Karlsruher Institut für Technologie

Institut für Programmstrukturen
und Datenorganisation (IPD)

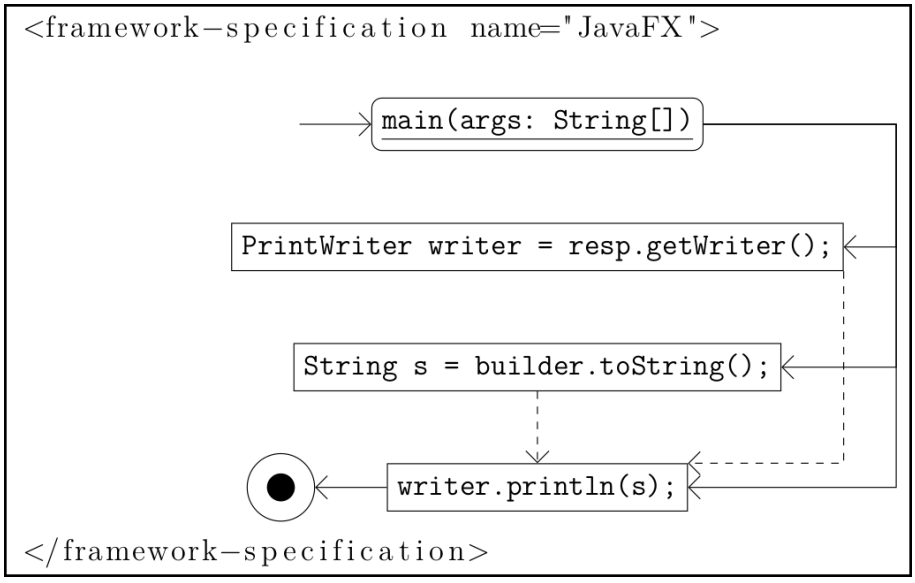
Lehrstuhl Prof. Dr.-Ing. Snelting

Eine Sprache zur Spezifikation von Lebenszyklen framework-basierter Anwendungen

Bachelorarbeit von

Martin Armbruster

an der Fakultät für Informatik



Erstgutachter:	Prof. Dr.-Ing. Gregor Snelting
Zweitgutachter:	Prof. Dr. rer. nat. Bernhard Beckert
Betreuende Mitarbeiter:	M. Sc. Simon Bischof
	Dipl.-Math. Dipl.-Inform. Martin Mohr
Abgabedatum:	17. Oktober 2018

Zusammenfassung

Sprach-basierte Informationsflusskontrolle analysiert Quell- oder Bytecode statisch auf Sicherheitsverletzungen hin [26]. Java Object-sensitive ANAlysis (JOANA) implementiert eine Informationsflusskontrolle für Java Bytecode, bei der ein Systemabhängigkeitsgraph für die zu analysierende Anwendung erzeugt, annotiert und hinsichtlich Sicherheitsverletzungen untersucht wird [25]. Hierbei geht JOANA von der `main`-Methode als Einsprungspunkt der Anwendung aus. Damit können framework-basierte Anwendungen, die in der Regel mehrere Einsprungspunkte definieren und vom nicht-deterministischen Nutzerverhalten abhängig sind, nicht analysiert werden.

Im Rahmen dieser Arbeit wurde der Ansatz untersucht, framework-basierte Anwendungen durch die Generierung einer künstlichen `main`-Methode, die das Framwork simuliert und für die Anwendung einen Einsprungspunkt setzt, für JOANA analysierbar zu machen. Dazu wurde durch die Analyse der Frameworks Java Servlet Specification, Version 4.0, Swing und JavaFX 8 eine Sprache entwickelt, mit der sich ein Framework spezifizieren lässt. Diese Framework-Spezifikation wird von der prototypischen Implementierung „Joana and Frameworks (JoFrames)“ mit dem Framework und einer Anwendung zu einer künstlichen `main`-Methode umgesetzt. Wie sich gezeigt hat, kann mit der künstlichen `main`-Methode eine Anwendung in JOANA auf Sicherheitsverletzungen hin untersucht werden. Für die Unterstützung weiterer Frameworks sind Erweiterungen der Sprache und zusätzliche Analysen für die jeweiligen Frameworks notwendig.

Inhaltsverzeichnis

1. Einführung

Java Object-sensitive ANAlysis (JOANA)[1] erzeugt aus einer Anwendung, dessen `main`-Methode den Einsprungspunkt der Anwendung markiert, intra-prozedural einen Programmabhängigkeitsgprahen und inter-prozedural einen Systemabhängigkeitsgraphen [25, 26]. Darauf kann JOANA eine sprach-basierte Informationsflusskontrolle durchführen und so Java Bytecode statisch auf Sicherheit hin untersuchen.

Das Code-Beispiel 1 zeigt ein Ausschnitt aus dem `edu.kit.ipd.pp.joframes.test.servlet.ip.IPServlet`, aufbauend auf der Java Servlet Specification, Version 4.0, einem Framework für `javax.servlet.Servlet`s mit Anfrage-Antwort-(Request-Response-)Verfahren [47].

```
1  public final class IPServlet extends HttpServlet {
2      private StringBuilder builder = new StringBuilder();
3
4      @Override
5      protected void doGet(final HttpServletRequest req, final
           HttpServletResponse resp) throws IOException {
6          String ip = req.getRemoteAddr(); // Quelle: HIGH
7          builder.append(ip);
8      }
9
10     @Override
11     protected void doPost(final HttpServletRequest req,
          final HttpServletResponse resp) throws IOException {
12         PrintWriter writer = resp.getWriter();
13         writer.println(builder.toString()); // Senke: LOW
14     }
15 }
```

Listing 1.1: Code-Beispiel: ein Ausschnitt aus dem `IPServlet`

Kompiliert bildet das `IPServlet` eine eigene Anwendung, die von einer Implementierung der Java Servlet Specification, Version 4.0, ausgeführt werden kann. Dabei können Benutzer beliebig viele Hypertext Transfer Protocol (HTTP) GET

[1] `http://joana.ipd.kit.edu`

und POST-Anfragen an das `IPServlet` stellen, die zum mehrmaligen, gleichzeitigen Aufruf der `doGet-` und `doPost`-Methoden genau einer Instanz des `IPServlets` führen [47]. Mit jeder GET-Anfrage wird in der `doGet`-Methode die zugehörige IP-Adresse, als vertrauenswürdige Datenquelle mit HIGH annotiert, in einem `java.lang.StringBuilder` gespeichert. Mit jeder POST-Anfrage kommt es in der `doPost`-Methode zu einer öffentlichen Antwort, die über einen `java.io.PrintWriter` geschrieben wird [43]. Daher ist die Methode `println` des `PrintWriter` als Senke mit LOW annotiert. In die Ausgabe wird der Inhalt des `StringBuilders` geschrieben, welcher alle vorher gespeicherten IP-Adressen umfasst. Somit besteht eine Verletzung der Vertraulichkeit im `IPServlet`.

JOANA kann diesen illegalen Informationsfluss nicht erkennen, da das `IPServlet` über keine `main`-Methode verfügt und sich sowohl für das `IPServlet` wie auch in echten Servlet-Anwendungen mit mehreren `Servlets` kein eindeutiger Einsprungspunkt definieren lässt. `Servlets` können hierbei getrennt voneinander Anfragen verarbeiten und über die Verwendung gemeinsamer Objekte Informationsflüsse etablieren [47]. In dieser Arbeit wird daher der Ansatz untersucht, für u. a. eine Servlet-Anwendung eine künstliche `main`-Methode zu generieren, die die Besonderheiten der Java Servlet Specification, Version 4.0, berücksichtigt und die Ausführung einer Anwendung simuliert. Um diesen Ansatz nicht nur auf ein Framework zu beschränken, sondern auch für weitere Frameworks nutzbar zu machen, wird über den Vergleich dreier Frameworks (in Kapitel 3.1.2) eine Sprache entwickelt (Kapitel 3.2), mit der ein beliebiges Framework beschrieben werden kann und mit der für die Anwendung eines beschriebenen Frameworks eine künstliche `main`-Methode generiert wird. Der Ansatz wird prototypisch implementiert (Kapitel 3.3) und hinsichtlich der Erkennung von Informationsflüssen in JOANA evaluiert (Kapitel 4.1). Zudem wird betrachtet, inwieweit zusätzliche Frameworks in der entwickelten Sprache ausgedrückt werden können (Kapitel 4.2).

2. Grundlagen und Verwandte Arbeiten

In diesem Kapitel werden zuerst die Grundlagen dieser Arbeit vorgestellt. Dies umfasst die sprach-basierte Informationsflusskontrolle, Programm- und Systemabhängigkeitsgraphen und Systemabhängigkeitsgraph-basierte Informationsflusskontrolle. Im Anschluss werden verwandte Arbeiten zur Informationsflusskontrolle framework-basierter Anwendungen vorgestellt und mit dem Ansatz dieser Arbeit verglichen.

2.1. Sprach-basierte Informationsflusskontrolle

Sprach-basierte Informationsflusskontrolle untersucht statisch den Quell- oder Bytecode einer Anwendung auf Einhaltung von Vertraulichkeit und Integrität [26]. Vertraulichkeit bedeutet hierbei, dass vertrauenswürdige Daten geschützt bleiben und nicht öffentlich zugänglich werden, und Integrität, dass kritische Berechnungen nicht durch nicht-vertrauenswürdige Daten beeinflusst werden können.

Es existieren verschiedene Anforderungen an die Informationsflusskontrolle, die miteinander konkurrieren: U. a. soll eine Informationsflusskontrolle alle Sicherheitsverletzungen finden (Korrektheit), ohne falsche Alarme auszulösen (Präzision) und dabei verschieden große Programme verarbeiten können (Skalierbarkeit) [26]. Diese drei Anforderungen lassen sich nicht gleichzeitig erfüllen, sodass korrekte Informationsflusskontrollalgorithmen konservativ arbeiten. Damit werden alle Sicherheitsverletzungen gefunden, es können aber auch falsche Alarme erzeugt werden.

Analysen zur Informationsflusskontrolle können mit unterschiedlichen Sensitivitäten eingestellt werden, die Einfluss auf die Präzision haben [26]. Je sensitiver dabei eine Analyse ist, desto präziser ist sie auch. Eine Analyse kann fluss-sensitiv, womit die Reihenfolge der Anweisungen beachtet wird, kontext-sensitiv, womit der Aufrufkontext einer Methode berücksichtigt wird und für jeden Aufruf ein eigener Kontext existiert, und objekt-sensitiv, bei dem bei Zugriffen auf ein Objekt-Feld zwischen verschiedenen Objekten unterschieden wird, sein.

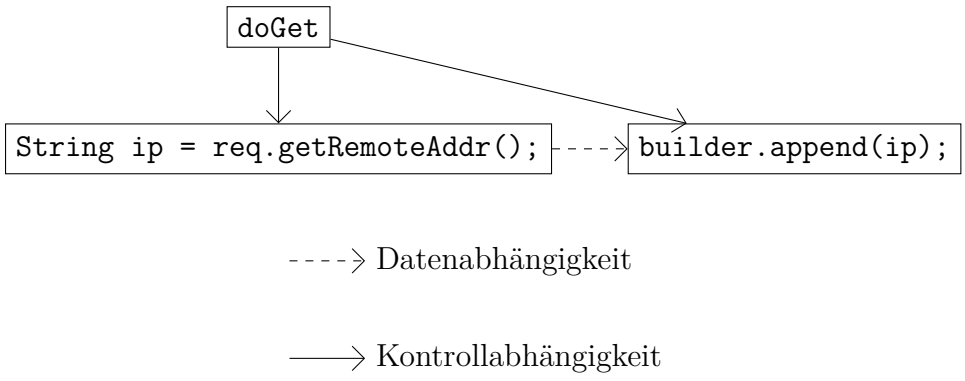


Abbildung 2.1.: Beispiel eines Programmabhängigkeitsgraphen für die `IPServlet.doGet`-Methode

2.2. Programmabhängigkeitsgraph

Ein Programmabhängigkeitsgraph ist eine Datenstruktur zur Modellierung von intra-prozeduralen Informationsflüssen innerhalb einer Anwendung [26]. Eine Anweisung bildet dabei einen Knoten. Dazu gibt es Daten- und Kontrollabhängigkeitskanten. Bei einer Datenabhängigkeitskante $x \rightarrow y$ wird in x eine Variable zugewiesen, die in y verwendet wird und bis dahin nicht nochmals zugewiesen wird. Bei einer Kontrollabhängigkeitskante $x \rightarrow y$ hängt die Ausführung von y direkt von x ab, wobei es sich bei x in der Regel um einen Ausdruck einer if-Bedingung oder while-Schleife handelt. Mittels Slicing kann der Backward-Slice $BS(y) = \{x \mid x \rightarrow^* y\}$ für eine Anweisung y bestimmt werden, der alle Anweisungen beinhaltet, die y beeinflussen können. Programmabhängigkeitsgraphen sind fluss-sensitiv.

Abbildung 2.1 zeigt ein Beispiel für einen Programmabhängikeitsgraphen.

2.3. Systemabhängigkeitsgraph

Systemabhängigkeitsgraphen erweitern die intra-prozedurale Anwendungs-Modellierung um die inter-prozedurale Modellierung [26]. Die Elemente des Programmabhängigkeitsgraphen werden um Abhängigkeiten zu Methodenaufrufen, Parameter und summary edges ergänzt. Jede Methode erhält für jeden Ein- und Ausgabeparameter einen formal-in und formal-out Knoten. Für jeden Methodenaufruf gibt es einen Knoten für den Methodenaufruf mit actual-in und actual-out Knoten für die eigentlichen Ein- und Ausgabeparameter. Zwischen den actual-in und actual-out Knoten werden summary edges etabliert, die außen sichtbare transitive Abhängigkeiten zwischen den Parametern repräsentieren. Damit ist ein Systemabhängigkeitsgraph

ebenfalls fluss- und dazu kontext-sensitiv. Durch die in [26] vorgestellte Repräsentation von geschachtelten Parameterobjekten als Bäume wird ein Systemabhängigkeitsgraph auch objekt-sensitiv.

2.4. Systemabhängigkeitsgraph-basierte Informationsflusskontrolle

Auf einem Systemabhängigkeitsgraphen lässt sich eine Informationsflusskontrolle durchführen [26]. Dazu können relevante Knoten als Quelle oder Senke mit Sicherheitsleveln, die einen Verband bilden und auf denen damit eine Halbordnung definiert ist, annotiert werden. Einfache Sicherheitslevel sind beispielsweise $HIGH \geq LOW$. Die Sicherheitslevel werden entlang der Kanten über den Systemabhängigkeitsgraphen propagiert. Für die Informationsflusskontrolle wird nun genutzt: Wenn $x \notin BS(y)$ gilt, dann gibt es keinen Informationsfluss zwischen x und y. Wenn sich entsprechend eine mit einem höheren Sicherheitslevel annotierte Quelle im Backward-Slice einer mit niedrigerem Sicherheitslevel annotierten Senke befindet, liegt eine Sicherheitsverletzung vor. Umgekehrt dürfen Daten von Knoten mit einem niedrigen Sicherheitslevel zu Knoten mit höherem Sicherheitslevel fließen.

In manchen Situationen ist der Informationsfluss von einem mit einem hohen Sicherheitslevel annotierten Knoten zu einem Knoten mit niedrigerem Sicherheitslevel erforderlich. Mit der Markierung von Knoten, an denen dieser Fluss erlaubt wird, als Deklassifikations-Knoten wird gewährleistet, dass der Informationsfluss nicht als illegal gekennzeichnet wird [26].

2.5. Java Object-sensitive ANAlysis (JOANA)

Bei JOANA handelt es sich um die prototypische Implementierung einer Systemabhängigkeitsgraph-basierten Informationsflusskontrolle für Java Bytecode [25]. Es stellt dafür ein Application Programming Interface (API) und die *IFC Console*, eine graphische Benutzerschnittstelle, zur Verfügung. Damit können für single- und multi-threaded Anwendungen der Systemabhängigkeitsgraph erstellt, annotiert und auf Sicherheitsverletzungen hin geprüft werden. JOANA skaliert bis zu einer Anwendungsgröße von 100 kLOC und basiert auf den T. J. Watson Libraries for Analysis (WALA)[1]. Für die Java API nutzt JOANA Stubs, die den inneren Informationsfluss modellieren und die Menge des zu analysierenden Codes reduzieren.

[1] `http://wala.sourceforge.net/wiki/index.php/Main_Page`

2.6. Verwandte Arbeiten

Im Nachfolgenden wird ein kurzer Blick auf Arbeiten, die ebenfalls auf die Informationsflusskontrolle framework-basierter Anwendungen abzielen, geworfen.

