVALIDLINKS = "http://(www\\.)?tiny\\.cc/(url|example|help|tools|qrcodes|api|contact|terms|domains)";

    public ArrayList<DownloadLink> decryptIt(CryptedLink param, ProgressController progress) throws Exception {
        ArrayList<DownloadLink> decryptedLinks = new ArrayList<DownloadLink>();
        final String parameter = param.toString();
        if (parameter.matches(INVALIDLINKS)) {
            logger.info("Link invalid: " + parameter);
            return decryptedLinks;
        }
        br.setFollowRedirects(false);
        br.getPage(parameter);
        if (br.containsHTML("Sorry, we weren't able to locate that URL")) {
            logger.info("Link offline: " + parameter);
            return decryptedLinks;
        }
        final String finallink = br.getRedirectLocation();
        if (finallink == null) {
            logger.warning("Decrypter broken for link: " + parameter);
            return null;
        }
        decryptedLinks.add(createDownloadlink(finallink));
        return decryptedLinks;
    }
}
```

D.2 Hoster-Plug-in

Hoster-Plug-ins (Anbieter-Plug-ins) stellen Implementationen der abstrakten Klasse *PluginForHost*[3]" dar. Sie bieten alle nötigen Funktionen, damit JDownloader einen Anbieter unterstützen und dessen Dateien und Inhalte für den Anwender herunterladen kann.

Verglichen mit den meisten Crawler-Plug-ins, ist die Entwicklung von Hoster-Plug-ins jedoch aufwändiger und fehleranfälliger. Dies kann unterschiedliche Gründe haben. Zum Beispiel kann die Entwicklung des Plug-ins durch mangelhafte oder fehlende Dokumentation erschwert sein. Auch können regionale Unterschiede zu unterschiedlichen Laufzeitverhalten des Plug-ins bei Entwickler und Anwender führen. Die meiste Entwicklungszeit fließt jedoch oftmals in die korrekte Erkennung und Verarbeitung möglicher Fehlermeldungen seitens des Anbieters, da es viel Zeit und Aufwand bedarf, diese zu finden oder zu verursachen.

Hoster-Plug-ins sollten in der Lage sein, den Anbieter in jeder Situation korrekt zu unterstützen, denn nur so ist ein fehlerfreies und automatisiertes Herunterladen von dessen Dateien und Inhalten möglich.

[2]Beispiel `http://tiny.cc/f5rxix`

[3]jd.plugins.PluginForHost

Zum Beispiel kann JDownloader, anhand des unten aufgezeigten Hoster-Plug-ins, bestimmte URLs von dem Anbieter *GitHub*[4] herunterzuladen.

```
@HostPlugin(revision = "$Revision: -1 $", interfaceVersion = 2,
 names = { "github.com" },
 urls = { "https://(www\\.)?github\\.com/[^<>\"]+/downloads" },
 flags = { 0 })
public class GitHubCom extends PluginForHost {

    public GitHubCom(PluginWrapper wrapper) {
        super(wrapper);
    }

    @Override
    public String getAGBLink() {
        return "https://github.com/site/terms";
    }

    @Override
    public AvailableStatus requestFileInformation(DownloadLink link) throws IOException, PluginException {
        this.setBrowserExclusive();
        br.setFollowRedirects(true);
        br.getPage(link.getDownloadURL());
        if (br.containsHTML("assets\\.github\\.com/images/modules/404/")) throw new PluginException(LinkStatus.ERROR_FILE_NOT_FOUND);
        String filename = new Regex(link.getDownloadURL(), ".*?/([^<>"/]*?)/downloads\$").getMatch(0);
        if (filename == null) throw new PluginException(LinkStatus.ERROR_PLUGIN_DEFECT);
        link.setName(Encoding.htmlDecode(filename.trim()));
        return AvailableStatus.TRUE;
    }

    @Override
    public void handleFree(DownloadLink downloadLink) throws Exception, PluginException {
        requestFileInformation(downloadLink);
        dl = BrowserAdapter.openDownload(br, downloadLink, downloadLink.getDownloadURL().replace("/downloads", "/zipball/master"), false, 1);
        if (dl.getConnection().getContentType().contains("html")) {
            br.followConnection();
            throw new PluginException(LinkStatus.ERROR_PLUGIN_DEFECT);
        }
        dl.startDownload();
    }

    @Override
    public void reset() {
    }

    @Override
    public int getMaxSimultanFreeDownloadNum() {
        return -1;
    }

