j-Algo

The Algorithm
Visualisation Tool

Entwicklerhandbuch

Inhaltsverzeichnis

1 Einleitung

Dieses Handbuch soll künftigen Entwicklern von **j-Algo** helfen, sich schnell mit der Struktur der Software auseinanderzusetzen. **j-Algo** ist eine Software, die sich mit der Visualisierung von Algorithmen beschäftigt. Sie soll dazu dienen, verschiedene Algorithmen zu veranschaulichen um sie so Studenten und anderen Interessierten verständlicher zu machen. Die Anwendung basiert auf einer Plugin-Struktur, die es ermöglicht, einzelne Module, die jeweils einen Algorithmus oder ein Themengebiet abdecken können, in das Programm zu integrieren und zu laden.

Sowohl **j-Algo** als auch die einzelnen Module entstanden im Rahmen des externen Softwarepraktikums im Studiengang Informatik der TU Dresden in Zusammenarbeit mit dem Lehrstuhl Programmierung. Die implementierten Module orientieren sich daher an den Lehrveranstaltungen „Algorithmen und Datenstrukturen“ sowie „Programmierung“ im Grundstudium Informatik an der TU Dresden. Das Einsatzgebiet soll vor allem die Vorlesung und das studentische Lernen zu Hause umfassen.

j-Algo ist eine freie Software, die beliebig oft kopiert werden darf.

2 Technische Hinweise

2.1 Systemvoraussetzungen

Folgende minimale Systemanforderungen werden für den reibungslosen Einsatz von **j-Algo** benötigt:

- IBM-kompatibler PC
- Mindestens 64 MB RAM
- Windows 98(SE)/ME/2000/XP , Linux SuSE/Red Had
- Java 2 Platform Standard Edition 5.0 (siehe: http://java.sun.com/)
- Maus und Tastatur
- Monitor mit einer Auflösung von mindestens 800x600

2.2 Installation

Windows

Entpacken Sie nach dem Herunterladen das ZIP-komprimierte Archiv in einen Ordner Ihrer Wahl. In diesem Ordner finden Sie eine Datei namens „j-algo.bat“. Öffnen Sie diese Datei mit einem Doppelklick, und das Programm wird gestartet.

Unix

Entpacken Sie nach dem Herunterladen das TGZ-komprimierte Archiv in einen Ordner Ihrer Wahl. In diesem Ordner finden Sie eine Datei namens „j-algo.sh“. Öffnen Sie die Konsole und starten sie mittles `sh j-algo.sh` das Programm.

2.3 Deinstallation

Der komplette Programmordner kann jederzeit gefahrlos von der Festplatte gelöscht werden.

3 CVS-Zugang

j-Algo ist als Projekt bei SourceForge registriert und gehostet. Es gibt 2 Arten, auf das CVS-Repository des Projektes zuzugreifen.

1. Lesezugriff. Als Beobachter des Projektes kann jeder auf das Projekt lesend zugreifen. Die Zugangsdaten sind:
   Verbindungsmethode: `pserver`
   Host: `cvs.sourceforge.net`
   Repository-Pfad: `/cvsroot/j-algo`
   Login: `anonymous`

2. Vollzugriff. Als registrierter Entwickler bei SourceForge und als eingetragenes Projektmitglied bei **j-Algo** kann das CVS unter folgenden Zugangsdaten im Vollzugriff erreicht werden:
   Verbindungsmethode: `extssh`
   Host: `cvs.sourceforge.net`
   Repository-Pfad: `/cvsroot/j-algo`
   Login: <SourceForge-Login>
   Passwort: <SourceForge-Passwort>
   Um als Projektmitglied eingetragen zu werden, wenden Sie sich bitte an den Projekt-Administrator. Dessen Kontaktdaten sind auf der SourceForge-Seite zugänglich.

Achtung: Wird das Projekt im Rahmen des Software-Praktikums an der TUD weiterentwickelt, gelten andere Bedingungen für den CVS-Zugang. Diese sind beim zuständigen Betreuer des Praktikums zu erfragen.

