Projekt DGPF

Erstellung einer grafischen Benutzeroberfläche

Fachgebiet Verteilte Systeme
Sommersemester 2006

Mirko Dietrich, mirko.dietrich@hrz.uni-kassel.de
Lado Kumsiashvili, lado@student.uni-kassel.de
Alexander Podlich, podlich@student.uni-kassel.de

25. September 2006

Inhaltsverzeichnis

1 Einführung **4**
1.1 Problemstellung 4
1.1.1 Grafische Benutzeroberfläche 5
1.1.2 Hintergrunddienst / Daemon 5
1.1.3 Netzwerkschnittstelle 6

2 Realisierung der Aufgaben **8**
2.1 Projektverwaltung 8
2.1.1 Ant 8
2.1.2 Compiler 9
2.1.3 Eclipse 10
2.2 Grafische Benutzeroberfläche 10
2.2.1 Verwendete Bibliothek 10
2.2.2 Aufbau der grafischen Benutzeroberfläche 11
2.2.3 Dialoge 12
2.2.4 Mnemonic 17
2.2.5 Hilfsprogramme 17
2.2.6 Sprachunterstützung 19
2.2.7 Darstellung der statistischen Daten 20
2.2.8 Client-Liste und Informations-Block 27
2.2.9 Grafische Darstellung der statistischen Daten 29
2.2.10 Automatische Protokollierung 32
2.3 Hintergrunddienst / Daemon 34
2.3.1 Starten von Suchaufgaben 34
2.3.2 Statusinformationen eines Suchvorgangs 35
2.3.3 Adaption an Suchparameter zur Laufzeit 35
2.4 Netzwerkkommunikation 35
2.4.1 Protokoll 36
2.4.2 Zirkulärer Zwischenspeicher 37
2.4.3 Events 37

3 Verschiedenes **39**
3.1 Bekannte Probleme 39
3.2 Verwendete externe Komponenten 39

3.2.1 JFreeChart . . . 39
3.2.2 JDT . . . 40
3.2.3 Tango Icons . . . 40
3.3 Metriken . . . 40
3.3.1 Arbeitszeit . . . 40
3.3.2 Lines of Code . . . 40
3.3.3 Packages, Klassen und Interfaces . . . 41
3.4 Coding Styles . . . 44
3.5 Ausblick . . . 45

4 Projektzusammenfassung **46**

1 Einführung

Dieses Dokument beschreibt die studentische Projektarbeit, die im Sommersemester 2006 im Fachgebiet *Verteilte Systeme* des Fachbereichs Elektrotechnik / Informatik der Universität Kassel durchgeführt wurde. Sie bildet ein Teilprojekt des *Distributed Genetic Programming Framework*[1] [6] [1] [2].
Zuerst sollen im folgenden Abschnitt 1.1 die bearbeitete Aufgabenstellung dargelegt werden. Die einzelnen Softwarekomponenten, die während des Projekts entstanden, werden anschließend im Abschnitt 2.2, 2.3 und 2.4 näher erläutert. Im Abschnitt 2.1 wird die Projektverwaltung erklärt. Im Kapitel 3 gibt es Informationen zu bestehenden Problemen, den verwendeten Softwarekomponenten, Softwaremetriken und Coding Style. Außerdem gibt es einen Ausblick auf zukünftige Erweiterungen des Projekts.

1.1 Problemstellung

Das DGPF Projekt implementiert ein Rahmenwerk für verteilte Suchalgorithmen. Die Algorithmen sowie die zu findenen Indivuen sind, wie auch die DGPF Bibliotheken selber in Java implementiert. Wollte man in der Vergangenheit eine Suchaufgabe mit dem DGPF lösen, so musste man es von Hand über die Kommandozeile starten. Auch die Ergebnisse wurden nur über eine Textausgabe dargestellt. Eine grafische Darstellung, beispielsweise mittels einer Tabellenkalkulation, konnte nur im Nachhinein und auf manuelle Weise geschehen. Wünschenswert war ein bequemes Starten und Beenden der verteilten Suchalgorithmen, die Möglichkeit, während der Laufzeit die Parameter der Suche zu ändern und ihren Fortschritt verfolgen zu können.
Es sollte ein erweiterbares Managementsystem mit Benutzeroberfläche für das DGPF-System entworfen werden. Als Programmiersprache wurde, wie auch beim Rest des DGPF Projekts, Java gewählt. Das System besteht aus drei Teilen:

1. Grafische Benutzeroberfläche (GUI)
2. Hintergrunddienst (Daemon)
3. Netzwerkschnittstelle zwischen GUI und Daemon

Die verschiedenen Ansätze des DGPF-Projekts zur verteilten Suche, wie Client-Server, Peer-To-Peer und Hybrid, werden in diesem Dokument nicht behandelt. Durch die Benutzeroberfläche lässt sich eine verteilte Suche steuern. Sie dient als Schaltzentrale für die einzelnen Rechenknoten. Auf diesen läuft jeweils ein DGPF-Dienst.

[1] `http://dgpf.sourceforge.net/`

1.1.1 Grafische Benutzeroberfläche

Die grafische Benutzeroberfläche des DGPF-Systems hat zwei Hauptaufgaben:

1. Die von den Suchalgorithmen erzeugten statistischen Daten grafisch darzustellen, sowie auszuwerten und aufzuzeichnen.

2. Dem Benutzer ermöglichen, die Suchalgorithmen durch die GUI zu steuern.

Die arbeitenden Maschinen stellen ein verteiltes Rechencluster dar. Auf diesem wird die Suche nebenläufig durchgeführt. Diese Knoten senden über den Hintergrunddienst jeweils Zustandsinformationen (z.B. Statistiken) zum Managementsystem. Der Suchalgorithmus an sich besitzt zur Kommunikation lediglich ein System um Ereignisse zu erzeugen und eine Schnittstelle, mit der dynamisch Einstellungen vorgenommen werden können. Die Datenkommunikation mit dem Knoten — auf dem das Managementsystem läuft — ist Teil der Aufgabe. Besonders wichtig ist, dass die Datenkommunikation zwischen Managementsystem und Daemon asynchron abläuft, also dass die Suche nicht durch zu langsame Kommunikation mit dem Managementsystem gebremst wird.
Ebenso soll die Steuerung sowohl des Gesamtsystems als auch der einzelnen Knoten möglich sein. Zur Datendarstellung sollen konfigurierbare Diagramme genutzt werden.
Der Benutzer soll einen guten Eindruck vom Fortschritt der Evolution erhalten und nachvollziehen können, wie sich seine Einstellungen auf das System auswirken — dafür ist eine verständliche und klar strukturierte Aufbereitung der Informationen notwendig.
Die GUI soll es dem Benutzer jeweils ermöglichen die Informationen des Gesamtsystems sowie der einzelnen Rechenknoten auszuwerten.
Das DGPF-Framework gliedert sich in mehrere Ebenen. Die Genetic-Engine an sich ermöglicht den Einsatz beliebiger Genetischer Algorithmen für beliebige Problemdomänen. Darüber befindet sich die Ebene der Genetischen Programmierung — diese erlaubt es, Algorithmen in einer Assembler-ähnlichen Notation zu erzeugen. Der Benutzer soll Steuerinformationen per Hand mit der GUI setzen und an das System übertragen können. Es soll möglich sein, das Managementsystem zu einem späteren Zeitpunkt so zu erweitern, dass Fitnessfunktionen ausgewählt und eine genetische Evolution gestartet werden kann. Wenn dies auch nicht Teil der Aufgabe ist, so soll zumindest diese zukünftige Entwicklung bereits im Entwurf und der Spezifikation bedacht werden.

1.1.2 Hintergrunddienst / Daemon

Ein Hintergunddienst (Daemon) läuft auf jedem der Rechner, die Teil des Rechenclusters sind. Jeder Daemon-Prozess beherbergt jeweils einen Suchalgorithmus. Er ist dafür verantwortlich alle nötigen Schritte vorzunehmen, damit ein Suchalgorithmus gestartet und beendet werden kann. Weiterhin muss er die Kommunikation zwischen der Benutzeroberfläche und dem Suchalgorithmus netzwerktransparent koordinieren.
Er nimmt Verbindungen an, die von einer GUI kommen. Dies soll komplett über ein TCP/IP-fähiges Netzwerk geschehen. Der Benutzer kann mit Hilfe der GUI eine Verbindung zu einem Daemon aufbauen. Dabei soll die Verbindung zwischen Daemon und

GUI beliebig oft auf- und abgebaut werden können, ohne dass der Suchalgorithmus davon beeinflusst wird.
Außerdem werden die Statistiken, die eine Suchaufgabe produziert, an die GUI weitergeleitet. Diese wurden bisher einfach als Text auf dem Bildschirm ausgegeben. Hier soll eine einheitlich Schnittstelle geschaffen werden.
Falls sich Suchparameter ändern, muss der Hintergrunddienst dafür sorgen, dass diese beim Suchprozess ankommen, so dass dieser sie adaptieren kann. Der Benutzer kann also Einstellungen vornehmen, die dann von den Suchalgorithmen befolgt werden müssen.
Über die Oberfläche kann der Benutzer Suchprozesse starten und beenden. Für das Starten muss ein Weg gefunden werden, die Jobinformationen, also den Suchalgorithmus selber über das Netzwerk zum Daemon zu transportieren, damit dieser die Suche initiieren kann. Hier ist es nicht möglich die verschiedenen Algorithmen direkt beim Daemon zu beherbergen, da die Algorithmen austauschbar sein können und der Daemon auch einen neuen Algorithmus beherbergen können muss, der zur Entwicklungszeit noch nicht existierte.

1.1.3 Netzwerkschnittstelle

Die Kommunikation zwischen Instanzen der grafischen Oberfläche und des Hintergrunddienstes laufen, wie bereits erwähnt, über ein TCP/IP-basiertes Netzwerk. Darüber hinaus gibt es folgende Anforderung für die Kommunikationsschicht des Managementsystems:

- Asynchronität der Kommunikation:
  Die Kommunikation darf nicht blockieren. Das bedeuted insbesondere, dass das Fortschreiten der Suchalgorithmen nicht von dem Austausch von Nachrichten mit der Benutzerschnittstelle beeinflusst werden darf. Dazu müssen evtl. entsprechende Puffer oder anders geartete Mechanismen ausgewählt und implementiert werden, die ein entsprechendes Verhalten garantieren.

- Flexibilität:
  Es können zu jedem Zeitpunkt eine unbeschränkte Anzahl Instanzen von DGPF-Hintergrunddiensten, sowie Benutzeroberflächen existieren. Die verteilte Suche findet typischerweise auf einer Menge von Rechenknoten statt. Diese werden von der Benutzeroberfläche gesteuert, potenziell über eine Fernverbindung, z.B. über das Internet. Unabhängig davon muss es jedoch möglich sein, falls ein Daemon schon eine Verbindung mit einer Benutzerschnittstelle unterhält, eine weitere Verbindung von einer anderen Stelle aufzubauen. Ein Beispiel soll die Situation klarer darstellen: Auf einem Cluster arbeiten eine Reihe Rechner an einem Suchproblem. Von einer zentralen Stelle aus wird der Fortschritt überwacht und gesteuert. Die Verbindungen zu den Rechnern ist immer offen, die Benutzeroberfläche also Tag und Nacht verbunden. Trotzdem soll es möglich sein, z.B. wenn ein Mitarbeiter von zuhause aus den Stand der Dinge einsehen möchte, weitere Verbindungen zu den DGPF-Diensten aufzubauen.

- Modularität:
  Obwohl für diese Implementierung eine Verbindung über TCP gewählt wurde, so darf diese Basis der Kommunikation nicht die einzig mögliche sein. Denkbar sind weitere Kommunikationsarten über beispielsweise performantere Arten der Interprozesskommunikation oder aber auch über weitere Abstraktionsebenen, wie z.B. Web Services. Die Form der Implementierung soll kommende Kommunikationsarten nicht negativ beeinflussen oder übermäßigen Implementationsaufwand an der GUI bzw. dem Hintergrunddienst nach sich ziehen. Dafür ist ein möglichst modularer Aufbau der Kommunikationsschicht zu wählen.

2 Realisierung der Aufgaben

2.1 Projektverwaltung

Über die Aufgabenstellungen heraus haben sich weitere Aufgaben ergeben, die zur Bewältigung des Projekts notwendig waren. In den folgenden Seiten sollen die wichtigsten Unteraufgaben vorgestellt werden.

2.1.1 Ant

Algemeines

Ant ist ein in Java geschriebenes Open Source Werkzeug zur automatisierten Projektverwaltung. Es ist als plattformunabhängigen Ersatz für das unter Unix stark verbreitete System *make* gedacht. Die Funktionen umfassen:

- Übersetzen von Quelltexten
- Einbinden von Bibliotheken
- Dateisystemoperationen
- uvm...

Weitere Informationen sind auf der Homepage des Projekts[5] zu finden.

