Abschlussbericht
Forschungspraktikum Japi2

Vera Christ[i] Daniel Vivas Estevao[ii] Maximilian Strauch[iii]

Koblenz, Sommersemester 2015

Inhaltsverzeichnis

[i]Universität Koblenz-Landau, `vchrist@uni-koblenz.de`
[ii]Universität Koblenz-Landau, `dvivas@uni-koblenz.de`
[iii]Universität Koblenz-Landau, `maxstrauch@uni-koblenz.de`

1 Abstract

Japi[1] ist eine Bibliothek mit dem Ziel, die Entwicklung von graphischen Benutzeroberflächen so einfach und intuitiv wie möglich zu gestalten. Dazu unterstützt Japi mehrere Programmiersprachen und ist so weitgehend plattformunabhängig. So können komplexe graphische Benutzeroberflächen, die in vielen Programmiersprachen, wie beispielsweise in C, nicht ohne weiteres realisiert werden können, über Japi mit wenigen und einfachen Befehlen leicht realisiert werden.

Im Kern ist Japi in Java und C programmiert. Der Japi *Kernel* ist in Java umgesetzt und stellt die graphische Benutzeroberfläche bereit, die über die Netzwerkschnittstelle vom Japi *Client* manipuliert wird. Der Client ist in der Programmiersprache C umgesetzt. Diese „C-Bibliothek“ ist die Schnittstelle zu anderen Programmiersprachen, so dass auch beispielsweise Fortran die C-Bibliothek verwenden kann und so über den Japi *Kernel* eine graphische Benutzeroberfläche erstellen kann.

In der Originalversion von Japi wurde die graphische Oberfläche mit dem AWT-Framework realisiert. Dieses ist in der Zwischenzeit veraltet und hat einen sehr eingeschränkten Funktionsumfang. Das Hauptziel dieses Forschungspraktikums ist die Portierung des Japi Kernels von AWT auf die moderne und heute stark verbreitete Swing API. Dabei soll die Schnittstelle zu anderen Programmiersprachen nicht verändert werden, um die Kompatibilität bereits bestehender Japi Programme auch weiterhin zu gewährleisten. Ein weiteres Ziel des Projekts ist es, neben der Nutzung von Japi unter Windows und Linux auch die Nutzung unter Mac OS möglich zu machen.

Im Rahmen dieses Projektes wurde die imperative Architektur des originalen Japi Kernels auf eine objektorientierte, leichter wartbarere Architektur umgestellt. Daneben wurde das moderne und *leichtgewichtige* Swing-Framework verwendet und die bisher existierenden GUI-Komponenten hierfür neu implmenentiert.

[1] Kurzform für *Java Application Programming Interface*.

2 Prolog

Dieses Kapitel soll kurz eine Begründung für die Wahl des GUI Frameworks Swing liefern sowie eine Übersicht über die wichtigsten Begriffe, die in dieser Dokumentation verwendet werden.

2.1 Auswahl eines GUI Frameworks in Java

Möchte man mit der Programmiersprache Java eine grafische Benutzeroberfläche (GUI) realisieren, so hat man die Wahl zwischen den folgenden Frameworks:

- **AWT**. Das *Abstract Window Toolkit* ist eine Standard API, welche die nativen GUI Elemente des Betriebssystems verwendet. Es existiert seit JDK 1.1 und ist somit stark veraltet. Da jedes Betriebssystem anders ist, steht nur eine kleine Grundmenge an Komponenten bereit, die auf allen Betriebssystemen vorhanden sind. Darüber hinaus gibt es keinen großen Spielraum für „speziellere“ Komponenten (z.B. einen Fortschrittsbalken) und man muss diese selbst zeichnen.
- **SWT**. Das *Standard Widget Toolkit* wurde 2001 von IBM für die IDE Eclipse entwickelt. Ähnlich wie AWT verwendet es auch die Betriebssystemkomponenten, beschränkt sich aber nicht - im Gegensatz zu AWT - auf einige wenige Komponenten, sondern bietet ein großes Spektrum an. Da viele Windows-Komponenten auf linuxbasierten Betriebssystemen nicht existieren, müssen diese nachgebildet werden, was die Performance einschränkt. Außerdem ist das Framework nicht Bestandteil der Stardard API und muss als externe Bibliothek (inklusive nativer Bibliotheken) mit der Anwendung verbreitet werden.
- **JavaFX** ist die Antwort von Oracle auf Adobe Flash und Flex sowie auf Microsoft Silverlight und stellt den Versuch dar, von Javaseite in das Geschäft der Rich Internet Applications einzusteigen. Es ist seit Java SE Runtime 7 Update 6 in die Standard API integriert und momentan in Version 2 verfügbar. Es verfügt über einen großen Funktionsumfang und bietet viele Komponenten. Da es sich um eine sehr junge API handelt, sind noch nicht alle Funktionen vollständig ausgereift.
- **Swing** ist eine seit Java Version 1.2 (1998) in Java integrierte GUI Bibliothek, die auf das AWT Framework aufsetzt, doch seine Komponenten selbst zeichnet. Hierdurch ist es möglich, eine Vielzahl von GUI Komponenten anzubieten, die auf verschiedenen Plattformen existieren. Es ist modular und objektorientiert aufgebaut und verfügt über viele weitere Funktionen: z.B. Java2D Rendering, Drag & Drop Unterstützung, automatisches Double Buffering, Tastaturkombinationen.

Schließlich wurde für dieses Projekt die Swing Bibliothek gewählt, da sie folgende Vorteile bietet:

- Plattformunabhängig - die Komponenten werden von Swing gezeichnet und sind somit auf den verschiedenen Betriebssystem verfügbar

- Große Beliebtheit - viele „große“ GUI Applikationen sind mit Java Swing realisiert, z.B. NetBeans IDE oder SAP-ERP. Abbildung 1 zeigt in blau die „Präsenz“ des Swing Frameworks. Hier ist deutlich zu erkennen, dass es eine sehr große Beliebtheit genoss und immer noch eine bedeutendere Rolle als die anderen Frameworks spielt. Allerdings bildet sich für die Zunkunft eine Niederlage gegenüber dem inoffiziellem „Nachfolger“ von Swing - JavaFX - ab.
- Erprobtheit - Swing ist seit 1998 weiterentwickelt worden und mittlerweile erprobt - im Gegensatz zur vielfältigen, aber fehleranfälligen JavaFX Bibliothek
- Keine weiteren Bibliotheken erforderlich - dies ist ein Argument gegen SWT. Hier müssen (größere) native Bibliotheken eingebunden werden und dafür gesorgt werden, dass die JVM diese auch zur Laufzeit findet
- Erfahrung des Teams - Nicht zu unterschätzen ist auch die Tatsache, dass das Team dieses Projekts zum Teil große Erfahrung mit Swing hat, die während der Entwicklung eingesetzt werden konnten, um ein gutes Resultat zu erzielen

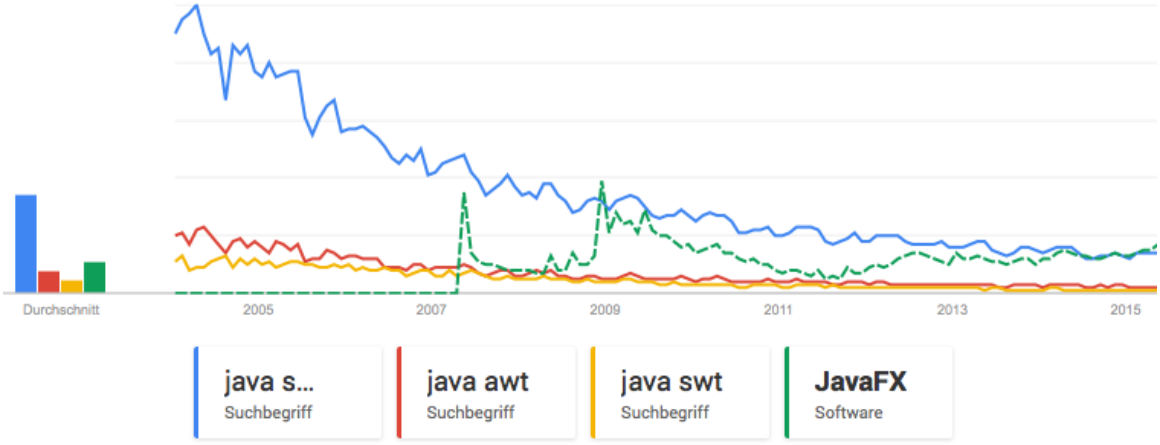


Abbildung 1: Vergleich der „Präsenz“ der vier vorgestellten GUI Frameworks anhand des Analysetools Google Trends. In blau ist das Java Swing Framework dargestellt. Quelle: `http://www.google.com/trends/explore#q=java%20swing%2C%20java%20awt%2C%20java%20swt%2C%20%2Fm%2F02r0tn1&cmpt=q&tz=`.

2.2 Wichtige Begriffe

Die nachfolgenden Begriffe werden im Verlauf dieser Ausarbeitung immer wieder vorkommen und sollen an dieser Stelle erläutert werden:

Client oder Japi Client oder `japilib`
Bei der Japi Bibliothek ist zwischen dem *Client* bzw. der `japilib` zu unterscheiden. Diese Komponente ist in C umgesetzt und wird von einer anderen Programmiersprache (C, Basic, Fortran, Ada, ...) als *native* Bibliothek verwendet. Dieser Teil der Japi Bibliothek wird als **Client** bezeichnet. Dem gegenüber steht der Japi Kernel.

Japi Kernel
Dies ist die Java Anwendung, die auf einem Networkport horcht und die Anforderungen zum Darstellen der GUI annimmt. Dieser Teil der Japi Bibliothek wurde im Rahmen dieses Projekts „modernisiert“.

Japi2 Kernel
Wenn in dieser Ausarbeitung bezug auf den „Japi2 Kernel“ genommen wird, ist immer die neu entwickelte Japi Kernel Applikation gemeint. Als „Japi Kernel“ oder „originaler Japi Kernel“ wird dementsprechend die gegebene Version des Japi Kernels bezeichnet.

Japi Call
Mit Japi Call wird die Anforderung bezeichnet, die vom Client an den Japi Kernel geschickt wird, damit dieser eine GUI Komponente erzeugt, manipuliert oder entfernt. Dabei beinhaltet der Japi Call das Senden einer ID („commandId“) und das Ausführen des entsprechend hinterlegten Quelltextes auf der Seite des Japi Kernels. Die Klasse `de.japi.Japi2Calls` beinhaltet alle möglichen IDs, die als Japi Calls aufgerufen werden können.

Command Socket & Action Socket
Der Japi Kernel hat zwei verschiedene Netzwerkverbindungen („sockets“) zu dem Client. Die Command-Verbindung ist synchron, d.h. der Client stellt eine Anfrage (z.B. Öffnen eines neuen Fensters) und erhält *sofort* eine Antwort darauf (z.B. Fenster ist jetzt geöffnet). Die Action-Verbindung ist Ereignisbasiert und informiert den Client nur über das Eintreten eines Ereignisses (z.B. Maus geklickt auf Komponente XY). Diese Verbindung ist also eine Einbahnstraße in Richtung des Clients.

3 Grundlagen

Dieses Kapitel verdeutlicht die Architektur von Japi2 und bereitet auf das Kapitel 4 vor, in dem beschrieben wird, wie Japi2 einfach um weitere Komponenten ergänzt werden kann. Daneben wird in diesem Kapitel noch eine kurze Betrachtung der Performance des Japi2 Kernels durchgeführt.