2.6.1. JoDroid

JoDroid stellt eine Erweiterung für JOANA dar, die die Analyse von Android-Anwendungen[2] ermöglicht [12]. Hierbei nutzt JoDroid ein Modell für Android-Anwendungen, das die wichtigsten Einsprungspunkte und ihre Beziehungen untereinander berücksichtigt. Mit dem Modell und einer Heuristik für das Auffinden weiterer Einsprungspunkte werden die vorhandenen Einsprungspunkte einer Android-Anwendung identifiziert und in der Reihenfolge ihres Aufrufes geordnet. Dazu werden auch XML-Dateien untersucht, die als Teil einer Android-Anwendung Einsprungspunkte definieren. Die Heuristik wählt Methoden aus, die mit „on“ beginnen. JoDroid generiert schließlich aus den aufgefundenen Einsprungspunkten eine synthetische Methode, die die Einsprungspunkte in ihrer Reihenfolge aufruft. Für die Generierung stehen verschiedene Optionen bereit, mit denen der Lebenszyklus der Android-Anwendung unterschiedlich simuliert werden kann. Dazu gehören eine sequentielle Variante, die zusätzliche Simulation von Benutzerinteraktionen, der Einfluss anderer Android-Anwendungen und der Neustart der Android-Anwendung, die bei Android zur Ausführung weiterer, spezieller Einsprungspunkte führt.

Die Idee hinter der Heuristik, nach Mustern in Methodennamen zu suchen, findet zum Teil Anwendung in der Regex-Regel der hier entwickelten Sprache (Kapitel 3.2). Ebenfalls wird in der künstlichen `main`-Methode Benutzerverhalten simuliert. Weitere Optionen für die Generierung wie bei JoDroid stehen nicht zur Verfügung, da diese bei JoDroid Besonderheiten von Android reflektieren, die bei den in dieser Arbeit analysierten Frameworks nicht vorliegen. Für die Generierung wird ein Modell als Ausdruck der Sprache (Kapitel 3.2), die für beliebige Frameworks entworfen wurde, genutzt, während JoDroid ein spezielles Modell, auf Android beschränkt, nutzt.

2.6.2. Taint Analysis for Java (TAJ)

Tripp *et al.* [50] präsentieren mit TAJ eine statische Taint-Analyse, die Anwendungen, die auf der Java Platform, Enterprise Edition (Java EE), basieren, hinsichtlich der am häufigsten vorkommenden Sicherheitsverletzungen untersuchen. TAJ nimmt zunächst

[2] `https://www.android.com/`

die Anwendung mit den benötigten Bibliotheken und einer Menge an Sicherheitsregeln entgegen. Eine Sicherheitsregel besteht aus einer Menge an Quellmethoden, einer Menge an Sanitizern, die Deklassifikationsknoten entsprechen, und eine Menge an Senken. Hierbei stellen die Quellmethoden Quellen nicht-vertrauenswürdiger Daten und die Senken Methoden, die sensitive Berechnungen vornehmen, dar. Damit prüft TAJ Integrität. Dazu wird ein spezieller Systemabhängigkeitsgraph genutzt, der bei Bedarf und auf dem ein hybrider Thin-Slice für jede Quelle berechnet wird. Dieser verwendet kontext-insensitives und -sensitives sowie fluss-sensitives Slicing als Balance zwischen Präzision und Skalierbarkeit. Befindet sich eine Senke ohne Sanitizer zur Quelle in einem hybriden Thin-Slice, liegt eine Sicherheitsverletzung vor. Zusätzlich zum hybriden Thin-Slice werden weitere Optimierungen genutzt. Native Methoden, die Reflections-API, `String`-Klassen und spezielle JavaEE-Bibliotheken werden durch einfachere Modelle ersetzt, die den Datenfluss simulieren und den Code reduzieren. Zudem werden XML-Dateien als Teil einer JavaEE-Anwendung in der Erstellung der einfacheren Modelle berücksichtigt. Für eine verbesserte Skalierbarkeit können auch bestimmte Parameter beschränkt werden, u. a. die Länge des hybriden Thin-Slices. Damit erfasst TAJ mit bestimmten Einstellungen nicht alle Sicherheitsverletzungen.

Die meisten Optimierungen von TAJ können für diese Arbeit nicht übernommen werden und stehen auch nicht im Fokus, da die prototypische Implementierung der Sprache JOANA nutzt und keine Änderungen vornimmt (Kapitel 3.3). Zudem wird damit ein korrektes System zur Informationsflusskontrolle erwartet. Die Einbeziehung von XML-Dateien bzw. allgemein weiterer Ressourcen neben dem eigentlichen Java Code wird mit dem Ressourcen-Lader (Kapitel 3.2.3) als Erweiterung der Sprache ebenfalls möglich. In der Evaluation wird sich zeigen, dass Optimierungen für die Analyse der künstlichen `main`-Methode mit JOANA in Betracht gezogen werden können (Kapitel 4.1.3). Hierunter kann in zukünftiger Arbeit auch die Entwicklung vereinfachter Modelle für Frameworks, wie in TAJ geschehen, fallen.

2.6.3. Framework For Frameworks (F4F)

Sridharan *et al.* [48] stellen mit F4F ein Framework für die Taint-Analyse von framework-basierten Webanwendungen vor. Dabei wird eine Anwendung von einem framework-spezifischen Generator verarbeitet, in dessen Folge eine Spezifikation in der Web Application Framework Language (WAFL) erzeugt wird. Diese beinhaltet das framework-spezifische Verhalten für die Anwendung. Dafür werden in WAFL synthetische Methoden generiert, die die Ausführung der Anwendung simulieren. Mit WAFL2JAVA wird aus einer WAFL-Spezifikation Java Bytecode erzeugt, der mit dem Anwendungscode von einem Taint-Analyse-Framework hinsichtlich Sicherheitsverletzungen analysiert werden kann. Weitere Frameworks können hinzugefügt werden, indem Generatoren für diese hinzugefügt werden.

Der Ansatz von F4F und dieser Arbeit ist ähnlich: Ein Framework und eine Anwendung werden in künstlichen Methoden simuliert. Der Verarbeitungsprozess unterscheidet sich allerdings. F4F benötigt für jedes Framework einen Generator, der für eine Anwendung eine WAFL-Spezifikation erzeugt, die dann unabhängig verarbeitet wird. Hingegen wird in dieser Arbeit eine Sprache erstellt, durch die ein Framework ausgedrückt wird. Diese Framework-Spezifikation wird mit einer Anwendung verarbeitet.

2.6.4. Andromeda

Tripp *et al.* [49] haben mit Andromeda ein korrektes Taint-Analyse-Framework implementiert, das ein ähnliches Prinzip wie TAJ (in Kapitel 2.6.2) aufweist: Als Eingabe nimmt Andromeda eine Anwendung mit den abhängigen Bibliotheken und Sicherheitsregeln entgegen. Für jede Quelle einer Sicherheitsregel werden verschiedene Analysen, die erst bei Betrachtung der Sicherheitsregel auf relevantem Code ausgeführt werden, durchgeführt und dabei geprüft, dass Daten der Quelle nicht ohne Sanitizer zur Senke fließen. Für framework-basierte Anwendungen wird die WAFL-Spezifikation von F4F (Kapitel 2.6.3) zunächst zu Java Code umgesetzt, der das Verhalten des Frameworks modelliert.

Andromeda und diese Arbeit arbeiten im ersten Schritt nach einem ähnlichen Prinzip: Aus einer Spezifikation wird Code generiert, der für eine Anwendung das Framework simuliert. Andromeda verwendet hierbei eine WAFL-Spezifikation, die bereits eine Anwendung berücksichtigt, während durch die Sprache aus dieser Arbeit ein Framework unabhängig von der Anwendung spezifiziert wird (Kapitel 3.2). Die folgende Informationsflusskontrolle ist in Andromeda mit eingebunden, was in dieser Arbeit nicht der Fall ist.

3. Entwurf und Implementierung

In diesem Kapitel wird die Entwicklung der Sprache durch den Vergleich dreier Frameworks, die Sprache selbst und ihre prototypische Implementierung betrachtet.

3.1. Analyse der Frameworks

Zunächst wurden die Java Servlet Specification, Version 4.0[1], Swing[2] und JavaFX 8[3] analysiert und miteinander verglichen, um Ansätze für die Entwicklung einer Sprache zu gewinnen.

3.1.1. Übersicht über die Frameworks

Vor dem eigentlichen Vergleich erfolgt hier ein kurzer Überblick über die einzelnen Frameworks. Dieser beinhaltet den Lebenszyklus und die Funktionsweise des Frameworks.

Java Servlet Specification, Version 4.0

Die Java Servlet Specification, Teil von Java EE, Version 8, definiert eine API, die den Abruf web-basierter Inhalte ermöglichen [47]. Im Kern der Servlet API steht dabei die Schnittstelle `javax.servlet.Servlet`, die das Anfrage/Antwort-Verfahren (Request/Response) unterstützt.

Der Lebenszyklus eines `Servlets` besteht aus der Instanziierung und dem Aufruf der Methoden `init`, `service` und `destroy`, wobei die `service`-Methode beliebig oft aufgerufen werden kann [47]. Abbildung 3.1 zeigt eine graphische Übersicht

[1] `https://www.jcp.org/en/jsr/detail?id=369`
[2] `https://docs.oracle.com/javase/8/javase-clienttechnologies.htm`
[3] `https://docs.oracle.com/javase/8/javase-clienttechnologies.htm`

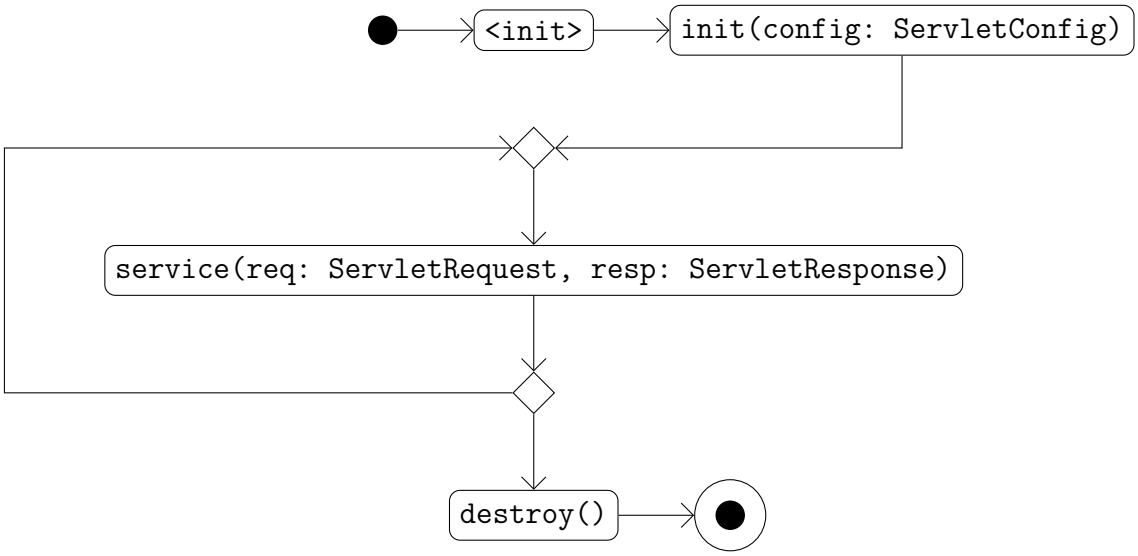


Abbildung 3.1.: Lebenszyklus eines `Servlets`

über den Lebenszyklus. Gesteuert wird dieser durch den Servlet-Container, in den ein `Servlet` installiert wird und der in der Regel Teil eines Web-Servers ist. Der Servlet-Container instanziiert ein `Servlet` genau einmal bei Bedarf, wenn die erste Anfrage für dieses eingegangen ist, und ruft direkt danach die `init`-Methode auf. Anschließend wartet das `Servlet` auf Anfragen, die vom Servlet-Container in ein `javax.servlet.ServletRequest` gepackt werden und mit einem `javax.servlet.ServletResponse` zur Ausgabe der Antwort an `service(ServletRequest, ServletResponse)` weitergegeben werden. Die tatsächliche Zahl an Anfragen hängt von den Zugriffen der Nutzer ab. Normalerweise existieren mehrere Threads, die für die Verarbeitung mehrerer Anfragen gleichzeitig auf das `Servlet` zugreifen können. Wird die Servlet-Anwendung beendet, ruft der Servlet-Container die `destroy`-Methode der `Servlets` auf.

Zusätzlich zum `Servlet` werden die abstrakten Klassen `javax.servlet.GenericServlet` und `javax.servlet.http.HttpServlet` definiert, die zusätzliche Funktionen bereitstellen [47]. So fungiert das `HttpServlet` als Oberklasse für `Servlets`, die HTTP unterstützen. Es besitzt eine spezielle `doX`-Methode für jede HTTP-Methode X, beispielsweise für GET `doGet` oder für POST `doPost`. Die `service`-Methode leitet die Verarbeitung an die `doX`-Methoden weiter.

Für Ereignisse, die während des Lebenszyklus eines `Servlets` entstehen, gibt es Schnittstellen, die die Schnittstelle `java.util.EventListener` erweitern, daher im Folgenden `EventListener` genannt werden und Methoden für die Verarbeitung der Ereignisse definieren [47]. Servlet-Anwendungen, die auf bestimmte Ereignisse reagieren möchten, benötigen dazu nur die passenden `EventListener` zu implementieren und zu registrieren. Die Registrierung kann über den deployment descriptor,

eine XML-Datei mit Auslieferungsdetails für `Servlets`, eine Annotierung der Implementierung mit der `javax.servlet.annotation.WebListener`-Annotation oder den Aufruf einer `addListener`-Methode erfolgen.

Im Kontext eines `Servlet`-Lebenszyklus werden `EventListener` beim Start des Servlet-Containers durch diesen geladen, instanziiert und registriert mit Ausnahme von `EventListenern`, die durch eine `addListener`-Methode registriert werden [47].

Besonders sind an dieser Stelle die `EventListener` `javax.servlet.ServletContextListener` und `javax.servlet.ServletRequestListener` zu beachten. Der `ServletContextListener` wird bei Erstellung bzw. Zerstörung des `javax.servlet.ServletContexts` vom Servlet-Container ohne Einfluss der Servlet-Anwendung aufgerufen [45]. Analog geschieht dies beim `ServletRequestListener` bei Erstellung oder Zerstörung eines `javax.servlet.ServletRequests` [46]. Bei Betrachtung der übrigen definierten `EventListener` ist deren Aufruf eine Folge der Servlet-Anwendung, die Funktionen der Servlet API nutzt und dabei Ereignisse auslöst.

Die Abbildung 3.2 zeigt einen erweiterten `Servlet`-Lebenszyklus mit `EventListenern`.

Swing

Swing ist Teil der Java API und ermöglicht die Erstellung graphischer Benutzerschnittstellen [29].

Für Swing gibt es keinen expliziten Lebenszyklus. Der Ablauf einer Swing-Anwendung ergibt sich allerdings aus der Dokumentation. Der empfohlene Start ist dabei ein Aufruf der statischen Methode `javax.swing.SwingUtilities.invokeLater` aus der `main`-Methode heraus [42]. `invokeLater` erhält ein `java.lang.Runnable`-Objekt, das auf dem Event Dispatch Thread ausgeführt wird und dann die graphische Benutzerschnittstelle erstellt und anzeigt. Hierbei stellt die Klasse `javax.swing.JFrame` als top-level Window das Fenster dar, in dem die graphische Benutzerschnittstelle angezeigt wird, und kann in einer Swing-Anwendung unterschiedlich benutzt werden, zum Beispiel als Oberklasse oder Attribut [37]. Über die Methode `setVisible` kann ein `JFrame` angezeigt werden, was als erstes Fenster den Start der Swing-Anwendung für den Nutzer markiert. Wird das letzte Fenster geschlossen oder die Swing-Anwendung anderweitig beendet, endet die Swing-Anwendung ohne den Aufruf einer speziellen, für das Ende bestimmten Methode.