Ant und das DGPF Projekt

Die Datei `build.xml` für das DGPF Projekt liegt wie üblich im Wurzelverzeichnis und besteht aus folgenden Variablen und Targets:

- Variablen
  - `project.title` Projekt Titel
  - `src.dir` Das Quelltext-Verzeichnis
  - `build.dir` Das Verzeichnis mit übersetzten Class-Dateien
  - `libs.dir` Das Verzeichnis für die verschiedenen Bibliotheken
  - `build.jar` Die zu erzeugende JAR Datei
  - `javadocs.dir` Das Verzeichnis für Javadoc Dateien

  - `jdt.jar` Der Eclipse-Java-Compiler
  - `jfreechart.jar` Die Bibliothek für die Erstellung von Diagrammen
  - `jcommon.jar` Wird von `jfreechart.jar` benötigt

- Targets
  - `prepare` - Bereitet das Dateisystem vor, legt die in `${src.dir}` und `${build.dir}` definierten Verzeichnisse an, falls diese nicht vorhanden sind.
  - `clean` - löscht das in `${src.dir}` Verezeichnis und die JAR-Datei `${build.jar}`.
  - `compile` Kompiliert alle Quelltexte in `${src.dir}` und kopiert diese nach `${build.dir}` (hängt vom `prepare` Target ab).
  - `jar` Dieses Target ist das **Default-Target** und baut aus den Javaklassen in `${build.dir}` die JAR-Datei und kopiert diese nach `${libs.dir}`. Dabei wird als Hauptklasse `org.dgpf.gui.DGPFClient` festgelegt (hängt vom `compile` Target ab).
  - `javadocs` Generiert Javadocs und kopiert die Dateien nach `${javadocs.dir}` (hängt vom `compile` Target ab).
  - `etags` Generiert eine TAGS-Datei[1] aus den Quelltexten.
  - `cleantags` Löscht die TAGS-Datei.
  - `run` Startet die Hauptklasse aus `${build.jar}` im Hintergrund.

Es können natürlich, je nach Bedarf, andere Targets definiert werden.

2.1.2 Compiler

Generische[2] Programmierung ist relativ neu in der Java-Welt, daher gehen die verschiedenen Java Compiler bisher noch unterschiedlich mit diesem Feature um. Diese Technik wird jedoch sehr intensiv im gesamten DGPF eingesetzt. Das DGPF-Projekt wurde bisher nur über den Eclipse-internen Compiler übersetzt. Erst jetzt hat sich herausgestellt, dass einige Sprachkonstrukte, die mit *Generics* verbunden sind vom Sun Java Compiler als ungültig eingestuft werden. Lässt man aber den Quelltext mit dem Eclipse-eigenen Compiler JDT übersetzen, so treten keine Fehler auf. Aber es verwenden nicht alle Softwareentwickler Eclipse, so dass Kompatibilitätsprobleme bestehen. Bis beide Compiler identische Ergebnisse liefern, muss mit einem Workaround gearbeitet werden. Dafür haben wir JDT (siehe Abschnitt 3.2.2) in das Projekt integriert, so dass der Compiler im CVS Repository gleich mitgeliefert wird. Er ist auch als Ant Task integriert. Dies ermöglicht nun allen Entwicklern, auch die, die nicht Eclipse verwenden, das Projekt zu kompilieren.

[1] Das Tool `etags` indiziert Feldvariablen, Methoden, usw., die in verschiedenen Entwicklungswerkzeugen und Editoren für die Schnellsuche verwendet werden.

[2] Nicht zu verwechseln mit Genetischer Programmierung!

2.1.3 Eclipse

Für die Entwicklung der DGPF Benutzeroberfläche wurde das Eclipse IDE[3] eingesetzt. Es erleichtert und beschleunigt den Entwicklungsprozess und hilft dabei, sauberen Quelltext zu erstellen, denn es hat viele, sehr bequeme Funktionen. Das DPGF Projekt liegt im CVS Repository als Eclipse CVS Projekt, so dass viele Entwickler, die das Eclipse IDE benutzen, ohne großen Aufwand beim DPGF Projekt mitmachen können.

2.2 Grafische Benutzeroberfläche

2.2.1 Verwendete Bibliothek

Es gibt verschiedene Bibliotheken zur plattformunabhängigen Oberflächenprogrammierung für Java. Die bekanntesten sind AWT, Swing und SWT.

- *AWT* ist Bestandteil von *JFC*[4] und bietet eine API zur Erzeugung und Darstellung von Benutzeroberflächen. Es benutzt die nativen Methoden des jeweiligen Betriebssystems.
- *Swing* baut auf AWT auf und bietet modernes *Look&Feel* und ist im Gegensatz zu AWT aber leichtgewichtiger. Es benutzt nicht die nativen Methoden des Betriebssystems, um die GUI-Elemente zu zeichnen, im Gegensatz zu AWT und SWT.
- *SWT* wurde etwas später von IBM entwickelt und nutzt, ähnlich AWT, die nativen Komponenten des Betriebssystems. Es wurde ursprünglich für Eclipse entwickelt, wird aber mittlerweile auch in anderen Projekten eingesetzt.

Die Wahl in DGPF Projekt ist aus verschiedene Gründen auf Swing gefallen:

- Swing besitzt mehr Features gegenüber AWT, u.a.:
  - Neue Widgets
  - Veränderbares Look & Feel
  - Mnemonics[5]
  - Tooltips
  - Drag'n Drop
- Es ist völlig plattformunabhängig, da es die GUI-Komponenten selbst zeichnet, anstatt den Auftrag an das Betriebssystem bzw. an ein anderes GUI-Toolkit weiterzuleiten.

[3] `http://www.eclipse.org`
[4] Java Foundation Classes
[5] Tastaturkürzel

- Ein weiterer Vorteil ist, dass Swing im Lieferumfang vom Java Runtime Environment enthalten ist. Man braucht also keine weiteren Bibliotheken, um die Benutzeroberfläche zum Laufen zu bringen.
- Außerdem sind mit Hilfe der Swing Bibliothek erstellte Benutzeroberflächen leichter wartbar, da es vielen Entwicklern vertrauter ist.
- Swing ist sehr einfach zu benutzen, da es sehr leichtgewichtig ist.

2.2.2 Aufbau der grafischen Benutzeroberfläche

Die grafische Benutzeroberfläche (siehe Abbildung 2.1) besteht aus einer Menüleiste, einer Arbeitsfläche, welche aus drei weiteren Komponenten besteht und einer Statusleiste. Die Komponenten der Arbeitsfläche stellen Panels (`JSplitPane`) dar, die sowohl neben- als auch übereinander platziert sind. Diese Komponenten werden dabei durch einen sichtbaren Separator voneinander getrennt. Der Anwender kann den Separator verschieben und so den Platz, der beiden Komponenten zur Verfügung steht, variieren.

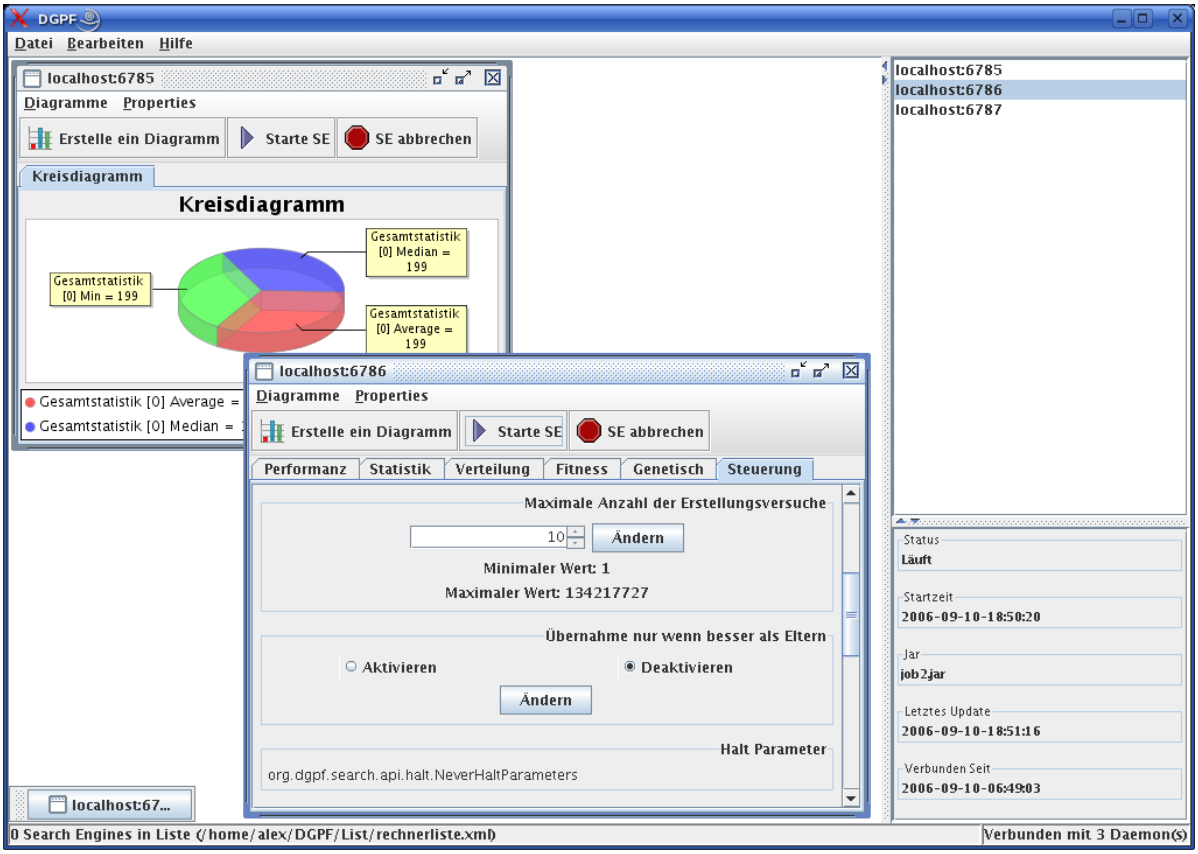

Abbildung 2.1: Die grafische Benutzeroberfläche.

Die Benutzeroberfläche im Detail:

- **Menüleiste**
  In der Menüleiste (`MenuBar`) können verschiedene Funktionen des Programms aufgerufen werden. Sie sind in Hauptmenüpunkte (*Datei*, *Bearbeiten* und *Hilfe*) unterteilt.

- **Interne Desktop**
  Der interne Desktop nimmt die Hauptfläche ein. Hier besitzt jeder verbundene Daemon ein eigenes Kindfenster. Alle Informationen, die zu einem Suchjob gehören, werden in diesem Kindfenster (`InternalFrame`) dargestellt. Nach Wahl kann der Benutzer die Kindfenster minimieren, maximieren oder schließen. Beim Schließen wird die Verbindung zum Daemon getrennt.

- **Fensterliste**
  Die Fensterliste (auf der rechten Seite angeordnet) werden alle zur Zeit verbundenen Daemons mit dem Namen des Rechners, dem jeweiligen Port und der eventuell vorhandenen Beschreibung aufgelistet. Über diese Liste `JList` ist es möglich den Job anzuhalten, sowie die oben beschriebenen Fensterfunktionalitäten auf ein oder alle Unterfenster anzuwenden.

- **Informations-Block**
  Der rechts unten angeordnete Informations-Block enthält einige wichtige Statusinformationen zu dem jeweiligen Suchjob, wie beispielsweise die Statusinformation, ob eine Suche läuft oder nicht, wann das letzte Update kam, etc.

- **Statusleiste**
  Ganz unten ist noch eine Statusleiste vorhanden, die einige globale Informationen anzeigt.

2.2.3 Dialoge

Die Benutzeroberfläche des DGPF Projekts besitzt eine Vielzahl von Dialogen. In diesem Abschnitt sollen einige wichtige vorgestellt werden. Außerdem wird das objektorientierte Grundkonzept der Dialoge im nächsten Abschnitt erläutert.

Der abstrakte Dialog

Alle Dialoge besitzen eine abstrakte Superklasse: Die Klasse `org.dgpf.gui.dialogs.DialogBase`. Sie bietet verschiedene Eigenschaften, die alle Dialoge gemeinsam haben:

- Ein Textfeld für allgemeine Beschreibungen und Hilfestellungen ganz oben im Dialogfenster.

- In der Mitte ist Platz für die spezifische Aufgabe des Dialogfensters. Dort liegt ein `Container`, in dem ein `LayoutManager` eingehängt werden kann.

- Unten existiert eine Reihe mit typischen Buttons. Dabei sind folgende Typen vorgesehen:

  - OK
  - OK und Abbruch
  - Wizard-Buttons (Fertigstellen, Abbrechen, Weiter, Zurück)
- Außerdem kümmert sich diese Klasse bereits um die Positionierung und Größe des Fenster, wobei die abgeleiteten Klassen jedoch noch Einfluss auf die Fenstergröße nehmen können.

Abbildung 2.2 zeigt den Dialog für die Benutzereinstellungen. Hier gibt es einen kurzen Erläuterungstext und unten die Standardbuttons *OK* und *Abbruch*.