4 Entwickeln unter Eclipse

Natürlich steht es jedem Entwickler frei, eine Programmierumgebung seiner Wahl zu benutzen. Da jedoch der Großteil der **j-Algo** -Entwickler unter Eclipse programmiert, und diese Plattform einige komfortable Features besitzt, sollen hier die wichtigsten Einstellungen für diese Umgebung erläutert werden. Für andere Programmierumgebungen gelten sie sinngemäß.
Da **j-Algo** die Java-Version 1.5 verwendet, ist eine Eclipse-Version 3.1 oder höher erforderlich.[1]

Das Projekt kann in der CVS-Ansicht von Eclipse ausgecheckt werden. Ab jetzt sind zwar alle nötigen Daten (Quellcodes, etc.) auf dem Rechner. Allerdings müssen noch einige Einstellungen vorgenommen werden, damit das Projekt kompiliert und gestartet werden kann:

- Unter den Projekteinstellungen->"Java Compiler"->„Compiler Compliance Level" ist „5.0" einzustellen. Es ist sicherzustellen, dass unter Projekteinstellungen->Libraries als „JRE System Library" die Version 1.5 eingestellt ist. Ist dies nicht der Fall, so ist diese mittels „Add Library..."->„JRE System Library" einzustellen.

- Unter Projekteinstellungen->Info->Text file encoding muss UTF-8 eingestellt werden. Dies garantiert reibungslose Unterstützung von Umlauten auf verschiedenen Betriebssystemen.

- Unter Projekteinstellungen->Java Build Path müssen jetzt einige Einstellungen für den ClassPath des Projektes vorgenommen werden:
  Unter Source darf nur der Ordner <PROJEKTORDNER>`/src` stehen. Ist dies nicht der Fall, ist dieser mittels „Add Folder..." aus der Ordnerliste auszuwählen. Andere Ordner sind zu entfernen.

- Als „Default Output Folder" ist <PROJEKTORDNER>`/bin` anzugeben.

- Unter „Libraries"->„Add JARs..." ist <PROJEKTORDNER>`/extlibs/jh.jar` hinzuzufügen. Dies ist die nötige Bibliothek für das Hilfe-System.

- Unter „Libraries"->„Add Class Folder..." sind folgende Ordner hinzuzufügen:
  <PROJEKTORDNER>`/runtime` (für die Erkennung der installierten Module)
  <PROJEKTORDNER>`/res/main` (für die Ressourcen zum Hauptprogramm)
  sowie alle verfügbaren Modulordner unter <PROJEKTORDNER>`/res/module/`, also z.B.
  <PROJEKTORDNER>`/res/module/testModule` (für die Ressourcen der einzelnen Module)

- Als nächstes werden die jUnit-Bibliotheken benötigt. Weil Eclipse diese bereits eingebaut hat, ist die einfachste Variante, diese hinzuzufügen, folgendermaßen:
  Projekteinstellungen übernehmen, Workspace neu kompilieren lassen, und dann im View „Problems" einen der vielen Fehler auswählen, die im Zusammenhang mit jUnit gebracht werden. Beim Öffnen des gewählten Source-Files zeigt Eclipse im Editor bei den entsprechenden Imports Fehler an. Drücken Sie genau dort auf das rote Kreuz, und Ihnen wird die Option angeboten „Add jUnit libraries". Wählen Sie diese aus, schließen das Source-File, und fertig.
  Jetzt sollte im View „Problems" kein Fehler mehr angezeigt werden.

[1] Achtung, auf der Seite des Eclipse-Projektes wird noch auf die Version 1.4.2 des JDK verlinkt

- Nun muss noch eine Startkonfiguration erstellt werden, und dann sind wir fertig: Unter dem Menüpunkt Run->Run... erstellen Sie eine neue Konfiguration vom Typ „Java Application“, vergeben einen sinnvollen Namen und wählen vom **j-Algo** -Projekt als „Main-Class“ `org.jalgo.main.JAlgoMain` aus.