3.1 Grundstruktur

Das Paket `de.japi` ist das Hauptpaket. Hierin befinden sich die wichtigsten Klassen, die für die Verwaltung des Japi Kernels und seiner Ressourcen zuständig sind. In Abbildung 2 ist eine Übersicht über die enthaltenen Dateien dargestellt. Diese erfüllen folgenden Zweck:

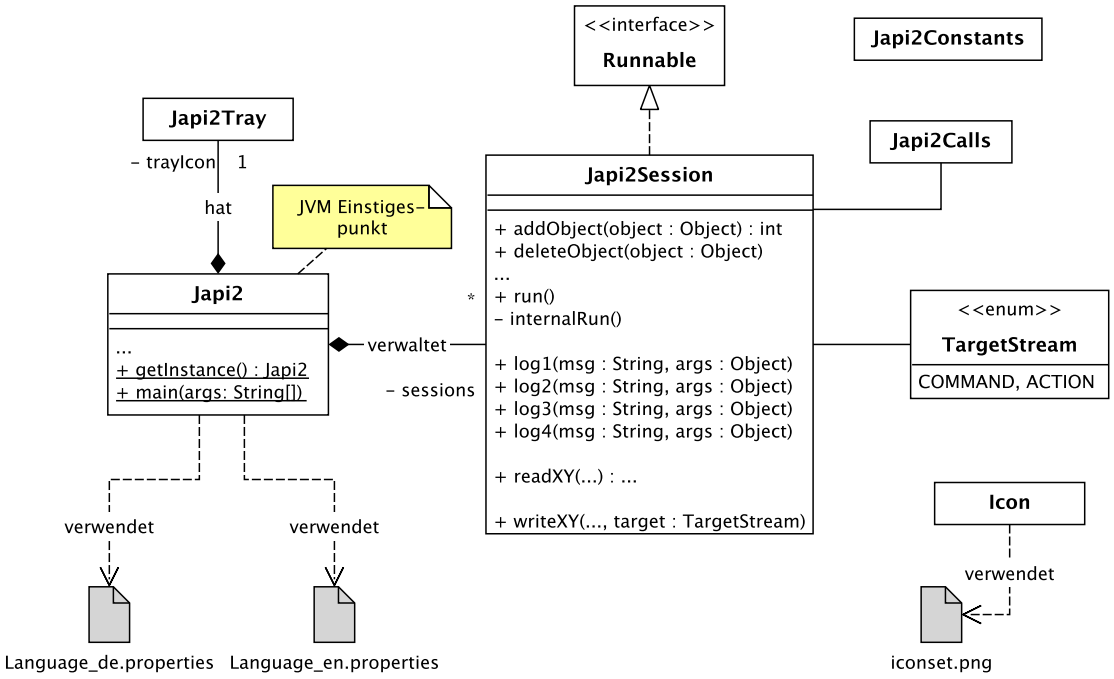


Abbildung 2: UML Klassendiagramm des Hauptpakets `de.japi`, das alle enthalteten Artefakte in Beziehung setzt.

- **Japi2** - Haupteinstiegspunkt der JVM. Diese Klasse ist ein *Singelton* und kann über eine statische Methode aus jedem Teil des Projekts aufgerufen werden. Sie bietet unter anderem Zugriff auf das Logging, sowie alle aktiven Sitzungen und stellt eine Methode bereit, um den gesamten Kernel ordungsgemäß zu beenden.

- **Japi2Tray** - Diese Klasse organisiert das Icon des Kernels im System-Tray (oder auch *Benachrichtigungsfeld*). Hierüber kann der Benutzer offene Sitzungen abbrechen oder den gesamten Kernel beenden. Zusätzlich stellt diese Klasse einige Methoden zur Verfügung, um Fehlermeldungen des Japi2 Kernels für den Benutzer darzustellen.
- **Japi2Constants** - alle Konstanten, die im Japi Kernel verwendet werden, sind hier gesammelt, so dass die Werte leicht geändert werden können und nicht über den gesamten Quelltext verteilt sind.
- **Japi2Calls** - diese Datei enthält alle Japi Call IDs als Konstanten, die im Switch-Statement von `Japi2Session` verwendet werden.
- **Japi2Session** - Diese Klasse stellt das wichtigste Bindeglied zwischen Client und Japi2 Kernel dar. Pro Clientverbindung[2] wird eine Instanz dieser Klasse erzeugt. Sie verwaltet jegliche Ressourcen, die in Zusammenhang mit dieser Clientsitzung stehen. Daher ist es möglich, dass auf einem physikalischen Rechner mehrere Sitzungen (und somit mehrere GUI's) geöffnet sind.
- **Icon** - Diese Klasse kümmert sich lediglich um das Auslesen der Icons für die verschiedenen Programmteile (Fehlermeldung, Programmicon, ...) aus der Icon-Map `iconset.png`.
- **Language_xy.properties** - Diese Dateien sind für die Internationalisierung zuständig. Sie enthalten die Programmtexte, die innerhalb des Kernels verwendet werden, in verschiedenen Sprachen. Um den Japi2 Kernel in eine neue Sprache zu übersetzen, muss hier nur eine weitere Datei erzeugt werden, wobei `xy` durch den Namen der entsprechenden Sprache nach ISO 639 Alpha-2 ersetzt wird. Die Werte der in dieser Datei vorhandenen Schlüssel müssen nun nur noch in die gewünschte Sprache übersetzt werden. Verwendet wird diese Sprache dann, wenn dies entweder explizit über Kommandozeilenparameter des Programms gefordert wird oder das System, auf dem der Japi2 Kernel läuft, diese Sprache verwendet. Mit dieser Internationalisierung sind aber nur alle „Systemmeldungen“ des Japi2 Kernels gemeint und nicht die benutzerspezifischen graphischen Oberflächen. Hier muss der Programmierer selbst für die Internationalisierung sorgen, falls benötigt.

3.2 Die Hauptklasse `Japi2Session`

Wie in Abbildung 2 angedeutet, stellt die Klasse `Japi2Session` eine breite Funktionalität zur Verfügung, um alle Japi Calls abzuarbeiten. Eine neue Sitzung wird durch den Aufruf des Konstruktors als eigenständiger Thread ausgeführt. Hierzu ist die Methode `run()` vorhanden, die durch das `Runnable` Interface vorgegeben ist und in dem eigenständigen Thread ausgeführt wird. Da während der Abarbeitung der Japi Calls auch Fehler *oder* Exceptions auftreten können, wird intern in `run()` eine weitere Methode aufgerufen, deren Exceptions abgefangen werden und eine ordentliche Beendigung der Sitzung im Falle eines fatalen Fehlers erlaubt.

Innnerhalb dieser `internalRun()`-Methode läuft eine einfache *while*-Schleife, in der immer wieder ein Kommando vom Client abgefragt wird. Dieses Kommando - der sogenannte Japi Call - wird dann verwendet, um den entsprechenden Code des Japi Calls auszuführen.

[2] Eine Clientverbindung besteht im Rahmen der Architektur des Japi Kernels aus zwei `Sockets`: ein *Action*-Socket und ein *Command*-Socket. Das *Action*-Socket ist nur für die Weiterleitung eventueller Action-Events (von beispielsweise Listenern) an den Client zuständig und das *Command*-Socket wird vom Client synchron angefragt, um GUI Objekte zu manipulieren.

Neben dieser Hauptschleife stellt die Klasse `Japi2Session` auch noch Methoden zum Lesen von dem Command-Socket und Schreiben auf das Command- und Action-Socket zur Verfügung. Entsprechend der Methodennamen, die immer mit „read“ beginnen, werden verschiedenste Typen wie Integer-Zahlen oder Byte-Werte eingelesen. Um eine hohe Geschwindigkeit beim Lesen des Stroms zu erreichen, ist es empfehlenswert, häufiger „low-level“ Operationen mit Byte-Buffern zu verwenden. Gerade wenn größere Datenmengen gelesen werden, stellen diese eine erheblich beschleunigte Möglichkeit dar, die Daten einzulesen.

Neben den Methoden zum Lesen bietet `Japi2Session` auch Methoden zum Schreiben auf das Command-Socket und das Action-Socket. Alle diese Methoden beginnen mit „write“ und verlangen als Argument eine Angabe des Enums `TargetStream`. Hiermit wird spezifiziert, auf welchen Strom die Daten geschrieben werden: entweder auf das Command-Socket für synchrone Rückmeldungen auf Anfragen vom Client oder auf das Action-Socket für Antworten von Listenern. Da wesentlich häufiger Antworten auf das Command-Socket geschrieben werden, stehen hierzu alle „write“-Methoden ohne das Argument `TargetStream` zur Verfügung. Gemeint ist dann immer das Command-Socket als Ziel der Daten. Diese Methoden sind lediglich syntaktischer Zucker.

Ein weiterer wichtiger Punkt ist die Organisation der vom Client erzeugten GUI Objekte. Auch diese Aufgabe übernimmt `Japi2Session`. Die Hauptmethode ist hierbei `int addObject(Object object)`, die ein GUI Objekt unter einer Sitzungseindeutigen Identifikationsnummer (ID) abspeichert. Über weitere Methoden können diese Objekte anhand ihrer ID oder die ID anhand eines Objekts wieder abgefragt werden. Wird die Sitzung beendet - z.B. durch einen Aufruf von `Japi2Session#exit()` - wird dafür gesorgt, dass alle GUI Objekte ordentlich vernichtet werden und der *Garbage Collector* wird explizit angestoßen, um wieder Speicher freizugeben.

3.3 Organisation der Quelltexte der Japi Calls

Die eigentliche Implementierung der einzelnen Japi Calls ist in dem Package `de.japi.calls` organisiert. Dort befinden sich Java Klassen, die ausschließlich statische Methoden enthalten. Jede statische Methode ist für die Abarbeitung eines spezifischen Japi Calls zuständig. Aufgerufen wird diese Methode dann, bei Anfrage des entsprechenden Japi Calls, aus dem Switch-Statement der `internalRun()`-Methode in `Japi2Session`.

Eine solche Methode hat folgendes generisches Aussehen:

```
public static void pack(Japi2Session session, Window window) throws
    IOException {
   session.log2("Pack {0}", window);
   window.pack();
}
```

- Die Methoden sind statisch, da sie zustandslos nur auf den gegebenen Eingaben agieren. Daher wird als erstes Argument immer die aktuelle Sitzung, für die dieser Japi Call ausgeführt werden soll, angegeben. Dies erlaubt dann den Zugriff auf das Logging für diese Sitzung sowie das Lesen und Schreiben von Argumenten auf die Datenströme.
- Es folgen weitere Argumente, die zur Ausführung benötigt werden. Hier wird beispielsweise das Fenster als Argument benötigt, das „gepackt“ werden soll.

- Die Methode kann beliebge Exceptions werfen. Beim Aufruf wird darauf geachtet, dass alle diese Exceptions ordnungsgemäß abgefangen werden. Sollte innerhalb einer solchen Japi Call Methode festgestellt werden, dass dieser auf einen gegebenen Parameter nicht anwendbar ist, so kann eine `UnsupportedOperationException` Exception geworfen werden (mit einer aussagekräftigen Nachricht). Diese wird vom Japi2 Kernel als solche erkannt und in einem entsprechenden Fehlerdialog dargestellt.
- Der Rückgabewert ist egal und wird nicht beachtet. Daher bietet sich hier `void` an.