Dialog für Benutzereinstellungen

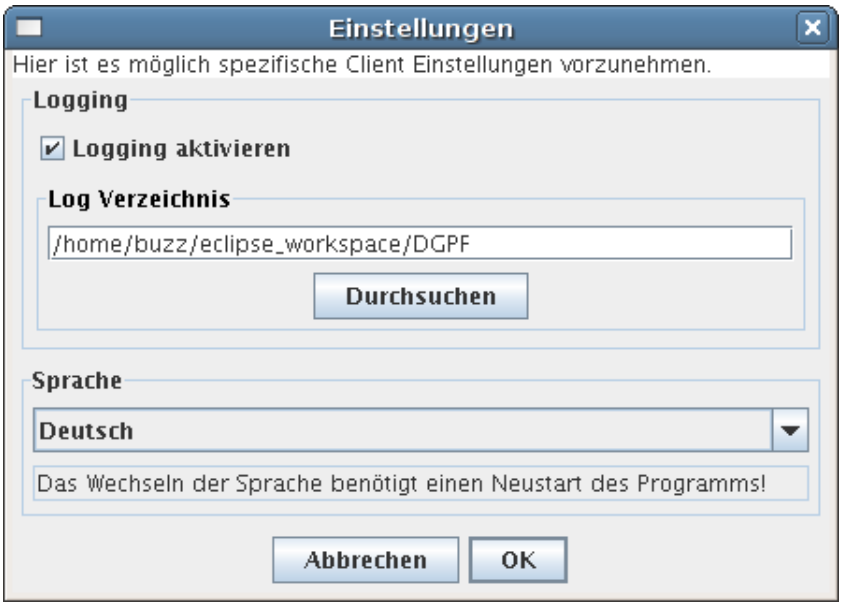

Abbildung 2.2: Dialog für Benutzereinstellungen

Die Benutzereinstellungen können über die Menüleiste erreicht werden: *Bearbeiten / Einstellungen*. Über diesen Dialog kann das Mitloggen von Ereignissen an- und ausgeschaltet und ein Ausgabeverzeichnis gewählt werden (siehe Abbildung 2.2). In dieses Verzeichnis werden später alle Loginformationen abgelegt (siehe auch Abschnitt 2.2.10). Außerdem kann die Sprache der GUI angepasst werden (siehe Abschnitt 2.2.6). Damit die neue Sprache aktiv wird, muss das Programm neu gestartet werden.

Editor für Rechnerlisten

Damit nicht jedesmal der entfernte Rechner manuell eingegeben werdem muss, wurde ein Editor für Rechnerlisten erstellt. Über den Menüpunkt *Bearbeiten / Rechnerlisten* lässt sich der Editor starten. Die Abbildung 2.3 zeigt das Editorfenster, in dem schon

zwei Rechner eingetragen sind. So ist es möglich die neuen Rechner anzugeben oder vorhandene zu entfernen.

Abbildung 2.3: Editieren von Rechnerliste.

Der Editor kann die Rechnerliste persistent auf das Dateisystem schreiben oder eine vorhandene Liste laden. Für die Speicherung der Liste kommt XML zum Einsatz. Zwar gibt es andere Dateiformate, die schneller zu verarbeiten sind als XML, aber die höhere Portabilität, Lesbarkeit und der intuitive Aufbau sprachen für XML. Die Liste wird durch folgende `Dokumenttypdefinition` beschrieben:

```
<!ELEMENT hosts (host*)>
<!ELEMENT host (hostname, port, description)>
<!ELEMENT hostname (#CDATA)>
<!ELEMENT port (#CDATA)>
<!ELEMENT description (#CDATA)>
```

Für die Handhabung der Listen wurde eine Klasse `HostListUtils` entwickelt, die zwei statische Methoden enthält, um Listen zu lesen und zu schreiben:

- `public static Vector<Host> getHosts(File p_file)`
- `public static void writeHosts(Vector<Host> p_hosts, File p_file)`

Um bequem mit der Rechnerliste zu arbeiten wurde noch eine Klasse `org.dgpf.gui.utils.Host` erstellt, deren Instanz jeweils einen entfernten Rechner repräsentiert. Sie enthält Hostname bzw. IP, Port und die Beschreibung zum jeweiligem Rechner.

Diagramm-Wizard

Um ein Diagramm zeichnen zu können wird dem Benutzer ein Diagramm-Wizard bereit gestellt (siehe Abbildung 2.4 und Abbildung 2.5).

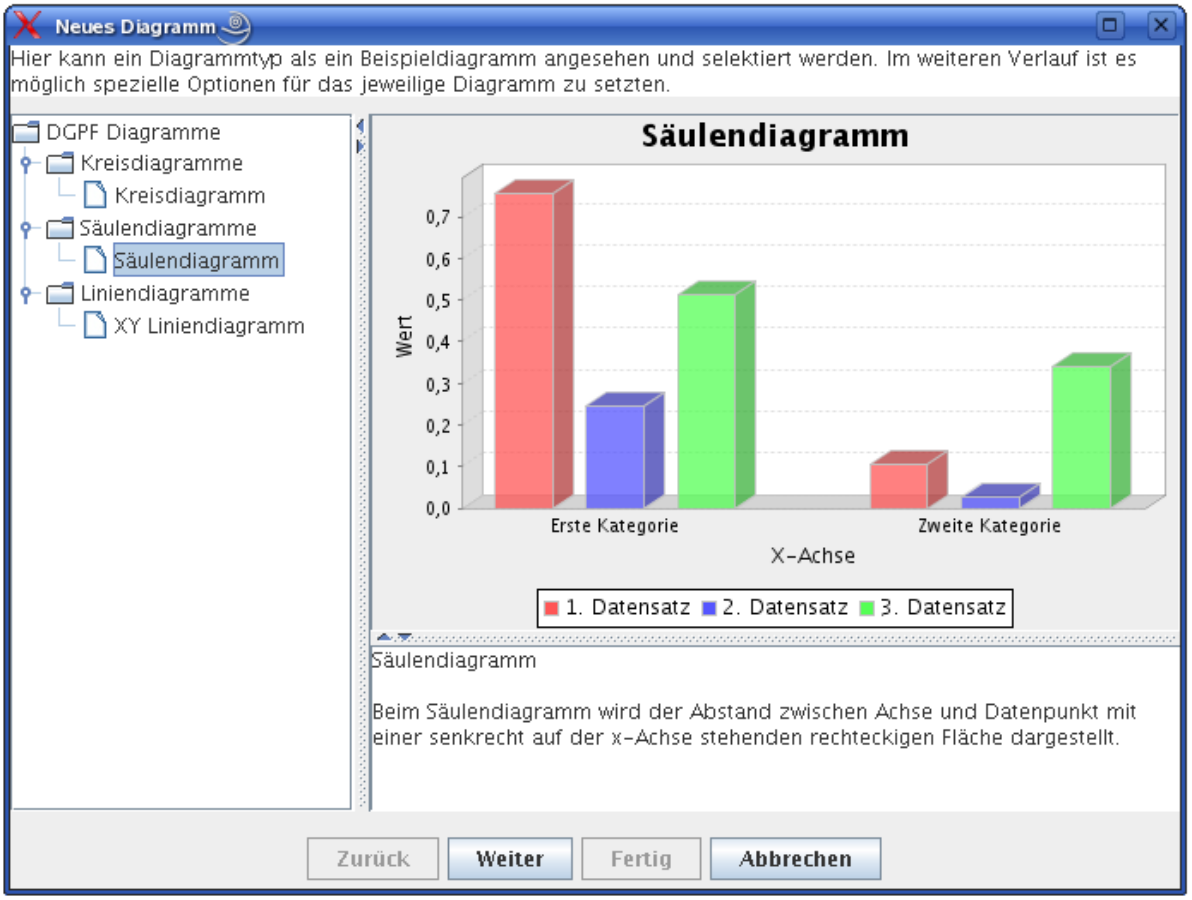

Abbildung 2.4: Auswahl des Diagramm-Typ mit dem Diagramm-Wizard.

Alle Wizards besitzen eine abstrakte Superklasse: Die von der Klasse `DialogBase` abgeleitete Klasse `WizardBase`. Sie bietet alle Eigenschaften, die alle Dialoge gemeinsam haben und stellt zusätzlich die Möglichkeit bereit, Wizard-Buttons (Fertigstellen, Abbrechen, Weiter, Zurück) zu aktivieren oder deaktivieren. So kann beispielsweise im Diagramm-Wizard nicht auf den Weiter-Button geklickt werden, bevor ein Diagramm-Typ ausgewählt wurde. Zusätzlich haben alle Wizards ein vorgegebenen Layout-Manager: `CardLayout`. Wie der Name andeutet, ordnet der `CardLayout`-Manager seine Komponenten wie einen Stapel Karteikarten an, von denen nur die oberste sichtbar ist. Alle Komponenten werden hierbei in der gleichen Größe dargestellt. Zum Blättern definiert der CardLayout-Manager vier Methoden:

1. Die Methode `next()`, mit der die nächste „Karte“ sichtbar gemacht wird.
2. Die Methode `previous()`, mit der die vorhergehende „Karte“ sichtbar gemacht wird.
3. Die Methode `last()`, mit der die letzte „Karte“ sichtbar gemacht wird.
4. Die Methode `first()`, mit der die erste „Karte“ sichtbar gemacht wird.

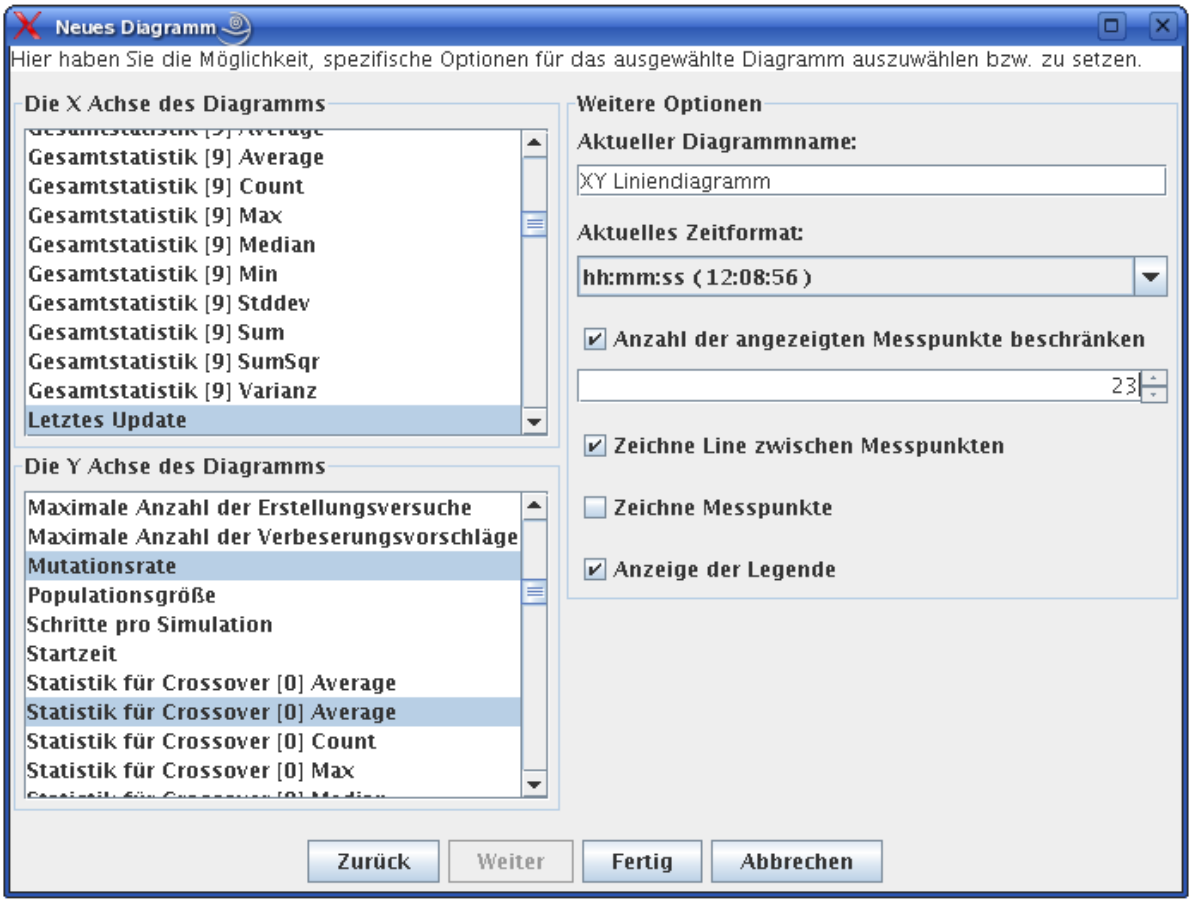

Abbildung 2.5: Diagrammeinstellungen für ein XY-Liniendiagramm.

Wie die Abbildungen 2.4 und 2.5 zeigen, besitzt der Diagramm-Wizard zwei Karten.

- Mittels der ersten Karte (`CCWzrdFirstCard`) wird dem Benutzer die Möglichkeit gegeben einen Diagramm-Typ zu selektieren. Hierzu werden Beispieldiagramme mit den jeweiligen Beschreibungen angezeigt. Alle Diagramme werden im Vektor `Vector<IChart> m_charts` der Klasse `CCWzrd` gespeichert. Bei der Selektion eines Diagramms wird die Beschreibung (`getDescription()`) und das Beispieldiagramm (`getSampleChart()`) aufgerufen. Die aufgerufenen Methoden werden von dem jeweiligen Diagramm implementiert.

- Mittels der zweiten Karte werden dem Benutzer Optionen und Einstellungen angezeigt, die für den jeweiligen Graphen zutreffen. So kann man beispielsweise bei dem XY-Liniendiagramm (siehe Abbildung 2.5 und 2.16) die Anzahl der gezeichneten Messpunkte beschränken und beim Kreisdiagramm (siehe Abbildung 2.14) die Diagramm-Rotation aktivieren. Jeder Diagramm-Typ hat somit ein eigene Karte, welche von der abstrakten Klasse `CCWzrdSecCard` abgeleitet ist.

2.2.4 Mnemonic

Um die Benutzeroberflächen bequem per Tastatur bedienen zu können, wurde für jeden Menüeintrag ein Tastaturkürzel definiert. Diese Mnemonics sind auch in den `Resource-Bundles` eingetragen.