Jetzt ist das Projekt kompilierbar und das Programm kann gestartet werden.

5 Projektstruktur

Es folgt ein kurzer Überblick über die bestehende Struktur des Projektes, so dass der Entwickler weiß, welche Teile er verändern darf, und welche besser unangetastet bleiben sollten...
Das Projekt fasst mehrere Ordner und einige „lose“ Dateien. Der Reihe nach:

- Der Ordner `bin` fasst die kompilierten Klassen. Sein Inhalt kann gelöscht werden, er wird bei jedem kompilieren neu erstellt. (Hinweis: Dieser Ordner gehört nicht unter die Versionskontrolle!)
- Der Ordner `doc` fasst die Projektdokumentation. Dies sind die Dateien zum Entwicklerhandbuch, zum Benutzerhandbuch, sowie einige Dateien, die gewisse aufgetretene Probleme und evtl. Abhilfen schildern.
- Im Ordner `examples` sind Beispieldateien für jedes Modul enthalten. Der komplette Ordner wird später in der Distribution enthalten sein.
- Im Ordner `extlibs` liegen Bibliotheken, die Fremdcode enthalten. Dies ist derzeit nur die Laufzeitbibliothek des Hilfesystems. Der komplette Ordner wird später in der Distribution enthalten sein.
- Der Ordner `relicts` fasst Codeteile und Ressourcendateien, welche derzeit nicht mehr verwendet werden. Sie wurden trotzdem aufgehoben, weil sie teilweise Funktionalität enthalten, die zu implementieren mal begonnen wurde, die jedoch nie ausgereift waren und daher derzeit nicht verwendet werden. Vielleicht bringen Sie einen Nutzen, wenn der Entwickler Ideen sucht.
- Im Ordner `res` liegen alle Ressourcendateien geordnet nach Programmteilen.
- Der Ordner `runtime` enthält leere, aber notwendige Dateien für die Laufzeit. Sie sind Teil der Pluginstruktur, und ermöglichen das Erkennen der installierten Module.
- Im Ordner `src` schließlich ist der Quellcode enthalten. Die Paketstruktur ist intuitiv gehalten. Unter `org.jalgo.main` findet sich alles, was zum Hauptprogramm gehört, und unter `org.jalgo.module` liegen alle Modulpakete.
- Die „losen“ Dateien sind diverse Build-Skripte, Manifest- und Start-Dateien für verschiedene Betriebssysteme sowie einige projektspezifische Dateien.

Teilweise wird in diesem Entwicklerhandbuch auf die API-Dokumentation von **j-Algo** verwiesen. Da an der Software permanent gearbeitet wird, ist diese nicht unter der Versionskontrolle verfügbar, sondern sollte vom Entwickler selbst in regelmäßigen Abständen generiert werden. Arbeitet der Entwickler unter Eclipse, so kann dies unter dem Menüpunkt Project->„Generate Javadoc...“ ganz einfach durchgeführt werden.

6 Implementieren eines neuen Moduls

Hier sollen nur die technischen Schritte angegeben sein, die nötig sind, ein neues Modul für j-**Algo** korrekt und vollständig aufzusetzen und in das Hauptprogramm zu integrieren. Es wird vorausgesetzt, dass der Entwickler selbst ein Konzept seines Moduls entwickelt, insbesondere, was Details der Visualisierung betrifft.

Es folgen fünf Abschnitte. Im ersten wird der Teil erklärt, der für das Implementieren eines neuen Moduls minimal notwendig ist. Der zweite Abschnitt enthält Erklärungen zur Funktionsweise der Pluginstruktur von j-**Algo** , und was der Entwickler für korrekte Erkennung des Moduls zu beachten hat. Der dritte Abschnitt zeigt, wie die Ressourcen des zu implementierenden Moduls organisiert sein sollten. Der vierte Abschnitt erklärt, wie die Dateien der Online-Hilfe für das Modul zu organisieren sind und der fünfte Abschnitt schließlich zeigt die hauptprogrammseitige Schnittstelle zwischen Modul und Hauptprogramm. Im folgenden wird abkürzend für „Das zu implementierende Modul“ nur „Das Modul“ geschrieben.