Die graphische Benutzerschnittstelle besteht aus Komponenten, deren Klassen von `javax.swing.JComponent` erben [36] und die meistens nicht thread-sicher sind, weswegen das Zeichnen und die Ereignisverarbeitung auf dem Event Dispatch Thread

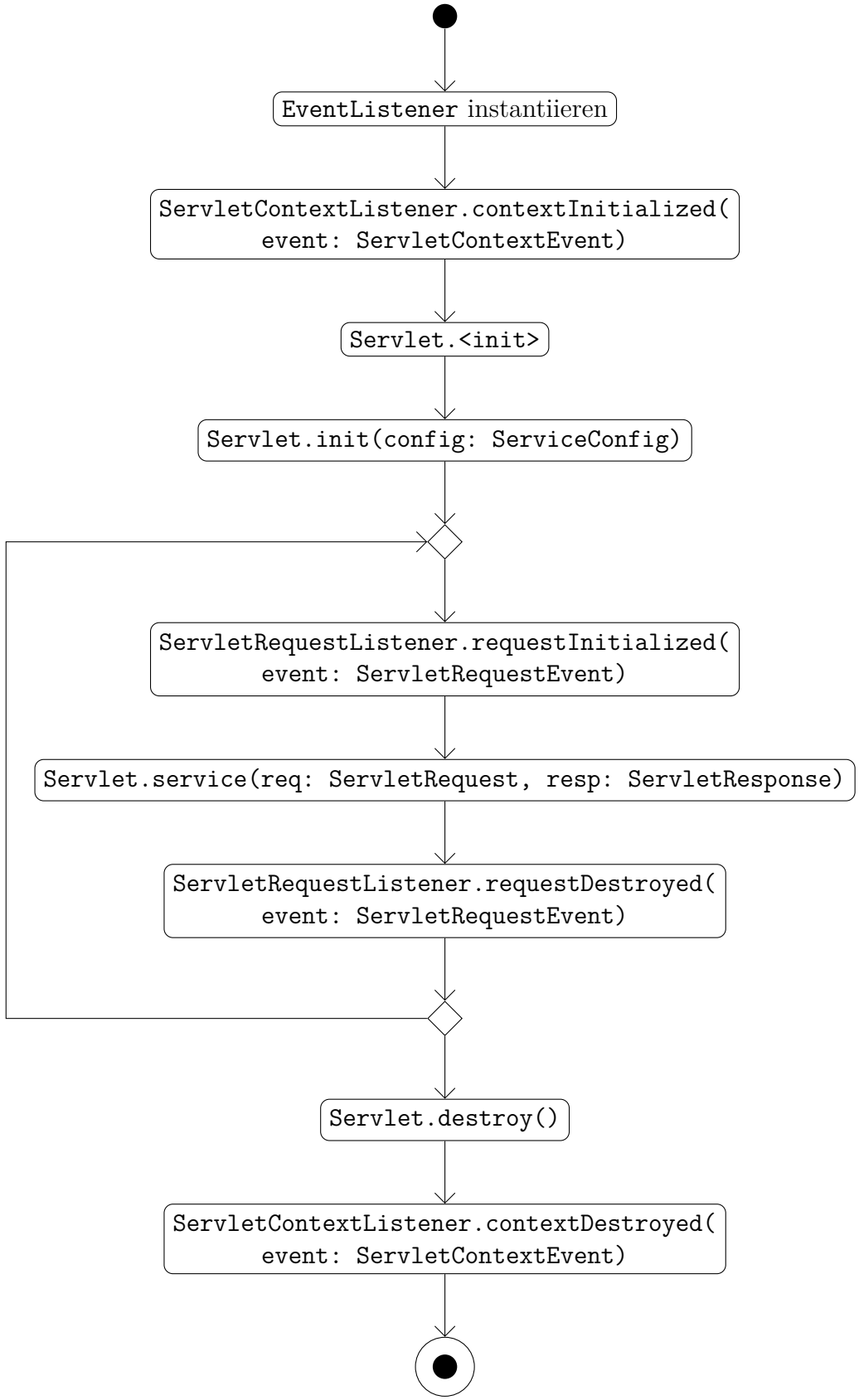


Abbildung 3.2.: Lebenszyklus eines `Servlets` unter Beachtung des Servlet-Containers, der `ServletContextListener` und `ServletRequestListener`

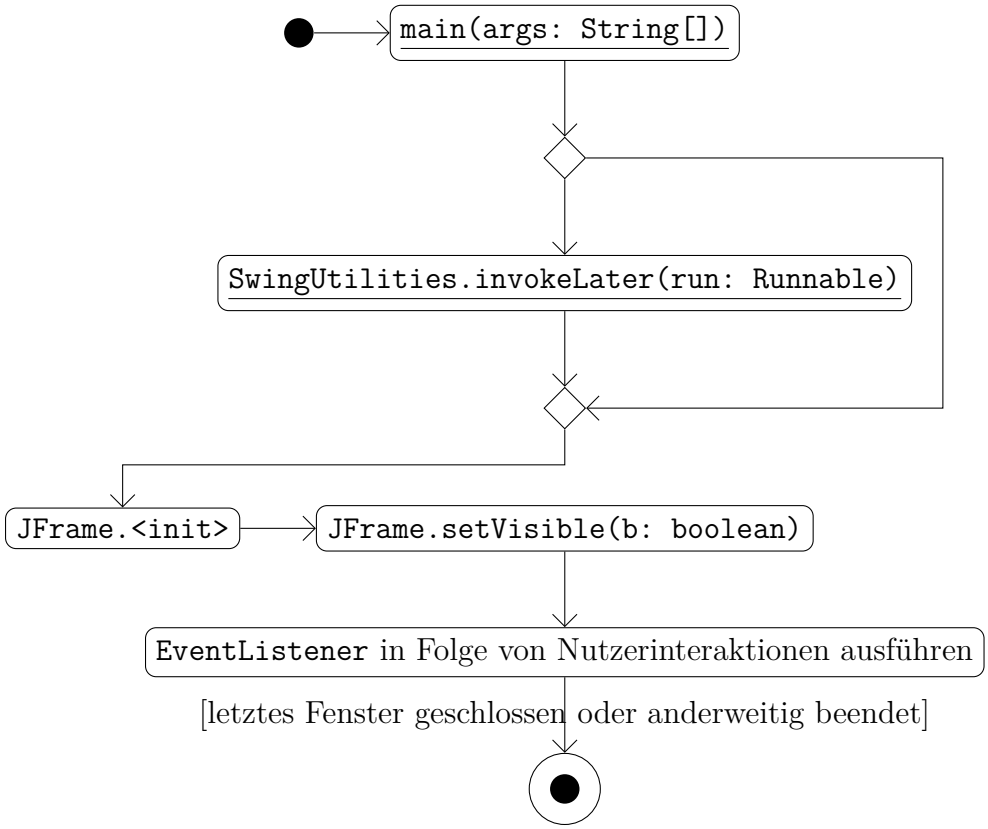


Abbildung 3.3.: Lebenszyklus einer Swing-Anwendung

abläuft [30].

Nachdem das erste `JFrame` dem Nutzer angezeigt wird, wartet es auf dessen Eingaben, die Ereignisse auslösen und von speziellen `EventListenern` verarbeitet werden. Dafür reihen sich die Ereignisse zunächst in die `java.awt.EventQueue` ein, die die Ereignisse über `dispatchEvent` sequentiell und Reihenfolge-treu direkt an die Komponente ausliefert, auf der das Ereignis ausgelöst wurde [35]. Die Zielkomponente verarbeitet schließlich das Ereignis, indem sie das Ereignis an die passenden `EventListener` übergibt. Eine Swing-Anwendung kann dazu jeder Komponente für ein bestimmtes Ereignis X, zum Beispiel das häufigste Ereignis Action, durch die Klasse `java.awt.event.ActionEvent` repräsentiert, mehrere `EventListener` über die Methode `addXListener` (für das Beispiel: `addActionListener`) hinzufügen [32].

Abbildung 3.3 zeigt einen graphischen Überblick über den Lebenszyklus einer Swing-Anwendung.

JavaFX 8

JavaFX 8 als Teil der Java API 8 ermöglicht ebenfalls die Erstellung graphischer Benutzerschnittstellen mit einfacher Medien- und Webnutzung [27].

Hierbei wird der Basis-Lebenszyklus einer JavaxFX-Anwendung durch die abstrakte `javafx.application.Application`-Klasse festgelegt und durch die JavaFX Runtime gesteuert [33]. Für den Start einer JavaFX-Anwendung gibt es drei Möglichkeiten: die Erstellung eines `javafx.embed.swing.JFXPanel`, wenn die JavaFX-Anwendung in eine Swing-Anwendung eingebunden wird, die empfohlene Variante über den Aufruf der statischen Methode `Application.launch` in der `main`-Methode oder über den direkten Aufruf der statischen Methode `javafx.application.Platform.startup`, die im Rahmen der ersten beiden Möglichkeiten ebenfalls aufgerufen wird. `Platform.startup` startet die JavaFX Runtime und erstellt eine neue Instanz der `Application`-Klasse, auf der die Methode `init` aufgerufen wird. Anschließend wird der JavaFX Application Thread erstellt, auf dem die JavaFX-Anwendung ab diesem Punkt weiter läuft. Dort wird die Methode `start` der `Application`-Klasse aufgerufen und die Anwendung beginnt ihre eigentliche Arbeit. Ist diese vollendet und die JavaFX-Anwendung wird über `Platform.exit` explizit oder durch das Schließen des letzten Fensters implizit zum Beenden aufgefordert, wird die `stop`-Methode der `Application`-Klasse aufgerufen. Damit endet die JavaFX-Anwendung.

Die erstellte graphische Benutzerschnittstelle wird durch den Szenengraphen als Baum repräsentiert [38]. Dessen Knoten sind graphische Benutzerelemente und Objekte von Unterklassen von `javafx.scene.Node`, auf denen, analog zur Java Servlet Specification, Version 4.0, und zu Swing, `EventListener` für die Verarbeitung verschiedener Ereignisse registriert werden können. Im Gegensatz zu den anderen beiden Frameworks kann für jedes Objekt und Ereignis nur ein `EventListener` registriert werden und zusätzlich definiert JavaFX 8 als einzigen `EventListener` die generische Schnittstelle `javafx.event.EventHandler`, die als Typ eine Unterklasse von `javafx.event.Event` entgegennimmt und damit das behandelte Ereignis festlegt [40].

Nach Anzeige der graphischen Benutzerschnittstelle, die mit dem Aufruf der `start`-Methode der `Application`-Klasse einhergeht, wartet die JavaFX-Anwendung auf Aktionen des Nutzers, die Ereignisse auslösen [28]. Dessen Zielobjekt im Szenengraph wird durch interne Regeln festgelegt. Damit wird auch die Route, die Event Dispatch Chain, im Szenengraphen vom Wurzelobjekt zum Zielobjekt berechnet, die zweimal durchlaufen wird. Zunächst durchläuft das Ereignis in der Event Capturing Phase die Event Dispatch Chain zum Zielobjekt und wird Filtern, ebenfalls `EventHandler`, übergeben. In der darauf folgenden Event Bubbling Phase steigt das Ereignis wieder zum Wurzelobjekt auf und wird den auf den `Nodes` eigentlich registrierten `EventHandlern` übergeben.

Der komplette Lebenszyklus einer JavaFX-Anwendung wird in Abbildung 3.4 graphisch dargestellt.

3.1.2. Vergleich der Frameworks

Dieser Abschnitt beinhaltet den eigentlichen Vergleich der Frameworks. Der Fokus liegt hierbei auf den Gemeinsamkeiten, die durch die Sprache ausgedrückt werden sollen.

Lebenszyklus

Für jedes Framework gibt es einen Lebenszyklus, der durch eine abstrakte Klasse oder eine Schnittstelle und dessen Methoden festgelegt ist. Eine Anwendung muss Implementierungen für die abstrakte Klasse oder Schnittstellen bereitstellen, die vom Framework verwaltet werden. So stellt `Servlet` die Schnittstelle für die Java Servlet Specification, Version 4.0, dar [47] und `Application` die abstrakte Klasse für JavaFX 8 [33]. Swing bildet hier eine Ausnahme, da es keine solche Klasse oder Schnittstelle bereitstellt und damit beliebige Klassen zulässt, die sich selbst verwalten können. Allerdings wird ein Fenster nur über ein `JFrame` angezeigt [37], sodass dieses als Swing-äquivalente Schnittstelle angesehen werden kann.

Wie zuvor bereits beschrieben, benötigen eine JavaFX- oder Swing-Anwendung eine `main`-Methode. Somit ist diese nicht immer im Framework zu erwarten.

Zudem unterteilt sich der Lebenszyklus in drei Phasen: eine feste Anfangs- und Endphase und eine Arbeitsphase. Die Anfangsphase eines `Servlets` besteht zum Beispiel aus der Instanziierung und der Methode `init` und die Endphase aus der Methode `destroy` [47]. Für JavaFX-Anwendungen ist die Anfangsphase die Instanziierung der `Application`-Klasse mit den Methoden `init` und `start` und die Endphase die Methode `stop` [33]. Für Swing kann die Anfangsphase als der Aufruf der `main`-Methode bis zur Anzeige des ersten `JFrame` angesehen werden. Eine feste Endphase gibt es in diesem Sinne nicht. Die Arbeitsphase besteht bei allen drei Frameworks schließlich daraus, dass sie auf Nutzereingaben warten. Bei `Servlets` führen diese zu einem Aufruf der `service`-Methode [47] und bei JavaFX- und Swing-Anwendungen zu Aufrufen von `EventListenern`. Damit ist die tatsächliche Arbeit des Frameworks und der Anwendung vom Nutzer abhängig, was sie nicht-deterministisch macht.

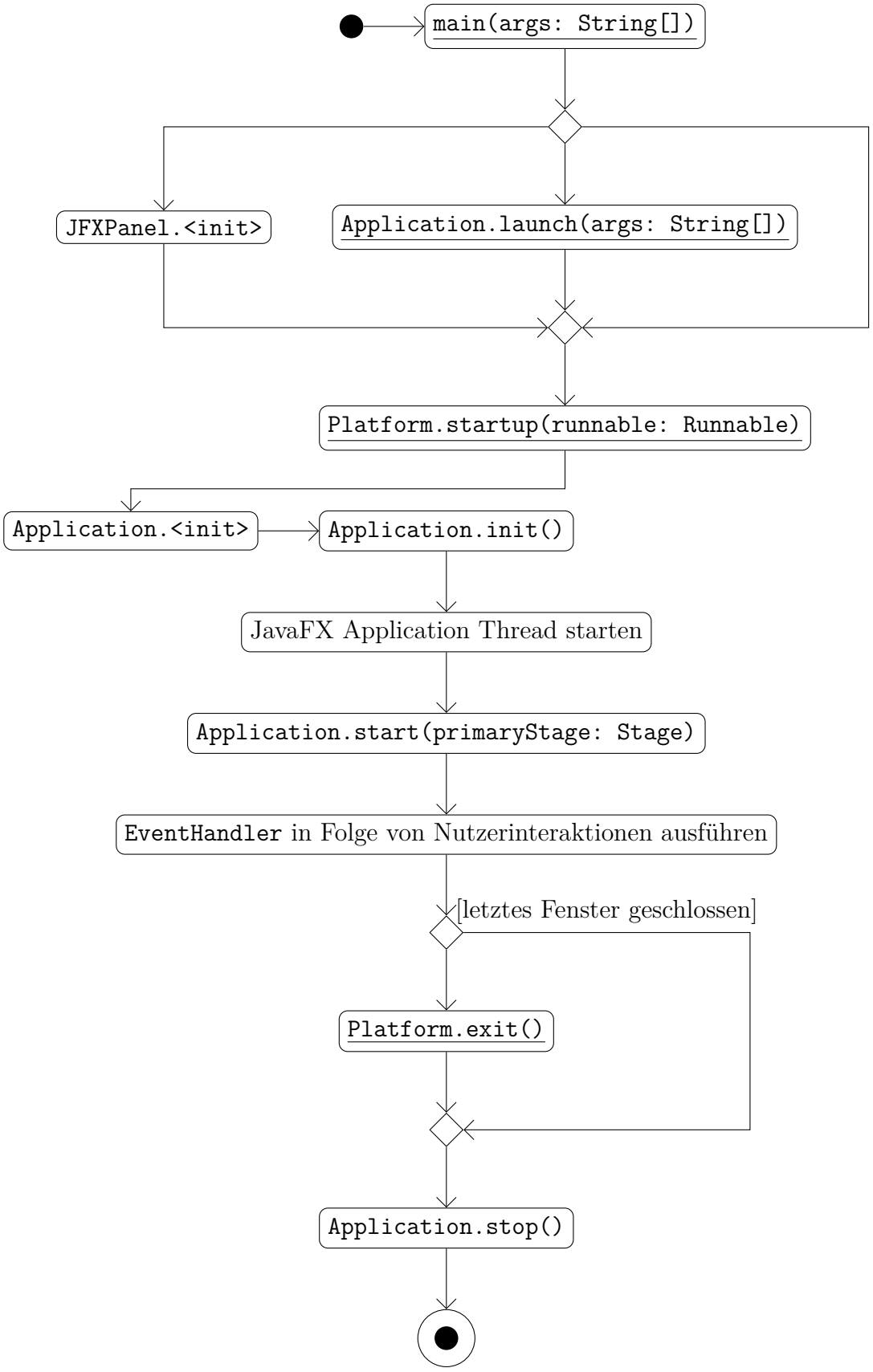


Abbildung 3.4.: Lebenszyklus einer JavaFX-Anwendung, speziell der `Application`-Klasse und der `EventHandler`

API

Jedes Framework stellt eine oder mehrere APIs bereit, mit denen Anwendungen auf weitere Funktionen und Funktionalitäten des Frameworks zugreifen und den Lebenszyklus beeinflussen können. Teilweise werden auch Objekte der API-Klassen als Methoden-Parameter übergeben oder stehen über `get`-Methoden zur Verfügung. In der Java Servlet Specification, Version 4.0, werden den `Servlets` in der `service`-Methode beispielsweise ein `ServletRequest` und `ServletResponse` übergeben [47]. Zudem erlaubt die Methode `getServletConfig` der Klasse `Servlet` den Zugriff auf die `ServletConfig` und die Methode `getServletContext` der Klasse `ServletConfig` Zugriff auf den `ServletContext` [44]. Der `ServletContext` wiederum besitzt die Methoden `createServlet` und `addServlet`, mit denen eine Servlet-Anwendung neue `Servlets` erstellen kann. Ähnlich wird in JavaFX-Anwendungen das erste Fenster, ein Objekt vom Typ `javafx.stage.Stage`, als Parameter an die `start`-Methode der `Application`-Klasse übergeben und der Anfang und das Ende der JavaFX-Anwendung kann über `Platform.startup` und `Platform.exit` gesteuert werden [33]. Ansonsten lassen JavaFX 8 und Swing den Anwendungen große Freiheiten in der Benutzung der APIs. So können zum Beispiel `Nodes` und `Stages` in JavaFX 8 und Komponenten und `JFrames` in Swing beliebig oft instanziiert und angezeigt werden. Insgesamt zeigt sich bei allen drei Frameworks, dass `EventListener` immer das Ereignis als Parameter übergeben bekommen.