2.2.5 Hilfsprogramme

Browser starten

Unter dem Menüpunkt *Hilfe*, sowie im *Aboutfenster* sind Links zu verschiedenen Webseiten. Aus einem Javaprogramm heraus Webseiten aufzurufen ist nicht trivial, denn auf jedem potentiellen Zielsystem funktioniert dieser Systemaufruf anders. Für diesen Zweck wurde eine Klasse `BrowserLaunch` geschrieben. Sie macht eine Fallunterscheidung für

- Microsoft Windows
- Mac OS X
- und Linux, sowie andere Unix-Varianten.

Bei Windows, sowie Mac OS X wird das Betriebssystem direkt nach dem zuständigen Programm gefragt. Bei den Unix-Varianten werden nacheinander die Browser *Firefox*, *Mozilla*, *Opera*, *Konqueror*, *Epiphany* und schließlich *Netscape* durchprobiert. Eine Erkennung des Standardbrowsers ist bei Unix sehr kompliziert, da es eine Vielzahl verschiedener *Fenster Manager* gibt.

Verwaltung der Benutzereinstellungen

Java bietet die Möglichkeit verschiedene Benutzer-, aber auch systemweite Einstellungen rund um ein Javaprogramm zu realisieren. Dafür existiert die Klasse `java.util.prefs.-Preferences`. Es ist möglich für die Speicherung verschiedene Backends zu wählen, etwa die Speicherung in der Registry bei Microsoft Windows oder im `.java` Verzeichnis bei POSIX-konformen Plattform, aber auch in einer SQL Datenbank. In jedem Fall ist der Einsatz des Backends dem Benutzer verborgen und die Speicherung und Gewinnung von verschiedenen Parametern geschieht über wohldefinierte Schnittstellen.
Desweiteren kann man verschiedene Sätze von Parametern gruppieren, indem man so genannte `Nodes` zu Sätzen zusammenschließt. Die `Nodes` sind dann in einer Baum Struktur gespeichert und man kann je nach Bedarf mit bekannten Baum-Algorithmen über diese `Nodes` traversieren.
Es existieren folgende `Nodes`:

- `UserRootNode` - Die Parametern werden flach und pro Benutzer gespeichert. Mit der statischen Methode `Preferences.getUserNode()` kann auf den Parametersatz in diesem `Node` zugegriffen werden.

- `SystemRootNode` - Die Parametern werden auch hier flach gespeichert. Mit der Methode `Preferences.getSystemNode()` kann man den Parametersatz aus diesem `Node` gewinnen. Im Gegensatz zu `UserNode` sind diese Parameter für jeden Benutzer gleich.

- `PackageNode` - `Preferences.getPackageNode(Class class)` liefert den Parametersatz des Packages zu dem die Klasse `class` gehört.

Per default benutzt die `java.util.prefs.Preferences` Klasse bei Microsoft Windows die Registry und unter POSIX-konformen Systemen das `.java` Verzeichnis im Homeverzeichnis des Benutzers. Diese Methode ist völlig ausreichend für die DGPF Oberfläche. Daher wurden die Standardeinstellungen beibehalten. Für die bequeme Speicherung verschiedener Parameter wurde die Wrapper-Klasse `org.dgpf.gui.utils.PrefUtils` erstellt.
Da es keine globalen oder packagespezifischen Parameter für die Benutzeroberfläche gibt, dafür aber sehr wohl benutzerspezifische, wurde die Speicherung in dem `UserRootNode` gewählt. Es werden z.B. die zuletzt gültige Position und die Grösse der Oberfläche gespeichert, die aktuelle Spracheinstellung, aber auch Pfade von den zuletzt geöffneten Dateien. Je nach Bedarf können weitere Parameter hinzukommen.
Es wurden mehrere statische Methoden erstellt:

- get<Built-in Type>(String key, <Built-in Type> alternativerWert) - Liefert zu dem angegeben `Key` den gespeicherten `Built-in Type` Wert, falls dieser nicht vorhanden ist, wird `alternativerWert` zurückgeliefert.[6]

- put<Built-in Type>(String key, <Built-in Type> wert) - Speichert den übergebenen `Built-in Type` zu dem `key`

- remove(String key) - Löscht den Wert zu dem `key`

Erweitertes Grid-Bag-Layout

Für die Darstellung der grafischen Komponenten wurde das flexible `GridBagLayout` aus folgenden Gründen benutzt:

- Eine grafische Komponente kann sich über einen Anzeigebereich von mehreren Gitterzellen erstrecken.

- Spalten und Zeilen können unterschiedlich breit bzw. hoch sein.

- Eine grafische Komponente kann die ihr zugewiesenen Gitterzellen voll ausfüllen oder aber in ihrer normalen Größe dargestellt werden. Wenn diese Komponente in ihrer normalen Größe dargestellt wird, kann angegeben werden, wie sie innerhalb der zugewiesenen Gitterzellen angeordnet wird.

[6] Unter `Built-in Type` fallen `String`, `float`, `double`, `int` und `boolean`.

- Falls die Größe des Containers vom Benutzer verändert wird, kann angegeben werden, zu welchen Anteilen die Höhen-/Breitenänderung auf die einzelnen Gitterzeilen und -spalten verteilt wird. So kann beispielsweise spezifiziert werden, dass sich Änderungen der Höhe eines Fensters ausschließlich auf die Höhe einer bestimmten Gitterzeile auswirken und alle anderen Gitterzeilen eine feste Höhe haben.

Die Einstellung dieser Eigenschaften erfolgt mit Exemplaren der Klasse `GridBagConstraints`, die im `awt`-Paket definiert ist. Hierzu müssen die Elemente eines `GridBagConstraints`-Objekts in der gewünschten Weise belegt werden. Um die Belegung zu vereinfachen, stellt `GridBagConstraints` einige Konstanten bereit.
Um den Vorgang zu vereinfachen, wurde die Klasse `Layout` erzeugt. Sie bietet eine statische Methode `set_constraints(...)`, mit der die `GridBagConstraints`-Objekte für eine dargestellte Komponente definiert und anschließend einem `Container` hinzugefügt werden können.

2.2.6 Sprachunterstützung

Moderne Programme sollten in Hinblick auf die Sprache anpassbar sein. Die Betriebssysteme bieten hier Möglichkeiten an, die jedoch je nach System auf verschiedene Weisen realisiert werden.
In Java existieren so genannte `ResourceBundles`, die sprachspezifische Informationen enthalten. Dabei gibt es für jede Sprache ein eigenes `ResourceBundle`, in dem sprachabhängigen Zeichenketten gespeichert werden. Ein einzelner `String` lässt sich über einen eindeutigen Schlüssel identifizieren. So existieren für jede Sprache eine einzelne Sprachdatei, die unabhängig vom Programmcode existiert. Soll das Programm nun in weitere Sprachen übersetzt werden, so muss bloß ein entsprechendes `ResourceBundle` geschrieben werden.
Auch die Benutzeroberfläche vom DGPF bietet Unterstützung für verschiedene Sprachen. Deutsch und Englisch sind zur Zeit als Sprachen verfügbar.
Dafür wurde eine Wrapper-Klasse `org.dgpf.gui.utils.LanguageSwitcher` um die Klasse `java.util.ResourceBundle` erstellt, die die Handhabung erleichtert. In dem Package `org.dgpf.resources.rb` befinden sich die `ResourceBundles DGPF_de.properties` und `DGPF_en.properties`.
In den Sprachdateien gibt es immer Schlüssel-Wert-Paare nach dem folgenden Aufbau:

```
....

# Actions
actions.close             = Schließen
actions.close_mnemo       = s
actions.close_all         = Alle schließen
actions.close_all_mnemo   = a
actions.abortse           = SE Abbrechen
actions.abortse_mnemo     = e
```

```
actions.unminimize         = Wiederherstellen
actions.unminimize_mnemo   = w
actions.minimize           = Minimieren
actions.minimize_mnemo     = m

....
```

Die Sprachunterstützung der DGPF Benutzeroberfläche umfasst folgende Features:

- In den `ResourceBundles` werden momentan `Strings` und `Chars` gespeichert. Die Klasse `LanguageSwitcher` bietet die statischen Methoden

  ```
  getString(String key):String
  ```

  und

  ```
  getChar(String key):char
  ```

  um über einen Schlüssel auf die entsprechende Werte im `ResourceBundle` zuzugreifen.

- Beim allerersten Start der Benutzeroberfläche wird die Sprache aus dem aktuellen `Locale` des Betriebssystems übernommen. Existiert kein `ResourceBundle` für diese Sprachen, so wird das Programm standardmäßig in Englisch angezeigt.

- Unter dem Menüpunkt *Bearbeiten / Einstellungen* hat der Benutzer die Möglichkeit, aus verfügbaren Sprachen zu wählen. Nach einem Neustart des Programms tritt die Änderung in Kraft. Diese Benutzerentscheidung wird permanent gespeichert (siehe Abschnitt 2.2.5) bis der Benutzer eine andere Sprache auswählt.

2.2.7 Darstellung der statistischen Daten

Übersicht

Die Datengewinnung der statistischen Daten aus der SearchEngine und die Darstellung dieser Daten in der grafischen Benutzeroberfläche ist in den folgenden Unterpunkten grob beschrieben und wird in weiteren Kapiteln näher erläutert.

1. Die SearchEngine generiert in regelmäßigen Abständen (beispielsweise immer nach einer Generation) ein `SearchUpdateEvent`. Nähere Informationen findet man im Unterabschnitt *Datengewinnung*.

2. Dieses `SearchUpdateEvent` enthält den `SearchState`, welcher die Daten der Suche enthält. Da für eine Suche verschiedene Suchmaschinen (Genetic, HillClimbing, SimulatedAnnealing) benutzt werden können, gibt es für jede Suchmaschinenart eine eigene Subklasse von `SearchState` mit verschiedenen Feldern.

3. Diese Felder werden über getter-Methoden (z.B. `getUpdateCount()`) gelesen und über setter-Methoden geschrieben. Nähere Informationen findet man im Unterabschnitt *Setzen und Abfragen statistischer Daten*.

4. Die Darstellung der Felder in der grafischen Benutzeroberfläche wird mittels Properties realisiert. Ein `Property` stellt ein Feld dar, welches einen eindeutigen Namen hat und neben anderen Methoden eine eindeutige setter- und getter-Methode besitzt. Diese beiden Methoden rufen dann über Java-Reflection die entsprechenden Methoden im `SearchState`- bzw. im `SearchParameters`-Objekt auf. Nähere Informationen findet man im Unterabschnitt *Properties*.

5. Erhält nun die grafische Benutzeroberfläche das erste `SearchUpdateEvent`, so werden die Properties, die das `SearchUpdateEvent` unterstützen, weiterverarbeitet. Diese Properties werden dann genutzt, um die Darstellungskomponenten der grafischen Benutzeroberfläche zu bauen. Da es verschiedene Arten von Properties (z.B. Für Integers, Fließkommazahlen oder Zeiten) gibt, existieren somit auch verschiedene Darstellungskomponenten. Properties, die Zahlen darstellen, werden zudem in eine separate Liste (`m_pdVector`) gespeichert. Diese Liste wird genutzt, um Diagramme erstellen zu können.

6. Für die Darstellung der statistischen Daten wird für jede `Property`-Gruppe (`PropertySubSet`) ein Tab erzeugt. In dieses Tab kommen alle Darstellungskomponenten, die zu der jeweiligen Gruppe gehören. Diagramme werden vom Benutzer erstellt und kriegen jeweils ein Diagramm-Tab.

7. Bei jedem weiteren `SearchUpdateEvent` werden alle Darstellungskomponenten und Diagramme aktualisiert.

Datengewinnung

Wie in der Übersicht erwähnt, generiert die SearchEngine in regelmäßigen Abschnitten ein Ereignis. In diesem Ereignis (`SearchUpdateEvent`) wird der Suchzustand mit der Auswertung der vergangenen Generation übermittelt und kann schließlich von allen empfangen werden, die sich als Listener bei der SearchEngine registriert haben. In der Regel befinden sich die grafische Benutzeroberfläche und die SearchEngine auf verschiedenen Rechnern. Die Kommunikation zwischen diesen Komponenten ist an dieser Stelle jedoch nicht relevant und wird deshalb im Abschnitt 2.4 *Netzwerkkommunikation* genauer behandelt. Für die Datengewinnung reicht es zu wissen, dass in der grafischen Benutzeroberfläche ein Client mit einem Unterfenster (`InternalFrame`) dargestellt wird. Damit auch das jeweilige Unterfenster die Ereignisse erhalten kann, muss es das Interface `IEventListener` implementieren. Sobald also die SearchEngine ein Ereignis erzeugt hat, wird die Methode `receive(final SfcEvent p_event)` des `InternalFrame` aufgerufen und man erhält den aktuellen `SearchState`. Die Abbildung 2.6 zeigt diesen Sachverhalt vereinfacht.

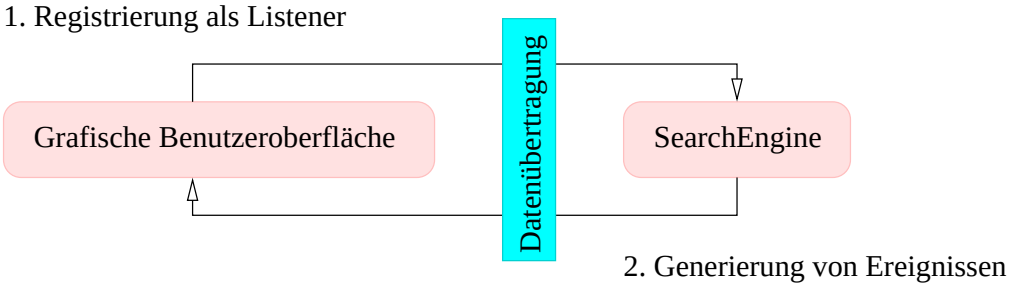

Abbildung 2.6: Gewinnung der Daten aus der SearchEngine.