    @Override
    public void resetDownloadlink(DownloadLink link) {
    }

}
```

[4] https://www.github.com

Anhang E Lazy-Plug-ins

Crawler- und Hoster-Plug-ins (siehe Anhang D) dienen dem JDownloader zum Auffinden und Herunterladen von Dateien und Inhalten eines Anbieters. Hierfür nutzt der JDownloader Funktionen und Informationen, welche durch die Plug-ins bereitgestellt werden. Dazu zählen auch die Informationen der *HostPlugin*[5]-beziehungsweise *DecrypterPlugin*[6]-Annotation des jeweiligen Plug-ins. Diese und weitere können aber im Normalfall nur durch eine Aktualisierung des Plug-ins verändert werden und bleiben somit während der gesamten Zeit für die gleiche Version des Plug-ins identisch.

Vielen Funktionen des JDownloaders reichen allein diese Informationen oftmals aus. So wird beispielsweise der für Datien und Inhalte verantwortliche Anbieter anhand der *HostPlugin*- oder *DecrypterPlugin*-Annotation[7] bestimmt. Wenn diese Informationen jedoch nicht ausreichen und JDownloader auf Funktionen eines Plug-ins zugreifen muss, gibt es zwei Möglichkeiten, wie dies geschehen kann. Je nach benötigter Funktion kann eine Plug-in Instanz mehrfach verwendet oder für den jeweiligen Aufruf neu instanziiert werden. Während dem Herunterladen einer Datei bedarf es zum Beispiel einer eigenen Instanz des Plug-ins, da es sich hierbei um einen zustandsbehafteten Ablauf handelt und Informationen zwischen JDownloader und dem Plug-in in beide Richtungen ausgetauscht werden. Im Gegensatz dazu steht das zustandslose Zeichnen von grafischen Elementen. Hier kann eine einzelne Plug-in Instanz mehrfach verwendet werden.

Somit unterscheidet der JDownloader drei verschiedene Zugriffsarten auf Plug-ins.

- Zugriff auf Informationen des Plug-ins
- Mehrfache Nutzung einer Plug-in Instanz
- Mehrfache Instanzierung eines Plug-ins

Der reine Zugriff auf Informationen von Plug-ins ist die häufigste Zugriffsart innerhalb des JDownloader. Um hieraus einen Vorteil ziehen zu können, wurde eine weitere Zwischenschicht zwischen dem JDownloader und den Plug-ins eingeführt.

Sie wird durch die abstrakte Klasse *LazyPlugin*[8] beschrieben und bietet alle drei Zugriffsarten auf das eigentliche Plug-in. Lazy-Plug-ins dienen primär als Zwischenspeicher für alle Informationen, auf die JDownloader am häufigsten zugreift. Zusätzlich stellen sie Funktionen zum korrekten Laden und Instanziieren von

[5]jd.plugins.HostPlugin
[6]jd.plugins.DecrypterPlugin
[7]im Package jd.plugins
[8]org.jdownloader.plugins.controller.LazyPlugin

Plug-ins bereit und enthalten Metainformationen[9] über das Plug-in, die von weiteren Komponenten (siehe Anhang G) von JDownloader benötigt werden. Hoster-Plug-ins implementieren sie in der Klasse LazyHostPlugin[10] und Crawler-Plug-ins in der Klasse LazyCrawlerPlugin[11].

Der große Vorteil gegenüber der direkten Verwendung der Plug-ins, liegt nun darin, dass JDownloader auf bestimmte Informationen der Plug-ins zugreifen kann, ohne das diese geladen oder instanziiert sein müssen. Lazy-Plug-ins unterstützen nicht nur den Entwickler bei dem Umgang mit den Plug-ins, sondern tragen auch einen Großteil zur Reduktion des Speicherverbrauches und der Ladezeit von JDownloader bei.

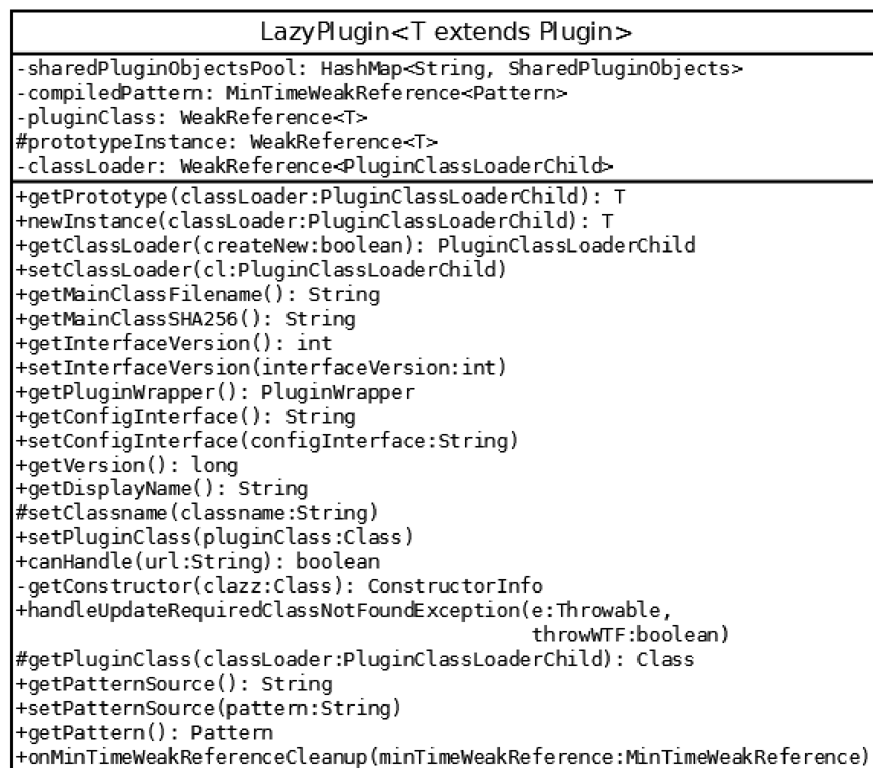


Abbildung 4.4: Klassendiagramm LazyPlugin

[9] Zeitstempel, Dateigröße, Prüfsumme der Plug-in Datei

[10] org.jdownloader.plugins.controller.host.LazyHostPlugin

[11] org.jdownloader.plugins.controller.crawler.LazyCrawlerPlugin

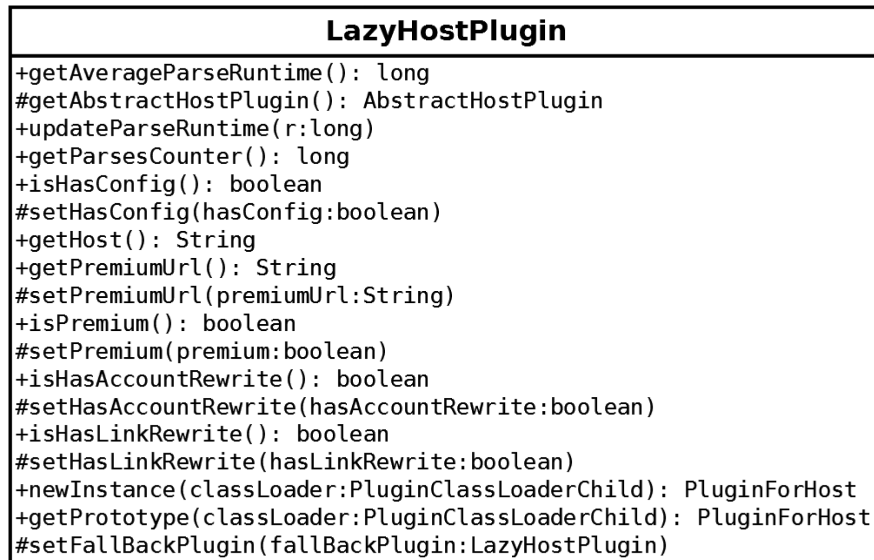


Abbildung 4.5: Klassendiagramm LazyHostPlugin

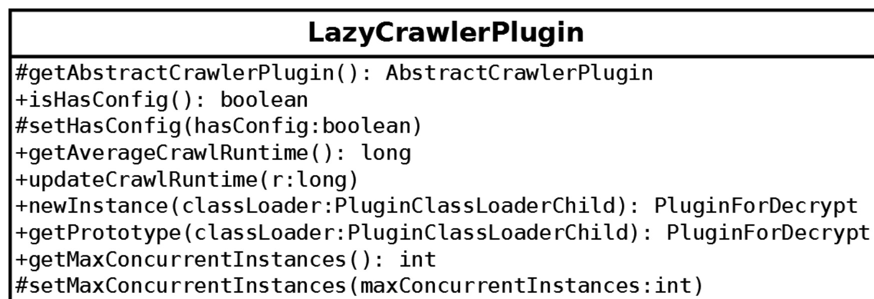


Abbildung 4.6: Klassendiagramm LazyCrawlerPlugin

Anhang F PluginClassLoader

Der *PluginClassLoader*[12] dient der Verwaltug und Instanzierung der für das Laden der Plug-ins verantwortlichen Klasse *PluginClassLoaderChild* (siehe Anhang F.1). Es bedarf einer korrekten Verwaltung aller Instanzen von *PluginClassLoaderChild*, um das mehrfache Laden von Plug-ins zu verhindern und den Speicherverbrauch des JDownloaders nicht unnötig zu erhöhen. Zusätzlich unterstützt der *PluginClassLoader* Entwickler im korrekten Umgang mit der Klasse *PluginClassLoaderChild*, um Problemen mit den Plug-ins, aufgrund falscher Verwendung, vorzubeugen.

[12] org.jdownloader.plugins.controller.PluginClassLoader