Nebenläufigkeit

Nebenläufigkeit ist essentiell in allen drei Frameworks. Dabei ist die Thread-Verwaltung im Framework verankert und verborgen.

In JavaFX 8 und Swing werden beispielsweise auf separaten Threads die graphischen Benutzerschnittstellen gesteuert und Ereignisse verarbeitet. Zudem stellen sie APIs und Hilfsklassen für parallele Operationen bereit. Dazu gehört in Swing der `javax.swing.SwingWorker` [39], der zuerst über die Methode `doInBackground` Arbeit auf einem Arbeitsthread ausführt und anschließend über `done` die Aktualisierung der graphischen Benutzerschnittstelle auf dem Event Dispatch Thread ermöglicht.

Nach der Java Servlet Specification, Version 4.0, werden Anfragen an `Servlets` parallel verarbeitet, sodass bei Anfragen an dasselbe `Servlet` dieses parallel ausgeführt wird [47].

`EventListener`

Während der Arbeit aller drei Frameworks treten Ereignisse auf, die eine Anwendung verarbeiten kann. Dazu stellen die Frameworks passende Methoden bereit, wobei es für jedes Ereignis eine Schnittstelle, die `EventListener` erweitert, mit passenden Methoden gibt. `EventListener` selbst definiert keine Methoden [41].

In Swing gibt es neben den Schnittstellen auch Implementierungen für Standardoperationen. Zum Beispiel stellt `javax.swing.text.DefaultEditorKit.CopyAction` ein `ActionListener` für das Kopieren von Daten in den Zwischenspeicher dar, den Anwendungen nutzen können [34].

JavaFX 8 bildet eine Ausnahme, da es nur den generischen `EventHandler` zur Verfügung stellt, der als Typ das zu behandelnde Ereignis entgegen nimmt [40].

Auch die Java Servlet Specification, Version 4.0, beinhaltet eine Besonderheit: Der Aufrufzeitpunkt der `EventListener` in Abhängigkeit des Lebenszyklus steht fest. Für Näheres sei hier auf die Übersicht über die Java Servlet Specification, Version 4.0 (in Kapitel 3.1.1), verwiesen.

Die Verteilung der Ereignisse im Framework unterscheidet sich zwischen den Frameworks. Bei Swing und der Java Servlet Specification, Version 4.0, werden die Ereignisse direkt an die `EventListener` ausgeliefert [47, 31]. In JavaFX 8 durchlaufen die Ereignisse den Szenengraphen und rufen mehrere `EventHandler` auf [28].

Beziehungen zwischen Objekten

Zwischen verschiedenen Objekten haben sich Beziehungen gezeigt, die sich durch Gleichheit ausdrücken lassen.

Innerhalb der `service`-Methode eines `Servlets` ist der Rückgabewert von `getServletConfig().getServletContext()` des `Servlets` gleich dem Rückgabewert von `getServletContext()` des übergebenen `ServletRequests` [47].

Für ein `java.awt.event.Event`, das einer `EventListener`-Methode in Swing übergeben wird, ist der Rückgabewert von `getSource()` gleich der Komponenten, auf der der `EventListener` registriert ist [31].

3.1.3. Folgerungen aus dem Vergleich

Im Vergleich hat sich die Aufteilung eines Lebenszyklus in die Anfangs-, Arbeits- und Endphase gezeigt. Dabei sind die Anfangs- und Endphase im Hinblick auf die beteiligten Klassen, Methoden und Aufrufreihenfolge fest modelliert, sodass sie explizit angegeben werden können.

Die Arbeitsphase verläuft nicht-deterministisch in Abhängigkeit vom Benutzer und nebenläufig. Damit ist hier das Ziel, das Benutzerverhalten, das Framework und die Nebenläufigkeit zu simulieren. Entsprechend sind auch framework-spezifische Regeln im Ablauf zu beachten.

Zusätzlich werden in der Arbeitsphase über Schnittstellen, die ein gemeinsames Muster aufweisen, die aufgerufenen Methoden bestimmt. Daher können Klassen und Methoden über Muster identifiziert oder explizit angegeben werden.

3.2. Die Sprache

Aus den Gemeinsamkeiten der analysierten Frameworks wurde eine Sprache entwickelt, die zur Beschreibung von Frameworks in Form von Framework-Spezifikationen in der entwickelten Sprache dient und deren Syntax und Umsetzung im nachfolgenden beschrieben wird.

3.2.1. Syntax

Die Syntax der Sprache besteht aus den folgenden Elementen. Hierbei bilden die ersten zwei Elemente Grundkonzepte der Sprache. Wie sich die Arbeit der Frameworks in eine Anfangs-, Arbeits- und Endphase gliedert, unterteilt sich eine Framework-Spezifikation ebenfalls in diese Teile, von denen alle drei mindestens einmal angegeben werden müssen.

Explizite Angabe

Eine explizite Angabe ist eine Liste von Methodenaufrufen, deren Reihenfolge durch die Liste gegeben ist und die entsprechend in dieser Reihenfolge erfolgen.

Der Aufruf einer statischen Methode besteht aus der Klasse und der statischen Methode.

Bei Methodenaufrufen auf Objekten werden die Methoden der Objekte, die für die explizite Angabe relevant sind, in der Reihenfolge ihres Aufrufes angegeben. Die Methoden sind an eine Klasse, die die Methodenaufrufe umgibt, gebunden. Die Klasse wird entweder durch einen äußeren Mechanismus oder durch die optionale Angabe einer Klasse aus dem Framework als Teil der expliziten Angabe vorgegeben. Bei der Angabe einer Klasse als Teil der expliziten Angabe wird ein „für alle“ mit angegeben. Damit wird klar gekennzeichnet, dass alle Aufrufe auf allen Instanzen dieser Klasse stattfinden. Wird in den Methoden ein Konstruktoraufruf angegeben, muss dieser als erstes angegeben werden. Dabei wird eine Instanz jeder Anwendungs-Klasse, die von der gebundenen Klasse erbt, erstellt und verwendet.

Eine explizite Angabe kann an jeder Stelle eine weitere explizite Angabe beinhalten.

Regeln

Eine Regel kann ein regulärer Ausdruck, die Angabe einer Klasse oder eines Interfaces als Supertyp oder eine Block-Regel sein.

Eine Block-Regel besteht aus einer Klasse, einem Quantor (momentan wird nur „für alle“ unterstützt, weitere können hinzugefügt werden) und dem eigentlichen Inhalt. Mit „für alle“ wird für jede Instanz, dessen Typ oder Supertyp die Klasse ist, ein Block mit dem Inhalt gebildet. Der Inhalt kann hierbei entweder eine Block-Regel oder eine explizite Angabe sein. Im Falle einer expliziten Angabe werden die Methoden in der expliziten Angabe an die Klasse der Block-Regel gebunden.

Anfangsphase

Die Anfangsphase besteht aus einer nicht-leeren Menge von expliziten Angaben, die alle eine Möglichkeit darstellen, wie das Framework beginnt und Anwendungscode aufruft. Es wird angenommen, dass nur eine Möglichkeit durch die Anwendung realisiert ist und daher bei Ausführung der Anwendung nur eine Möglichkeit ausgeführt wird. Eine Möglichkeit wird aus der Menge nicht-deterministisch ausgewählt.

Optional kann durch die Angabe der main-Methode deren Position im Kontext des Lebenszyklus festgelegt werden. Der Nutzer hat die Möglichkeit, von eventuell mehreren vorhandenen main-Methoden in verschiedenen Klassen eine spezielle Klasse anzugeben, die für die Analyse genutzt wird.

Arbeitsphase

Eine Arbeitsphase besteht aus einer Menge von Regeln, die alle mit derselben Priorität behandelt werden. Aus den Regeln wird eine Menge aus Methoden und Blöcken gebildet, die durch ihre Ausführung nicht-deterministisches Nutzerverhalten simulieren.

Für jede Arbeitsphase muss angegeben werden, ob sie single- oder mutli-threaded ist. Bei einer single-threaded Arbeitsphase läuft diese in einem Thread ab. Der Anwendungscode kann dennoch weitere Threads starten oder Framework-Funktionen nutzen, die das Erstellen und Starten weiterer Threads zur Folge hat. Bei einer mutli-threaded Arbeitsphase werden explizit mehrere Threads mit demselben Arbeitsablauf erstellt und gestartet, da die Abläufe parallel stattfinden.

Es ist möglich, mehrere Arbeitsphasen anzugeben. Es wird angenommen, dass sie in der Reihenfolge ihrer Angabe auch im Framework ablaufen, weswegen sie in der angegeben Reihenfolge abgearbeitet werden. Über die Angabe mehrerer Arbeitsphasen kann auch eine Reihenfolge von Regeln festgelegt werden.

Endphase

Die Endphase besteht aus genau einer expliziten Angabe.

3.2.2. Umsetzung zu Code

Wie die einzelnen Elemente der Sprache umgesetzt werden, wird hier beschrieben.

Dabei gilt allgemein für alle Methodenaufrufe: Bevor der Aufruf geschieht, werden alle Argumente instanziiert.

Nicht-deterministische Schleife / Bedingung

Nicht-deterministische Schleifen bzw. Bedingungen bezeichnen Schleifen bzw. if-else-if-Bedingungen, die nicht-deterministisches Verhalten simulieren und deren Inhalt beliebig ist.

Bei einer nicht-deterministischen Schleife wird eine zufällige Zahl als Zahl der Iterationen gewählt.

Bei einer nicht-deterministischen Bedingung wird eine zufällige Zahl zur Entscheidung für einen Zweig der Bedingung ausgewertet.

Explizite Angabe

Ist die explizite Angabe an keine Klasse gebunden, wird die explizite Angabe genau einmal umgesetzt.

Aus einem Aufruf einer statischen Methode wird ein Aufruf auf der Klasse mit der angegebenen statischen Methode.

Bei Methodenaufrufen auf Objekten werden zu verwendende Instanzen, wenn kein Konstruktoraufruf enthalten ist, oder zu verwendende Klassen, wenn ein Konstruktoraufruf enthalten ist, eingegeben. Ist ein Konstruktoraufruf enthalten, wird jede Klasse genau einmal mit dem Konstruktoraufruf am Anfang instanziiert und die dabei erstellte Instanz in den folgenden Methodenaufrufen verwendet. Mit einer for-Schleife wird über jedes Element der Eingabe iteriert und alle angegebenen Methoden in der Reihenfolge aufgerufen.

Beinhaltet die explizite Angabe die optionale Angabe einer Klasse, an die die explizite Angabe gebunden ist, werden ohne enthaltenen Konstruktoraufruf alle Instanzen aus der Anwendung, die von der Klasse erben bzw. im Falle eines Interfaces dieses implementieren, und alle erstellten Instanzen, wenn eine frühere explizite Angabe einen Konstruktoraufruf für die Klasse oder einen Supertypen dieser Klasse beinhaltet, eingegeben. Mit Konstruktoraufruf wird jede Anwendungs-Klasse, die von der Klasse erbt, eingegeben.

Im Falle einer weiteren, geschachtelten expliziten Angabe wird diese am angegebenen Punkt wie hier beschrieben umgesetzt.

Regeln

Bei Angabe einer Block-Regel wird mit dem Quantor „für alle“ für jede Instanz aus der Anwendung und jede erstellte Instanz, die als Supertyp die angegebene Klasse habt, ein Block gebildet. Beinhaltet die Block-Regel eine explizite Angabe, wird diese für jeden Block wie oben beschrieben mit der Instanz des Blockes als Eingabe umgesetzt. Beinhaltet die Block-Regel eine weitere, innere Block-Regel, wird diese ebenfalls wie hier beschrieben umgesetzt. Dabei unterteilen sich die Blöcke der äußeren Block-Regel in weitere Blöcke mit den Instanzen der inneren Block-Regel.

Bei Angabe eines regulären Ausdrucks werden alle Methoden, die im Framework deklariert oder im Framework Methoden von Supertypen außerhalb des Frameworks überschreiben und deren Methodennamen mit dem regulären Ausdruck übereinstimmen, ausgewählt.

Bei Angabe eines Supertyps werden alle Methoden ausgewählt, die in der angegebenen Klasse oder Interface und Subtypen im Framework deklariert sind oder Methoden von Supertypen außerhalb des Frameworks überschreiben.

Die durch die regulären Ausdrücke und Supertypen ausgewählten Methoden werden mit allen Instanzen aus der Anwendung gesammelt, deren Klassen von einer der Klassen, die ausgewählte Methoden beinhalten, erben oder eines der Interfaces, das ausgewählte Methoden beinhaltet, implementieren.

Anfangsphase

Es gibt eine nicht-deterministische Bedingung über alle expliziten Angaben der Anfangsphase, die wie oben beschrieben umgesetzt werden.

Arbeitsphase

Jede Arbeitsphase wird für sich separat wie hier beschrieben umgesetzt. Im umgesetzten Code sind Arbeitsphasen in der Reihenfolge vorhanden, in der sie angegeben wurden.

Zunächst werden die Methoden (und zugehörige Instanzen) aus Regeln und Blöcke aus Block-Regeln gesammelt. Anschließend wird eine Runnable-Klasse erstellt, deren run-Methode aus einer nicht-deterministischen Schleife besteht. Innerhalb der nicht-deterministischen Schleife gibt es eine nicht-deterministische Bedingung über alle gesammelten Methoden und Blöcke. Mit der Angabe von multi-threaded wird in einer for-Schleife mehr als ein Thread erstellt, die jeweils eine eigene Instanz der zuvor erstellten Runnable-Klasse erhalten, und gestartet. Bei Angabe von single-threaded wird eine Instanz der Runnable-Klasse direkt ausgeführt.

Endphase

Die explizite Angabe der Endphase wird wie oben beschrieben umgesetzt.

3.2.3. Ressourcen-Lader

Als Erweiterung der Sprache und für das Laden von zusätzlichen Ressourcen neben dem Anwendungscode kann in einer Framework-Spezifikation optional zuerst eine Datei mit kompilierten Java-Code referenziert werden, die eine vorgegebene Schnittstelle implementiert. Die Ressourcen werden nach einer ersten Analyse geladen, damit deren Informationen Einfluss auf das weitere Vorgehen nehmen und besser die Eigenschaften des Frameworks abbilden können.

3.3. Implementierung

Die Sprache und ihre Umsetzung wurde im Rahmen von „Joana and Frameworks (JoFrames)“[4] implementiert (Paket-Präfix ist `edu.kit.ipd.pp.joframes`, der im Folgenden weggelassen wird). Als konkrete Syntax für die Sprache wurde eine Beschreibung einer Framework-Spezifikation durch XML gewählt. Anhang A.1 zeigt als Beispiel die Framework-Spezifikation der Java Servlet Specification, Version 4.0. Dazu existiert ein XML-Schema, das die Form der XML-Dateien vorgibt. Mit der Klasse `api.logging.Log` besteht außerdem die Möglichkeit, die Analyse von JoFrames ohne Ausgabe oder durch eine Ausgabe auf die Standardausgabe oder in eine Datei zu loggen.

JoFrames stellt als API die Klasse `api.Pipeline` bereit, die eine Framework-Spezifikation, Jar-Dateien mit dem Framework und einer Anwendung, einer optionalen Jar-Datei als Ausgabe und einer optionalen Klasse mit einer `main`-Methode für die Verarbeitung der Anwendung entgegen nimmt. JoFrames erwartet für das Framework den vollständigen Bytecode und im Falle, dass wie bei der Java Servlet Specification, Version 4.0, das Framework nur als API vorliegt, eine Framework-Implementierung. Über `process()` wird die eigentliche Verarbeitung in drei Schritten gestartet.

Im ersten Schritt wird die Framework-Spezifikation intern an eine Instanz des `api.FrameworkSpecificationParser` weitergegeben, die die XML-Datei zunächst gegen das Schema validiert, anschließend einliest und dabei einen abstrakten Syntaxbaum generiert. Abbildung 3.5 zeigt einen Überblick über die Klassenhierarchie und den Aufbau des abstrakten Syntaxbaums. Dieser wird in Form eines `ast.Framework`-Objektes an die `Pipeline` zurückgegeben.