Sollte ein Japi Call mehrere verschiedene Objekte behandeln, so werden einfach verschiedene statische Methoden geschrieben, mit verschiedenen Parametern. Beispielsweise ist der Japi Call 2.078 (`JAPI_ENABLE`) dafür zuständig, verschiedene GUI Elemente zu aktivieren. Daher liegen hierzu verschiedene statische Methoden vor, die als Parameter verschiedene Objekte haben. In dem großen *switch*-Statement in `Japi2Session` sieht dies folgendermaßen aus:

```
...
 case 2078 /* JAPI_ENABLE */:
    if (obj instanceof Component)
          CommandCalls.enable(this, (Component) obj);
    else if (obj instanceof Japi2Menu)
          CommandCalls.enable(this, (Japi2Menu) obj);
    else if (obj instanceof Japi2MenuItem)
          CommandCalls.enable(this, (Japi2MenuItem) obj);
    else if (obj instanceof JCheckBoxMenuItem)
          CommandCalls.enable(this, (JCheckBoxMenuItem) obj);
    else
          throw new NotHandledException();
    break;
 ...
```

Zunächst wird geprüft, welchen Typ der gegebene Parameter des Japi Calls, `obj`, hat. Falls dieser auf eine verfügbare Methode passt, so wird diese aufgerufen. Andernfalls hat der Client einen illegalen Japi Call getätigt, der mit einer Fehlermeldung geahndet wird. Es ist natürlich denkbar, dass jede gegebene Methode einen anderen Namen hat. Zum besseren Verständnis wurde jedoch bei der Entwicklung auf einheitliche Namen Wert gelegt.

Nach diesem Stil sind alle Japi Calls implementiert und thematisch auf die entsprechenden Dateien aufgeteilt:

- **CommandCalls** - beinhaltet alle Japi Calls, die Kommandos (wie beispielsweise pack oder enable) enthalten.
- **ConstructionCalls** - alle Japi Calls, die GUI Elemente erzeugen.
- **GraphicCalls** - graphische Japi Calls, die der Veränderung von Grafiken dienen.
- **LayoutCalls** - Japi Calls, die Layouts erzeugen.
- **ListenerCalls** - alle Japi Calls, die Listener erzeugen und an Objekte anhängen.
- **QuestionCalls** - Japi Calls, die „Fragen beantworten“. I.d.R. kommt dies einem *Getter*-Aufruf gleich. Ein Beispiel hierfür wäre also 3.092 (`JAPI_GETXPOS`), um die X-Koordinate der übergebenen Komponente zu ermitteln.

3.4 Japi2 GUI Komponenten und weitere Klassen

Neben den vorgestellten Paketen stellt das Paket `de.japi.components` ein weiteres wichtiges Paket dar. Es beinhaltet die GUI Elemente des Japi2 Kernels. Da es sich bei diesem Projekt um eine Portierung von dem AWT-Framework auf das Swing-Framework handelt, und die API des Japi Kernels beibehalten werden muss, um eine Funktion mit den bisherigen Clients zu gewährleisten, müssen viele Komponenten an die Verhaltensweise von AWT angepasst werden. Aus diesem Grund bietet dieses Paket fast alle durch den Benutzer erzeugbaren GUI Elemente als Spezialisierungen der Swing GUI Elemente. Die Japi2 GUI Elemente enthalten Anpassungen, um die Funktion den AWT Komponenten nachzuempfinden. Weitere Details hierzu sind im Kapitel „Kurzübersicht aller GUI Komponenten“ (Seite 22) zu finden.

Die übrigen Pakete beinhalten folgende Dateien:

- Das Paket `de.japi.components.layout` enthält alle Japi2 Kernel spezifischen Layouts. Diese wurden aus dem originalen Japi Kernel übernommen. Es handelt sich hierbei lediglich um ein fixes Layout (in Swing auch *Null*-Layout genannt) und ein horizontales sowie vertikales Flusslayout.
- Im Paket `de.japi.components.listeners` sind alle Listenerklassen enthalten, die im Japi2 Kernel verwendet werden. Sobald ein Listener ein Event empfängt (z.B. wenn der Benutzer einen Button klickt und ein ActionEvent erzeugt), wird die ID des Listeners über den Action-Stream übertragen. Auf der Client Seite von Japi kann nun auf dieses Event reagiert werden.

3.5 Bauen des Japi2 Kernels

Der Japi2 Kernel wurde unter Verwendung der kostenlos verfügbaren NetBeans IDE in der Version 8.0 für Java entwickelt[3]. Das Projekt kann jederzeit wieder importiert werden und mit der NetBeans IDE weiterentwickelt werden. Über dieses Programm kann auch das gesamte Projekt bequem kompiliert und in ein JAR-Archiv verpackt werden.

Daneben ist es auch möglich, das Projekt über die Kommandozeile zu komplieren und zu packen. Dazu befinden sich im Projekthauptverzeichnis zwei spezielle Dateien: das `Makefile`, welches mit dem Kommando `make` aufgerufen wird und die Kontrolle an die zweite Datei, `make.sh`, übergibt. Dieses Shell-Skript kompiliert die Java-Quelldateien, verpackt diese in ein JAR-Archiv, komprimiert das JAR-Archiv noch zusätzlich und platziert dieses im Hauptverzeichnis des Projekts. Die Ausgabe auf der Kommandozeile sieht folgendermaßen aus:

```
$ make

--- Japi2 Kernel Build Script ---

--> Creating tmp directory and copying files
--> Compiling Japi2 Kernel
Note: ./de/japi/components/Japi2Led.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.
```

[3] `https://netbeans.org/`

```
Note: Some input files use unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
--> Cleanup of java resources
--> Packaging into JAR file
--> Generating manifest
--> Compressing JAR file
     Original size....:   197269 bytes
     Compressed size..:   169634 bytes

Finished. File: /home/.../japi2/japi2.jar
```

Es ist zu beachten, dass hierbei das JDK-Programm `pack2000` verwendet wird, welches das erzeugte JAR-Archiv nochmals komprimiert (u.a. Debug-Informationen entfernt). Wie zu erkennen, werden hiermit weitere 27 KB gespart. Es ist also ratsam, Versionen für den *produktiven* Einsatz über dieses Skript zu kompilieren.

3.6 Kommandozeilenparameter

Das lauffähige JAR-Archiv des Japi2 Kernels verfügt über vier Kommandozeilenparameter, über die gewisse Grundfunktionen gesteuert werden können. Der generische Aufruf sieht folgendermaßen aus:

```
$ java -jar japi2.jar [0-65535] [-d|o|l:[iso639]]
```

- Der erste optionale Parameter ist die Portnummer, auf dem der Japi2 Kernel wartet. Hier kann eine Zahl zwischen 0 und 65535 angegeben werden. Es ist aber zu beachten, dass i.d.R. die Portnummer größer als 1024 sein sollte, da viele Ports kleiner als 1024 für Systemdienste reserviert sind und hier eine Firewall intervenieren könnte. Falls dieser Parameter nicht angegeben ist, wird der Wert der Konstante `JAPI_PORT` (Klasse `Japi2Constants`) verwendet.
- Der Schalter `-d` aktiviert den erweiterten Debug-Modus. Es wird empfohlen, diese Option nur während der Entwicklungszeit zu verwenden, da hierbei viele Informationen auf der Standardausgabe ausgegeben werden. Diese Option hat nichts mit dem Debugging zu tun, der vom Client durch die Funktion `j_setdebug()` gesetzt wird.
- Der Schalter `-o` aktiviert „optimiertes Rendering“. Hierbei wird global für fast alle Japi2 Komponenten, die selbst gezeichnet werden, Antialiasing aktiviert. Dadurch werden die Zeichnungen geglättet dargestellt. Eine Komponente, die dies unterstüzt ist z.B. `Japi2SevenSegment`.
- Der Schalter `-l:[iso639]` ist zur Internationalisierung des Japi2 Kernels zuständig. Der Teil `[iso639]` kann vom Benutzer frei festgesetzt werden und stellt einen Platzhalter für das Sprachkürzel nach ISO 639 alpha-2 dar. Ein Beispiel wäre `de` für Deutsch oder `en` für Englisch. Durch das Setzen dieses Parameters wird die Oberfläche des Japi2 Kernels (Fehlermeldungen, Systemtray, ...) in der entsprechenden Sprache dargestellt, oder, falls diese nicht verfügbar ist, in Englisch dargestellt. Es ist zu beachten, dass jegliche GUI-Objekte, die vom Client geöffnet werden, hiervon unberührt sind. Dieser muss selbst für eine enventuelle Internationalisierung sorgen.

3.7 Performancevergleich

In diesem Abschnitt soll kurz überprüft werden, wie sich die Laufzeit des neu entwickelten Japi2 Kernels im Vergleich zum originalen Kernel verhält. Es ist dabei aber zu beachten, dass der Schwerpunkt dieses Projekts nicht auf der Entwicklung einer performanteren Version lag, sondern auf der Entwicklung eines modernen Japi Kernels, der nicht mehr das veraltete und einschränkende AWT-Framework verwendet.

In dem Diagramm 3 ist die gemessene Zeit für die Ausführung des Beispiels `canvas.c` dargestellt. Dieses Beispiel wurde gewählt, da es einen sehr hohen Traffic hat - es werden zufällig viele Pixel gezeichnet - und sich somit gut zum „Belastungstest“ eignet. Das Beispiel wurde so abgeändert, dass es genau 500.000 Punkte zeichnet und sich dann beendet. Die Messwerte wurden auf einem Standardnotebook ermittelt[4].

An dem Diagramm kann deutlich abgelesen werden, dass der „originale“ Japi Kernel performanter läuft, d.h. mehr Japi Calls in weniger Zeit abarbeiten kann: für dieses Beispiel ist der Japi2 Kernel rund 11,4% langsamer. Um den Grund hierfür aufzudecken, wurde das gleiche Beispiel nochmals laufen gelassen und währenddessen wurde der Japi2 Kernel „geprofiled“ . Das Ergebnis ist in Abbildung 4 dargestellt. Dabei können folgende Beobachtungen gemacht werden:

34,1% der Zeit, die die Anwendung im Switch-Statement verbringt, werden dafür aufgewendet, vom Netzwerkstream zu lesen (nächster Japi Call, Argumentobjekt). Betrachtet man die Methode `drawLine` näher, so kann man dort auch erkennen, dass 99,1% der Zeit in dieser Methode (4861 ms von total 4905 ms) für das Lesen von dem Netzwerkstream (Koordinaten) benötigt wird und nur 0,8% für die Ausführung des eigentlichen Japi Calls aufgewendet wird. Das Lesen vom Netzwerkstream benötigt also einen Großteil der Ausführungszeit. Es gibt aber noch weitere Faktoren, die dazu beitragen, dass der Japi2 Kernel langsamer ist als der „originale“:

[4]Intel Core i5, 1,8 GHz, 4GB Arbeitsspeicher, Mac OS 10.9.4, Java 1.8.0_11.