```
PluginClassLoader
-sharedClasses: WeakHashMap<Class, String>
-sharedLazyPluginClassLoader: WeakHashMap<PluginClassLoaderChild, WeakReference<LazyPlugin>>
-sharedPluginClassLoader: WeakHashMap<PluginClassLoaderChild, String>
-threadPluginClassLoader: WeakHashMap<Thread, WeakReference<PluginClassLoaderChild>>
-threadGroupPluginClassLoader: WeakHashMap<ThreadGroup, WeakReference<PluginClassLoaderChild>>
+getInstance(): PluginClassLoader
+getChild(): PluginClassLoaderChild
+getThreadPluginClassLoaderChild(): PluginClassLoaderChild
+setThreadPluginClassLoaderChild(theadChild:PluginClassLoaderChild,
                                 threadGroup:PluginClassLoaderChild)
+getSharedChild(plugin:LazyPlugin)
+getSharedChild(plugin:Plugin)
+findSharedClass(name:String): Class
-getCacheID(plugin:Plugin): String
-fetchSharedChild(plugin:Plugin,putIfAbsent:PluginClassLoaderChild): PluginClassLoaderChild
-fetchSharedChild(plugin:LazyPlugin,putIfAbsent:PluginClassLoaderChild): PluginClassLoaderChild
```

Abbildung 4.7: Klassendiagramm PluginClassLoader

Hierfür stehen dem Entwickler nachfolgende Funktionen zur Verfügung:

- getChild() → *PluginClassLoaderChild*

  Erzeugt eine neue Instanz von *PluginClassLoaderChild*.

- getThreadPluginClassLoaderChild() → *PluginClassLoaderChild*

  Gibt den für den aktuellen Prozessfaden oder dessen Gruppe definierten *PluginClassLoaderChild* zurück. Falls weder für den Prozessfaden, noch für dessen Gruppe, zuvor eine Instanz definiert wurde, wird zusätzlich die gesamte Hierarchie der entsprechenden Prozessgruppe durchlaufen um zu prüfen, ob nicht gegebenfalls für diese eine Instanz definiert wurde.

- setThreadPluginClassLoaderChild(*PluginClassLoaderChild,PluginClassLoaderChild*)

  Definiert, je nach Wahl der Parameter, einen *PluginClassLoaderChild* für den aktuellen Prozessfaden und/oder dessen Prozessgruppe. Mittels leerem Parameter (null) kann eine vorherige Defintion wieder aufgehoben werden.

- getSharedChild(*LazyPlugin*) → *PluginClassLoaderChild*

  Gibt den mit dem Lazy-Plug-in verknüpften *PluginClassLoaderChild* zurück. Sollte noch keine Verknüpfung existieren, wird eine neue Instanz von *PluginClassLoaderChild* erzeugt und mit dem Lazy-Plug-in verknüpft.

- getSharedChild(*Plugin*) → *PluginClassLoaderChild*

  Diese Funktion prüft zunächst auf eine existierende Verknüpfung von dem Lazy-Plug-in Pendant des Plug-ins zu einem *PluginClassLoaderChild*. Wird der entsprechende *PluginClassLoaderChild* gefunden, wird er zurück gegeben. Ansonsten wird nach einem mit dem Plug-in verbundenen *PluginClassLoaderChild* gesucht und dieser gegebenfalls zurück gegeben. Existiert auch hier keine Verknüpfung, so wird eine neue Instanz von *PluginClassLoaderChild* erzeugt und mit Plug-in selbst verknüpft.

F.1 PluginClassLoaderChild

Normalerweise werden Klassen in Java mittels einem hierarchischen Delegations-Modell geladen. Ein jeder Klassenlader (ClassLoader[13]) besitzt immer einen ihm übergeordneten Klassenlader. Einzig beim sogenannten „Bootstrap“-Klassenlader ist dies nicht der Fall, da er die Spitze des Delegations-Modells darstellt. Wenn nun ein Klassenlader eine Klasse laden soll, so prüft dieser zunächst, ob er die gewünschte Klasse in der Vergangenheit bereits geladen hat. Falls dem nicht so ist, delegiert er die Aufgabe an seinen übergeordneten Klassenlader. Dies geschieht solange, bis die Spitze des Modells erreicht wurde oder ein übergeordneter Klassenlader die gewünschte Klasse bereits geladen hat. Erst wenn sein übergeordneter Klassenlader nicht in der Lage ist, die Klasse zu laden, darf ein Klassenlader selbst die Klasse laden. Durch dieses Delegations-Modell sorgt Java dafür, dass immer der in der Hierarchie am höchsten stehende Klassenlader (kürzeste Entfernung zum „Bootstrap“-Klassenlader) das Laden einer Klasse übernimmt.