Im zweiten Schritt gibt die `Pipeline` das `Framework`-Objekt mit den Jar-Dateien des Frameworks und der Anwendung und der optionalen Klasse mit einer `main`-

[4] `https://github.com/HansMartinA/joframes`

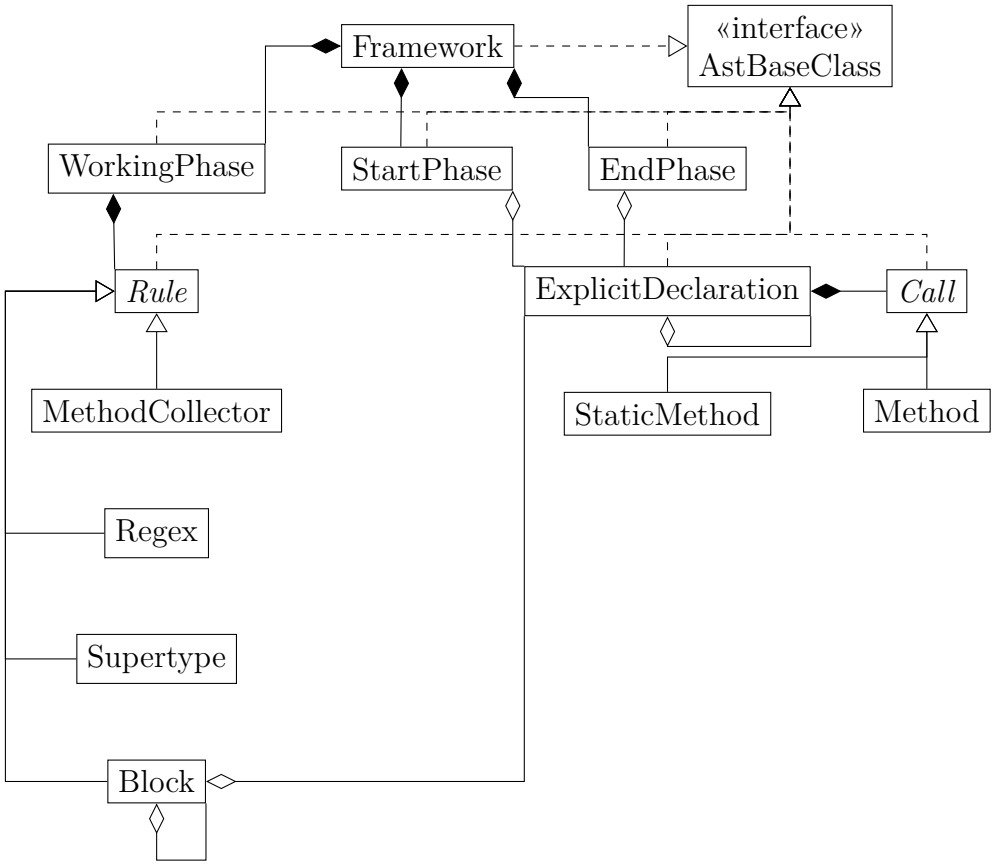


Abbildung 3.5.: Überblick über die Klassenhierarchie des abstrakten Syntaxbaums

Methode an eine Instanz von `api.ClassHierarchyAnalyzer` weiter. Dort wird mit Hilfe von WALA die Klassenhierarchie für die Java APIs, das Framework und die Anwendung erstellt, die nach den Klassen und Methoden aus der Framework-Spezifikation hin analysiert wird. Die Ergebnisse werden direkt im abstrakten Syntaxbaum gespeichert. Die einzige Ausnahme bilden die Regeln `ast.ap.Regex` und `ast.ap.Supertype` aus der Arbeitsphase. Diese werden pro Arbeitsphase durch die Regel `ast.acha.MethodCollector` ersetzt. Die erzeugte Klassenhierarchie wird mit dem `Framework`-Objekt und einer Sammlung aller in der Analyse gefundenen Klassen aus dem Framework in einer Instanz des `api.FrameworkWrapper` an die `Pipeline` zurückgegeben.

Im dritten Schritt erfolgt die eigentliche Bytecode-Erzeugung im `BytecodeInstrumenter`. Dazu bekommt dieser den `FrameworkWrapper` mit den Jar-Dateien des Frameworks und der Anwendung und der optionalen Jar-Datei zur Ausgabe.

Zunächst wird der Anwendungscode dahin gehend instrumentiert, dass Instanzen von Klassen aus dem Framework, die während der Klassenhierarchieanalyse gefunden wurden, oder deren Unterklassen in statischen Listen im `api.external.InstanceCollector` gesammelt werden, damit die Instanzen dem generierten Bytecode zur Verfügung stehen. Anwendungen können die Instanzen in Form von anonymen oder privaten inneren Klassen oder mit spezifischen Parametern aus der Anwendung erzeugen und in privatem Rahmen speichern, sodass von außen ohne die Instrumentierung kein Zugriff möglich ist. Durch die alternative, anwendungsunabhängige Erzeugung der Instanzen mit beliebigen Parametern können Informationsflüsse verborgen bleiben, da gerade die speziellen Parameter aus der Anwendung Informationsflüsse innerhalb der Anwendung ermöglichen.

Mit dem `InstanceCollector` und der `api.external.ArtificialClass` steht ein Gerüst für die Bytecode-Erzeugung bereit. Die `ArtificialClass` besitzt eine `main`-Methode, von der aus die Methoden `start`, `working` und `end` aufgerufen werden, die die entsprechenden Phasen des Lebenszyklus des Frameworks repräsentieren. In `start` und `end` wird Bytecode entsprechend der vorgesehenen Umsetzung erzeugt. Für die `working`-Methode ergibt sich eine abgewandelte Form. In der `working`-Methode werden nacheinander die verschiedenen Arbeitsphasen behandelt, wobei im Wesentlichen passende `Runnable`-Objekte der vorgegebenen inneren Klasse `WorkingWorker` der `ArtificialClass` für die Arbeitsphasen instanziiert und aufgerufen werden. Der `WorkingWorker` nimmt unter anderem die Nummer der Arbeitsphase entgegen, für die das Objekt zuständig ist und ruft, auf der Nummer basierend, eine passende Methode der `ArtificialClass` auf. Der Name der Methode weist dabei die Form `w + Nummer der Arbeitsphase` auf. Für jede Arbeitsphase wird eine solche Methode der `ArtificialClass` mit dem eigentlichen Inhalt, der nicht-deterministischen Schleife, hinzugefügt.

Bevor eine Methode der Anwendung im generierten Bytecode aufgerufen wird, werden

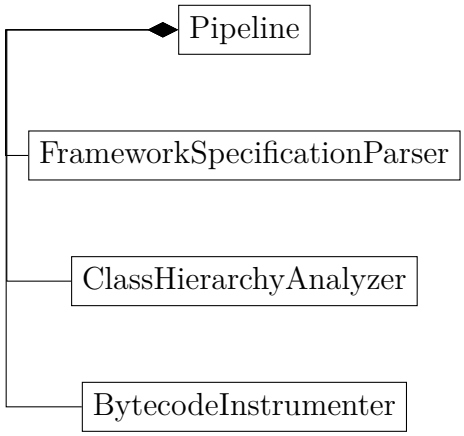


Abbildung 3.6.: Überblick über die *Pipeline*

alle Parameter instanziiert. Im Falle primitiver Typen wird die Konstante 0 geladen. Für ein Objekt wird in der Klassenhierarchie nach der ersten Klasse gesucht, die einen Konstruktor und keine Unterklassen besitzt, um so die Wahrscheinlichkeit für eine konkrete Klasse zu erhöhen. Aus unbekannten Gründen werden nämlich konkrete Klassen aus dem Framework als abstrakt gekennzeichnet. Benötigt der gewählte Konstruktor ebenfalls Parameter, werden diese rekursiv erzeugt. Um lange Ketten an Parameterinstanziierungen und Nicht-Determination durch zirkuläre Abhängigkeiten zu vermeiden, werden die Parameter nur bis zu einer Tiefe von drei rekursiven Schritten erzeugt. Anschließend wird die Konstante `null` für Objekte geladen.

Der `BytecodeInstrumenter` verwendet eine eigens entwickelte API, die auf dem Shrike-Modul von WALA, mit dem Bytecode gelesen und geschrieben werden kann [2], aufsetzt, deren Details abstrahiert und den Code lesbarer macht. In Shrike werden neue Bytecode-Instruktionen über `com.ibm.wala.shrikeBT.MethodEditor.Patch`-Objekte eingefügt, die im `BytecodeInstrumenter` zu einer hohen Zahl an anonymen Klassen geführt hätten.

Die instrumentierten Klassen aus `api.external` und der Anwendung werden mit den übrigen Klassen der Anwendung und des Frameworks in eine Jar-Datei, entweder der Ausgabedatei oder, wenn keine angegeben ist, in eine Standarddatei, gepackt. Diese steht anschließend für die Analyse mit JOANA bereit.

Die Abbildung 3.6 gibt nochmals einen Überblick über die beteiligten Klassen der `Pipeline`.

4. Evaluation

Mit diesem Kapitel wird näher die Evaluation JoFrames beleuchtet. Dabei wurden verschiedene Testfälle ausgeführt, um die Funktion und Leistung von JoFrames zu untersuchen. Zudem wurden weitere Frameworks hinsichtlich der Erweiterbarkeit der Sprache und JoFrames analysiert.

4.1. Tests

Zunächst wurde untersucht, ob und inwiefern JoFrames hilft, Sicherheitsverletzungen zu erkennen. Dazu wurde eine Reihe von Testfällen für die einzelnen Frameworks erstellt, ausgeführt und ausgewertet.

4.1.1. Testfälle

Für die Evaluation wurden 61 Testfälle verwendet. Hierbei teilen sich die Testfälle auf 48 Servlet-Testfälle, fünf Swing-Testfälle, vier JavaFX-Testfälle und vier Leistungstests auf.

Von den 48 Servlet-Testfällen stammen 43 von SecuriBench Micro 1.08[1], das eine Menge kleiner Testfälle für das Testen statischer Sicherheitsanalysierer bereitstellt [1]. Jeder Testfall beschränkt sich dabei auf ein `Servlet` und deckt eine Möglichkeit ab, wie es zu einer Sicherheitsverletzung kommen kann. Zu den Möglichkeiten, die hier genutzt wurden, gehören das intra- und inter-prozedurale Aliasing von Variablen, die Nutzung von Arrays, `java.util.Collection`s und eigens angelegten Datenstrukturen, die Aufteilung der Ausführung auf verschiedene Methoden (Testfälle **Inter***), bedingte Ausführungspfade (Testfälle **Pred***), die Verwendung von Sanitizern und verschiedener API-Funktionen (Testfälle **Basic***, **Session2**) und **StrongUpdates***.

[1] `https://suif.stanford.edu/~livshits/work/securibench-micro/`

Zu den übrigen fünf Servlet-Testfällen zählt das `IPServlet`, das bereits in der Einführung (Kapitel 1) vorgestellt wurde. Im Testfall **Conf1** kommt zusätzlich zu einem `Servlet` ein `ServletContextListener` hinzu, der ein geheimes Attribut im `ServletContext` setzt, das vom `Servlet` in die Antwort geschrieben wird. Der Testfall **Conf2** erweitert **Conf1** um einen `ServletRequestListener`, der das geheime Attribut vor der Ausführung des `Servlets` überschreibt, sodass es zu keiner Sicherheitsverletzung kommt. Ein ähnliches Prinzip kommt auch im Testfall **Int2** zum Einsatz. Hier wird ein geheimes Attribut durch einen `ServletContextListener` nach Erstellung des `ServletContexts` gesetzt, das von einem zweiten `ServletContextListener` bei der Zerstörung des `ServletContexts` verarbeitet wird. Zwischendurch kann das Attribut durch ein `Servlet` überschrieben werden. Im Testfall **Int1** wird auch die Integrität verletzt, indem Daten aus einem öffentlich verfügbarem `Servlet` in einem zweiten, eingeschränkt zugänglichen `Servlet` verarbeitet werden.

Der Aufbau der fünf Swing-Testfälle ist ähnlich: Die graphischen Benutzerschnittstellen bestehen aus Textfeldern und Knöpfe, deren `EventListener` einen Informationsfluss induzieren. In **Conf1** wird ein geheimer Wert durch das Drücken eines Knopfes in einem Textfeld angezeigt. In **Conf2** ist die Eingabe eines Passworts im `javax.swing.JPasswordField`, einem spezialisierten Textfeld, möglich, das nach Drücken eines Knopfes ausgegeben wird. Der Testfall **ExtendedPassword** erweitert die Passworteingabe um die Eingabe eines Nutzernamens, die beide über jeweils einen eigenen Knopf geprüft werden können. Mit dem Passwort wird hierbei nichts unternommen, während der Nutzername über das Internet übertragen wird. **NoIf** stellt ein Testfall dar, bei dem eine Textfeldeingabe eine Ausgabe beeinflusst, die wieder überschrieben wird und daher keinen illegalen Informationsfluss verursacht. Der **SwingWorker**-Testfall nutzt einen `SwingWorker` zur Simulation einer lang laufenden Operation, die am Ende eine Ausgabe vornimmt, die ein Geheimnis beinhaltet.

Für fast jeden Swing-Testfall gibt es einen vom Prinzip her äquivalenten JavaFX-Testfall. So sind die **ExtendedPassword**- und **Conf1**-Testfälle, **Conf2** und **Password** sowie **SwingWorker** und **Worker** äquivalent.

Die Leistungstests zielen darauf ab, JoFrames mit realen Anwendungen zu testen und dabei die Leistung zu messen. Hierfür wird als Servlet-Anwendung die HTTP-Server-Implementierung von JGit 5.1.1.201809181055-r[2], einer Git-Implementierung in Java, genutzt [19]. Als Swing-Anwendung kommt JPass 0.1.17-SNAPSHOT[3] zum Einsatz. JPass ist eine Passwortverwaltungssoftware [11]. Als JavaFX-Anwendungen werden die Simulation View und Monitoring View von OPC UA Simulator for Industrial Plants (OSIP) 1.1[4] genutzt. Die OSIP Simulation View simuliert eine industrielle

[2] `https://www.eclipse.org/jgit/`
[3] `https://github.com/gaborbata/jpass`
[4] `https://github.com/ByteHamster/PSE`

Produktionsanlage, von denen Sensorwerte über OPC UA bereit gestellt werden [9]. Diese wiederum werden von der OSIP Monitoring View abgerufen und angezeigt.

4.1.2. Testumgebung

Die Testfälle wurden zehn Mal auf einem Computer mit Ubuntu 16.04.5, einer 3,6 GHz CPU mit acht Kernen, 64 GB Arbeitsspeicher, dem OpenJDK 1.8.0_181 und aktiviertem Logging auf der Standardausgabe ausgeführt. Für die `Servlets` wurde als Framework-Implementierung Apache Tomcat 9.0.12[5] verwendet. JavaFX 8 musste als Framework ebenfalls dazu geladen werden. Swing ist als Teil der Java API in der Analyse von JoFrames immer enthalten.

4.1.3. Testergebnisse

In diesem Kapitel bezieht sich der Begriff Instruktion auf Bytecode Instruktionen.

Die Tabellen 4.1 und 4.2 geben einen Überblick über die Testergebnisse. Apache Tomcat besitzt dabei 366051 Instruktionen und JavaFX 365832. Die Instruktionen von Swing wurden aufgrund dessen, dass Swing Bestandteil der Java API ist, nicht gezählt. Die sich im Anhang befindlichen Tabellen A.1 und A.2 geben die empirischen Standardabweichungen für die Zeiten an.