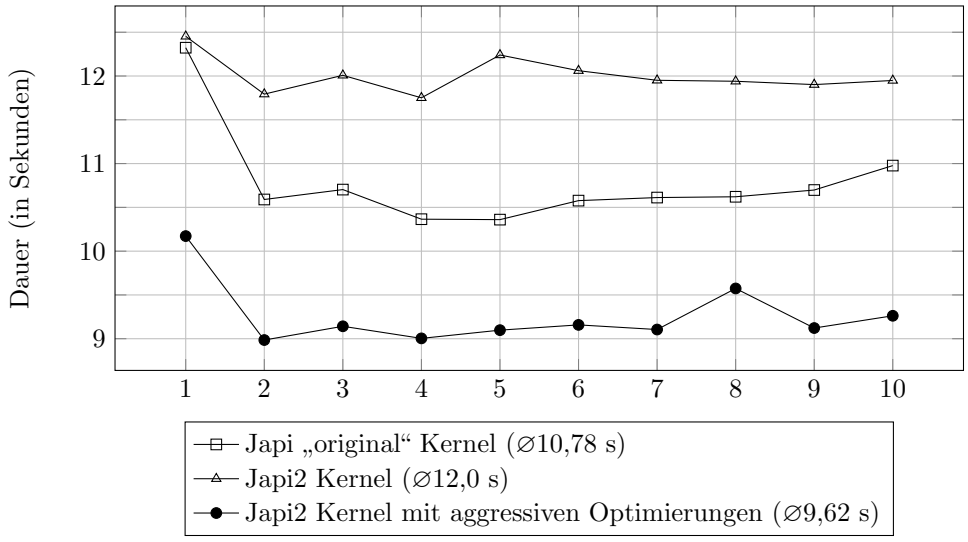
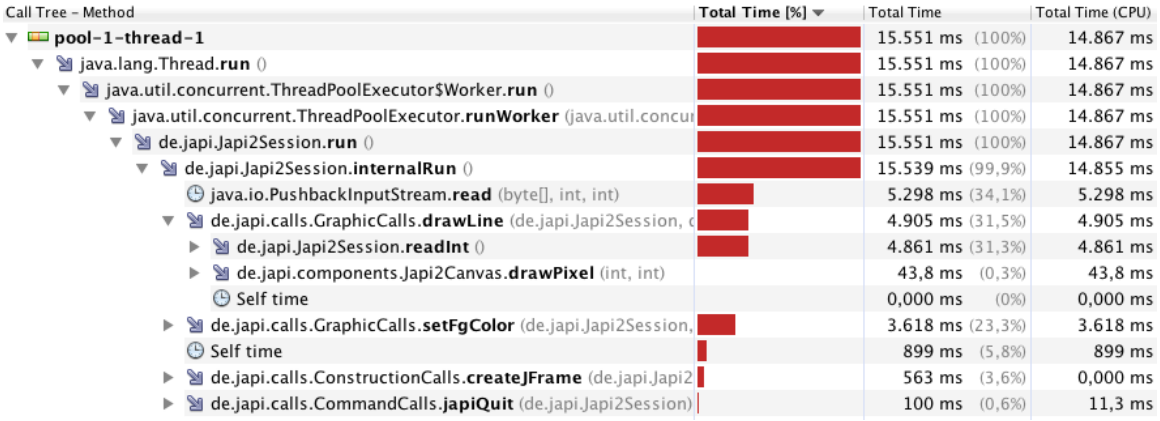


Abbildung 3: Vergleich der Laufzeiten des „originalen“ Japi Kernels und des Japi2 Kernels für das modifizierte `canvas.c`: zeichnen von 500.000 Punkten.



Call Tree – Method	Total Time [%]	Total Time	Total Time (CPU)
pool-1-thread-1		15.551 ms (100%)	14.867 ms
java.lang.Thread.**run** ()		15.551 ms (100%)	14.867 ms
java.util.concurrent.ThreadPoolExecutor$Worker.**run** ()		15.551 ms (100%)	14.867 ms
java.util.concurrent.ThreadPoolExecutor.**runWorker** (java.util.concur		15.551 ms (100%)	14.867 ms
de.japi.Japi2Session.**run** ()		15.551 ms (100%)	14.867 ms
de.japi.Japi2Session.**internalRun** ()		15.539 ms (99,9%)	14.855 ms
java.io.PushbackInputStream.**read** (byte[], int, int)		5.298 ms (34,1%)	5.298 ms
de.japi.calls.GraphicCalls.**drawLine** (de.japi.Japi2Session, c		4.905 ms (31,5%)	4.905 ms
de.japi.Japi2Session.**readInt** ()		4.861 ms (31,3%)	4.861 ms
de.japi.components.Japi2Canvas.**drawPixel** (int, int)		43,8 ms (0,3%)	43,8 ms
Self time		0,000 ms (0%)	0,000 ms
de.japi.calls.GraphicCalls.**setFgColor** (de.japi.Japi2Session,		3.618 ms (23,3%)	3.618 ms
Self time		899 ms (5,8%)	899 ms
de.japi.calls.ConstructionCalls.**createJFrame** (de.japi.Japi2		563 ms (3,6%)	0,000 ms
de.japi.calls.CommandCalls.**japiQuit** (de.japi.Japi2Session)		100 ms (0,6%)	11,3 ms

Abbildung 4: Ergebnis des Profilings des Japi2 Kernels für das modifizierte `canvas.c` Beispiel. Das Profile wurde mit dem in die NetBeans IDE integrierten Profiler erstellt.

- Der Japi2 Kernel hat eine kompliziertere Struktur. Der „originale“ Japi Kernel beinhaltet in der Klasse `JAPI_Thread` das Switch-Statement mit allen Japi Calls. Dadurch ist diese Klasse sehr umfangreich und nur schwer editierbar. Es entfallen jedoch Aufrufe weiterer Methoden und andere Aufenthaltspunkte.
- Das Lesen vom Netzwerkstream ist im Japi2 Kernel „hochsprachlicher“ umgesetzt. Während im „originalen“ Kernel mehr auf byte-Buffer gesetzt wurde, wird im Japi2 Kernel auf Methoden gesetzt, die diese Funktionalität übernehmen und so eine wartbarere Struktur ergeben. Das Ergebnis ist jedoch ein vergeichsweise langsamerer Code, da die objektorientierten Strukturen Zeit „beanspruchen“. Der Japi2 Kernel kann jedoch durch die Verwendung von aggressiven Optimierungen - Japi Call Implementierungen in das Switch-Statement integrieren und eine stärkere Verwendung von byte-Buffern - eine bessere Performance als der „originale“ Japi Kernel erreichen, wie in Diagramm 3 dargestellt wird (dritter Datensatz, „Japi2 Kernel mit aggressiven Optimierungen“ , hat im Durchschnitt eine Laufzeit von 9,62 Sekunden und ist damit schneller als die originale Implementierung).

Da der Aspekt der hohen Performance nicht im Vordergrund dieses Projekts stand, wurden keine weiteren Maßnahmen ergriffen, um diese zu erhöhen. Es ist jedoch bei den hier vorgestellten Zahlen zu beachten, dass es sich nur um die Auswertung eines Beispiels handelt. Dieses Beispiel stellt durch seine Struktur, das Setzen von 500.000 willkürlichen Pixeln, einen *worst-case* dar. Die meisten GUI-Anwendungen benötigen keine hohe Performance, da Ereignisse durch den Benutzer ausgelöst werden und dies um Größenordnungen langsamer passiert. Dies relativiert die scheinbar schlechten Ergebnisse für den Japi2 Kernel. Durch die zuletzt dargestellten Zahlen (aggressive Optimierung) kann jedoch der Japi2 Kernel angepasst werden. Es ist sogar denkbar, nur performancerelevante Stellen zu optimieren, da beispielsweise bei der Erzeugung eines Textfeldes ein Großteil der Anwendungszeit für Objekterzeugung und Initialisierung verwendet wird und dies nicht weiter beschleunigt werden kann.

3.8 Maßnahmen zur Qualitätssicherung

Ein Hauptziel der Qualitätssicherung ist es zu überprüfen, ob eine Software die Aufgabenspezifikation erfüllt. Im Rahmen dieses Projekts wurde der erstellte Quelltext in einem sogenannten *Schreibtischtest* von den Teilnehmern einzeln durchgegangen. Zusätzlich wurde der Code in Form eines *Walk-throughs* von allen Projektteilnehmern gemeinsam inspiziert. Darüber hinaus wurde der Quelltext während der Entwicklung immer wieder getestet.

Im Bereich des Testen von Software gibt es zahlreiche Ansätze. Für dieses Projekt wurde eine Mischung aus *Black- und White-Box-Tests* verwendet. Dabei bezeichnet *Black-Box testen* die testgetriebene Entwicklung (*test-driven-development, TDD*), die eigentlich heute als besser angesehen, trotzdem aber oftmals nicht verwendet wird. Dabei werden die Testfälle „vor“ der Entwicklung des Programmes, nur anhand der Spezifikation, geschrieben. Im Gegensatz dazu werden *White-Box-Tests* während oder nach der Programmentwicklung geschrieben. Der offensichtliche Nachteil des *White-Box-Tests* ist der, dass der Entwickler das System bereits gut kennt und daher die gleichen Fehler, die ihm bereits beim Programmieren unterlaufen sind, auch beim Testen wiederholt werden.

Da die Spezifikation des Projekts in Form einer Software (Originalversion von Japi) vorlag, konnten keine Testfälle im herkömmlichen Sinne aus den Anforderungen abgeleitet werden. Die einzelne Komponenten wurden deshalb mit Hilfe der bereits in der Vorgängerversion existierenden Beispiele getestet, die dann als *Black-Box-Tests* galten.

Es ist zu beachten, dass durch die Methode des Testens niemals die Abwesenheit von Fehlern bewiesen werden kann, nur durch ausreichende Anzahl von Testfällen die Wahrscheinlichkeit ihres Auftretens gering gehalten werden kann. Dabei ist anzumerken, dass auch Testfälle Software sind, und somit nicht frei von Fehlern sind.

Im Rahmen dieses Projekts wurde besonderer Wert auf die Benutzbarkeit und Wartbarkeit gelegt. Hierbei kommt es darauf an, wie gut und vor allem wie schnell die Software von anderen Personen (nicht den Entwicklern) verstanden und verändert werden kann. Daher folgt das Projekt den *Code Conventions* für Java, die 1999 von Sun festgelegt und seitdem kontinuierlich erweitert wurden. Als Beispiele sind hier Konventionen zur Benennung von Methoden und Variablen oder zur Instanziierung von Variablen zu nennen.

Weiterhin wurde der Quelltext ausführlich dokumentiert. Dabei wurden nicht nur einzelne Quelltextstellen dokumentiert, sondern auch die Klassen, Methoden und Attribute. Die Dokumentation erfolgte nach dem Java Standard *javadoc*. Dies bedeutet, dass eine Dokumentation zum Quelltext automatisch generiert werden kann[5]. Diese befindet sich dann im Projektverzeichnis (Unterverzeichnis `dist/javadoc/`).

Ein wichtiger Aspekt bei der Entwicklung von Japi war das Thema der Übertragbarkeit. Einzelne Komponenten sollten leicht austauschbar sein. So war es z.B. ein Ziel, Komponenten, die in zukünftigen Versionen eventuell nicht mehr gebraucht werden, leicht finden und löschen zu können. Ebenso einfach sollten Erweiterungen möglich sein. Eine komplette Anleitung zur Erweiterung von Japi2 um eine neue Komponente findet sich in Kapitel 4.

Der Japi2 Kernel wurde anhand der Beispiele auf den Betriebssystemen Mac OS, Linux und Windows erfolgreich getestet.

[5] Aufruf des Befehls `ant javadoc` im Projektverzeichnis erzeugt die Dokumentation.

4 Vorgehen zur Implementation einer neuen Komponente

In diesem Kapitel wird an einem einfachen Beispiel demonstriert, wie einfach weitere JAPI Calls in den im Rahmen dieses Projekts entwickelten Japi2 Kernel eingebaut werden können. Gezeigt wird dies am Beispiel eines `JSplitPane`*.*

4.1 Einleitung

Um die theoretischen Architekturüberlegungen des entwickelten Japi2 Kernels aus Kapitel 3 zu vertiefen und zu verdeutlichen, wie einfach dieser erweitert werden kann, wird in diesem Kapitel eine bisher nicht im Japi2 Kernel implementierte Komponte Schritt für Schritt entwickelt.

In Abbildung 5 ist die zu implementierende Komponente dargestellt: das `JSplitPane`. Diese Komponente ermöglicht es, zwei beliebige weitere Komponenten nebeneinander (*horizontal*) oder übereinander (*vertikal*) darzustellen und dynamisch per *Drag-n-Drop* die Komponenten zu vergrößern oder zu verkleinern. Hieraus ergibt sich sofort der Nutzen dieser Komponente: der Benutzer kann wichtige Elemente je nach Bedarf vergrößern oder verkleinern.

Abbildung 5: Dargestellt ist ein Screenshot (unter Mac OS X 10.9.4) der fertigen Demo des Japi2 Kernels für das `JSplitPane`. Das SplitPane hat eine horizontale Ausrichtung beider Komponenten (nebeneinander).