6.1 Grundimplementierung

Der Code für das Modul wird im Paket `org.jalgo.module.`<Modulkürzel> abgelegt. Das Modulkürzel sollte aussagekräftig, jedoch relativ kurz gehalten sein. Es wird später im Code oft benötigt, wenn es um Ressourcen-Zugriffe geht.
Eine schlechte Wahl wären also zum Beispiel `dijkstrasShortestPathAlgorithm` oder vielleicht `syntaxDiagramsAndEBNF`.

Es gibt zwei Schnittstellen, um das Hauptprogramm mit einem Modul zu vernetzen. Die erste, modulseitige, Schnittstelle besteht aus zwei Klassen. Jedes Modul muss eine Verbindungseinheit und eine Informationseinheit anbieten. Die Verbindungseinheit muss abgeleitet sein von `org.jalgo.main.AbstractModuleConnector`. Hier sind Methoden zu implementieren, die die Interaktion des Moduls mit dem Hauptprogramm spezifizieren. Für Details dazu sollte die API-Dokumentation von j-**Algo** konsultiert werden. Nachfolgend ist die Schnittstelle von `AbstractModuleConnector` abgebildet.

Achtung! Die Verbindungseinheit muss sich an eine Namenskonvention halten: Paket und Name der Klasse muss `org.jalgo.module.`<Modulkürzel>`.ModuleConnector` lauten. Dies ist ein notwendiger Teil der Pluginstruktur von j-**Algo** . Für Details dazu lesen Sie bitte den nächsten Abschnitt.

```
public abstract class AbstractModuleConnector {
    public abstract void init();
    public abstract void run();
    public abstract void setDataFromFile(ByteArrayInputStream data);
    public abstract ByteArrayOutputStream getDataForFile();
    public abstract void print();

    public boolean close();

    public final IModuleInfo getModuleInfo();
    public enum SaveStatus {
        NOTHING_TO_SAVE,
        NO_CHANGES,
        CHANGES_TO_SAVE
    }
    public final SaveStatus getSaveStatus();
    public final void setSaveStatus(SaveStatus status);
    public final void setSavingBlocked(boolean blocked);
    public final boolean isSavingBlocked();
    public final String getOpenFileName();
    public final void setOpenFileName(String filename);
}
```

Die Informationseinheit ist dazu da, Informationen über das Modul bereitzustellen, die dem Benutzer entsprechend aufbereitet dargeboten werden, wenn er ein Modul zur Benutzung auswählen will. Sie muss das Interface `org.jalgo.main.IModuleInfo` implementieren. Für Details dazu sei hier wieder auf die API-Dokumentation von **j-Algo** verwiesen. Nachfolgend ist die Schnittstelle von `IModuleInfo` abgebildet.

```
public interface IModuleInfo {
    public String getName();
    public String getVersion();
    public String getAuthor();
    public String getDescription();
    public URL getLogoURL();
    public String getLicense();
    public URL getHelpSetURL();
}
```

Achtung! Die Informationseinheit muss sich an eine Namenskonvention halten: Name und Paket der Klasse muss `org.jalgo.module.`<Modulkürzel>`.ModuleInfo` lauten. Dies ist notwendiger Teil der Pluginstruktur von **j-Algo** .
Weiterhin hat die Informationseinheit das **Singleton**-Entwurfsmuster zu implementieren mit der Zugriffsmethode `public static IModuleInfo getInstance();`
Als Klassenmethode kann diese nicht in das Interface `IModuleInfo` aufgenommen werden. Es sei jedoch ausdrücklich darauf hingewiesen, dass das Fehlen dieser Methode dazu führt, dass das Modul nicht korrekt erkannt wird und dass Laufzeitfehler beim Starten des Programmes auftreten.