Setzen und Abfragen statistischer Daten

Um die Funktionsweise der grafischen Benutzeroberfläche richtig verstehen zu können, ist ein Einblick in die Arbeitsweise der SearchEngine unumgänglich. Dazu stellt die SearchEngine auf der einen Seite die SearchEngine-Parameter und auf der anderen Seite den SearchEngine-Zustand (Siehe Abbildung 2.7) bereit.

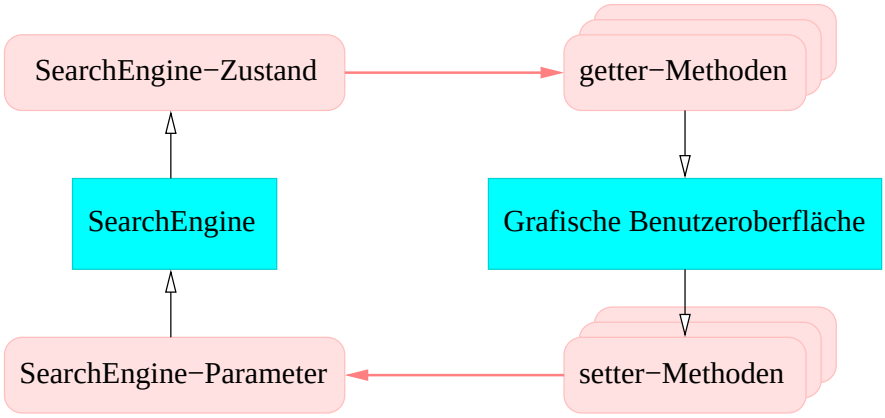

Abbildung 2.7: Setzen und Abfragen der statistischen Daten.

Bei den SearchEngine-Parametern werden Einstellungen vorgenommen, die erst für die nächste Generation gelten werden. So kann man beispielsweise zwar die Mutationsrate in der grafischen Benutzeroberfläche in einer laufenden Generation erhöhen oder auch erniedrigen, wirksam wird diese Änderung jedoch erst für die nächste Generation.
Die aktuelle Mutationsrate — sowie auch alle anderen aktuellen Informationen einer Suche — stehen im Gegensatz zu den SearchEngine-Parametern hierbei in dem SearchEngine-Zustand. Diese Trennung ist erforderlich, da während einer Generation die Parameter für diese jeweilige Generation nicht geändert werden dürfen. Die Abbildung 2.7 zeigt hierzu, auf welchen Objekten die getter- und setter-Methoden der grafischen Benutzeroberfläche arbeiten.

Der SearchEngine-Zustand

Wie bereits in der Übersicht erwähnt, wird in einem Ereignis (`SearchUpdateEvent`) der aktuelle SearchEngine-Zustand übermittelt. Dadurch, dass die SearchEngine unterschiedliche Suchalgorithmen oder auch verschiedenen Verteilungsformen verwenden kann, existieren auch verschiedene Eigenschaften in Abhängigkeit von dem verwendeten Suchalgorithmus. So gibt es beim Hill Climbing Suchalgorithmus beispielsweise kein *Crossover* und daher auch keine *Crossover-Rate*. Aus diesem Grund enthält der SearchEngine-Zustand verschiedene Felder, je nachdem, welcher Suchalgorithmus angewendet wird (Siehe hierzu die Abbildung 2.8).

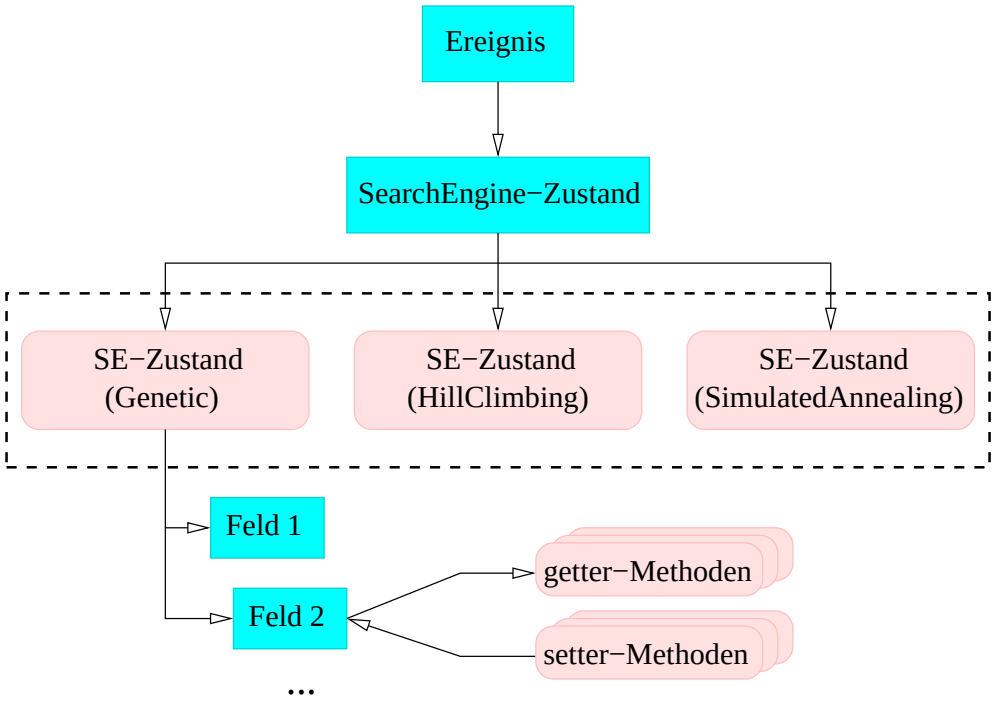

Abbildung 2.8: Der SearchEngine-Zustand.

Das Gleiche gilt auch für die im Abschnitt 2.2.7 erwähnten Parameter, welche die Steuerdaten für die nächste Generation enthalten. Diese Felder können mit getter- und setter-Methoden abgefragt und gesetzt werden. Für eine Verdeutlichung kann hier das Feld UpdateCount vorgestellt werden, welcher mit der Methode `getUpdateCount()` abgefragt wird.

Properties

In der Übersicht wurde erwähnt, dass so genannte Felder, die im Unterabschnitt *Der SearchEngine-Zustand* vorgestellt wurden, die statistischen Informationen enthalten. Diese Felder gilt es in der grafischen Benutzeroberfläche darzustellen. Zum Auslesen und setzen dieser Felder existieren getter- und setter-Methoden. Wie kann man diese Felder nun in der grafischen Benutzeroberfläche darstellen? Dazu gibt es zwei Möglichkeiten:

1. Man implementiert für jeden Suchalgorithmus und somit auch für jedes Feld eine gesonderte Darstellung. Kämen zu einem späteren Zeitpunkt neue Suchalgorithmen dazu oder würde ein Suchalgorithmus geändert werden, dann müsste man die Darstellungskomponenten ebenfalls anpassen.

2. Man implementiert Dastellungskomponenten, welche nicht an einen Suchalgorithmus gebunden sind. Die Darstellung würde somit automatisch erstellt werden (beispielsweise anhand des Typs des jeweiligen Feldes).

In diesem Projekt wurde der zweite Ansatz von Beginn an verfolgt und auch erfolgreich umgesetzt. Realisiert wurde es mit so genannten Properties, also Eigenschaften einer Suche.
Jede Instanz der Klasse `Property` stellt somit eine Meta-Eigenschaft dar und beschreibt ein Feld von einer oder mehreren Zustands-Klassen. Die Properties können ebenfalls mittels getter- und setter-Methoden gesetzt und gelesen werden. Es ergibt sich der in der Abbildung 2.9 dargestellte Verlauf.

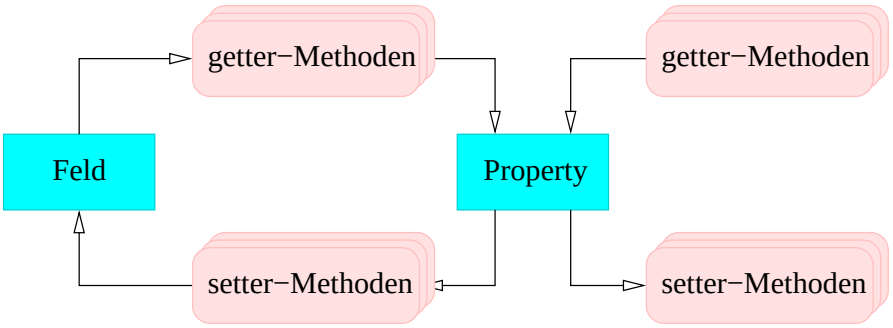

Abbildung 2.9: Zusammenarbeit zwischen Feldern und Properties

Die Schwierigkeit bestand darin, dass man erst zur Laufzeit weiß, welche Felder benutzt werden und welche Methoden zur Laufzeit aufgerufen werden sollen. Für die Lösung des Problems wurde die Reflektion der Programmiersprache Java zur Hilfe genommen. Die dazugehörige Reflection-API ist ein Bestandteil der Laufzeitumgebung und ermöglicht damit die Analyse von Klassen, Strukturen und Schnittstellen zur Laufzeit. Damit ist es möglich, auf die Felder und deren Methoden zuzugreifen, deren Existenz oder genaue Ausprägung zur Zeit der Programmerstellung nicht bekannt war.
Der Vorteil dieser Vorgehensweise ist, dass man in der grafischen Benutzeroberfläche sich nicht mehr darum kümmern muss, welche Felder dargestellt werden sollen und welche Methoden hierfür aufgerufen werden müssen. Die Darstellung der statistischen Werte werden über Properties abgewickelt, welche wiederum die Felder auslesen und gegebenenfalls setzen
Zur Wiederholung sei nochmal erwähnt, dass die getter- und setter-Methoden nicht auf den gleichen Objekten arbeiten. Wie in der Abbildung 2.7 gezeigt, arbeiten die getter-Methoden der Properties auf dem SearchEngine-Zustand und die setter-Methoden der Properties auf dem SearchEngine-Parameter.

Schaltflächen

Im Unterabschnitt *Properties* wurde erläutert, dass man mit Meta-Eigenschaften, so genannten Properties, auf die statistischen Daten der Suche zugreifen kann. Die dazugehörige Darstellung dieser Meta-Daten in der grafischen Benutzeroberfläche wurde mittels so genannten Controls realisiert. Ein `Control` stellt dabei eine Schaltfläche dar und repräsentiert eine `Property`.
Da der Zustand einer Suche Properties verschiedener Arten hat, wie beispielsweise statistische oder performanztechnische, werden die vielen Properties eines Suchzustandes in Gruppen (`PropertySubSet`) eingeteilt. In der grafischen Benutzeroberfläche (siehe Abbildung 2.10) wird für jede Gruppe ein Tab erzeugt, in welcher sich die Controls/Properties befinden, die zu der jeweiligen Gruppe gehören. Die Tabs können dabei beliebig ein- und ausgeblendet werden. Dafür existiert in jedem `InternalFrame` eine Menüleiste.

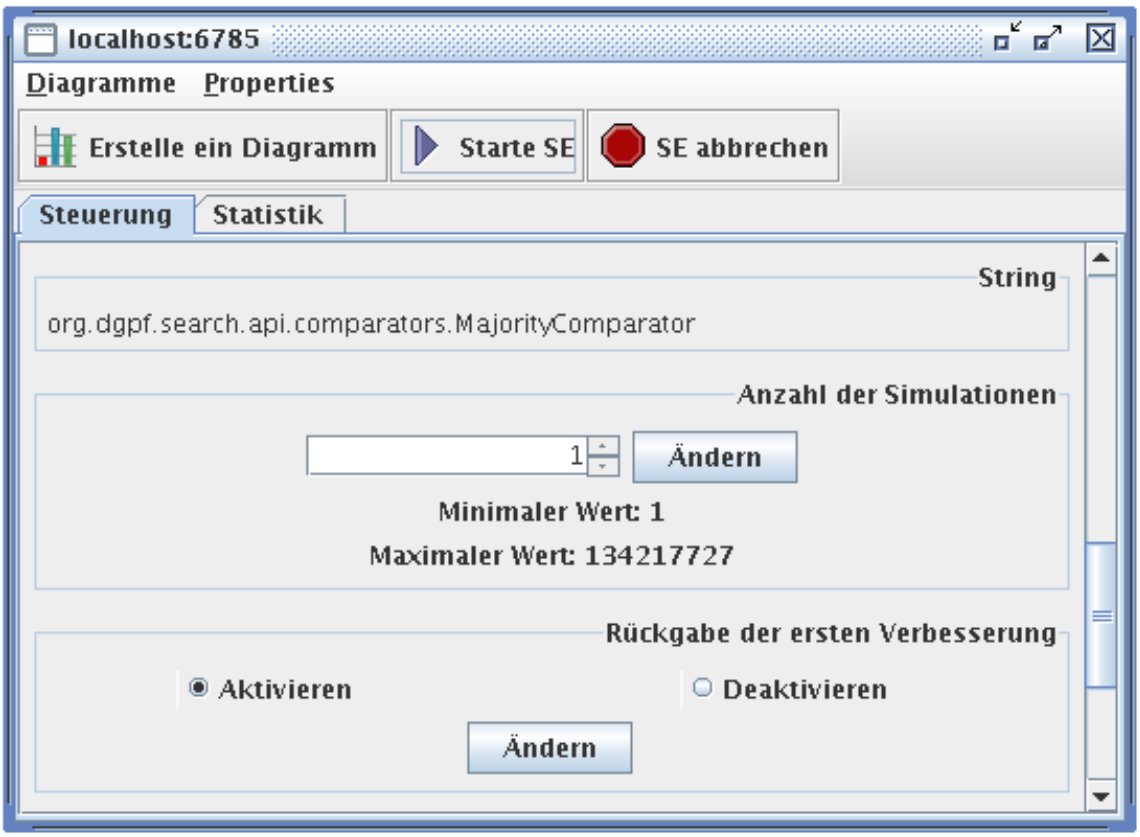

Abbildung 2.10: `Property`-Gruppen in Form von Tabs.