Um Plug-ins in JDownloader zur Laufzeit austauschen und entladen zu können eignet sich dieses Delegations-Modell jedoch nicht, da hierfür sowenig Referenzen wie möglich auf einen Klassenlader existieren sollten, damit dieser durch den Garbage Collector aufgeräumt werden kann. Denn erst dieses Aufräumen des Klassenlader ermöglich das saubere Entladen von Klassen zur Laufzeit. Im Idealfall sollte der Klassenlader deshalb nur durch das Plug-in selbst referenziert sein.

Aus diesem Grund übernimmt im JDownloader das Laden der Plug-ins ein modifizierter Klassenlader, der in der Klasse *PluginClassLoaderChild*[14] implementiert ist. Die Funktionen zum Laden von Klassen wurden dahingehend abgeändert, das das Laden der Plug-ins nicht länger mittels dem Delegations-Modell abläuft, sondern direkt im *PluginClassLoaderChild* geschieht.

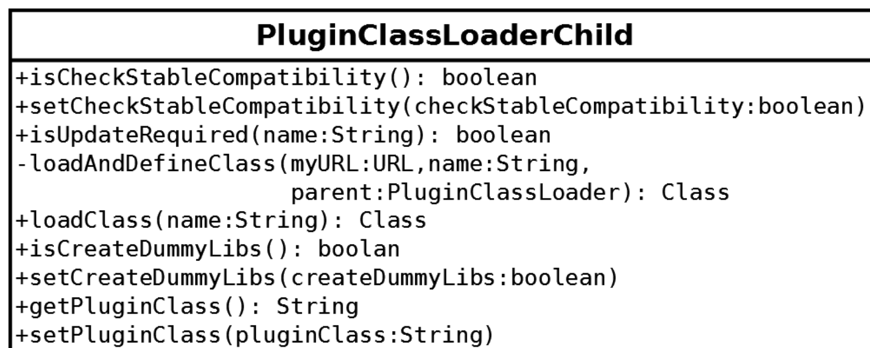


Abbildung 4.8: Klassendiagramm PluginClassLoaderChild

[13] java.lang.ClassLoader
[14] org.jdownloader.plugins.controller.PluginClassLoader

Anhang G PluginController

Der *PluginController*[15] stellt dem JDownloader die Funktion zum Auffinden von Plug-ins zur Verfügung. Diese Funktion durchsucht ein vorgegebenes Verzeichnis nach allen Java-Klassen (.class Dateien), die bestimmte Kriterien[16] aufweisen, und gibt diese anschließend an den Aufrufer zurück. Um den Ablauf der Funktion zu beschleunigen, verwendet sie Informationen und Metadaten (siehe Anhang E) bereits bekannter Lazy-Plug-ins. Ein erneutes Laden von Plug-ins wird vermieden, wenn sie bereits zuvor geladen wurden und deshalb schon bekannt sind. Neben einer Reduktion des Speicherbedarfs, kann so auch die Laufzeit der Funktion spürbar reduziert werden. Beispielsweise beschleunigt sich dadurch die Funktion für die Hoster-Plug-ins von JDownloader 2 von durchschnittlich[17] sechs Sekunden auf unter eine Sekunde.

Abbildung 4.9: Klassendiagramm PluginController

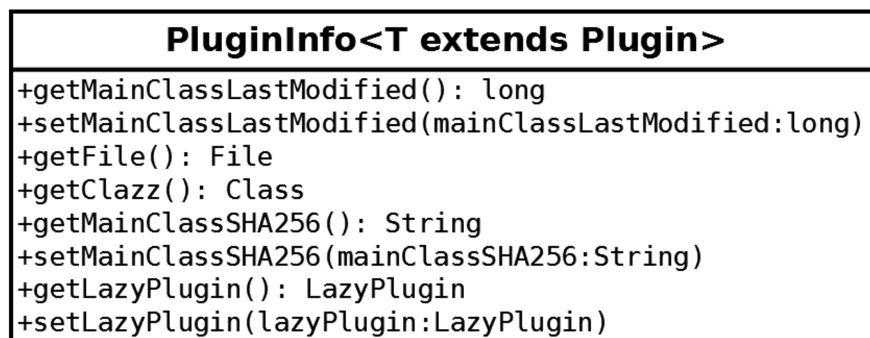


Abbildung 4.10: Klassendiagramm PluginInfo

Nachdem die Plug-ins gefunden wurden, kümmern sich die nachfolgend beschriebene Klassen um deren Verwaltung und stellen dem JDownloader eine zentrale Stelle für Zugriff und Aktualisierung bereit. Um dabei den Speicherverbrauch möglichst gering zu halten, werden nicht die eigentlichen Hoster- und Crawler-Plug-ins verwaltet, sondern lediglich deren LazyHost- und LazyCrawler-Plug-in Pendants. Um Ladezeiten zu minimieren, werden alle gefundenen LazyPlug-ins auf der Festplatte gespeichert, so dass JDownloader auch nach einem Neustart von deren Vorteilen profitieren kann.

[15] org.jdownloader.plugins.controller.PluginController

[16] zum Beispiel keine „inneren“ Klassen

[17] Gemessen auf Windows 7,128 GB SSD, 8GB RAM und 2.8 GHz DualCore-CPU

G.1 HostPluginController

Der *HostPluginController*[18] verwaltet alle Hoster-/LazyHost-Plug-ins. Um eine eindeutige Zuordnung zwischen Anbieter und Hoster-Plug-in zu gewährleisten und Inkompatibilitäten zu vermeiden, muss JDownloader für jeden Anbieter jeweils das gleiche Plug-in verwenden. Es kann jedoch auch vorkommen, dass für einen Anbieter mehrere Plug-ins zur Auswahl stehen. Dies kann zum Beispiel der Fall sein, wenn sich der Anwender zusätzlich externe Plug-ins installiert hat oder ein Entwickler gerade an einem neuen Plug-in für einen bereits unterstützen Anbieter arbeitet. In einem solchen Fall wird deterministisch entschieden, welches Plug-in zum Einsatz kommen wird. Die Klasse kümmert sich außerdem noch um den Austausch veralteter Hoster-/LazyHost-Plug-ins, sobald das Plug-in durch das Update-System aktualisiert wurden.