Als Beispielcode ist ein minimalistisches Modul implementiert namens `testModule`. Es ist ein korrekt implementiertes Modul, jedoch hat es keinerlei Funktionalität. Der Entwickler kann den Code bei Bedarf als Skelett zum Aufsetzen eines neuen Moduls nehmen. Zum aktuellen Zeitpunkt existieren für **j-Algo** 2 Module: **AVL-Bäume** und **Dijkstra**. Es sei dem Entwickler freigestellt, diese als Anleihe zu nehmen.

Mit diesen beiden Klassen ist die modulseitige Schnittstelle fertiggestellt. Damit das Modul als solches auch vom Hauptprogramm erkannt wird, ist noch ein Schritt notwendig.

6.2 Pluginstruktur von j-Algo

An dieser Stelle scheint es angebracht, kurz die Pluginstruktur von **j-Algo** zu erläutern. In der Distribution wird das Hauptprogramm in ein JAR-Archiv verpackt. Es muss unabhängig von den Modulen sein. Daher wird auch jedes Modul in ein eigenes JAR-Archiv verpackt mit allem, was zu diesem Modul gehört: Code und Ressourcendateien. Dies garantiert, dass bei Erscheinen eines neuen Modules nur das entsprechende JAR-Archiv vom Benutzer heruntergeladen werden muss.

Achtung! Die JAR-Archive für die Module müssen als Namen das Modulkürzel tragen und im Ordner `runtime/modules` liegen. Nur so kann das Modul vom Hauptprogramm erkannt werden.

Während der Laufzeit wird zum Start des Hauptprogramms der Ordner `runtime/modules` nach JAR-Archiven durchsucht. Dabei wird der Name des Archives als Paketname angenommen, und es wird in jedem Archiv nach den beiden Verbindungsklassen (siehe oben) gesucht:
`org.jalgo.module.`<ARCHIVNAME>`.ModuleConnector` und
`org.jalgo.module.`<ARCHIVNAME>`.ModuleInfo`
Sind diese korrekt implementiert, wird das Modul in die Liste der installierten Module aufgenommen und kann vom Benutzer ausgewählt werden.

Der Entwickler hat also nun noch eine leere Datei mit dem Namen <MODULKÜRZEL>`.jar` im Ordner `runtime/modules` zu erstellen. Wird **j-Algo** aus der Entwicklungsumgebung gestartet, kann das Modul nun aufgerufen werden.

6.3 Organisation der Ressourcen

Sicher soll das Modul irgendwelche Ressourcendateien halten, wie zum Beispiel Icons oder ausgelagerte Algorithmentexte.

Da, wie erwähnt, zur Laufzeit das Modul in einem JAR-Archiv vorliegt, kann auf die Ressourcen nur über den Klassenlader zugegriffen werden, indem die Methoden
`getClass().getResource(String)` (liefert eine URL) oder
`getClass().getResourceAsStream(String)` (liefert einen InputStream)
verwendet werden. Es erweist sich als vorteilhaft, wenn die Pfade zu den Ressourcendateien nicht direkt im Code verankert werden, sondern in einer externen Textdatei abgelegt werden. Um einen einfachen Ressourcenzugriff zu ermöglichen, bietet das Hauptprogramm mit der Klasse `org.jalgo.main.util.Messages` die Methode
`getResourceURL(String bundleKey, String key)`
an, welche direkt die URL einer Ressource zurückgibt. Der erste Parameter ist der Schlüssel, mit welchem das Ressourcenpaket ausgewählt wird, aus dem der Pfad zu entnehmen ist. Dies ist wieder das Modulkürzel, also der Hauptpaketname, wenn moduleigene Ressourcen geladen werden sollen, und `main`, wenn Ressourcen des Hauptprogramms, z.B. Standard-Icons verwendet werden sollen. Der zweite Parameter ist der Schlüssel der Ressource. Dafür ist im Hauptpaket des Moduls eine Textdatei zu erstellen, in welcher Ressourcenpfade zu Schlüsseln zugeordnet werden. Es sei als Beispiel auf die existierenden Dateien von Hauptprogramm und den bestehenden Modulen verwiesen.