Die Komponente `JSplitPane` hat vielfältige Funktionalitäten. Damit dieses Kapitel nicht den Rahmen sprengt, werden nur folgende grundlegende Funktionen implementiert:

- Erzeugung eines neuen `JSplitPane` mit der Orientierung (*horizontalen*/*vertikale*) und der initialen Position des Trennstrichs

- Zuweisen der linken (oberen) Komponente
- Zuweisen der rechten (unteren) Komponente

4.2 Implementation des SplitPane in der `japilib`

Zunächst muss die Unterstützung für das SplitPane auf der Client-Ebene von Japi2, in der `japilib`, geschaffen werden. Diese Implementierung wird direkt in C vorgenommen. Da sich dieses Projekt hauptsächlich mit der Modernisierung der Japi2 Kernel Seite beschäftigt hat, in der Programmiersprache Java, werden die Änderungen nicht sonderlich detailliert dargestellt.

Zunächst müssen jedoch die neuen JAPI Calls in der Datei `japicalls.def` „registriert“ werden:

```
...
CMDGROUP JAPI_COMMANDS
...
JAPI_SETSPLITPANELEFT
JAPI_SETSPLITPANERIGHT
...

CMDGROUP JAPI_CONSTRUCTORS
...
JAPI_SPLITPANE
...
```

Nach der Definition der JAPI Calls kann nun mit der Implementierung der Japi Client Funktionen in C, in der Datei `japilib.c`, fortgefahren werden:

Listing 1: Erweiterung der `japilib` um drei weitere Funktionen für das „SplitPane“.

```
int japi_splitpane(int parent, int orientation, int init_position) {
  return send_4int_get_int(JAPI_SPLITPANE, parent, orientation, init_position);
}

int japi_setsplitpaneleft(int sp, int component) {
  send_3int(JAPI_SETSPLITPANELEFT, sp, component);
}

int japi_setsplitpaneright(int sp, int component) {
  send_3int(JAPI_SETSPLITPANERIGHT, sp, component);
}
```

Dabei sind die definierten Funktionen zur Bereitstellung folgender Funktionalitäten zuständig:

- Die Funktion `japi_splitpane` erzeugt ein neues SplitPane und fügt dieses zu dem Objekt `parent` hinzu. Das neu erzeugte SplitPane hat darüber hinaus noch folgende Eigenschaften, die mit angegeben werden müssen:
  - `orientation` gibt an, ob die Komponenten vertikal oder horizontal ausgerichtet werden. Hierzu sind nur die Konstanten `J_HORIZONTAL` oder `J_VERTICAL` zu verwenden.

  - `init_position` gibt die initiale Position des Trennstreifens in Pixeln an, gemessen von links (horizontale Orientierung) oder von oben (horizontale Orientierung).

- Die Funktion `japi_setsplitpaneleft` setzt die Komponente `component` für ein SplitPane `sp` als linke oder obere Komponente (entsprechend der gesetzten Orientierung).
- Die Funktion `japi_setsplitpaneright` setzt die Komponente `component` für ein SplitPane `sp` als rechte oder untere Komponente (entsprechend der gesetzten Orientierung).

Mithilfe dieser drei Funktionen lässt sich nun das Quellcodebeispiel zu Abbildung 5 formulieren, dass die Grundfunktionalität des `JSplitPane` verdeutlicht:

Listing 2: Beispiel in C zur Demonstration der neuen Funktionalität „SplitPane“.

```
#include <stdio.h>
#include <stdlib.h>
#include "japi.h"

int main() {
   int frame, sp, panel1, panel2;
   if(!j_start())
      printf("can't connect to server\n"),exit(0);

   frame = j_frame("Split Pane Demo");
   j_setborderlayout(frame);

   sp = j_splitpane(frame, J_HORIZONTAL, 142);
   j_setborderpos(sp, J_CENTER);

   panel1 = j_panel(sp);
   j_setcolorbg(panel1, 255, 0, 0);
   j_label(panel1, "Linkes Panel");

   panel2 = j_panel(sp);
   j_setcolorbg(panel2, 255, 255, 0);
   j_label(panel2, "Rechtes Panel");

   j_setsplitpaneleft(sp, panel1);
   j_setsplitpaneright(sp, panel2);

   j_show(frame);

   while (1)
      if (j_nextaction() == frame)
         break;

   j_quit();
   exit(0);
}
```

Dabei wird das SplitPane folgendermaßen erzeugt: In Zeile 13 wird ein neues horizontales SplitPane erzeugt, mit dem Trennstreifen bei einer Postion von `142px`, gemessen von der linken Seite des SplitPanes. Das neu erzeuge SplitPane wird zu dem Fenster hinzugefügt und die Referenz zurück

gegeben. Dann werden in den Zeilen 16-18 und 20-22 die beiden Komponenten erzeugt. Hier werden einfach farbige Panels mit einem Label zur visuellen Identifizierbarkeit erzeugt. Die Anweisungen in den Zeilen 24 und 25 sorgen dann dafür, dass die Panels zu dem SplitPane hinzugefügt werden.

Achtung: Es ist zu beachten, dass der JAPI Call `panel1 = j_panel(sp)` ein Panel erzeugt, aber dieses nicht zum ScrollPane hinzufügt. Dies ist auch nicht möglich, da hier die Information, an welcher Seite (links oder rechts / oben oder unten) das Panel eingefügt werden soll, nicht vorliegt. Es ist daher immer noch ein Aufruf der Funktion `j_setsplitpaneleft` oder `j_setsplitpaneright` nötig. Für alle anderen Komponenten gilt jedoch, dass `j_panel(parent)` ein neues Panel erzeugt und direkt zu `parent` hinzufügt.

4.3 Implementation der Japi Calls im Japi2 Kernel (Java)

Nun soll es darum gehen, die Unterstützung für die Komponente SplitPane auch auf der Seite des Japi2 Kernel (der GUI) bereitzustellen. Dies wird direkt im Java-Projekt gemacht, am Besten mit einer entsprechenden IDE wie beispielsweise NetBeans.

Sobald ein Client mit dem Japi2 Kernel verbunden wurde, wird für diesen eine neue Sitzung eröffnet. Dies wird durch die Klasse `Japi2Session` übernommen. Wie in Kapitel 3 bereits beschrieben, übernimmt diese Klasse sämtliche Aufgaben bezüglich Objektspeicherung, Kommunikation mit dem Client sowie das sitzungsbasierte Logging. Daneben ist in dieser Klasse die Hauptmethode `internalRun()` zu finden. In dieser Methode befindet sich das Switch-Statement, das immer wieder einen Japi Call entgegennimmt und dann die entsprechende Methode aufruft, die dann die nötige Logik ausführt. Um also die neuen drei Japi Calls

- `JAPI_SPLITPANE` (4136)
- `JAPI_SETSPLITPANELEFT` (2114)
- `JAPI_SETSPLITPANERIGHT` (2115)

zu unterstützen, muss das Switch-Statement um drei weitere Fälle erweitert werden, je einer für jeden neuen Japi Call. Insgesamt ist das Switch-Statement also um den nachfolgenden Quellcode zu erweitern.

Listing 3: Erweiterung des Switch-Statemens in der Klasse `Japi2Session` um drei neue Fälle zur Realisierung der Funktionalität des SplitPanes.

```
...
// Find the right method for handling
try {
   switch (command) {
     ...
     case Japi2Calls.JAPI_SETSPLITPANELEFT:
          if (obj instanceof JSplitPane)
             CommandCalls.setLeftSplitPaneComponent(this, (JSplitPane) obj);
          else
             throw new NotHandledException();
          break;
       case Japi2Calls.JAPI_SETSPLITPANERIGHT:
          if (obj instanceof JSplitPane)
             CommandCalls.setRightSplitPaneComponent(this, (JSplitPane) obj);
```

```
            else
               throw new NotHandledException();
            break;
        ...
        case Japi2Calls.JAPI_SPLITPANE:
            if (obj instanceof Container)
               ConstructionCalls.japiSplitPane(this, (Container) obj);
            else
               throw new NotHandledException();
            break;
         ...
      }
   } catch(Exception ...) {
      ...
   }
```

Wie in dem Listing erkennbar, ist die Erweiterung um einen Japi Call einfach: Es wird zunächst ein neuer Fall im Switch-Statement mit der entsprechenden Japi Call ID aus `Japi2Calls` erzeugt. In diesem Fall befindet sich nun eine Verzweigung. Die Variable `obj` steht im Bereich des Switch-Statements zur Verfügung und beinhaltet das Objekt, auf dem dieser Japi Call ausgeführt werden soll, oder ist `null`, falls der Japi Call auf keinem Objekt ausgeführt wird. Dieser Fall ist aber sehr selten und tritt in der Regel nur bei zustandslosen Aktionen auf, wie z.B. `JAPI_BEEP` oder `JAPI_SYNC`.

Der Zweck dieser Verzweigung ist es, zu definieren, auf welche Argumenttypen der aufgerufene Japi Call anwendbar ist. Der Else-Zweig einer solchen Verzweigung ist dafür zuständig, das Ereignis abzufangen, dass der Japi Call auf einem Argument aufgerufen wird, auf das der Japi Call *oder seine Semantik* nicht angewendet werden kann (nur ein Soundfile kann abgespielt werden mit diesem Japi Call). Beispielsweise könnte der Japi Call `JAPI_PLAYSOUNDFILE` nicht auf ein Label angewendet werden. Daher wird in diesem Else-Zweig stets eine `NotHandledException` geworfen. Diese wird vom Japi2 Kernel erkannt und es wird darauf entsprechend reagiert. Über die einzelnen If-Zweige werden dann alle zulässigen Argumenttypen definiert. Ein Beispiel hierzu wäre der Japi Call `JAPI_DISPOSE`, der für das „Schließen“ eines Fensters oder „Entfernen“ einer Komponente von ihrer Elternkomponente zuständig ist:

Listing 4: Fall für den Japi Call Japi Call `JAPI_DISPOSE` im Switch-Statement mit mehren Argumenttypen und Überladung der einzelnen Methoden.

```
...
case Japi2Calls.JAPI_DISPOSE:
   if (obj instanceof Window)
      CommandCalls.japi2Dispose(this, (Window) obj);
   else if (obj instanceof Component)
      CommandCalls.japi2Dispose(this, (Component) obj);
   else if (obj instanceof JComponent)
      CommandCalls.japi2Dispose(this, (JComponent) obj);
   else if (obj instanceof Japi2PrintJob)
      CommandCalls.japi2Dispose(this, (Japi2PrintJob) obj);
   else
      throw new NotHandledException();
   break;
```

...

Wichtig ist hierbei noch, dass man eine Reihenfolge über die Platzierung der einzelnen Zweige definieren kann: tauscht man beispielsweise den Zweig aus Zeile 5/6 mit dem Zweig aus Zeile 3/4, so wird der Kernel nicht mehr erwartungsgemäß funktionieren, wenn man ihn auf einem Window-Objekt aufruft. Denn: Ein `Window` ist gleichzeitig auch ein `Component`. Es wird also niemals der Fall, der speziell für Argumente vom Typ `Window` geschaffen wurde, ausgeführt werden. Und im Falle einer Component wird versucht, dieses aus seiner Elternkomponente zu entfernen. Das ist aber nicht für Window definiert (ein Window ist kein Teil einer Elternkomponente) und es wird einfach ein Aufruf der `dispose`-Methode erwartet. Stattdessen wird eine Exception geworfen und das Programm verabschiedet sich ungraziös.