```
HostPluginController
-list: Map<String, LazyHostPlugin>
-INSTANCE: HostPluginController
+getInstance(): HostPluginController
+init(): Map<String, LazyHostPlugin>
-loadFromCache(logger:LogSource): List<LazyHostPlugin>
-update(logger:LogSource,updateCache:List<LazyHostPlugin>): List<LazyHostPlugin>
+isCacheInvalidated(): boolean
+invalidateCache()
#validateCache()
-save(save:List<AbstractHostPlugin>)
+list(): Collection<LazyHostPlugin>
+ensureLoaded(): Map<String, LazyHostPlugin>
+get(displayName:String): LazyHostPlugin
+invalidateCacheIfRequired()
```

Abbildung 4.11: Klassendiagramm HostPluginController

Für den Zugriff auf die Plug-ins gibt es zwei unterschiedliche Funktionen:

- list() → Collection<*LazyHostPlugin*>

  Gibt eine Sammlung aller verwalteter LazyHost-Plug-ins zurück, welche mittels eines Iterators durchwandert werden kann. Sie ist alphabetisch (a-z) nach den Namen der Anbieter sortiert und der Iterator durchwandert die Sammlung immer in der gleichen Reihenfolge.

- get(*String*) → *LazyHostPlugin*

  Gibt das LazyHost-Plug-in für den angegebenen Anbieter zurück. Aufgrund der internen *LinkedHashMap* vom *HostPluginController* ist der Zugriff über diese Funktion entsprechend performant und bietet in den meisten Fällen eine Laufzeit von O(1).

[18]org.jdownloader.plugins.controller.host.HostPluginController

G.2 CrawlerPluginController

Der *CrawlerPluginController*[19] verwaltet alle Crawler-/LazyCrawler-Plug-ins. Im Gegensatz zum *HostPluginController*, dürfen dabei auch mehrere Crawler-Plug-ins pro Anbieter existieren. Dennoch bedarf es in solchen Situationen ebenfalls eines deterministisches Verhaltens der entsprechenden Crawler-Plug-ins. Aus diesem Grund wird die interne Liste aller Crawler-Plug-ins anhand fester Regeln sortiert, bevor JDownloader auf sie zugreifen kann. Da Crawler-Plug-ins hauptsächlich während dem Hinzufügen von Dateien und Inhalten der Anbieter benötigt werden, kann der Speicherverbrauch weiter reduziert werden, indem die Plug-ins nur vorgehalten werden, wenn sie auch zum Einsatz kommen. In der restlichen Zeit kann die Liste durchaus verworfen/freigegeben werden, da das erneute Laden der LazyCrawler-Plug-ins, aufgrund der Optimierungen des *PluginController*, kaum ins Gewicht fällt.