SecuriBench Micro	**MEV**	**V**	**AI**	**I**	**JFZ**	**SDGZ**	**JOZ**	**GZ**
Aliasing1	1	1	28	430	1,9	83,9	0,3	116,3
Aliasing2	0	0	30	430	1,4	79,0	0,2	109,9
Aliasing5	1	1	43	430	1,3	78,0	0,2	109,1
Aliasing6	1	1	126	430	1,3	77,1	0,2	108,9
Arrays2	1	1	45	430	1,3	83,8	0,2	115,3
Arrays4	1	1	39	430	1,3	84,2	0,2	115,7
Arrays5	1	1	39	430	1,3	81,1	0,2	111,7
Arrays10	1	1	54	430	1,3	77,9	0,2	108,3
Basic4	1	1	38	430	1,3	77,3	0,2	108,1
Basic7	1	1	46	430	1,3	78,2	0,2	109,5
Basic8	0	1	57	430	1,3	77,8	0,2	109,5
Basic14	1	1	35	430	1,3	48,4	0,1	57,3
Basic23	2	2	44	430	1,3	80,1	0,3	111,0
Basic28	1	1	236	430	1,3	79,7	0,2	110,1
Basic31	1	1	61	430	1,3	52,9	0,1	62,5

[5] `http://tomcat.apache.org/`

Basic35	1	1	51	430	1,3	46,0	0,1	55,2
Collections3	0	1	51	430	1,3	77,1	0,2	108,0
Collections6	1	1	48	430	1,3	77,4	0,2	109,1
Collections7	1	1	53	430	1,3	429,2	5,4	11978,3
Collections8	0	1	50	430	1,4	81,3	0,2	114,2
Collections11	1	1	51	430	1,2	77,4	0,2	108,9
Collections13	1	1	76	430	1,3	78,3	0,2	109,9
Collections14	1	1	54	430	1,3	78,7	0,2	110,1
Datastructures2	1	1	67	430	1,3	77,8	0,2	108,3
Datastructures4	0	0	72	430	1,3	76,2	0,2	106,9
Datastructures6	1	1	86	430	1,2	76,6	0,2	107,7
Inter2	1	1	44	430	1,3	78,5	0,2	109,8
Inter3	1	1	113	430	1,3	75,9	0,2	110,4
Inter4	1	1	32	430	1,3	79,0	0,2	110,1
Inter7	1	1	50	430	1,3	77,2	0,2	108,2
Inter8	1	1	51	430	1,3	77,8	0,2	108,5
Inter11	1	1	46	430	1,2	75,7	0,2	105,5
Inter13	1	1	43	430	1,2	75,0	0,2	105,0
Pred1	0	0	20	430	1,3	86,4	0,1	114,7
Pred3	0	1	39	430	1,3	83,6	0,2	115,0
Pred7	0	1	43	430	1,3	83,3	0,2	115,1
Pred9	1	1	44	430	1,3	83,1	0,2	114,7
Sanitizers2	1	1	103	430	1,3	78,1	0,2	110,0
Sanitizers4	1	1	90	430	1,3	77,8	0,2	108,9
Session2	1	1	46	430	1,2	78,0	0,2	110,6
StrongUpdates3	0	1	47	430	1,3	78,4	0,2	109,2
StrongUpdates4	1	1	31	430	1,3	78,2	0,2	108,4
StrongUpdates5	0	1	44	430	1,3	75,5	0,2	105,3
Servlets	**MEV**	**V**	**AI**	**I**	**JFZ**	**SDGZ**	**JOZ**	**GZ**
Conf1	1	1	26	436	1,4	53,6	0,1	63,3
Conf2	0	1	35	442	1,3	51,2	0,1	62,0
Int1	1	1	18	550	1,4	90,1	0,2	130,6
Int2	1	1	36	442	1,4	86,9	0,2	123,2
IPServlet	1	1	26	430	1,8	50,2	0,2	62,3
Swing	**MEV**	**V**	**AI**	**I**	**JFZ**	**SDGZ**	**JOZ**	**GZ**
Conf1	1	1	57	1761	0,4	86,6	0,1	94,8
Conf2	2	4	98	1849	0,4	92,8	0,4	131,7
ExtendedPassword	0	0	117	1789	0,4	86,2	0,1	95,5
NoIf	0	1	68	1376	0,4	84,8	0,1	93,1
SwingWorker	1	1	121	1761	0,4	89,4	0,1	100,7

JavaFX	**MEV**	**V**	**AI**	**I**	**JFZ**	**SDGZ**	**JOZ**	**GZ**
Conf1	1	1	74	293	1,5	148,5	1,1	2034,3
ExtendedPassword	0	1	170	354	1,6	153,0	2,2	2450,0
Password	1	1	71	293	1,5	147,6	1,2	1946,5
Worker	1	1	142	293	1,5	151,8	1,4	2495,8

Tabelle 4.1.: Ergebnisse der Testfälle ohne Leistungstests
Legende:
MEV = Minimale Zahl an Sicherheitsverletzungen, die gefunden werden sollen. Enstpricht genau der Zahl an vorhandenen Sicherheitsverletzungen im Testfall.
V = Zahl der von JOANA angezeigten Sicherheitsverletzungen.
AI = Zahl der Instruktionen der Anwendung.
I = Zahl der instrumentierten Instruktionen von JoFrames.
JFZ = Mittlere Zeit in Sekunden, die JoFrames für die Analyse und Instrumentierung benötigt hat.
SDGZ = Mittlere Zeit in Sekunden, die JOANA für die Erstellung des Systemabhängigkeitsgraphen benötigt hat.
JOZ = Mittlere Zeit in Sekunden, die JOANA für die Informationsflusskontrolle benötigt hat.
GZ = Mittlere Gesamtzeit in Sekunden für die Analyse und Instrumentierung von JoFrames und die Analyse von JOANA. Dies beinhaltet auch die Annotation der Quellen und Senken und eine Konvertierung der Sicherheitsverletzungen, die teilweise viel Zeit in Anspruch nehmen kann. Daher kann die Gesamtzeit von der Summe von JFZ, SDGZ und JOZ abweichen.

Leistungstests	AL	AS	AI	I	JFZ
JGit	11	12	395833	2918	2,9
JPass	1	41	332968	12778	3,0
OSIP Monitoring View	1	0	490711	232	3,5
OSIP Simulation View	1	0	876703	230	5,0

Tabelle 4.2.: Ergebnisse der Leistungstests
Legende:
AL = Anzahl gefundener Anwendungs-Klassen, die den Lebenszyklus aus dem Framework implementieren.
AS = Anzahl gefundener Instanzen aus der Anwendung, die bei der Instrumentierung von JoFrames gesammelt wurden.
AI = Zahl der Instruktionen der Anwendung.
I = Zahl der instrumentierten Instruktionen von JoFrames.
JFZ = Mittlere Zeit in Sekunden, die JoFrames für die Analyse und Instrumentierung benötigt hat.

Die Testfälle enthalten insgesamt 45 Sicherheitsverletzungen, die alle von JOANA erkannt wurden. Entsprechend unsichere Anwendungen wurden auch als unsicher markiert. Zusätzlich wurden zwölf falsche Sicherheitsverletzungen gemeldet und zehn von 14 sicheren Anwendungen als unsicher markiert.

Die Testfälle von SecuriBench Micro zeigen, dass die Zahl der instrumentierten Instruktionen unabhängig von der Zahl an Anwendungs- und Framework-Instruktionen ist. Die Testfälle bestehen jeweils aus einem `Servlet` und weisen genau 430 instrumentierte Instruktionen auf. In Kombination mit dem Servlet-Testfall **Int1**, das zwei `Servlets` und 550 instrumentierte Instruktionen enthält, lässt sich 120 als die Zahl an Instruktionen bestimmen, die mindestens für ein `Servlet` instrumentiert werden. Mit `EventListenern` erhöht sich die Zahl. In JGit wurden elf `Servlets` und zwölf `Servlet`-Instanzen von JoFrames gefunden (siehe auch im Anhang A.3 zur JoFrames-Ausgabe) und in den instrumentierten Instruktionen verwendet. Damit müsste JGit $430 + 22 \cdot 120 = 3070$ instrumentierte Instruktionen aufweisen, was dem tatsächlichen Wert von 2918 instrumentierten Instruktionen nahe kommt.

JoFrames hat für OSIP keine `EventHandler` und keine `EventHandler`-Instanzen gefunden, obwohl OSIP `EventHandler` in Form von Lambda-Ausdrücken implementiert [8, 10]. Dies weist daraufhin, dass JoFrames mit Lambda-Ausdrücken nicht umgehen kann und dahingehend erweitert werden müsste.

Für die Laufzeit von JoFrames lassen sich verschiedene Faktoren identifizieren. Einen Einfluss hat das Laden der Framework-Klassen. So weisen die Swing-Testfälle, bei denen keine Framework-Klassen geladen werden müssen, die kürzesten Laufzeiten mit 0,4 Sekunden auf. Bei den Servlet-Testfällen liegen die Laufzeiten zum Beispiel zwischen 1,2 und 1,9 Sekunden. Weiteren Einfluss hat die Anwendungsgröße: so benötigte JoFrames für die OSIP Simulation View 5 Sekunden bei 876703 Anwendungs- und 230 instrumentierten Instruktionen. Umgekehrt hat auch die Zahl der instrumentierten

Instruktionen einen Einfluss: die Swing-Anwendung JPass benötigte 3 Sekunden bei 332968 Anwendungs- und 12778 instrumentierten Instruktionen. Insgesamt lässt sich damit die Laufzeit von JoFrames nur grob abschätzen und für eine reale Anwendung können einige Sekunden erwartet werden.

Die Laufzeit von JoFrames lässt wiederum keinen Schluss auf die Analyse von JOANA und ihrer Laufzeit zu. So benötigte der Servlet-Testfall **Collections7** 5,4 Sekunden für die Informationsflusskontrolle mit JOANA, während JoFrames für deren Analyse und Instrumentierung 1,3 Sekunden benötigte, damit einer der schnelleren Testfälle ist und die übrigen Zeiten der Informationsflusskontrolle der SecuriBench Micro-Testfälle zwischen 0,1 und 0,3 Sekunden liegen.

Ähnliches lässt sich bei den JavaFX-Testfällen beobachten: Die Zahl an instrumentierten Instruktionen liegt mit 293 und 354 unter denen der `Servlets`. Die Informationsflusskontrolle dauert allerdings mit 1,1 bis 2,2 Sekunden um das 3,7- bis 7,3-fache im Vergleich zu den oberen 0,3 Sekunden länger. Die Zahl an instrumentierten Instruktionen lässt somit generell nur bedingt einen Schluss auf die Laufzeit von JOANA zu. Die JavaFX-Testfälle weisen daraufhin, dass das Framework, das unverändert mitanalysiert wird, deutlichen Einfluss auf die Laufzeit von JOANA nehmen kann. Zur Verbesserung dieser können Maßnahmen zur verbesserten Analyse von Frameworks untersucht werden. Beispielsweise war für das Testen das Pruning nicht eingeschalten.

4.2. Zusätzliche Frameworks

Als letztes wurde evaluiert, ob und wie weitere Frameworks mit der Sprache ausgedrückt werden können. Untersucht wurden dabei Android, JUnit und das Standard Widget Toolkit.

4.2.1. Android

Android-Anwendungen bestehen aus mehreren Komponenten [21], die sich von `android.app.Activity` [20], `android.app.Service` [24], `android.content.BroadcastReceiver` [22] oder `android.content.ContentProvider` [23] ableiten. `Activities` stellen die Hauptkomponenten einer Android-Anwendung dar, in denen die graphische Benutzerschnittstelle angezeigt wird [21]. `Services` bieten die Möglichkeit, Hintergrundaufgaben und lang dauernde Operationen auszuführen. `BroadcastReceiver` werden bei bestimmten Systemereignissen ausgelöst und können so auf

diese reagieren. `ContentProvider` gewähren Android-Anwendungen über eine API Zugriff auf eine Menge an geteilten Anwendungsdaten.

Jede Komponente besitzt ihren eigenen Lebenszyklus, der hauptsächlich die Erstellung und Vernichtung regelt [21]. Exemplarisch wird nun genau eine `Activity` betrachtet, deren grundlegender Lebenszyklus in Abbildung 4.1 dargestellt ist. Dieser ist nicht vollständig. Die Dokumentation [20] zeigt, dass es weitere Methoden gibt, die im Verlauf des Lebenszyklus aufgerufen werden. Ein Beispiel dafür ist `onRetainNonConfigurationInstance`.

Der grundlegende Lebenszyklus zeigt, dass unterschiedliche Nutzerinteraktionen zu verschiedenen Abläufen im Lebenszyklus führen [20]. So wird eine `Activity`, die ausgeführt wird, immer durch `onPause` pausiert und kann direkt über `onResume` weiter ausgeführt werden. Die `Activity` kann aber auch direkt beendet werden, sodass `onStop` und `onDestroy` statt `onResume` aufgerufen werden.

Ein solcher Lebenszyklus kann in dieser Form nicht durch eine Framework-Spezifikation ausgedrückt werden. Alternativ besteht allerdings die Möglichkeit, den Lebenszyklus aufzubrechen. Es lässt sich nämlich eine Anfangs- und Endphase erkennen. Die einzelnen Verläufe dazwischen können als Block-Regeln Bestandteil der Arbeitsphase werden. Abbildung 4.2 zeigt dies beispiel- und schemenhaft für den grundlegenden Lebenszyklus einer `Activity`. Für die tatsächliche Umsetzung müssen die vollständigen Lebenszyklen aller Komponenten berücksichtigt werden. Damit müssen die Lebenszyklen von `Activities`, `Servicen`, `BroadcastReceivern` und `ContentProvidern` vereinigt werden. Zusätzlich muss beachtet werden, dass mit mehreren gleichartigen Komponenten, zum Beispiel mit mehreren `Activities`, der aufgebrochene Lebenszyklus nicht mehr genutzt werden kann. Zwischen `onStop` und `onRestart` kann beispielsweise eine andere `Activity` ausgeführt werden [20]. Außerdem wurden bisher keine direkten Nutzerinteraktionen oder Statusänderungen des Gerätes betrachtet, die zum Aufruf weiterer Methoden führen.

4.2.2. JUnit

JUnit 4.12[6] ist ein Testframework für Java [6]. Es verwendet Annotationen zur Kennzeichnung von Methoden, die in der Ausführung aufgerufen werden. Hierzu gehören die Annotationen `@Test` [5], mit der ein einzelner Testfall gekennzeichnet wird, `@Ignore` [3], mit der ein Testfall ignoriert und so nicht ausgeführt wird, `@BeforeClass`, `@Before`, `@After` und `@AfterClass` [4]. Die letzteren vier dienen zum Aufsetzen und Aufräumen von Testressourcen. Dabei wird `@BeforeClass` einmal für die Testklasse zu Beginn der Ausführung der Testfälle in dieser Klasse aufgerufen. Analog verhält es

[6] `https://junit.org/junit4/`

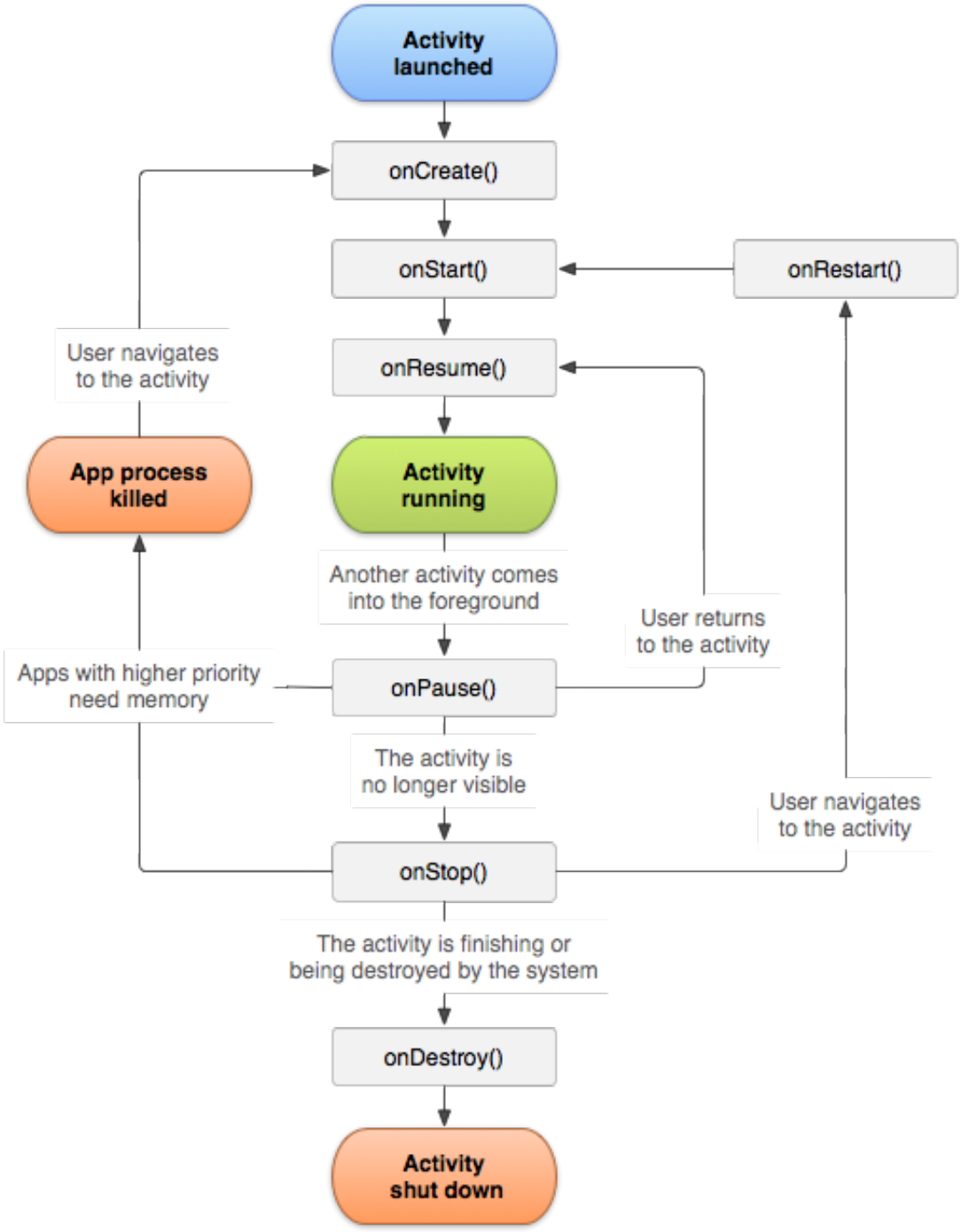


Abbildung 4.1.: Grundlegender Lebenszyklus einer `Activity` [20]