Die eigentliche Ausführung der Aktionen des Japi Calls findet allerdings in einer seperaten Methode statt. Alle diese Methoden sind in einer Klasse in dem Paket `de.japi.calls` zu finden. Diese Methoden sind alle statisch und nach Kategorien auf die verschiedenen Klassen aufgeteilt, um eine gewisse Übersicht zu gewähren. Ein weiterer Grund für die Auslagerung der Semantik ist die Übersichtlichkeit. Natürlich könnte man in das Switch-Statement den gesamten Code aus den statischen Methoden einfügen. Dann wäre jedoch eine unübersichtliche Codestruktur gegeben. So kann die Semantik übersichtlich in einzelne Methoden verpackt werden, die auch noch wiederverwertet werden können. Das nachfolgende Listing zeigt die Methoden, die für die Japi Calls des SplitPane-Beispiels implementiert werden müssen:

Listing 5: Implementation der Methoden mit der Semantik der Japi Calls, auf die in den neuen Fällen des Switch-Statments aus Listing 4 verwiesen wurde.

```
// Klasse: de.japi.calls.ConstructionCalls
...
public static void japiSplitPane(Japi2Session session, Container c) throws
    IOException {
   int orient = session.readInt();
5  int initPos = session.readInt();
   session.log1("SplitPane in parent object {0} with orientation {1} and "
         + "divider location at {2}", c, orient, initPos);

   if (orient != Japi2Constants.J_HORIZONTAL &&
10       orient != Japi2Constants.J_VERTICAL) {
      throw new UnsupportedOperationException("SplitPane can only be horizontal
          or vertical.");
   }

   JSplitPane splitPane = new JSplitPane(
15       orient == Japi2Constants.J_HORIZONTAL ?
             JSplitPane.HORIZONTAL_SPLIT :
             JSplitPane.VERTICAL_SPLIT
   );
   splitPane.setOneTouchExpandable(false);
20 splitPane.setDividerLocation(initPos);
   c.add(splitPane);

   int oid = session.addObject(splitPane);
   session.writeInt(oid);
```

```
}
...

// Klasse: de.japi.calls.CommandCalls
...
public static void setLeftSplitPaneComponent(Japi2Session session, JSplitPane
    sp) throws IOException {
   int argId = session.readInt();
   session.log2("Set as left split pane component: {0} for {1}", argId, sp);
   Component c = session.getObjectById(argId, Container.class);
   sp.setLeftComponent(c);
}

public static void setRightSplitPaneComponent(Japi2Session session, JSplitPane
    sp) throws IOException {
   int argId = session.readInt();
   session.log2("Set as right split pane component: {0} for {1}", argId, sp);
   Component c = session.getObjectById(argId, Container.class);
   sp.setRightComponent(c);
}
...
```

Per Konvention hat so eine statische Methode als erstes Argument immer die Klasse `Japi2Session`, da diese alle notwendigen Methoden zur Datenkommunikation und zum Logging enthält, auf die während der Abarbeitung des Japi Calls sicherlich zugegriffen werden muss. Handelt es sich um einen zustandslosen Japi Call (wie `JAPI_BEEP` oder `JAPI_SYNC`), so sind keine weiteren Argumente erforderlich. Andernfalls folgt die Komponente, auf der der Japi Call ausgeführt werden soll. Da jegliche Typüberprüfungen im Switch-Statement vorgenommen werden, kann die entsprechende Methode direkt den expliziten Typ verlangen. Wie in Listing 4 dargestellt, kann dann die Methodenüberlagerung von Java verwendet werden, um die entsprechende Methode aufzurufen. Die Benennung der Methoden ist egal und auch Methodenüberlagerung muss nicht verwendet werden.

Weiterhin kann die Methode beliebige Exceptions werfen. Diese werden automatisch im Switch-Statement abgefangen und entsprechend behandelt. Sie enden zumeist in einer Benachrichtigung des Benutzers und der Option, die aktuelle Sitzung abzubrechen. Eine besondere Bedeutung kommt jedoch der `UnsupportedOperation`-Ausnahme zu. Diese kann geworfen werden, wenn der Japi Call weitere Argumente erwartet und diese nicht in der gewünschten Form vorliegen. Die Ausnahme sollte dann unbedingt eine Fehlermeldung haben - diese wird dem Benutzer mit angezeigt.

Praktisch ist dieses „Feature“ am Beispiel der Methode `japiSplitPane` im Listing 5 zu sehen (Zeilen 9 - 13). Der Japi Call erwartet eine Orientierung für das SplitPane. Die Orientierung ist als einfache Integer-Zahl kodiert. Um das System erweiterbar zu halten, wurden Konstanten definiert. Für die Orientierung sind jedoch nur `J_HORIZONTAL` oder `J_VERTICAL` semantisch valide Optionen. Vom reinen Typ her könnte aber auch eine in diesem Kontext unsinnige Konstante, wie beispielsweise `J_NULL`, übergeben werden. Das Verhalten des Japi Calls ist in diesem Fall nicht definiert. Daher wird eine `UnsupportedOperation`-Ausnahme geworfen.

Darüber hinaus sind im Listing 5 noch weitere „Eigenschaften“ oder „Konventionen “ für Implementationen von Japi Calls erkennbar:

- Zunächst werden über das `Japi2Session` Objekt weitere Argumente gelesen und in Variablen zur Verwendung gehalten: Hier in Zeile 4/5 sowie 32 und 39.
- Darauf folgt meistens eine entsprechende Debug-Ausgabe über eine entsprechende Methode zum Logging der Klasse `Japi2Session`. Da diese Logging-Ausgaben sitzungsbehaftet sind, werden diese nur auf der Konsole bei global eingeschaltetem Debug-Modus ausgegeben[6], oder - falls vom Client aktiviert - in dem „DebugWindow“ aufgeführt. Diese Logging-Ausgaben dienen der Nachvollziehbarkeit des Kontroll- und Datenflusses.
- Als nächstes wird die eigentliche Operation durchgeführt (siehe Zeilen 11 - 21 sowie 33/34 und 40/41).
- Bei der Erzeugung von Komponenten oder Japi Calls mit Rückgaben müssen diese noch übermittelt werden. Dies wird am Ende der Methode vorgenommen. In diesem Beispiel wird in den Zeilen 23 und 24 zunächst die Referenz auf das neu erzeugte SplitPane in der Sizung gespeichert (damit ein späterer Zugriff darauf möglich ist) und dann über das `Japi2Session` Objekt die ID als Referenz zurück übertragen.

Rückgabewerte sind für diese statischen Methoden nicht vorgesehen und werden auch nicht benötigt. Allerdings kann die Architektur jederzeit um dieses Feature erweitert werden.

4.4 Abschließende Gedanken

In diesem Kapitel wurde der gesamte Weg zum Einbau einer neuen Funktionalität - hier am Beispiel des SplitPane - gezeigt: von der Erweiterung der `japilib` in C bis hin zur Änderung des Japi2 Kernels in Java.

Die Komponente SplitPane ist durch dieses Kapitel bereits in der aktuellen Version umgesetzt, funktioniert mit dem vorgestellten Beispiel und ergibt das in Abbildung 5 dargestellte GUI Beispiel.

Natürlich sind weitere Japi Calls denkbar, die die Benutzung des SplitPane noch angenehmer gestalten. Solche wären beispielsweise:

- Setzen der minimalen und maximalen Größe der linken und rechten Komponente.
- Aktivieren und deaktivieren der „One Touch“- Funktionalität. Hierduch werden im Trennerstrich kleine Pfeile dargestellt, die bei einem Klick die gesamte Komponente verstecken und die andere Komponente mit der gesamten zur Verfügung stehenden Größe darstellt.
- Abfrage der Position des Trennerstrichs, *auch wenn der Nutzen fragwürdig ist, da man diesen Wert nur setzt.*

Um dieses Beispiel jedoch kompakt zu halten und auf die Erklärung der Struktur besser eingehen zu können, wurde diese weitere Funktionalität ausgelassen. Außerdem ist das SplitPane in dieser Implementierung vollkommen funktionsfähig und folgt somit dem Motto von Japi, die Komponenten so einfach wie möglich zu halten.

[6] Es wird empfohlen, diesen Debug-Modus nur während der (Weiter-)Entwicklung des Japi2 Kernels durch den entsprechenden Kommandozeilenparameter zu aktivieren.

5 Kurzübersicht aller GUI Komponenten

In diesem Kapitel werden die einzelnen Komponenten und ihre Funktionsweise erklärt und auf Unterschiede zwischen der orignal- und der neuen Version eingegangen.

Nachfolgend wird nun jede Klasse des Pakets `de.japi.components` kurz erläutert und ihre *Besonderheiten* angegeben (falls vorhanden).

1. **AbstractDrawable**

   Kurzbeschreibung: Diese Klasse erbt von `JComponent` und überschreibt deren paint-Methoden.

   Besonderheiten: Sie ist Superklasse für die meisten Klassen, die sich selbst zeichnen. Dabei kapselt `AbstractDrawable` die verschiedenen `paint`-Methoden und stellt eine einheitliche Methode zum Zeichnen bereit, die einfach überschrieben werden kann.

2. **AbstractJapi2ValueComponent**

   Kurzbeschreibung: Die Klasse erweitert `AbstractDrawable` und stellt Funktionalitäten für selbstgezeichnete Komponenten zur Verfügung, die Integer-Werte darstellen. Ein Beispiel wäre das `Japi2Meter`, das diese Klasse implementiert. Das Setzen von (Anzeige-)Werten für ein Meter wird komplett von der abstrakten Klassen übernommen.

3. **Japi2AboutWindow**

   Kurzbeschreibung: Die Klasse erbt von `JFrame` und stellt das „Über ...“-Fenster des Japi2 Kernels dar, das Versions- und Lizenzinformationen bietet.

4. **Japi2DebugWindow**

   Kurzbeschreibung: Diese Klasse erbt von `JDialog`. In einem separaten Fenster werden zur einzelnen `Japi2Session` die Logging Nachrichten, die durch deren logging-Methode (`log1, log2, log3 und log4`) für die einzelnen Komponenten erzeugt werden, angezeigt.

   Besonderheiten: Wie in der Originalversion werden verschiedene Logging-Level unterstützt. Eine Log-Nachricht wird genau dann in der Debug-Konsole angezeigt, wenn ihr Level geringer ist als das globale Level, das mit `setLevel(int)` für diese Konsole vorher gesetzt wurde.

5. **Japi2Frame**

   Kurzbeschreibung: `Japi2Frame` erbt von der Swing Klasse `JFrame`. Die Klasse erzeugt einen neuen Frame. Als Standardlayout wird ein `Japi2FixLayout` gesetzt.

   Besonderheiten: Im Gegensatz zu der AWT-Komponente `Frame` unterstützt `JFrame` keine `Inset` mehr. Um die alte Funktionalität trotzdem zu gewährleisten, besitzt die Klasse eine privates `JPanel` mit einem leerem Rand, für den über die Methode `setInsets(Insets)` die Randbreite gesetzt wird. So wird das AWT-Konzept `Inset` in Swing nachgebildet.

6. **Japi2Canvas**

   Kurzbeschreibung: `Japi2Canvas` ist eine Klasse, die ein rechteckiges Feld, auf das gezeichnet werden kann, erstellt. Dementsprechend erbt die Klasse von `AbstractDrawable`.

   Besonderheiten: Diese Komponente ist automatisch doppelt gepuffert, d.h. sie zeichnet sich automatisch neu, falls dies erforderlich ist. Darüber hinaus geschieht ein Neuzeichnen auch nur, wenn der Benutzer eine Änderung an der Zeichenfläche vornimmt (oder das Betriebssystem dies verlangt). Hierdurch wird eine geringe CPU-Auslastung erreicht, da in einem Großteil der Zeit keine Zeichenoperationen durchgeführt werden und dann auch kein „repaint“ Ereignis ausgelöst wird.

7. **Japi2Button**

   Kurzbeschreibung: Die `Japi2Button` Klasse ist eine geradlinige Erweiterung der `JButton` Klasse, die einen anklickbaren Button erstellt. Sie enthält einen ActionListener, um auf Nutzereingaben zu reagieren.

8. **Japi2MenuBar**

   Kurzbeschreibung: Erweiterung der Klasse `JMenuBar`, die eine Menu-Zeile erstellt. Einzelne Menüs können in diese Menu-Zeile eingeordnet werden. Die Schriftart, die für die MenuBar gewählt wird, gilt für alle ihre Menüs.