Um die richtige Schaltfläche für ein `Property` erstellen zu können, führen Properties Typ-Informationen mit sich. Mit der Hilfe eines Hashes erhält man über die Typ-Informationen für die jeweilige Property-Art schließlich eine Factory, die dazu benutzt

wird, eine entsprechende Schaltfläche zu erzeugen. Aus den verschieden `Property`-Arten ergeben sich verschiedene Schaltflächen. Die unten stehende Tabelle gibt hierzu eine Übersicht über die vorhandenen Properties und die dazugehörigen Schaltflächen.

Property	Schaltfläche
`BoundedByteProperty`	`PropertyControl`
`BoundedDoubleProperty`	`BoundedDoublePropertyControl`
`BoundedIntProperty`	`BoundedIntPropertyControl`
`BoundedLongProperty`	`BoundedLongPropertyControl`
`Property`	`PropertyControl`
`TimeProperty`	`TimePropertyControl`

Tabelle 2.1: Properties und die dazugehörigen implementierten Schaltflächen.

Die Abbildungen 2.11, 2.12 und 2.13 zeigen hierzu drei verschieden Schaltflächen, die an Hand des `Property`-Typs in der grafischen Benutzeroberfläche automatisch erzeugt wurden.

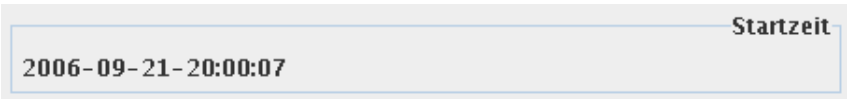

Abbildung 2.11: Schaltfläche für eine `TimeProperty`.

Abbildung 2.12: Schaltfläche für eine `Boolean-Property`.

Man sieht an der Tabelle und an den Beispielen (siehe Abbildung 2.11, 2.12 und 2.13), dass für wichtige und oft genutzte Datentypen Schaltflächen existieren. Trotzdem kann der Wunsch nach neuen Datentypen und neuen Schaltflächen bestehen. So könnte man sich das Szenario vorstellen, dass eine Suche mit komplexen Zahlen arbeitet. Eine sinnvolle und mögliche Schaltfläche könnte eine separate Darstellung des Real- und Imaginär-Anteils ermöglichen. Es ergeben sich daraus zwei wichtige Fälle, die es zu beachten gilt:

1. Die Suche arbeitet mit komplexen Zahlen. Es existiert aber keine dafür vorgesehene Schaltfläche. Werden die statistischen komplexen Werte überhaupt angezeigt? Wenn ja, wie?

2. Die Suche arbeitet mit komplexen Zahlen. Die statistischen komplexen Werte will man in einer gesonderten Schaltfläche anzeigen. Was muss dafür gemacht werden?

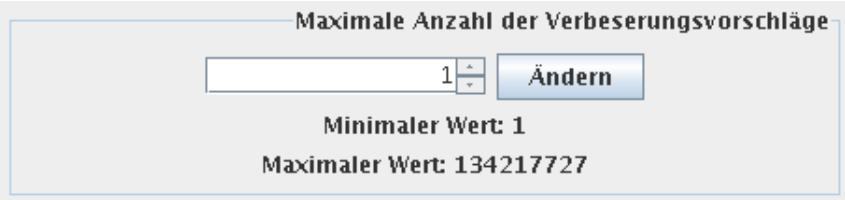

Abbildung 2.13: Schaltfläche für eine `BoundedIntProperty`.

Die momentane Implementierung sieht es vor, dass alle statistischen Daten angezeigt werden. Falls keine Schaltfläche für eine spezielle `Property` existiert, dann wird rekursiv geprüft, ob für die Superklasse der `Property` eine Schaltfläche existiert. Da alle Properties von der Klasse `Property` abgeleitet sind, ist die einfachste Darstellung einer beliebigen `Property` durch die `PropertyControl` immer gegeben. Die Antwort auf den ersten Fall würde somit lauten, dass nicht notwendigerweise eine Schaltfläche für jeden Typ existieren muss, um die statistischen Daten anzuzeigen. Reicht jedoch die Ausgabe der Schaltfläche `PropertyControl` nicht aus und man möchte eine gesonderte Schaltfläche erstellen, dann sind die folgenden zwei Punkte auszuführen:

1. Die neue Schaltfläche muss das Interface `IControl` implementieren. Die wichtigen Methoden sind hier die `get()`-Methode, welche die jeweilige JComponente zurück liefert und die `update(...)`-Methode, die das Control aktualisiert.

2. Diese neue Schaltfläche muss anschließend der `Map<Class, IControlFactory>` in der Klasse `FactoryMap` hinzugefügt werden. Der Schlüssel ist hierbei die Klasse der jeweiligen `Property` und der Wert ist die neue Schaltfläche.

2.2.8 Client-Liste und Informations-Block

Um die Übersicht über die verbundenen Daemons zu erleichtern, wurde in das Hauptfenster, wie bereits im Abschnitt 2.2.2 erwähnt, zwei Unterfenster eingebaut. Beide Fenster lassen sich bequem per Maus verkleinert / vergrößern.

Client-Liste

Die Liste von verbunden Clients wird flach, mittels einer `JList` dargestellt (siehe Abbildung 2.1). In Zukunft wäre auch eine baumartige Darstellung denkbar, falls Clients zu Gruppen zusammengefasst werden sollen. In der Liste sind alle verbundenen Clients dargestellt. Mittels dieser Liste ist es möglich, die zu einem Client gehörigen Fenster zu manipulieren (siehe Abschnitt 2.2.2). Ferner besitzt jeder Eintrag ein Submenü. Für jeden Eintrag wurden so genannte `Actions` entwickelt. Jede `Action` ist eine eigene Klasse und im Package `org.dgpf.gui.actions` enthalten. Jede dieser Klassen implementiert das Interface `java.awt.event.ActionEvent`. Die Klassen werden beim Erscheinen des Popupmenüs instanziiert und bekommen den gewählten Client aus der Liste als Parameter übergeben. Wird dann eine `Action` ausgelöst, so reagiert sie indem es die Metho-

de `public void actionPerformed(ActionEvent event)` ausführt. Folgende `Actions` sind momentan vorhanden:

- `AbortSeAction` - Bricht den Suchjob ab.
- `CloseAction` - Schließt das aktuelle Fenster des Clients.
- `CloseAllAction` - Schließt alle Fenster.
- `MinimizeAction` - Minimiert das aktuelle Fenster.
- `UnminimizeAction` - Stellt die ursprüngliche Größe des aktuellen Fensters wieder her.

Informations-Block

Zwar werden viele Informationen in internen Fenstern dargestellt, aber meistens haben diese Fenster viele Reiter, und man muss immer zwischen diesen Reitern blättern. Um das zu vermeiden, wurde ein Fenster im unteren rechten Teil des Hauptfensters (siehe Abbildung 2.1) eingebaut, welches die wichtigsten Informationen zu dem ausgewählten Client anzeigt. Es sind aber auch einige Angaben vorhanden, die sonst nirgendwo direkt zu sehen sind. Momentan werden folgende Informationen dargestellt:

- `Status` - Zeigt, ob der Suchjob auf dem ausgewählten Daemon gerade läuft. Dieses Feld wird durch das `RunningStatusEvent` (siehe Abschnitt 2.4.3) aktualisiert.
- `Starttime` - Zeigt wann der Suchjob angestoßen wurde.
- `Jar` - Zeigt den Namen des Programms, dass momentan auf dem Client läuft.
- `Last Update` - Zeit seit der letzten Aktualisierung. Dieses Feld und `Starttime` werden aus dem `SearchUpdateEvent` gelesen.
- `Connected since` - Seit wann die Benutzeroberfläche mit dem Daemon verbunden ist.

Statusleiste

Im unterem Infopanel werden links die Anzahl der Daemons angezeigt, die in der aktuellen Rechnerliste geladen wurden. Falls die Clients aus einer Datei geladen wurde, so wird auch noch der absolute Pfad zu dieser Datei angezeigt. Rechts wird die Anzahl von verbundenen Daemons angezeigt.

2.2.9 Grafische Darstellung der statistischen Daten

Mit zunehmenden statistischen Daten im Verlauf einer Suche, gehen für den Benutzer relevante Informationen unter. Sinnvolle Grafiken bringen hierzu nicht nur Abwechslung, sondern heben wichtige Werte hervor. Die leistungsfähige Bibliothek JFreeChart[4] hilft diesem Projekt, mit Java aussagefähige Diagramme zu zeichnen und statistische Daten grafisch darzustellen.
In Abschnitt 3.2.1 wird die JFreeChart-Bibliothek vorgestellt und in den folgenden Abschnitten gezeigt, wie die damit erzeugten Diagramme in das Projekt integriert wurden.

JFreeChart

Die Bibliothek JFreeChart der Firma Object-Refinery ist ein SourceForge-Projekt, welches seit dem Jahr 2000 entwickelt wird. JFreeChart bietet zwei Möglichkeiten Diagramme zu erstellen:

1. Ein Diagramm interaktiv anfertigen.

2. Ein Diagramm selbstständig implementieren.

In diesem Projekt wurden die Diagramme selbstständig implementiert. Dazu wurde die zentrale Klasse der Bibliothek `org.jfree.chart.JFreeChart` in das Projekt integriert. Sie ist ein Container für Titel und Legenden sowie Objekte vom Typ `org.jfree.chart.plot.Plot` und `org.jfree.data.Dataset`. Die Klasse `Plot` ist eine abstrakte Klasse. Subklassen wie `XYPlot` oder `PiePlot` implementieren die eigentliche Funktionalität. `Dataset` ist eine Schnittstelle (ein Interface), die von einer Vielzahl von Klassen implementiert wird. Ein `Dataset` besteht meist aus mehreren Datenserien, beispielsweise setzt sich das `XYDataset` aus Objekten vom Typ `XYSeries` zusammen. Eine solche Datenserie umfasst einen Namen und eine Menge von Punkten (x, y). Für Kreisdiagramme gibt es ein `DefaultPieDataset`, für Balkendiagramme ein `CategoryDataset` und so weiter.

Erstellung von Diagrammen mit `JFreeChart`

Um ein vorkonfiguriertes Diagramm mittels `JFreeChart` zu zeichnen, kommt in aller Regel eine statische Methode von `ChartFactory` zum Einsatz:

```
JFreeChart chart = ChartFactory.createPieChart(...);
Plot plot = chart.getPlot();
```

Über das `Plot`-Objekt lassen sich einige Aspekte des Plots manipulieren. So läßt sich mit der Methode `setBackgroundImage()` beispielsweise ein Hintergrundbild setzen. Zwei weitere GUI-Klassen sind sehr nützlich:

1. `ChartPanel`

2. `ChartFrame`

Letztere ist eine einfache Subklasse von `JFrame` mit einem `ChartPanel` als `Content-Pane`. Das `ChartPanel` wurde in diesem Projekt verwendet und stellt all jene nützliche Operationen zur Verfügung, die für eine interaktive Bearbeitung des Diagramms wichtig sind:

- Drucken
- Als PNG[7] speichern
- Zoomen
- Bearbeiten der Eigenschaften über ein Menü
- usw.

Mit dem Wissen um diese Klassen ist der Weg zum eigenen Diagramm nicht weit und vollzieht sich in drei wesentlichen Schritten:

1. Zuerst erzeugt man ein geeignetes `Dataset`.
2. Anschließend erzeugt man ein `JFreeChart` über die `ChartFactory`.
3. Zum Schluss fehlt nur noch ein `ChartFrame` für die Anzeige. Soll das Diagramm in eine Anwendung integriert werden, bietet sich das `ChartPanel` an.

Realisierte Diagramme

In dem vorgehendem Abschnitten wurde erläutert, was `JFreeChart` ist und wie diese Bibliothek benutzt werden kann, um Diagramme zu erstellen.
In diesem Projekt wird dem Benutzer die Möglichkeit gegeben, statistische Daten mittels `JFreeChart`-Diagrammen grafisch darzustellen. Jedes Diagramm wird bei der Erstellung in ein Diagramm-Tab `ChartPanel` hinzugefügt und kann nach Wahl auch geschlossen werden. Folgende drei Diagramme stehen dem Benutzer zur Verfügung:

1. Kreisdiagramm (siehe hierzu die Abbildung 2.14).
2. Säulendiagramm (siehe hierzu die Abbildung 2.15).
3. XY-Liniendiagramm (siehe hierzu die Abbildung 2.16).

Auf dieselbe Art und Weise, wie im Abschnitt 2.2.7 *Properties* erläutert wurde, wie man statistische Daten in der grafischen Benutzeroberfläche anzeigt, so wird auf die gleiche Art und Weise auf die statistischen Daten zugegriffen, um die Daten für Diagramme zu gewinnen.

[7] PNG steht für `Portable Network Graphics` und ist ein Grafikformat für Rastergrafiken.