Achtung! Die Textdatei mit den Ressourcenpfaden muss einer Namenskonvention folgen: Sie hat den Titel `res.properties` zu tragen und muss im Hauptpaket des Moduls liegen. Anderenfalls wird sie von der Klasse `Messages` nicht gefunden.

Die Ressourcendateien selbst werden im Ordner `res/module/`MODULKÜRZEL abgelegt. Wird unter Eclipse programmiert, ist dieser Ordner unter „Projekteinstellungen“->„Libraries“->„Add Class Folder...“ hinzuzufügen, damit die Ressourcen in der Entwicklungsumgebung freigegeben sind.
Um Namenskonflikten unter den Ressourcendateien vorzubeugen (letzlich liegen alle Ressourcenpfade hinter `res/main/` und `res/module/`<MODULKÜRZEL>`/` auf dem Klassenpfad), empfiehlt es sich, im angelegten Ressourcenordner eine angemessene Struktur zu entwickeln, so zum Beispiel einen Unterordner <MODULKÜRZEL>`_pix` für Bilddateien. Jetzt kann auch ein Dateiname wie `icon.gif` ohne Probleme verwendet werden.

j-Algo ist ein Programm, welches mehrere Sprachen unterstützt. Zum aktuellen Zeitpunkt sind sämtliche Programmteile in Deutsch und Englisch verfügbar. Dem Entwickler wird nahegelegt, auch das neue Modul in diesen Sprachen zu veröffentlichen. Dazu ist es notwendig, alle Zeichenketten, die dem Benutzer dargeboten werden sollen, in externen Textdateien zu speichern. Auch für den einfachen Zugriff auf diese Zeichenketten bietet die Klasse `org.jalgo.main.util.Messages` eine Methode an:
`getString(String bundleKey, String messageKey)`
Die Verwendung dieser Methode erfolgt analog zu der oben erwähnten Methode für die Ressourcen.

Achtung! Die Textdatei mit den ausgelagerten Zeichenketten muss einer Namenskonvention folgen: Auch sie hat die Endung `.properties` zu tragen. Der Name der Datei ist einfach das Kürzel der Sprache, für welche sie Zeichenketten enthält. Für deutsch also `de.properties`, für englisch `en.properties`. Auch diese Textdateien haben im Hauptpaket des Moduls zu liegen.

Liegen die Textdateien korrekt vor, so wird vom Hauptprogramm automatisch auf die eingestellte Sprache umgestellt. Der Modulentwickler muss hierzu nichts mehr beachten.
Auch hier wieder sei als Beispiel auf die existierenden Dateien von Hauptprogramm und Modulen verwiesen.

Letzlich sei noch erwähnt, dass jedes Modul die Möglichkeit hat, persistente Benutzereinstellungen anzubieten. Will der Modulentwickler solche Einstellungen einbauen, so muss eine Textdatei mit Zuordnungen zwischen Schlüsseln und Werten angelegt werden. Es sind hier alle Einstellmöglichkeiten als Schlüssel zu erwähnen; die zugeordneten Werte sind jeweils die Default-Werte für die entsprechenden Einstellungen.

Achtung! Die Textdatei mit den Default-Einstellungen muss einer Namenskonvention folgen: Sie muss den Namen <MODULKÜRZEL>`.prefs` tragen und im Ressourcenverzeichnis des Moduls (`res/`<MODULKÜRZEL>`/`) liegen. Außerdem muss ein Schlüssel namens „Version“ in der Datei stehen, dessen Wert die Version der Einstellungsdatei angibt. Wird in einer späteren Version etwas am Bestand der Schlüssel geändert, gewährleistet der geänderte Wert von „Version“, dass beim Benutzer die Einstellungsdatei neu erstellt wird und an die neuen Einstellungen anpasst wird.