9. **Japi2Menu**

   Kurzbeschreibung: Die Klasse erweitert `JMenu`. Sie erstellt ein Menu in einer MenuBar.

10. **Japi2MenuItem**

    Kurzbeschreibung: Die Klasse erweitert `JMenuItem`. Sie erstellt ein MenuItem, das jeweils einem Menü zugeordnet ist. `Japi2MenuItem` besitzt einen ActionListener zum Erkennen, ob das MenuItem angeklickt wurde.

11. **Japi2CheckMenuItem**

    Kurzbeschreibung: `Japi2CheckMenuItem` ist der Klasse `Japi2MenuItem` sehr ähnlich. Die Klasse erzeugt Menüeinträge, die selektiert und deselektiert werden können. Sie besitzt einen ItemListener, um die Selektierung des Eintrages zu erkennen.

12. **Japi2Panel**

    Kurzbeschreibung: Die Klasse erweitert `JPanel`. Ein Panel ist ein Container-Objekt zum Kapseln anderer Komponenten. Es enthält einen ComponentListener, um auf Veränderungen der Position, Größe oder Visibilität einer Komponente reagieren zu können. Als Standardlayout ist ein `Japi2FixLayout` gesetzt.

    Besonderheiten: Ein Unterschied zu der Originalversion von Japi ist an dieser Stelle, dass der Rahmen nicht von dieser Komponente selbst in der `paint`-Methode gezeichnet wird, sondern eine eigene `Japi2PanelBorder` Klasse verwendet wird, welche diese Operation übernimmt.

13. **Japi2PanelBorder**

Kurzbeschreibung: Neue Klasse, die von `AbstractBorder` erbt. Hiermit ist es möglich, die vier Bordertypen LINEDOWN, LINEUP, AREADOWN und AREAUP darzustellen, die in der Originalversion zur Verfügung stehen. Durch die Abstraktion in eine eigene Border-Klasse kann diese auf jede beliebige Swing-Komponente angewendet werden.

14. **Japi2PopupMenu**

Kurzbeschreibung: Einfache Abstraktion, um das AWT `PopupMenu` mit der gleichen Japi Semantik (getrennte Erzeugung und Anzeige) weiterhin mit der Swing Komponente, `JPopupMenu`, zu verwenden.

15. **Japi2Label**

Kurzbeschreibung: `Japi2Label` basiert auf `Label` und erzeugt einen Bereich, auf dem Texte oder Bilder dargestellt werden können.

16. **Japi2CheckBox**

Kurzbeschreibung: Diese Klasse erbt von `JCheckBox` und erstellt selektierbare Check Boxen. Um auf die Selektion zu reagieren, besitzt die Klasse einen ItemListener.

17. **Japi2RadioGroup**

Kurzbeschreibung: Erweiterung von `ButtonGroup`, die dazu verwendet wird, dass jeweils immer nur ein Element selektiert sein kann.

Besonderheiten: Im Gegensatz zum Original erbt diese Klasse nicht von `CheckboxGroup`, sondern von `ButtonGroup`. Während es in AWT für die verschiedenen Button Typen auch verschieden Gruppen gibt, werden in Swing alle Arten von Buttons in der `ButtonGroup` zusammengefasst. Die Gruppierung in einer `ButtonGroup` ist eine rein logische, keine physikalische Gruppierung.

18. **Japi2RadioButton**

Kurzbeschreibung: `Japi2RadioButton` erweitert die Klasse `JRadioButton`, die einen selektierbaren Button erstellt und aus diesem Grund auch einen ItemListener besitzt.

Besonderheiten: Der Konstruktor der Klasse nimmt hier nur noch den Titel der RadioGroup als Argument und nicht mehr auch die ButtonGroup, zu der der Button gehört. Diese wird in der `showRadioButton()`-Methode der `ConstructionCalls`-Klasse gesetzt.

19. **Japi2List**

Kurzbeschreibung: `Japi2List` ist eine Anpassung der Komponente `JList` für den Japi2 Kernel, um Listen darzustellen.

Besonderheiten: Da die AWT-Komponente `List` automatisch einen Scrollbalken besitzt, wurde diese Funktionalität durch `Japi2List` nachgebildet. `Japi2List` ist also zunächst ein `JScrollPane`, das ein `JList` Objekt *hat* und listenspezifische Methoden implementiert und direkt an die `JList` weiterreicht. So wird die Illusion geschaffen, dass sich `Japi2List` genau so wie die AWT-Komponente `List` verhält. Daneben wurden noch kleinere Anpassungen vorgenommen, wie z.B. die Erweiterung um einen `ActionListener` für die Selektion eines Eintrags, da in Swing Listenselektionen anders behandelt werden, aber die Kompatibilität zu der Japi API gewährleistet werden muss.

20. **Japi2Choice**

Kurzbeschreibung: `Japi2Choice` erweitert die `JComboBox` Komponente, die in Swing die alte `Choice`-Komponente ersetzt hat. `JComboBox` stellt eine Drop-Down Liste zur Verfügung. `Japi2Coice` besitzt einen ItemListener, um die Selektion einzelner Listenelemente zu erkennen.

21. **Japi2TextArea**

Kurzbeschreibung: Die Klasse spannt ein mehrzeiliges Feld zur Texteingabe auf.

Besonderheiten: Im Unterschied zur AWT Komponente `TextArea` besitzt `JTextArea` nicht automatisch Scrollbalken. Dieses Problem wird hier gelöst, indem die Komponente eine `JScrollPane` erweitert und darin eine `JTextArea` angelegt wird. Der `DocumentListener` erfasst Änderungen des Textes und ruft ggf. die Japi spezifischen Listenermethoden auf.

22. **Japi2TextField**

Kurzbeschreibung: `Japi2TextField` ist eine Klasse, die von `JTextField` erbt. Sie erstellt ein einzeiliges Text Feld, in das Text, bis zu einer maximalen Anzahl ein Zeichen, eingegeben werden kann.

Besonderheiten: In AWT gibt es nur *eine* Textfeld-Komponente, die sich um die Eingabe von normalem Text und um die Eingabe von maskiertem Text, z.B. Passwörtern, kümmert. In Swing sind diese Funktionen in zwei Komponenten aufgeteilt. Deshalb wird hier die interne Klasse `Japi2EchoCharDocument` erstellt, die es ermöglicht, auch in einem normalen `JTextField` die eingegebenen Zeichen zu maskieren.

23. **Japi2Dialog**

Kurzbeschreibung: Die Klasse erweitert `JDialog` und erstellt ein Dialogfenster. Als Standardlayout wird das `Japi2FixLayout` gesetzt.

24. **Japi2Window**

Kurzbeschreibung: `Japi2Window` erstellt ein neues Fenster und erbt von der Klasse `JWindow`.

25. **Japi2ScrollPane**

Kurzbeschreibung: Die `Japi2ScrollPane` Klasse ist eine neu hinzugekommene Klasse und erbt von `JScrollPane`. Sie stellt das Gegenstück zu der AWT-Komponente `ScrollPane` dar und bietet die gleiche Funktionalität.

26. **Japi2AlertDialog**

Kurzbeschreibung: Diese Klasse stellt einen modalen Informationsdialog, Fehlerdialog sowie mehrere Fragedialoge zur Verfügung, die dem Nutzer der GUI kurze Informationen oder Entscheidungen präsentieren.

27. **Japi2GraphicButton**

Kurzbeschreibung: Die Klasse erweitert die `JButton` Klasse und erlaubt die Darstellung von Icons auf Buttons.

28. Japi2GraphicLabel

Kurzbeschreibung: `Japi2GraphicLabel` erbt von `JLabel` und stellt ein Bild auf einem Label dar.

29. Japi2Ruler

Kurzbeschreibung: `Japi2Ruler` erweitert die Klasse `AbstractDrawable` und überschreibt deren `draw(Graphics2D, int, int)`-Methode, um eine Trennlinie zu zeichnen (vgl. Swing Komponente `JSeparator`).

30. Japi2PrintJob

Kurzbeschreibung: Diese Klasse ist eine komplette Neuentwicklung, um das Drucken von Komponenten und Grafiken auch weiterhin zu ermöglichen.

Besonderheiten: Der Grund, aus dem hier eine zusätzliche Klasse erstellt wurde und nicht einfach die javaeigene `PrintJob` Klasse verwendet wurde, ist, dass es mit `PrintJob` nicht möglich ist, die Größe des Randes zu berücksichtigen. Deshalb kann es passieren, dass außerhalb des Druckbereichs gezeichnet wird. Um das Problem zu umgehen, kümmert sich die `Japi2PrintJob` Klasse automatisch um die Beachtung des Randes.

31. Japi2Image

Kurzbeschreibung: Auch `Japi2Image` ist eine neu hinzugekommene Klasse. Sie erbt von `BufferedImage` und wird verwendet, um einen verfrühten Aufruf der Methode `dispose()` zu verhindern.

Besonderheiten: Wenn die `dispose()` Methode des Grafik-Objekts aufgerufen wird, wird das gesamte Bild gelöscht. Führt man dann später weitere graphische Aktionen auf diesem Bild aus, ist das Bild „leer“. Es muss die `dispose()`-Methode der `Japi2Image` Klasse aufgerufen werden, um das Bild ordnungsgemäß zu schließen. Aufrufe der `dispose()`-Methode der Grafik haben keinen Effekt.

32. Japi2Led

Kurzbeschreibung: Diese Klasse erbt von `AbstractJapi2ValueComponent` und simuliert eine LED-Leuchte, die an- oder ausgeschaltet werden kann.

Besonderheiten: Anstelle der Methoden `setState()` und `getState()` der Originalversion sind hier die Methoden `setOn()` und `isOn()` getreten. Ihre Funktionsweise ist allerdings gleich geblieben. Die LED wird dann als „an“ erkannt, wenn der Wert der Superklasse auf 0xff gesetzt ist, und als „aus“, wenn der Wert 0x00 beträgt.

33. Japi2SevenSegment

Kurzbeschreibung: Sie erweitert die Klasse `AbstractJapi2ValueComponent`. Die Klasse erlaubt die Darstellung von 7-Segment-Anzeigen. Mit der `setValue(int)`-Methode kann ein Wert zwischen 0 und 15 eingegeben werden, der dann über die Anzeige im Stil einer 7-Segment-Anzeige ausgegeben wird.

34. Japi2Meter

Kurzbeschreibung: Diese Klasse erweitert die `AbstractJapi2ValueComponent` Klasse. Sie stellt eine Messuhr graphisch dar.

35. **Japi2Slider**

    Kurzbeschreibung: Diese Klasse erbt von `JSlider` und implementiert die Interfaces `Adjustable` und `ChangeListener`. Sie ermöglicht dem Anwender, einen Zahlenwert in einem bestimmten Intervall durch die Bewegung eines Slider auszuwählen.

    Besonderheiten: Diese Klasse ersetzt die vormals verwendete Scrollbar-Komponente aus AWT, um einen Schieberegler für Werte zu realisieren. In Swing ist mit `JSlider` bereits so eine Komponente umgesetzt. Die Japi Calls `setUnitIncrement` und `setBlockIncrement` haben nun keine Funktion mehr, da sie Scrollbar spezifisch sind und nicht mehr in die Semantik des `JSlider` passen.

Im Anschluss an die GUI Komponenten werden nun die Layout Komponenten, die sich im Paket `de.japi.components.layout` befinden, vorgestellt.

1. **Japi2FixLayout**

    Kurzbeschreibung: Die Klasse `Japi2FixLayout` implementiert das Interface `LayoutManager`. Sie erlaubt die Nutzung von Komponenten mit fixer Positionsangabe.