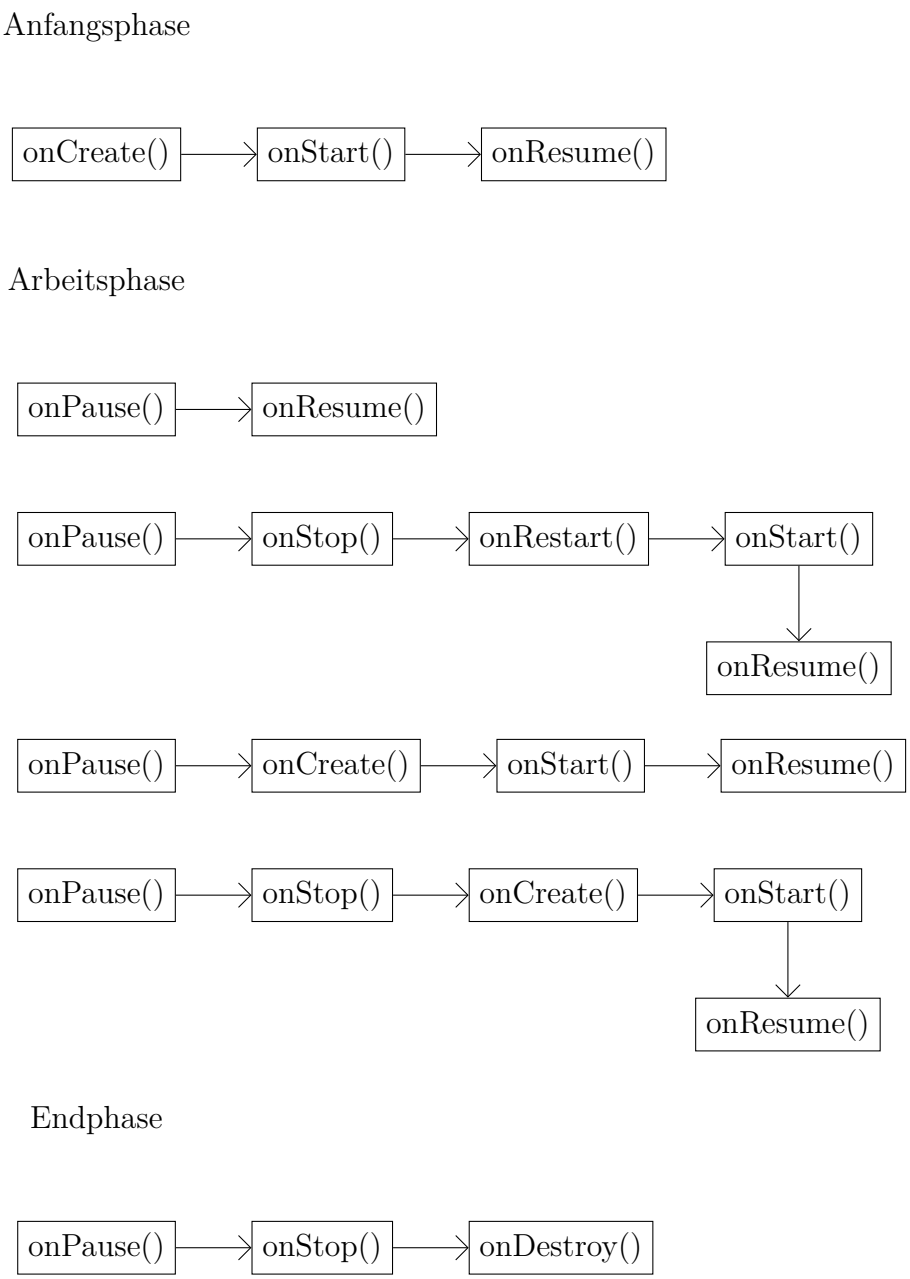


Abbildung 4.2.: Schema einer Framework-Spezifikation mit dem aufgebrochenen grundlegenden Lebenszyklus einer `Activity`

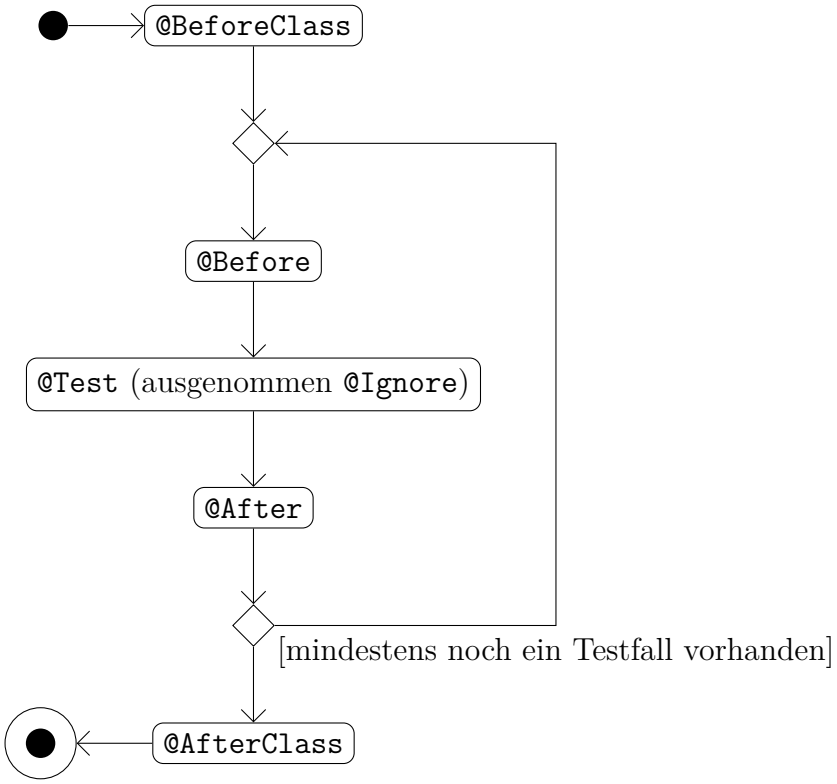


Abbildung 4.3.: Lebenszyklus einer Testklasse in JUnit 4.12

sich mit `@AfterClass` nach Ausführung der Testfälle. `@Before` wird vor und `@After` nach jedem Testfall aufgerufen. Ohne weitere Annotationen werden die Testfälle einer Klasse in beliebiger Reihenfolge ausgeführt [7]. Abbildung 4.3 zeigt den so entstehenden Lebenszyklus einer Testklasse.

Der Lebenszyklus zeigt eine mögliche Einteilung für genau eine Testklasse: `@Before Class` stellt die Anfangsphase, `@AfterClass` die Endphase und die Paarung `@Before - @Test - @After` den Ausdruck für die Arbeitsphase dar. In einer Framework-Spezifikation können allerdings keine Annotationen angegeben werden, sodass JoFrames aktuell Testklassen von JUnit nicht verarbeiten kann. Die Sprache und in der Folge JoFrames können dafür dahin gehend erweitert werden, dass beispielsweise auch Annotationen für die Klassenangabe der expliziten Angabe oder Block-Regel erlaubt sind. Für `@Ignore` bestehen die Möglichkeiten, diese Annotation auszulassen und somit alle Testfälle einzubeziehen oder einen neuen Mechanismus für diese Art des Methodenausschlusses einzuführen.

4.2.3. Standard Widget Toolkit (SWT)

SWT[7] ist ein weiteres Framework für die Erstellung graphischer Benutzerschnittstellen in Java [15]. Es ist dabei eng mit nativem Code verbunden, der die Darstellung der graphischen Benutzerschnittstelle an das Betriebssystem delegiert. Der konkrete Lebenszyklus wird durch eine SWT-Anwendung selbst gesteuert, wobei der generelle Lebenszyklus durch SWT vorgegeben ist [17]. Abbildung 4.4 gibt einen Überblick darüber. Dieser zeigt auch, dass es eine Anfangs-, Arbeits- und Endphase gibt. Eine SWT-Anwendung startet mit der `main`-Methode in die Anfangsphase. In dieser wird das `org.eclipse.swt.widgets.Display` erstellt, das die Verbindung zum Betriebssystem herstellt [13]. Im Anschluss können `org.eclipse.swt.widgets.Shell`s, die ein Fenster repräsentieren [18], und `org.eclipse.widgets.Widget`s, die die Elemente der graphischen Benutzerschnittstelle darstellen [17], erstellt und für die Anzeige vorbereitet werden. Über die Methode `open` eines `Shell`-Objektes wird das Fenster angezeigt. Damit beginnt die Arbeitsphase, in der analog zu Swing und JavaFX 8 ein Thread auf Ereignisse wartet und diese verarbeitet [16]. In SWT ist dies derselbe Thread, in dem das `Display`-Objekt erstellt wurde. Zusätzlich muss eine SWT-Anwendung die Schleife, in der die Ereignisse verarbeitet werden, bereitstellen. Dafür eignet sich in den meisten Fällen, als Schleifenbedingung auf das Schließen der `Shell` mittels `isDisposed` zu warten und in der Schleife die `Display`-Methode `readAndDispatch` aufzurufen, in der die eigentliche Ereignisverarbeitung implementiert ist. Das `Display` liest das nächste Ereignis vom Betriebssystem ein und liefert es an den passenden `EventListener` aus [14]. Sind alle `Shells` geschlossen, werden über die Methode `dispose` des `Displays` Betriebssystem-Ressourcen frei gegeben [17] und die SWT-Anwendung kann enden, sodass diese Methode die Endphase darstellt.

Solange die Bedingungen für den Lebenszyklus eingehalten werden, kann eine SWT-Anwendung beliebige Formen annehmen. U. a. ist es damit möglich, den beschriebenen Lebenszyklus komplett in der `main`-Methode auszuführen. Damit führt die Angabe der `main`-Methode in der Anfangsphase einer Framework-Spezifikation dazu, dass der Code der Endphase in der Anfangsphase mit analysiert wird. Aufgrund der engen Verbundenheit zwischen SWT und nativem Code mit Delegation an das Betriebssystem ist an dieser Stelle offen und zu prüfen, welchen Einfluss die Position der `dispose`-Methode in der Anfangsphase hat. Die Arbeitsphase lässt sich durch die Angabe der `EventListener`-Schnittstelle als Supertyp-Regel realisieren.

[7] `https://www.eclipse.org/swt/`

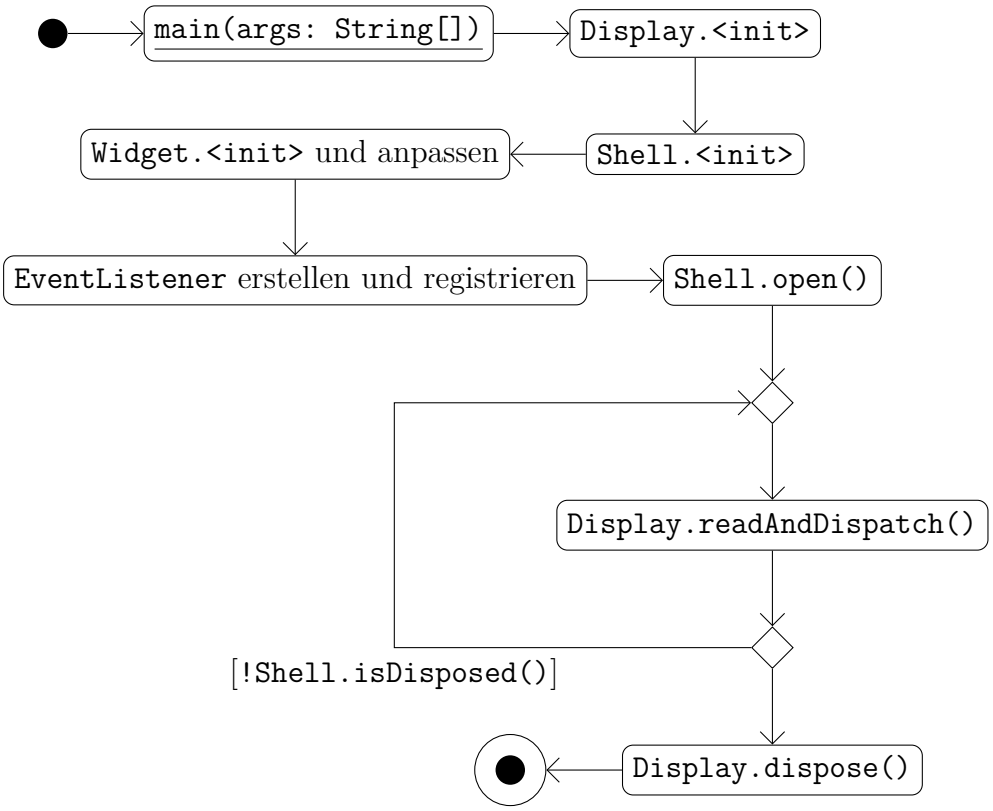


Abbildung 4.4.: Lebenszyklus einer SWT-Anwendung

5. Fazit und Ausblick

In dieser Arbeit wurde eine Sprache vorgestellt, mit der sich ein Framework in Form einer Framework-Spezifikation ausdrücken lässt. In Kombination mit den Framework- und Anwendungsklassen generiert JoFrames eine künstliche `main`-Methode daraus, die einen Einsprungspunkt für die Anwendung darstellt und das Verhalten des Frameworks simuliert. Die künstliche `main`-Methode eignet sich für die Analyse und Informationsflusskontrolle in JOANA.

Aktuell werden die Frameworks Java Servlet Specification, Version 4.0, Swing und JavaFX 8 einzeln unterstützt. Somit bleibt die Erweiterung auf Anwendungen, die mehrere Frameworks nutzen, offen. Wie sich gezeigt hat, bieten die Sprache und JoFrames die Möglichkeit, durch einfache Erweiterungen weitere Frameworks zu unterstützen. JUnit beispielsweise benötigt eine Erweiterung für die Angabe von Annotationen. Android und SWT benötigen darüber hinaus zusätzliche Untersuchungen und Arbeit für die Umsetzung in eine Framework-Spezifikation, da sie eigene Besonderheiten und im Falle von Android eine große API aufweisen.

Literaturverzeichnis

[1] *Stanford SecuriBench Micro*. https://suif.stanford.edu/~livshits/work/securibench-micro/. Version: 26. Mai 2006. – Zugriff: 08. Oktober 2018.

[2] *Shrike technical overview*. http://wala.sourceforge.net/wiki/index.php/Shrike_technical_overview. Version: 24. April 2010. – Zugriff: 08. Oktober 2018.

[3] *Ignoring tests*. https://github.com/junit-team/junit4/wiki/Ignoring-tests. Version: 29. Juli 2015. – Zugriff: 08. Oktober 2018.

[4] *Test fixtures*. https://github.com/junit-team/junit4/wiki/Test-fixtures. Version: 18. Juni 2017. – Zugriff: 08. Oktober 2018.

[5] *Annotation Type Test*. https://junit.org/junit4/javadoc/latest/org/junit/Test.html. Version: 2018. – Zugriff: 09. Oktober 2018.

[6] *JUnit 4*. https://github.com/junit-team/junit4. Version: 2018. – Zugriff: 09. Oktober 2018.

[7] *Test execution order*. https://github.com/junit-team/junit4/wiki/Test-execution-order. Version: 28. März 2018. – Zugriff: 08. Oktober 2018.

[8] Armbruster, Martin ; Kahles, David ; Lehmann, Hans-Peter ; Schwarzmann, Maximilian ; Wilhelm, Niko: *MonitoringController.java*. https://github.com/ByteHamster/PSE/blob/master/src/osip-monitoring-controller/src/main/java/edu/kit/pse/osip/monitoring/controller/MonitoringController.java. Version: 02. Mai 2017. – Zugriff: 12. Oktober 2018.

[9] Armbruster, Martin ; Kahles, David ; Lehmann, Hans-Peter ; Schwarzmann, Maximilian ; Wilhelm, Niko: *OPC UA Simulator for Industrial Plants (OSIP)*. https://github.com/ByteHamster/PSE. Version: 05. Mai 2017. – Zugriff: 08. Oktober 2018.

[10] Armbruster, Martin ; Kahles, David ; Lehmann, Hans-Peter

; Schwarzmann, Maximilian ; Wilhelm, Niko: *SimulationController.java*. https://github.com/ByteHamster/PSE/blob/master/src/osip-simulation-controller/src/main/java/edu/kit/pse/osip/simulation/controller/SimulationController.java. Version: 14. April 2017. – Zugriff: 12. Oktober 2018.

[11] Bata, Gabor: *Password manager application with strong encryption (AES-256). [Java/Swing]*. https://github.com/gaborbata/jpass. Version: 06. Juni 2018. – Zugriff: 08. Oktober 2018.

[12] Blaschke, Tobias: *Automatische Modellierung des Lebenszyklus von Android-Anwendungen*. April 2014

[13] Eclipse Foundation, Inc.: *Class Display*. http://help.eclipse.org/luna/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Freference%2Fapi%2Forg%2Feclipse%2Fswt%2Fwidgets%2FDisplay.html. – Zugriff: 10. Oktober 2018.

[14] Eclipse Foundation, Inc.: *Events*. http://help.eclipse.org/luna/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Fguide%2Fswt_widgets_events.htm&cp=2_0_7_0_1. – Zugriff: 10. Oktober 2018.

[15] Eclipse Foundation, Inc.: *The Standard Widget Toolkit*. http://help.eclipse.org/luna/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Fguide%2Fswt.htm. – Zugriff: 10. Oktober 2018.

[16] Eclipse Foundation, Inc.: *Threading issues*. http://help.eclipse.org/luna/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Fguide%2Fswt_threading.htm&cp=2_0_7_2. – Zugriff: 10. Oktober 2018.

[17] Eclipse Foundation, Inc.: *Widgets*. http://help.eclipse.org/luna/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Fguide%2Fswt_widgets.htm&cp=2_0_7_0. – Zugriff: 10. Oktober 2018.