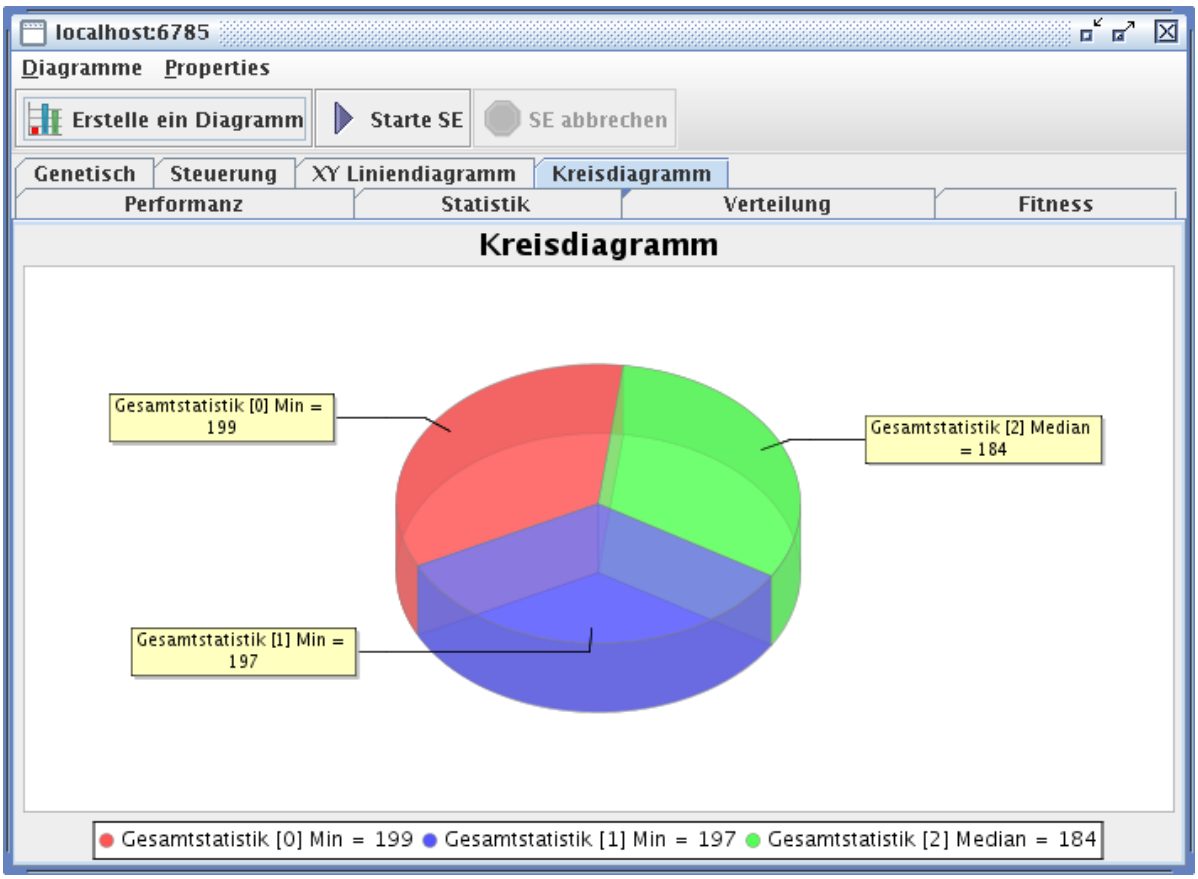

Abbildung 2.14: `JFreeChart`-Kreisdiagramm.

Hierzu werden Properties, die in einem Diagramm dargestellt werden können, zunächst in dem Vektor `m_pdVector` gespeichert. Welche `Property` zum Zeichnen geeignet ist und welche nicht, wird beim ersten `SearchUpdateEvent` in der Methode `addElement(...)` des `InternalFrame` bestimmt. Dies sind zum Einen die Properties, deren Typ eine `StatisticInfo` ist und zum Anderen die Properties, die einen `Integer`-, `Long`- oder `Double`-Wert zurück liefern. Die eigentlichen Diagramme implementieren alle das Interface `IChart`. Die wichtigen Methoden sind hierbei:

- Die `update()`-Methode, welche das Diagramm bei jedem `SearchUpdateEvent` aktualisiert.
- Die `getSampleChart()`-Methode, welche ein Beispieldiagramm zurück liefert. Benutzt wird diese Funktion beispielsweise vom Diagramm-Wizard.
- Die `createChart()`-Methode, welche das eigentliche Diagramm erzeugt.

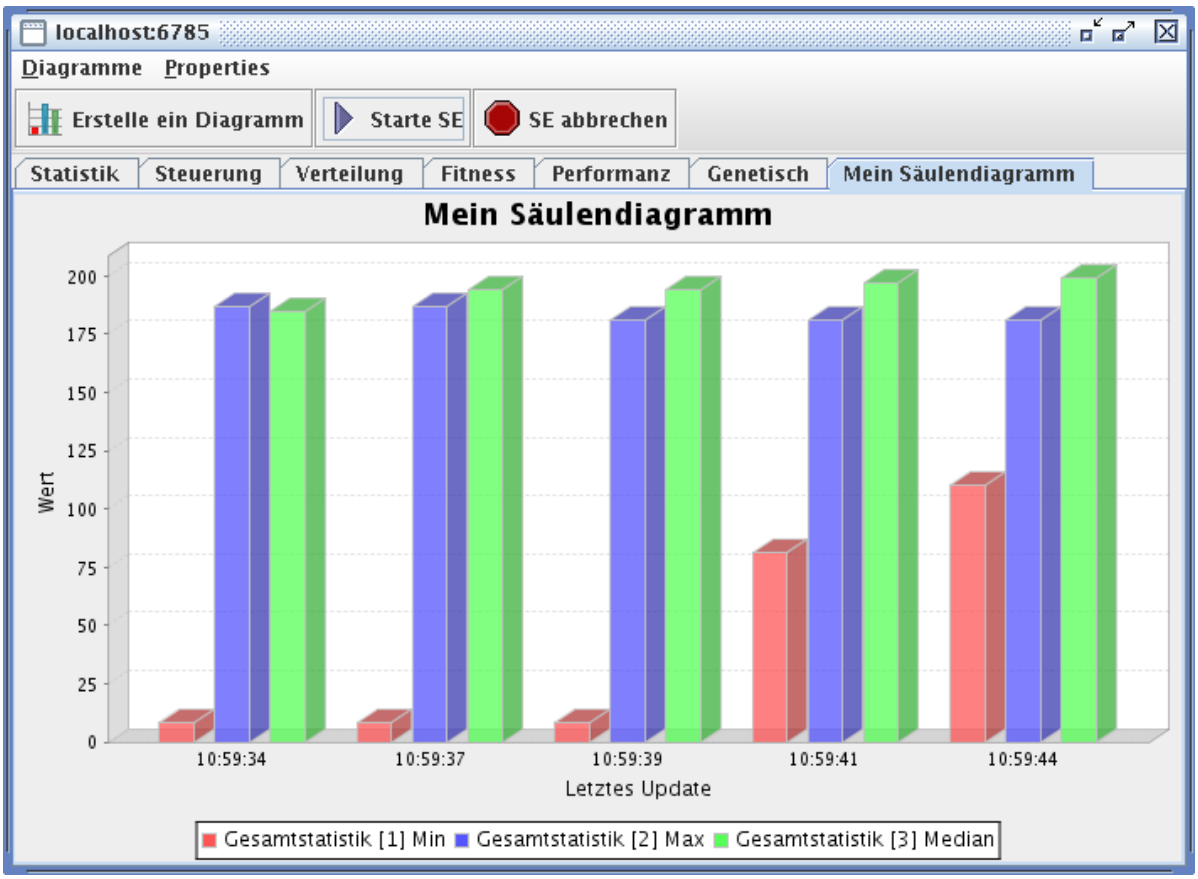

Abbildung 2.15: `JFreeChart`-Säulendiagramm.

2.2.10 Automatische Protokollierung

Bei der Protokollierung (auch Logging genannt) können Ereignisse oder Aktionen des Benutzers mitprotokolliert werden. Dies ermöglicht beispielsweise das Lokalisieren von Fehlfunktionen oder Nachvollziehen von Suchverläufen. Die Logging-Informationen können auf verschiedenen Arten und Weisen gespeichert werden:

- Die Abspeicherung eines Diagramms im PNG-Format. Das Diagramm beinhaltet die vom Benutzer gewählten statistischen Werte. Die Protokollierung kann sowohl über die komplette Suchdauer erfolgen, als auch über eine vom Benutzer festgegebene Zeitdauer.
- Die Abspeicherung des Suchverlaufs in eine Textdatei. Die Protokollierung erfolgt parallel zu der eigentlichen Suche in einer für den Menschen gut lesbaren Form.
- Die Abspeicherung des Suchverlaufs in eine Textdatei. Die Protokollierung erfolgt parallel zu der eigentlichen Suche im CSV[8]-Format. Eine CSV-Datei dient dabei zur Speicherung oder zum Austausch einfach strukturierter Daten. Die einzelnen

[8]Das Kürzel CSV steht für Character Separated Values oder Comma Separated Values

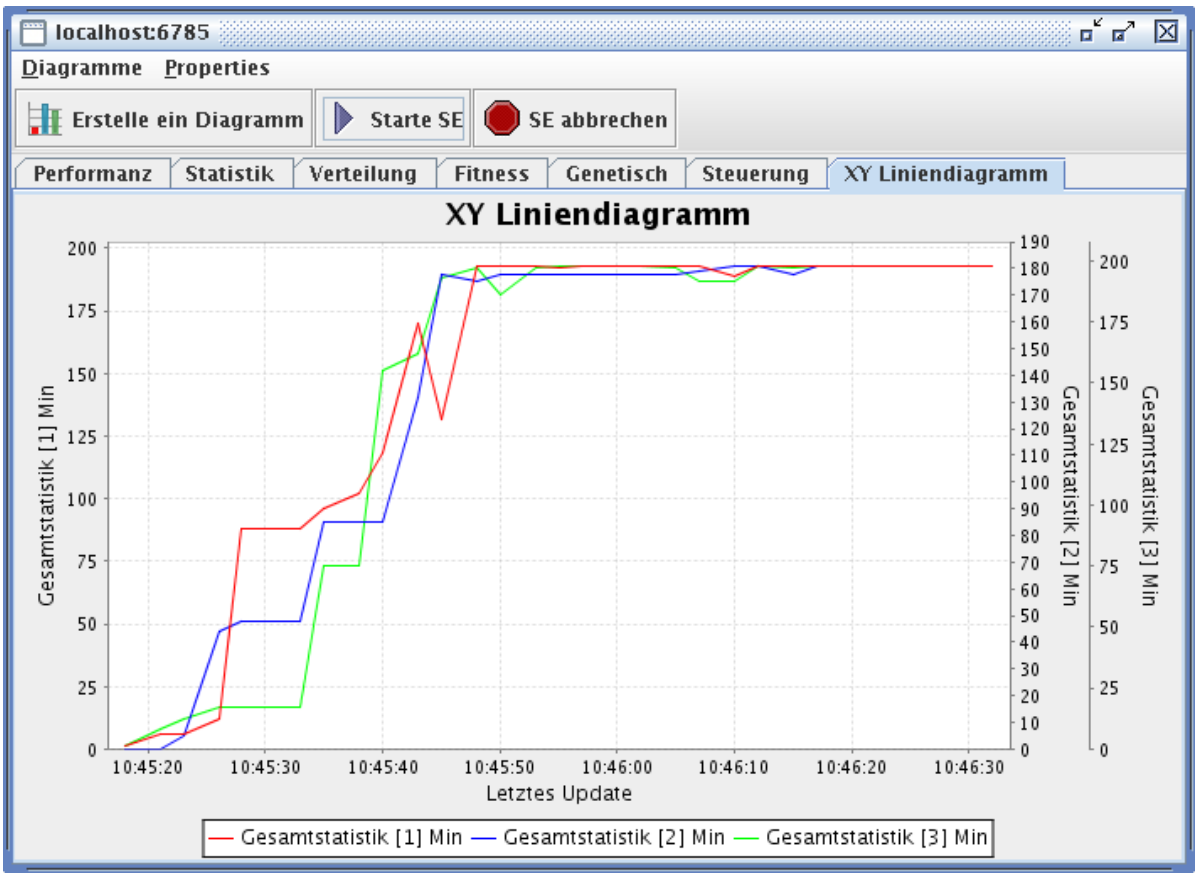

Abbildung 2.16: `JFreeChart`-XY-Liniendiagramm.

Werte werden in dieser Datei durch ein spezielles Trennzeichen, beispielsweise das Komma, getrennt.

Alle drei Formen der Protokollierung können mittels der grafischen Benutzeroberfläche realisiert werden. Die Protokollierung der statistischen Daten durch ein Diagramm wurde in den Abschnitten 2.2.9 *Realisierte Diagramme* und 2.2.3 *Diagramm-Wizard* behandelt und wird an dieser Stelle nicht weiter erläutert. Auf die Abspeicherung des Suchverlaufs in eine Textdatei wird jedoch im weiteren Verlauf eingegangen.

Multiplexer

Wie in Abschnitt 2.2.7 *Datengewinnung* grob erläutert, registriert sich die grafische Benutzeroberfläche bei der SearchEngine als `Listener`. Anhand der Ereignisse, die das `InternalFrame` anschließend erhält, werden die Schaltflächen aufgebaut und aktualisiert. Der Protokollverlauf eines Loggers (eines Protokollanten) verläuft symmetrisch. Jeder Protokolant muss sich ebenfalls als Listener bei der SearchEngine anmelden. Dadurch ist es beispielsweise möglich, die einzelnen Logger ohne die grafische Benutzeroberfläche zu benutzen.