2. **Japi2GridLayout**

    Kurzbeschreibung: Diese Klasse wird dazu verwendet das Layout auf die Komponente erneut anzuwenden, nachdem die Anzahl der Spalten oder Zeilen geändert wurde. So wird diese immer korrekt angezeigt. Während der Entwicklung hat sich herausgestellt, dass ein einfacher Aufruf der Methoden zum setzen der Zeile oder Spalte nicht ausreicht und das Layout erneut angewendet werden muss.

3. **Japi2HorizontalFlowLayout**

    Kurzbeschreibung: Die Klasse `Japi2HorizontalFlowLayout` erbt von `FlowLayout` und erlaubt eine variable, von der Fensterhöhe abhängige Positionierung von Elementen.

4. **Japi2VerticalFlowLayout**

    Kurzbeschreibung: Die Klasse erbt von `FlowLayout` und ermöglicht eine variable, von der Fensterbreite abhängige Positionierung ihrer Elementen.

Zum Schluss werden nun die Listener Komponenten im Paket `de.japi.components.listeners` erklärt.

1. **AbstractJapi2Listener**

    Kurzbeschreibung: `AbstractJapi2Listener` ist eine neu hinzugekommen Klasse, die als Elternklasse für alle folgenden Listener verwendet wird.

    Besonderheiten: Diese Klasse ist dafür zuständig, dass für die konkreten Listener korrekte Logging-Informationen ausgeben werden und das Debug Level gesetzt wird. Außerdem kapselt diese Klasse den Mechanismus, um den Japi Client über den Action-Stream über das Eintreten eines Listenerevents zu informieren.

2. **Japi2ActionListener**

    Kurzbeschreibung: Diese Klasse erbt von `AbstractJapi2Listener`, implementiert das Interface `ActionListener` und ist für das Empfangen von Action Events zuständig.

3. **Japi2AdjustmentListener**

   Kurzbeschreibung: Diese Klasse erbt von `AbstractJapi2Listener` und implementiert das Interface `AdjustmentListener`.

4. **Japi2TextListener**

   Kurzbeschreibung: `Japi2TextListener` erbt von `AbstractJapi2Listener` und implementiert das Interface `TextListener`. Wenn sich der Text einer Komponente ändert, wird die entsprechende Methode dieser Klasse aufgerufen.

5. **Japi2ItemListener**

   Kurzbeschreibung: Die Klasse erbt von `AbstractJapi2Listener`, implementiert das Interface `ItemListener` und reagiert darauf, wenn ein Item selektiert wird.

6. **Japi2MouseListener**

   Kurzbeschreibung: Diese Klasse erbt von `AbstractJapi2Listener`, implementiert das Interface `MouseListener` und empfängt Mausklicks.

   Besonderheiten: Im Konstruktor wird die ID per default auf -1 gesetzt. Um keine Fehlermeldung zu erhalten, muss die ID manuell mit `setId(int)` auf einen gültigen Wert gesetzt werden. Diese Besonderheit gilt auch für den nun folgenden `Japi2MouseMotionListener`.

7. **Japi2MouseMotionListener**

   Kurzbeschreibung: Diese Klasse erbt von `AbstractJapi2Listener`, implementiert das Interface `MouseMotionListener` und empfängt Mausbewegungen.

8. **Japi2WindowListener**

   Kurzbeschreibung: Diese Klasse erbt von `AbstractJapi2Listener` und implementiert das Interface `WindowListener`. Sie erkennt Änderungen im Status eines Fensters, z.B. Statusänderungen wie geöffnet, geschlossen, aktiviert, deaktiviert usw.

   Besonderheiten: Auch hier gilt, dass die ID auf -1 gesetzt wird und manuell mit der Methode `setId(int)` gesetzt werden muss.

9. **Japi2KeyListener**

   Kurzbeschreibung: Diese Klasse erbt von `AbstractJapi2Listener` und implementiert das Interface `KeyListener`. Sie ist für das empfangen von Tastertureingaben zuständig.

   Besonderheiten: Im Konstruktor des Listeners wird dieser bereits als Objekt der Sitzung (durch `Japi2Session#addObject()`) registriert und die ID gespeichert. Dies wird in den beiden folgenden Listenern ebenso gehandhabt, aber nur einmal an dieser Stelle erwähnt.

10. **Japi2FocusListener**

    Kurzbeschreibung: Diese Klasse erbt von `AbstractJapi2Listener` und implementiert das Interface `FocusListener`. Sie erkennt, ob eine Komponente aktuell den Focus hat oder nicht.

11. **Japi2ComponentListener**

    Kurzbeschreibung: Diese Klasse erbt von `AbstractJapi2Listener` und implementiert das Interface `ComponentListener`. Hier werden Änderungen der Sichtbarkeit, Größe und Position der einzelnen Komponenten erkannt.

6 Ergebnis & Ausblick

Abschließend wird in diesem Kapitel noch einmal geprüft, welche Anforderungen erfüllt wurden und wie das Projekt zukünftig weiterentwickelt werden könnte.

6.1 Ergebnis dieses Projekts

Das Ziel dieses Projektes war die Entwicklung eines „modernen“ Japi Kernels, der nicht mehr auf dem veralteten AWT-Framework basiert, sondern auf dem Swing-Framework. Die drei Kerneigenschaften der Japi Bibliothek - intuitive Benutzung, Sprachenunabhängigkeit, Plattformunabhängigkeit - wurden in diesem Projekt auch erhalten. Im Folgenden werden die Anforderungen aus dem Anforderungsprofil zu diesem Projekt aufgelistet und es wird evaluiert, inwiefern diese erfüllt wurden:

- *„Die Sourcen [...] sind [...] alt [und] der Schwerpunkt [...] liegt daher in der Aufgabe, den JAPI Kernel auf einen aktuellen Stand zu portieren“* - Diese Anforderung wurde erfüllt, da der Japi2 Kernel das modernere Swing-Framework verwendet und auf einem modernen Sprachlevel der Programmiersprache Java, Version 8 (ohne funktionale Features), entwickelt wurde.
- *„[...] neben der SWING GUI auch moderne Aspekte der objektorientierten Programmierung [...]“* - In Kapitel 3 ist die Architektur des Japi2 Kernels beschrieben, die objektorientierte Ansätze verwendet, um die Struktur und Funktion zu unterstützen. So konnte beispielsweise im Japi2 Kernel an einigen Stellen durch Vererbungsbeziehungen (z.B. `AbstractDrawable`, `AbstractJapi2ValueComponent` oder `AbstractJapi2Listener`) der bestehende Code des Japi Kernels vereinfacht werden.
- *„[...] Portierung auf das Mac-OS [...]“* - der Japi2 Kernel kann auch problemlos unter Mac OS verwendet werden und integriert sich dank Implementation Mac spezifischer Funktionen (Dock-Icon, Mac Menu Bar) in das Aussehen von Mac OS. Daneben wurde der Japi2 Kernel auch unter Linux und unter Windows an den Beispielen erfolgreich getestet. Swing verwendet jeweils das System Look-And-Feel und integriert sich somit bestmöglich in das System.
- *„[...] eine Erweiterung der Oberflächenobjekte [...]“* - Im Rahmen des Kapitels 4 wurde am Beispiel des SplitPane beschrieben, wie man den Japi2 Kernel um eine neue Komponente oder generell Japi Calls erweitert. Diese neue Komponente SplitPane ist natürlich in die finale Version integriert und kann verwendet werden.
- *„Eine ausführliche Dokumentation [...]“* - Neben diesem Dokument ist auch der Quelltext extensiv dokumentiert und erlaubt es, eine automatische HTML-Dokumentation mittels `javadoc` zu erzeugen. Diese Dokumentation enthält alle Hinweise, die für eine Weiterentwicklung von Japi2 nötig sind.

Nachdem alle Anforderungen aus dem Anforderungsprofil erfüllt wurden, konnten noch weitere sinnvolle Funktionen in den Japi2 Kernel eingebaut werden, die die Benutzung vereinfachen. Bei diesen neuen Features handelt es sich um:

- **Integration in das Benachrichtigungsfeld** *(system tray)* - Der Japi2 Kernel läuft im Hintergrund. Dies macht es beispielsweise schwer, ihn zu beenden. Um dieses Problem zu lösen, wurde eine Integration des Japi2 Kernels in das Benachrichtigungsfeld *(system tray)* geschaffen. Dort wird das Programmicon angezeigt und mit einem Rechtsklick kann ein Kontextmenü geöffnet werden, das folgende Operationen erlaubt:
  - Beenden - Der Japi2 Kernel kann „geordnet“ beendet werden und Systemressourcen werden „geordnet“ wieder frei geben.
  - Entfernte Verbindungen erlauben - Da der Japi2 Kernel seine Befehle über Netzwerk erhält, wäre es denkbar, den GUI-Teil der Anwendung auf einem entfernten Rechner laufen zu lassen. Dieses Vorgehen kann durchaus nicht erwünscht sein. Daher kann diese Option verwendet werden, um entfernte Verbindungen (also nicht *localhost*) zu dem Japi2 Kernel zu unterbinden.
  - Logging aussetzen - Für Anwendungen, die Grafiken zeichnen, werden in der Regel viele Japi Calls benötigt. Da das Logging die Abarbeitung erheblich verlangsamt, kann dieses hierüber global für den Japi2 Kernel deaktiviert werden. Sollte zu dieser Zeit ein `Japi2DebugWindow` geöffnet sein, wird auch in diesem das Logging deaktiviert.
  - Über Japi2 - Bietet Informationen über den Japi2 Kernel wie Versionsnummer und Lizenz.
- **Internationalisierung** - Die Fehlermeldungen sowie das Menü des Benachrichtigungsfeldes sind durch Sprachdateien internationalisiert. Je nach System werden diese momentan[7] in Deutsch oder Englisch angezeigt. Auf Systemen mit einer anderen Sprache wird die englische Sprachdatei verwendet. Die Unterstützung einer neuen Sprache ist sehr einfach: eine existierende Sprachdatei wird verdoppelt, mit dem Kürzel der neuen Sprache benannt und alle Werte in der Datei in die neue Sprache übersetzt.
- **Aktualisierung der Icons** - Neben dem Programmicon wurden auch die Icons für Benachrichtigungsdialoge modernisiert und bilden nun ein einheitliches Gesamtbild. Die Icons liegen im Vektorgrafikformat vor, im Unterordner `art/` des Projektverzeichnisses.

6.2 Ausblick

Wie bereits vorgestellt (Kapitel 3.7), lässt sich die Performance des Japi2 Kernels durch weitere Optimierungen am Quelltext noch erhöhen. Ob dies immer sinnvoll ist, muss abgewägt werden, da GUI Applikationen durch den Nutzer mit Ereignissen versorgt werden, die Änderungen an der GUI vornehmen. Hierfür ist die vom Japi2 Kernel erreichte Geschwindigkeit mehr als ausreichend. Bei Zeichenoperationen werden jedoch in der Regel viele Japi Calls abgesetzt und hier würde sich eine Optimierung der Performance auszahlen. Daher ist es möglicherweise sinnvoll, den Japi2 Kernel nur an kritischen Stellen zu optimieren. Denn eine Optimierung auf Geschwindigkeit ist immer ein Kompromiss zwischen der Wartbarkeit bzw. Lesbarkeit des Quelltextes und der Geschwindigkeit: je stärker die Optimierungen sind, umso schlechter wird der Quelltext unter dem Aspekt der Wartbarkeit bzw. Lesbarkeit. Dennoch würde sich eine Optimierung auszahlen, wie die Abbildung 3 im Kapitel 3.7 verdeutlicht.

Mit der Portierung von AWT nach Swing steht der Erweiterung des Japi2 Kernels um Komponenten wie eine Baumansicht oder eine Tabelle nichts mehr im Weg.

[7] Mit Fertigstellung dieses Projekts sind Sprachdateien für die Sprachen Deutsch und Englisch angelegt.