```
CrawlerPluginController
-INSTANCE: MinTimeWeakReference<CrawlerPluginController>
-list: List<LazyCrawlerPlugin>
+isCacheInvalidated(): boolean
+invalidateCache()
#validateCache()
+getInstance(): CrawlerPluginController
+init(): List<LazyCrawlerPlugin>
-loadFromCache(): List<LazyCrawlerPlugin>
-update(logger:LogSource,updateCache:List<LazyCrawlerPlugin>): List<LazyCrawlerPlugin>
-save(save:List<AbstractCrawlerPlugin>)
+list(): List<LazyCrawlerPlugin>
+list(ensureLoaded:boolean): List<LazyCrawlerPlugin>
+ensureLoaded(): List<LazyCrawlerPlugin>
+get(displayName:String): LazyCrawlerPlugin
+getAll(displayName:String): List<LazyCrawlerPlugin>
+invalidateCacheIfRequired()
```

Abbildung 4.12: Klassendiagramm CrawlerPluginController

Für den Zugriff auf die Plug-ins gibt es mehrere Möglichkeiten:

- list() → List<*LazyCrawlerPlugin*>

  Gibt eine Liste aller vorhandener LazyCrawler-Plug-ins zurück.

- list(*Boolean*) → List<*LazyCrawlerPlugin*>

  Falls der Parameter „true“ ist, wird ebenfalls eine Liste aller bekannter LazyCrawler-Plug-ins zurück gegeben. Für den Parameter „false“ hängt die Rückgabe jedoch vom internen Zustand des *CrawlerPluginController* ab. Hält dieser zu dem Zeitpunkt eine interne Liste an Plug-ins parat, so gleicht die Rückgabe einem Aufruf mit dem Parameter „true“. Existiert keine interne Liste, so wird auch nichts (null) zurückgegeben.

[19] org.jdownloader.plugins.controller.crawler.CrawlerPluginController

- get(String) → *LazyCrawlerPlugin*

  Durchläuft alle verfügbaren LazyCrawler-Plug-ins und gibt das erste gefundene LazyCrawler-Plug-ins zurück, welches für den angegebenen Anbieter zuständig ist. Aufgrund der Tatsache, dass die interne Liste im *CrawlerPluginController* nicht anhand der Namen der Anbieter sortiert ist, ist ihre Laufzeit im schlimmsten Fall mit O(n) zu bewerten.

- getAll(*String*) → List<*LazyCrawlerPlugin*>

  Liefert alle verfügbaren LazyCrawler-Plug-ins für den angegebenen Anbieter zurück. Die Laufzeit dieser Funktion ist immer mit O(n) zu bewerten, da dem *CrawlerPluginController* die Information fehlt, wie viel Plug-ins pro Anbieter vorhanden sind. Aus diesem Grund müssen immer alle Plug-ins durchlaufen werden.

Anhang H schwache Referenzen

Ein jedes Objekt in Java ist normalerweiße mittels einer „starken Referenz“ erreichbar (referenziert). Eine „starke Referenz“ auf ein Objekt entsteht zum Beispiel durch die Zuweisung einer Variable oder dem Hinzufügen in eine Liste/Sammlung von Objekten. Auch können lebendige Prozessfäden eine solche Referenz auf ein Objekt darstellen.

Für die automatisierte Speicherverwaltung bedient sich der Garbage Collector unterschiedlicher Methoden, um die Erreichbarkeit eines jeden Objektes festzustellen. Ab dem Zeitpunkt, ab dem ein Objekt durch keine „starke Referenz“ mehr erreichbar ist, ist es dem Garbage Collector erlaubt, den Speicherbereich des Objektes anhand unterschiedlicher Regeln freizugeben und der JVM wieder bereitzustellen.

Seit der Version 1.2 bietet Java, neben der „starken Referenz“, auch schwächere Referenzmöglichkeiten an. Objekte können nun zum Beispiel mittels einer „schwachen Referenz[20]“ referenziert und verwendet werden. Im Gegensatz zu „starken Referenzen“ halten solche „schwache Referenzen“ den Garbage Collector nicht davon ab, den Speicherbereich des Objektes wieder freizugeben. Mittels der abgeschwächten Referenzen kann sich der Entwickler indirekt den Garbage Collector zu Nutze machen.

So lassen sich zum Beispiel Objekt-Caches realisieren (Monson, 1998), dessen Einträge vom Garbage Collector automatisiert freigegeben werden, sobald sie durch keine „starke Referenz“ mehr erreichbar sind. Anhand des nachfolgenden Pseudocodes wird dies demonstriert.

Pseudocode:

```
String sr = new String("not null");
WeakHashMap<String, Boolean> wm = new WeakHashMap<String, Boolean>();
wm.put(sr, Boolean.TRUE);
System.out.println("(Before GC) StrongReference=" + sr);
System.out.println("(Before GC) MapKeys=" + wm.keySet()+"|MapSize:"+wm.size());
System.gc();
System.out.println("(After 1st GC) StrongReference=" + sr);
System.out.println("(After 1st GC) MapKeys=" + wm.keySet()+"|MapSize:"+wm.size());
sr = null;
System.gc();
System.out.println("(After 2nd GC) StrongReference=" + sr);
System.out.println("(After 2nd GC) MapKeys=" + wm.keySet()+"|MapSize:"+wm.size());
```

Ausgabe des Pseudocodes:

```
(Before GC) StrongReference=not null
(Before GC) MapKeys=[not null]|MapSize:1
(After 1st GC) StrongReference=not null
(After 1st GC) MapKeys=[not null]|MapSize:1
(After 2nd GC) StrongReference=null
(After 2nd GC) MapKeys=[]|MapSize:0
```

[20] java.lang.ref.WeakReference

H.1 MinTimeWeakReference

Der unbedachte Einsatz von „schwachen Referenzen“ kann sich aber auch nachteilig auf eine Anwendung auswirken. Je nach Zugriffsmuster der Anwendung auf schwach referenzierte Objekte kann es passieren das Objekte neu erstellt, anschließend aber vom Garbage Collector gleich wieder freigegeben werden. Solche Situationen führen nicht selten zu erhöhtem Rechen- und Speicheraufwand.

Anhand „weicher Referenzen[21]“ ließe sich der genannte Fall einfach lösen. Im Gegensatz zu „schwachen Referenzen“ wird dabei der Speicherbereich von Objekten durch den Garbage Collector nicht sofort, sondern erst bei Speicherknappheit freigegeben. Der Nachteil von „weichen Referenzen“ ist aber, dass nun neben der Erreichbarkeit eines Objektes, auch der Speicherverbrauch der gesamten Anwendung als Kriterium dient, wann ein Objekt vom Garbage Collector freigegeben wird. Dies kann dazu führen, dass viele Objekte den Speicher belegen und „zumüllen“, obwohl sie gar nicht mehr erreichbar oder gar gebraucht werden.

Um die Vorteile von „schwachen Referenzen“ beizubehalten und die oben beschriebene Situationen zu vermeiden, wurde die Klasse *WeakReference* von der Klasse *MinTimeWeakReference*[22] erweitert. Diese hält neben der ursprünglichen „schwachen Referenz“ ebenfalls eine zeitlich beschränkte „starke Referenz“. Bei der Erzeugung einer *MinTimeWeakReference* legt man fest, wie lange nach dem letzten Aufruf der „get()“-Methode der Referenz, das Objekt weiterhin vorgehalten werden soll, bis der Garbage Collector das Objekt wieder freigeben darf. Der folgende Pseudocode soll dies veranschaulichen.

Pseudocode:

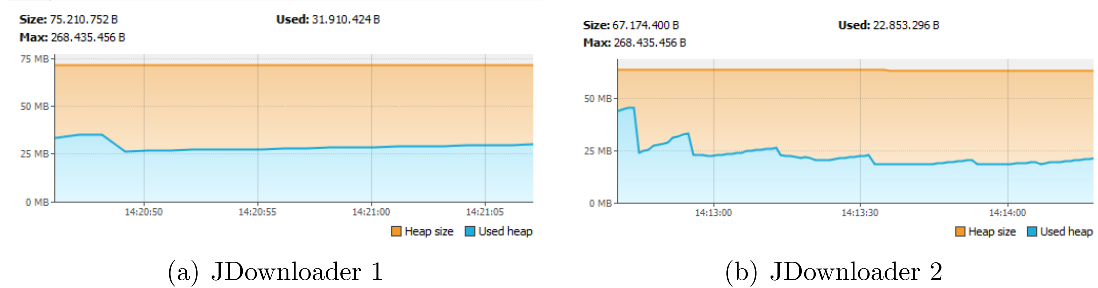
```
MinTimeWeakReference<String> mtwr=new MinTimeWeakReference<String>(new String("not null"), 100, "");
System.out.println("(Before GC) MinTimeWeakReference=" +mtwr.get());
System.gc();
System.out.println("(After 1st GC) MinTimeWeakReference=" +mtwr.get());
Thread.sleep(150);
System.gc();
System.out.println("(After 150 ms and 2nd GC) MinTimeWeakReference=" +mtwr.get());
```

Ausgabe des Pseudocodes:

```
(Before GC) MinTimeWeakReference=not null
(After 1st GC) MinTimeWeakReference=not null
(After 150 ms and 2nd GC) MinTimeWeakReference=null
```

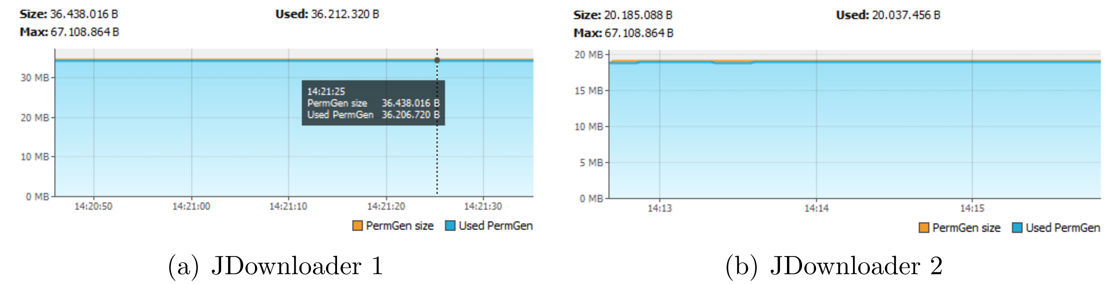
[21] java.lang.ref.SoftReference
[22] org.appwork.storage.config.MinTimeWeakReference

Anhang I VisualVM-Analyse für 2.4.3



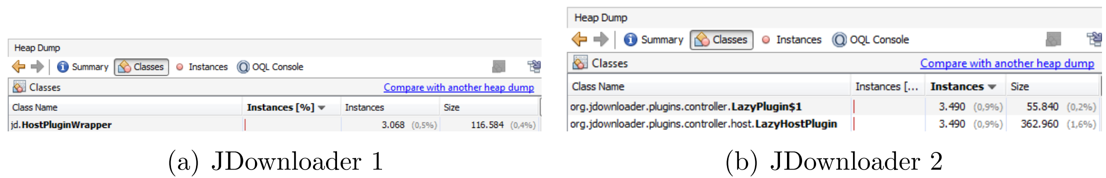
(a) JDownloader 1 (b) JDownloader 2

Abbildung 4.13: Heap-Speicherverbrauch



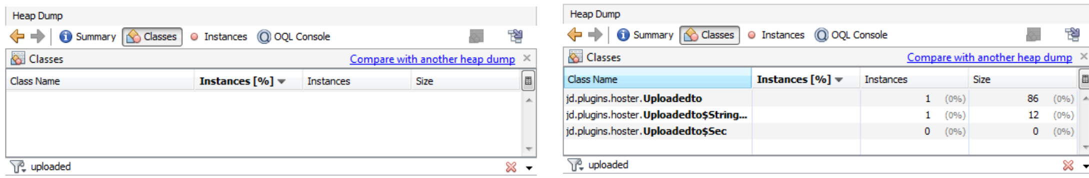
(a) JDownloader 1 (b) JDownloader 2

Abbildung 4.14: PermGen-Speicherverbrauch



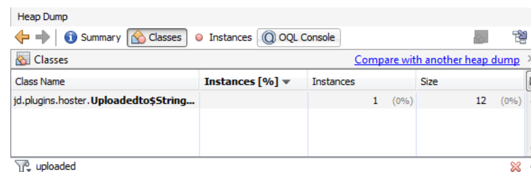
(a) JDownloader 1 (b) JDownloader 2

Abbildung 4.15: Anzahl verwalteter Plug-ins



(a) Kein Plugin geladen

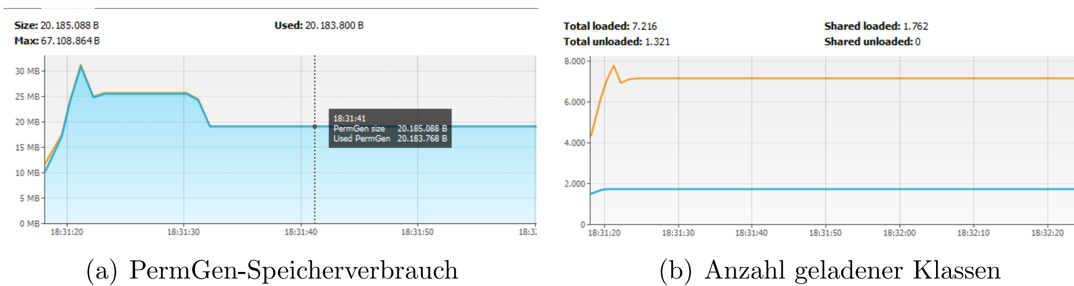
(b) Plugin geladen



(c) Plugin entladen

Abbildung 4.16: Beispiel von Laden/Entladen des Uploaded-Plug-ins

Anhang J VisualVM-Analyse für 2.5



(a) PermGen-Speicherverbrauch

(b) Anzahl geladener Klassen

Abbildung 4.17: Entladen von Klassen

Literaturverzeichnis

Abran, A., Khelifi, A., Suryn, W. & Seffah, A. (2003). Usability meanings and interpretations in iso standards. Software Quality Journal, 11 (4), 325–338.

Alliance, O. (2014). Osgi service platform, core specification, release 6. OSGi Specification.

Birsan, D. (2005). On plug-ins and extensible architectures. Queue, 3 (2), 40–46.

Chatley, R., Eisenbach, S., Kramer, J., Magee, J. & Uchitel, S. (2004). Predictable dynamic plugin systems. In Fundamental approaches to software engineering (S. 129–143). Springer.

Chung, L. & do Prado Leite, J. C. S. (2009). On non-functional requirements in software engineering. In Conceptual modeling: Foundations and applications (S. 363–379). Springer.

Crockford, D. (2006). Json: The fat-free alternative to xml. In Proc. of xml (Bd. 2006).

Duvigneau, M. (2010). Konzeptionelle Modellierung von Plugin-Systemen mit Petrinetzen (Bd. 4). Logos Verlag Berlin GmbH.

Fowler, M. (2002). Patterns of enterprise application architecture. Addison-Wesley Longman Publishing Co., Inc.

Mayer, J., Melzer, I. & Schweiggert, F. (2003). Lightweight plug-in-based application development. In Objects, components, architectures, services, and applications for a networked world (S. 87–102). Springer.

McAffer, J., VanderLei, P. & Archer, S. (2010). Osgi and equinox: Creating highly modular java systems. Addison-Wesley Professional.

Metsker, S. J. (2002). The design patterns java workbook. Addison-Wesley Longman Publishing Co., Inc.

Monson, L. (1998). Caching & weakreferences. JAVA Developer's Journal, 3 (8), 32–36.

Rathlev, J. (2011). Anwendungsentwicklung mit Plug-in-Architekturen: Erfahrungen aus der Praxis. In Software engineering (S. 183–196).

Rechenmacher, T. (2013). Online-Update-Systeme am Beispiel JDownloader (Bachelor's Thesis (Projektarbeit)). University of Erlangen.

Rechenmacher, T. (2014). The JDownloader continuous deployment immune system (Diploma thesis). University of Erlangen.

Riehle, D., Siberski, W., Bäumer, D., Megert, D. & Züllighoven, H. (1997). Serializer. In Pattern languages of program design 3 (S. 293–312).

Shams, Z. & Edwards, S. H. (2013). Reflection support: Java reflection made easy. Open Software Engineering Journal, 7, 38–52.