[18] Eclipse Foundation, Inc.: *Class Shell*. http://help.eclipse.org/luna/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Freference%2Fapi%2Forg%2Feclipse%2Fswt%2Fwidgets%2FShell.html. Version: 2014. – Zugriff: 10. Oktober 2018.

[19] Eclipse Foundation, Inc.: *JGit*. https://www.eclipse.org/jgit/. Version: 2018. – Zugriff: 08. Oktober 2018.

[20] Google Inc.: *Activity*. https://developer.android.com/reference/

android/app/Activity.html. Version: 25. September 2018. – Zugriff: 08. Oktober 2018.

[21] Google Inc.: *Application Fundamentals*. https://developer.android.com/guide/components/fundamentals. Version: 30. April 2018. – Zugriff: 08. Oktober 2018.

[22] Google Inc.: *BroadcastReceiver*. https://developer.android.com/reference/android/content/BroadcastReceiver.html. Version: 06. Juni 2018. – Zugriff: 08. Oktober 2018.

[23] Google Inc.: *ContentProvider*. https://developer.android.com/reference/android/content/ContentProvider. Version: 06. Juni 2018. – Zugriff: 08. Oktober 2018.

[24] Google Inc.: *Service*. https://developer.android.com/reference/android/app/Service.html. Version: 25. September 2018. – Zugriff: 08. Oktober 2018.

[25] Graf, Jürgen ; Hecker, Martin ; Mohr, Martin: Using JOANA for Information Flow Control in Java Programs - A Practical Guide. In: *Proceedings of the 6th Working Conference on Programming Languages (ATPS'13)*, Springer Berlin / Heidelberg, Februar 2013 (Lecture Notes in Informatics (LNI) 215), S. 123–138

[26] Hammer, Christian ; Snelting, Gregor: Flow-Sensitive, Context-Sensitive, and Object-sensitive Information Flow Control Based on Program Dependence Graphs. In: *International Journal of Information Security* 8 (2009), Dezember, Nr. 6, S. 399–422. http://dx.doi.org/10.1007/s10207-009-0086-1. – DOI 10.1007/s10207–009–0086–1

[27] Oracle Corporation: *1 JavaFX Overview*. https://docs.oracle.com/javase/8/javafx/get-started-tutorial/jfx-overview.htm. Version: 2014. – Zugriff: 22. September 2018.

[28] Oracle Corporation: *1 Processing Events*. https://docs.oracle.com/javase/8/javafx/events-tutorial/processing.htm. Version: 2014. – Zugriff: 28. April 2018.

[29] Oracle Corporation: *About the JFC and Swing*. https://docs.oracle.com/javase/tutorial/uiswing/start/about.html. Version: 2017. – Zugriff: 22. September 2018.

[30] Oracle Corporation: *The Event Dispatch Thread*. https://docs.oracle.

com/javase/tutorial/uiswing/concurrency/dispatch.html. Version: 2017. – Zugriff: 28. April 2018.

[31] Oracle Corporation: *General Information about Writing Event Listeners*. https://docs.oracle.com/javase/tutorial/uiswing/events/generalrules.html. Version: 2017. – Zugriff: 28. April 2018.

[32] Oracle Corporation: *Class AbstractButton*. https://docs.oracle.com/javase/10/docs/api/javax/swing/AbstractButton.html. Version: April 2018. – Zugriff: 23. April 2018.

[33] Oracle Corporation: *Class Application*. https://docs.oracle.com/javase/10/docs/api/javafx/application/Application.html. Version: April 2018. – Zugriff: 23. April 2018.

[34] Oracle Corporation: *Class DefaultEditorKit.CopyAction*. https://docs.oracle.com/javase/8/docs/api/javax/swing/text/DefaultEditorKit.CopyAction.html. Version: 2018. – Zugriff: 02. Oktober 2018.

[35] Oracle Corporation: *Class EventQueue*. https://docs.oracle.com/javase/10/docs/api/java/awt/EventQueue.htmll. Version: April 2018. – Zugriff: 23. April 2018.

[36] Oracle Corporation: *Class JComponent*. https://docs.oracle.com/javase/8/docs/api/javax/swing/JComponent.html. Version: 2018. – Zugriff: 02. Oktober 2018.

[37] Oracle Corporation: *Class JFrame*. https://docs.oracle.com/javase/10/docs/api/javax/swing/JFrame.html. Version: April 2018. – Zugriff: 23. April 2018.

[38] Oracle Corporation: *Class Node*. https://docs.oracle.com/javase/10/docs/api/javafx/scene/Node.html. Version: April 2018. – Zugriff: 23. April 2018.

[39] Oracle Corporation: *Class SwingWorker<T,V>*. https://docs.oracle.com/javase/8/docs/api/javax/swing/SwingWorker.html. Version: 2018. – Zugriff: 02. Oktober 2018.

[40] Oracle Corporation: *Interface EventHandler<T extends Event>*. https://docs.oracle.com/javase/10/docs/api/javafx/event/EventHandler.html. Version: April 2018. – Zugriff: 23. April 2018.

[41] Oracle Corporation: *Interface EventListener*. https://docs.oracle.com/javase/10/docs/api/java/util/EventListener.html. Version: April 2018. – Zugriff: 23. April 2018.

[42] Oracle Corporation: *Package javax.swing*. https://docs.oracle.com/javase/8/docs/api/javax/swing/package-summary.html. Version: 2018. – Zugriff: 02. Oktober 2018.

[43] Oracle Corporation ; The Apache Software Foundation: *Interface HttpServletResponse*. http://download.oracle.com/otndocs/jcp/servlet-4-final-spec/index.html. Version: 2017. – Zugriff: 23. April 2018.

[44] Oracle Corporation ; The Apache Software Foundation: *Interface ServletConfig*. http://download.oracle.com/otndocs/jcp/servlet-4-final-spec/index.html. Version: 2017. – Zugriff: 23. April 2018.

[45] Oracle Corporation ; The Apache Software Foundation: *Interface ServletContextListener*. http://download.oracle.com/otndocs/jcp/servlet-4-final-spec/index.html. Version: 2017. – Zugriff: 23. April 2018.

[46] Oracle Corporation ; The Apache Software Foundation: *Interface ServletRequestListener*. http://download.oracle.com/otndocs/jcp/servlet-4-final-spec/index.html. Version: 2017. – Zugriff: 23. April 2018.

[47] Shing Wai Chan, Ed B.: *Java Servlet Specification - Version 4.0*. https://jcp.org/en/jsr/detail?id=369. Version: Juli 2017. – Zugriff: 23. April 2018.

[48] Sridharan, Manu ; Artzi, Shay ; Pistoia, Marco ; Guarnieri, Salvatore ; Tripp, Omer ; Berg, Ryan: F4F: taint analysis of framework-based web applications. In: *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, 2011

[49] Tripp, Omer ; Pistoia, Marco ; Cousot, Patrick ; Cousot, Radhia ; Guarnieri, Salvatore: ANDROMEDA: Accurate and Scalable Security Analysis of Web Applications. In: *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering*. Berlin, Heidelberg : Springer-Verlag, 2013 (FASE'13). – ISBN 978–3–642–37056–4, 210–225

[50] Tripp, Omer ; Pistoia, Marco ; Fink, Stephen J. ; Sridharan, Manu ; Weisman, Omri: TAJ: Effective Taint Analysis of Web Applications. In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA : ACM, 2009 (PLDI '09). – ISBN 978–1–60558–392–1, 87–97

Erklärung

Hiermit erkläre ich, Martin Armbruster, dass ich die vorliegende Bachelorarbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis beachtet habe.

Ort, Datum	Unterschrift

A. Anhang

A.1. Framework-Spezifikationen

```
1  <?xml version="1.0" encoding="UTF-8"?>
2      <framework-specification xmlns:xsi="http://www.w3.org
           /2001/XMLSchema-instance" xsi:schemaLocation="
           framework_specification_language_model.xsd" xmlns="
           fwspec" name="Servlets">
3           <phases>
4               <start>
5                   <explicit-declaration>
6                       <explicit-declaration for_all="Ljava/
                            util/EventListener">
7                           <method-call>Constructor</method-
                                call>
8                       </explicit-declaration>
9                       <explicit-declaration for_all="Ljavax/
                            servlet/ServletContextListener">
10                          <method-call>contextInitialized(
                                Ljavax/servlet/
                                ServletContextEvent;)V</method-
                                call>
11                      </explicit-declaration>
12                      <explicit-declaration for_all="Ljavax/
                            servlet/Servlet">
13                          <method-call>Constructor</method-
                                call>
14                          <method-call>init(Ljavax/servlet/
                                ServletConfig;)V</method-call>
15                      </explicit-declaration>
16                  </explicit-declaration>
17              </start>
18              <end>
19                  <explicit-declaration>
20                      <explicit-declaration for_all="Ljavax/
```

```
                    servlet/Servlet">
21                   <method-call>destroy()V</method-call
                        >
22                  </explicit-declaration>
23                  <explicit-declaration for_all="Ljavax/
                        servlet/ServletContextListener">
24                   <method-call>contextDestroyed(Ljavax
                        /servlet/ServletContextEvent;)V</
                        method-call>
25                  </explicit-declaration>
26              </explicit-declaration>
27          </end>
28          <working-phases>
29              <working threads="multi">
30                  <rule-block quantor="for_all" class="
                        Ljavax/servlet/Servlet">
31                   <explicit-declaration>
32                       <explicit-declaration for_all="
                            Ljavax/servlet/
                            ServletRequestListener">
33                        <method-call>
                              requestInitialized(Ljavax
                              /servlet/
                              ServletRequestEvent;)V</
                              method-call>
34                       </explicit-declaration>
35                       <method-call>service(Ljavax/
                            servlet/ServletRequest;Ljavax
                            /servlet/ServletResponse;)V</
                            method-call>
36                       <explicit-declaration for_all="
                            Ljavax/servlet/
                            ServletRequestListener">
37                        <method-call>
                              requestDestroyed(Ljavax/
                              servlet/
                              ServletRequestEvent;)V</
                              method-call>
38                       </explicit-declaration>
39                   </explicit-declaration>
40                  </rule-block>
41              </working>
42          </working-phases>
43      </phases>
```

```
44      </framework-specification>
```

Listing A.1: Framework-Spezifikation der Java Servlet Specification, Version 4.0

A.2. Empirische Standardbweichungen

SecuriBench Micro	JFZS	SDGZS	JOZS	GZS
Aliasing1	0,2	5,8	0,0	7,2
Aliasing2	0,1	5,4	0,0	6,4
Aliasing5	0,1	5,3	0,0	6,6
Aliasing6	0,1	5,4	0,0	6,5
Arrays2	0,1	6,8	0,0	7,9
Arrays4	0,1	6,1	0,0	7,3
Arrays5	0,1	5,6	0,0	6,5
Arrays10	0,1	4,6	0,0	5,1
Basic4	0,1	5,0	0,0	5,7
Basic7	0,1	5,0	0,0	5,6
Basic8	0,1	4,8	0,0	5,5
Basic14	0,0	3,4	0,0	3,7
Basic23	0,0	5,2	0,0	5,9
Basic28	0,0	4,8	0,0	5,6
Basic31	0,0	3,4	0,0	3,7
Basic35	0,0	2,9	0,0	3,3
Collections3	0,1	5,0	0,0	6,1
Collections6	0,1	5,1	0,0	6,2
Collections7	0,0	2,0	0,2	479,2
Collections8	0,1	6,9	0,0	8,1
Collections11	0,0	5,2	0,0	6,0
Collections13	0,1	5,3	0,0	6,1
Collections14	0,1	5,2	0,0	6,0
Datastructures2	0,1	4,5	0,0	5,1
Datastructures4	0,0	3,3	0,0	3,6
Datastructures6	0,0	3,0	0,0	3,7
Inter2	0,1	5,4	0,0	6,1
Inter3	0,1	7,0	0,0	6,6
Inter4	0,0	6,2	0,0	7,0
Inter7	0,1	4,8	0,0	5,5
Inter8	0,1	5,3	0,0	5,2
Inter11	0,0	2,5	0,0	2,2
Inter13	0,0	1,7	0,0	1,5
Pred1	0,1	7,1	0,0	7,8
Pred3	0,1	6,8	0,0	8,1

Pred7	0,1	7,1	0,0	8,3
Pred9	0,1	6,6	0,0	7,7
Sanitizers2	0,1	5,5	0,0	6,1
Sanitizers4	0,1	5,2	0,0	5,9
Session2	0,0	1,9	0,0	1,7
StrongUpdates3	0,1	4,7	0,0	5,5
StrongUpdates4	0,0	4,3	0,0	4,1
StrongUpdates5	0,0	2,5	0,0	1,9
Servlets	**JFZS**	**SDGZS**	**JOZS**	**GZS**
Conf1	0,1	4,7	0,0	5,4
Conf2	0,1	3,9	0,0	4,1
Int1	0,1	7,2	0,0	8,6
Int2	0,1	8,1	0,0	9,3
IPServlet	0,0	0,2	0,0	0,4
Swing	**JFZS**	**SDGZS**	**JOZS**	**GZS**
Conf1	0,0	1,0	0,0	1,1
Conf2	0,0	0,7	0,0	1,2
ExtendedPassword	0,0	0,6	0,0	1,0
NoIf	0,0	0,9	0,0	1,0
SwingWorker	0,0	1,0	0,0	1,2
JavaFX	**JFZS**	**SDGZS**	**JOZS**	**GZS**
Conf1	0,0	2,3	0,0	118,5
ExtendedPassword	0,0	1,9	0,1	189,2
Password	0,0	1,7	0,1	102,9
Worker	0,0	1,4	0,0	137,6

Tabelle A.1.: Empirische Standardabweichungen der Zeitmessungen der Testfälle ohne Leistungstests
Legende:
JFZS = Standardabweichung in Sekunden für die Analyse und Instrumentierung durch JoFrames.
SDGZS = Standardabweichung in Sekunden für die Erstellung des Systemabhängigkeitsgraphen durch JOANA.
JOZS = Standardabweichung in Sekunden für die Informationsflusskontrolle durch JOANA.
GZS = Standardabweichung in Sekunden für die Gesamtzeit.

Leistungstests	JFZS
JGit	0,0

JPass	0,0
OSIP Monitoring View	0,0
OSIP Simulation View	0,0

Tabelle A.2.: Empirische Standardabweichungen der Zeitmessungen der Leistungstests
Legende:
JFZS = Standardabweichung in Sekunden für die Analyse und Instrumentierung durch JoFrames.

A.3. JoFrames: Ausgabe

```
1  [...]
2  Explicit declaration: Searching for application sub classes
       of Ljavax/servlet/Servlet. [...]
3  Found sub class Lorg/eclipse/jgit/http/server/glue/
       ErrorServlet. [...]
4  Found sub class Lorg/eclipse/jgit/http/server/glue/
       MetaServlet. [...]
5  Found sub class Lorg/eclipse/jgit/http/server/
       UploadPackServlet. [...]
6  Found sub class Lorg/eclipse/jgit/http/server/
       ObjectFileServlet$PackIdx. [...]
7  Found sub class Lorg/eclipse/jgit/http/server/
       ObjectFileServlet$Loose. [...]
8  Found sub class Lorg/eclipse/jgit/http/server/
       ObjectFileServlet$Pack. [...]
9  Found sub class Lorg/eclipse/jgit/http/server/
       ReceivePackServlet. [...]
10 Found sub class Lorg/eclipse/jgit/http/server/
       TextFileServlet. [...]
11 Found sub class Lorg/eclipse/jgit/http/server/
       InfoPacksServlet. [...]
12 Found sub class Lorg/eclipse/jgit/http/server/
       InfoRefsServlet. [...]
13 Found sub class Lorg/eclipse/jgit/http/server/GitServlet.
14 [...]
15 Adding instance of Lorg/eclipse/jgit/http/server/glue/
       ErrorServlet; [...]
16 Adding instance of Lorg/eclipse/jgit/http/server/
       UploadPackServlet; [...]
```

```
17 Adding instance of Lorg/eclipse/jgit/http/server/
     ReceivePackServlet; [...]
18 Adding instance of Lorg/eclipse/jgit/http/server/
     InfoRefsServlet; [...]
19 Adding instance of Lorg/eclipse/jgit/http/server/glue/
     ErrorServlet; [...]
20 Adding instance of Lorg/eclipse/jgit/http/server/
     TextFileServlet; [...]
21 Adding instance of Lorg/eclipse/jgit/http/server/
     TextFileServlet; [...]
22 Adding instance of Lorg/eclipse/jgit/http/server/
     TextFileServlet; [...]
23 Adding instance of Lorg/eclipse/jgit/http/server/
     InfoPacksServlet; [...]
24 Adding instance of Lorg/eclipse/jgit/http/server/
     ObjectFileServlet$Loose; [...]
25 Adding instance of Lorg/eclipse/jgit/http/server/
     ObjectFileServlet$Pack; [...]
26 Adding instance of Lorg/eclipse/jgit/http/server/
     ObjectFileServlet$PackIdx;
27 [...]
```

Listing A.2: Ausschnitt aus der Ausgabe von JoFrames zur Analyse und Instrumentierung von JGit