Es wird als Beispiel auf die Dateien `main.prefs` des Hauptprogramms und `avl.prefs` des Moduls AVL-Bäume verwiesen.
Zugegriffen wird auf die persistenten Einstellungen über die Klasse
`org.jalgo.main.util.Settings`
Für Details ist die API-Dokumentation von **j-Algo** zu konsultieren. Nachfolgend ist der relevante Teil von `Settings` abgebildet:

```
public class Settings {

    public static boolean getBoolean(String resourceKey, String settingKey);
    public static void setBoolean(String resourceKey, String key,
        boolean value);
    public static String getString(String resourceKey, String key);
    public static void setString(String resourceKey, String key,
        String value);
}
```

Wie schon beim Zugriff auf Ressourcenpfade und ausgelagerte Zeichenketten gibt der erste Parameter jeder Methode hier an, in welcher Einstellungsdatei die Einstellmöglichkeit gesucht werden soll. Dies ist für modulspezifische Einstellungen wieder das Modulkürzel. Es ist aber auch möglich, an die Einstellungen des Hauptprogramms zu gelangen mittels des `resourceKeys` `main`.

6.4 Organisation der Hilfe-Dateien

Dem Entwickler angeraten, ebenfalls eine Online-Hilfe zu seinem Modul zu erstellen. Die Online-Hilfe von j-**Algo** nutzt die Technologie von **JavaHelp**. Es folgt eine kurze Einführung in dieses System.

...

TODO: Matthias, du bist dran, denk auch daran, dass Du mir noch die Änderungen für das Gewährleisten der Kontextsensitivität schilderst.

6.5 Schnittstelle zum Hauptprogramm

Die zweite erwähnte Schnittstelle ist die auf Seiten des Hauptprogramms, namentlich die Klasse `org.jalgo.main.gui.JAlgoGUIConnector`. Der Entwickler hat hier nichts zu implementieren, jedoch hat er Kenntnis von dieser Schnittstelle zu haben. Hierüber laufen alle Anfragen, die das Modul an die graphische Oberfläche des Hauptprogramms richtet. Für Details dazu sei auf die API-Dokumentation von j-**Algo** verwiesen. Nachfolgend ist die Schnittstelle von `JAlgoGUIConnector` abgebildet. Gerade die letzten 3 Methoden sind interessant, um an die moduleigenen GUI-Komponenten zu gelangen.

```
public class JAlgoGUIConnector {

    public static JAlgoGUIConnector getInstance();

    public void saveStatusChanged(AbstractModuleConnector moduleInstance);
    public void showErrorMessage(String msg);
    public void showWarningMessage(String msg);
    public void showInfoMessage(String msg);
    public int showConfirmDialog(String question, int optionType);
    public void setStatusMessage(String msg);
    public String showOpenDialog(boolean openAsJAlgoFile,
                                 boolean useCurrentModuleInstance);
    public AbstractModuleConnector newModuleInstanceByName(String moduleName);

    public JComponent getModuleComponent(AbstractModuleConnector module);
    public JMenu getModuleMenu(AbstractModuleConnector module);
    public JToolBar getModuleToolbar(AbstractModuleConnector module);
}
```

Diese Klasse implementiert das **Singleton**-Entwurfsmuster. Somit kommt man über die Zugriffsmethode `getInstance()` an die Instanz.

7 Bekannte Fehler und Schwachstellen

Derzeit wird vom Hauptprogramm kein Drucken unterstützt. Daher ist die Methode `AbstractModuleConnector.print()` derzeit nutzlos. Sie wird trotzdem aus Kompatibilitätsgründen mitgeführt, um für zukünftige Implementationen gerüstet zu sein.

8 Weiterführende Links

Die Software nutzt für die graphische Oberfläche die **Swing**-Technologie. Informationen hierzu findet man unter
http://java.sun.com/docs/books/tutorial/,
http://java.sun.com/products/jfc/
oder in der API-Dokumentation von Java.