Damit man sich nicht mehrfach bei der SearchEngine anmelden muss, wurde ein `MultiplexingWriter` implementiert, welcher die ankommenden Informationen / Ereignisse multiplext. Somit ist gewährleistet, dass sich die grafische Benutzeroberfläche nur ein mal bei der SearchEngine als `Listener` registriert. Mit der Methode `add(final Writer p_o)` ist es dem `MultiplexingWriter` möglich beliebig viele `Writer` hinzuzufügen.
So wird im `InternalFrame` beispielsweise ein `MultiplexingWriter` benutzt, welcher zum Einen die statistischen Daten in eine CSV-Logdatei auf die Festplatte speichert und zum Anderen dieselbe Ausgabe in einem Log-Tab (`JTextPane`) mitprotokolliert.

Starten und Anzeigen einer Protokollierung

Im Abschnitt 2.2.3 wurde der Dialog für Benutzereinstellung vorgestellt. Die Option *Logging* bietet die Möglichkeit, die Protokollierung zu aktivieren. Ist diese Option aktiv, so werden zwei grundlegende Funktionalitäten im `InternalFrame` aktiviert:

- Sowohl für den menschenlesbaren Logger, als auch für den CSV Logger werden Tabs erzeugt, in welchen die Suche protokolliert wird. Diese Tabs können aus- und wieder eingeblendet werden. Beim Ausblenden wird im Hintergrund dennoch die Protokollierung weitergeführt.
- Sowohl für den menschenlesbaren Logger, als auch für den CSV Logger werden Logdateien auf der Festplatte erzeugt. Das Verzeichnis ist das vom Benutzer festgelegte Verzeichnis aus dem Dialog für Benutzereinstellungen.

2.3 Hintergrunddienst / Daemon

Der Daemon arbeitet als Hintergrundprozess. Das heisst er wird einmal gestartet und läuft dann ohne weitere Benutzerinteraktion im Hintergrund. Er öffnet, nachdem er initiiert wurde, einen Netzwerkport[9] und horcht auf ihm. Eingehende Verbindungen von einer GUI werden – nach einem knappen Handshaking – angenommen und mit Statusinformationen des Suchalgorithmus versorgt, sofern bereits eine Suchaufgabe verarbeitet wird.

2.3.1 Starten von Suchaufgaben

Der Daemon startet Suchaufgaben auf Verlangen des Benutzers. Hier soll nun auf den Ablauf aus Sicht des Daemons eingegangen werden. Empfängt der Daemon eine entsprechend aufgebaute JAR-Datei von der GUI, so speichert er sie temporär, läd sie zur Laufzeit und startet den enthaltenen Suchalgorithmus per Reflection-API. Dabei ist auf eine korrekte Einhaltung des Formats der JAR Datei zu achten. Die Spezifizierung eines einheitlichen Formats war nicht Teil des Projekts, soll aber der Vollständigkeit halber

[9] Der von der IANA zugewiesene Standardport für das das DGPF Projekt ist 6785. Er kann aber auch manuell angepasst werden.

hier kurz erwähnt sein. Sie muss ein Package names `startup` besitzen, welches eine Klasse

`Factory` (abgeleitet von `org.dgpf.search.api.utils.SearchFactory`)

enthält. Sie implementiert die Methode

`public final SearchEngine<?> create()`.

Diese Methode übernimmt alle notwendigen Schritte um die Suchaufgabe zu initialisieren. Danach wird mit dem Aufruf von `public final void start ()` die `SearchEngine` gestartet. Auf diesem Wege ist eine einheitliche Schnittstelle geschaffen, so dass der Daemon sich nicht um die Eigenheiten aktueller oder kommender Suchalgorithmen kümmern muss. Eine automatisierte Erstellung der JAR-Dateien ist sinnvoll, jedoch nicht Teil der Aufgabenstellung und soll in einem Folgeprojekt bearbeitet werden (siehe auch Abschnitt 3.5).

2.3.2 Statusinformationen eines Suchvorgangs

Jeder Suchjob erzeugt in seinem Lebenszyklus eine Vielzahl von Statusinformationen in Form von Ereignissen (`SearchUpdateEvents`). Diese werden schlussendlich von der GUI angezeigt (siehe Abschnitt 2.2). Dafür registriert sich der Daemon bei der Suchaufgabe und erhält so alle produzierten Ereignisse. Der Daemon leitet diese über den in Abschnitt 2.4 näher erläuterten Mechanismus an die GUI weiter, so dass diese immer über die aktuelle Entwicklung informiert ist.

2.3.3 Adaption an Suchparameter zur Laufzeit

Benutzerinteraktionen, die auf Seite der grafischen Benutzeroberfläche passieren, werden angenommen und an den Suchauftrag weitergeleitet. Diese Ereignisse kommen bei der Netzwerkkomponente an (Abschnitt 2.4). Auch dort hat sich der Daemon registriert, so dass er alle von der GUI kommenden Ereignisse empfängt. Hier handelt es sich um `PropertyUpdateEvents`. Sie werden an den Suchauftrag weitergeleitet, der diese zu interpretieren hat und sein Verhalten entsprechend anpassen muss.

2.4 Netzwerkkommunikation

Die Kommunikation der beiden Komponenten Daemon und GUI läuft über ein TCP/IP-fähiges Netzwerk. So ist es möglich, eine Vielzahl von arbeitenden Computern über das Netzwerk zu kontrollieren. Dabei stehen die folgenden beiden Möglichkeiten (auch gemischt) zur Verfügung:

- Eine GUI kontrolliert eine Vielzahl von Daemons.
- Zwei oder mehrere GUIs verbinden sich mit einem Daemon.

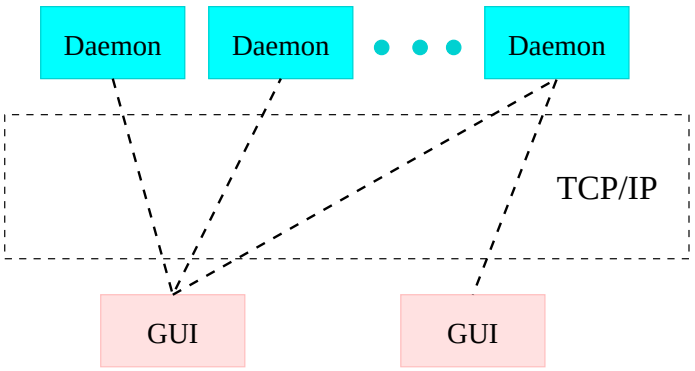

Abbildung 2.17: Verbindung zwischen zwei grafischen Benutzeroberflächen und einer Reihe von Hintergrundprozessen über ein TCP/IP-basiertes Netzwerk.

Das eine GUI sich zu mehreren Daemons verbindet ist wichtig und eine Kernfunktionalität der Software. Auf der anderen Seite ist der Aspekt, dass mehr als eine GUI zur selben Zeit eine Verbindung zu einem einzelnen Daemon aufbauen kann ebenfalls notwendig. Lässt jemand eine GUI über einen längeren Zeitraum auf, so könnte in dieser Zeit niemand mehr auf die Daemons zugreifen (siehe Abbildung 2.17).

2.4.1 Protokoll

Die Verbindung wird durch ein eigenständiges primitives Netzwerkprotokoll abgewickelt. Es besteht aus Paketen, die eines der in der Klasse

`org.dgpf.search.control.protocol.Commands`

definierten Kontrollwörtern gefolgt von den eigentlichen Nutzdaten enthalten. Die Daten sind serialisierte Java-Objekte. Auf beiden Seiten – also auf Daemon-, wie auch auf GUI-Seite – befindet sich jeweils eine Instanz der Klasse

`org.dgpf.search.control.protocol.NetworkCommunicationPeer`.

Sie bietet eine beidseitige Kommunikation. Dafür implementiert sie die Interfaces `IEventListener`, sowie `IEventSource`. Der Daemon erhält Statusinformation von seinem aktuellen Suchjob. Die leitet er an seinen `NetworkCommunicationPeer`, der sie wiederum über das Netzwerk verschickt. Der `NetworkCommunicationPeer` auf der GUI Seite wiederum leitet die Ereignisse an die grafische Oberfläche weiter, wo sie beispielsweise in Form von Diagrammen aufbereitet werden (siehe Abschnitt 2.2.9). Die grafische Oberfläche, sowie der Hintergrunddienst besitzen beide einen `NetworkCommunicationPeer` und verständigen sich ausschließlich über Events. Diese werden über jeweils einen symmetrischen NetworkCommunicationPeer (NCP) transparent über ein TCP/IP-basiertes Netzwerk geschickt. Die SearchEngine produziert SearchUpdateEvents. Die GUI produziert u.a. `PropertyUpdateEvents`. Dieser Vorgang wird in Abbildung 2.18 gezeigt.

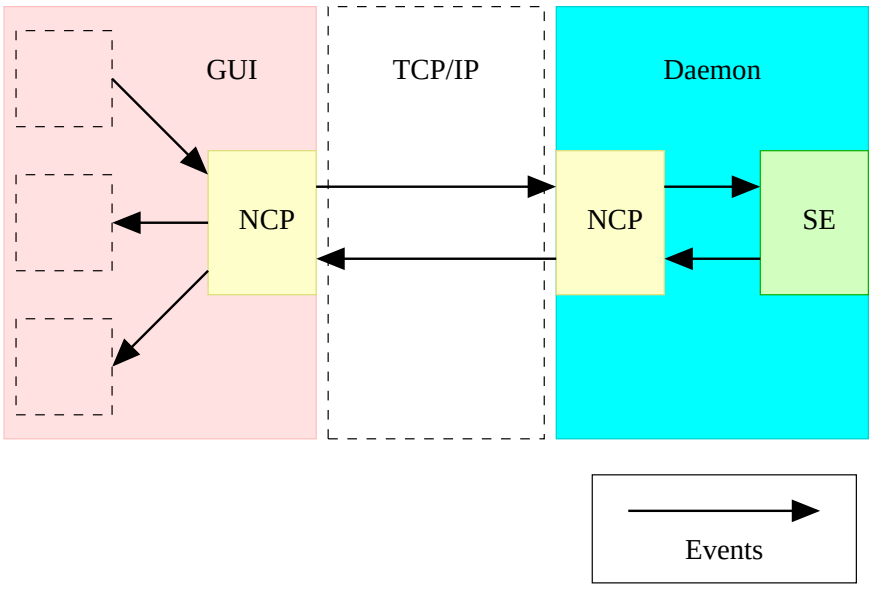

Abbildung 2.18: Die Netzwerkkommunikation über Events

Auch die GUI schickt ihre Daten über das Netzwerk zum Daemon. Wenn der Benutzer Parameter verändert, so erzeugt er implizit ein `PropertyUpdateEvent`, welches zum Daemon übertragen wird. Außerdem kann die GUI eine Suchaufgabe initiieren. Der Benutzer kann eine Suchaufgabe (in Form einer JAR-Datei) auswählen. Diese Datei wird dann an die verbundenen Daemons verschickt und von ihnen ausgeführt (Abschnitt 2.3.1).

2.4.2 Zirkulärer Zwischenspeicher

Aus der Tatsache, dass die Nachrichten vom `SearchEngine` abgeschickt werden, sobald dieser anfängt zu arbeiten, ergibt sich ein Problem: Will der Benutzer die Daten des Suchjobs von Anfang an in ein Diagramm schreiben, so müssen die `SearchUpdateEvents` komplett mitgeschrieben werden. Es können jedoch erst Diagramme erzeugt werden, nachdem das erste `SearchUpdateEvent` aufgefangen und aufgebaut wurde (vergl. Abschnitt 2.2.9). Da aber dann bereits ständig neue SearchUpdateEvents ankommen und der Benutzer u.U. nicht schnell genug seine Einstellungen vornehmen könnte, wurde im NetworkCommunicationPeer ein Ringpuffer eingebaut. Er hat eine variable Größe. Wenn er voll ist, so werden die Einträge gelöscht, die als erstes eingefügt wurden. Mit Hilfe dieses Puffers ist sichergestellt, dass der Benutzer in Ruhe seine Diagramme definieren kann diese trotzdem den Verlauf von Anfang anzeigen.

2.4.3 Events

Wie bereits erwähnt, wird die Kommunikation zwischen die beiden Komponenten der entwickelten Software, GUI und Daemon, nur über Events realisiert. Die beteiligten

Events sollen in den folgenden Abschnitten näher beschrieben werden.

Kommunikation vom Daemon zur GUI

Die Kommunikation in Richtung grafische Oberfläche beinhaltet, abgesehen vom Verwaltungsoverhead die Statusinformationen, die erzeugt werden. Die im Daemon eingebettete `SearchEngine` produziert in jedem Suchzyklus jeweils einen `SearchState`, der wiederum in ein `SearchUpdateEvent` verpackt wird. Dieses wandert vom `SearchEngine` über den Daemon, die beiden `NetworkCommunicationPeers` schließlich zum GUI. Dort wird es entsprechend interpretiert, d.h.

- als Diagramm angezeigt (Abschnitt 2.2.9)
- als statische `Property` dargestellt (Abschnitt 2.2.7)
- als vom User veränderbare `Control` dargestellt (Abschnitt 2.2.7)
- sowie auf Wunsch als Datei protokolliert 2.2.10.

Außerdem produziert der Daemon selbst noch das `RunningStatusEvent`. Es unterrichtet die GUI darüber, ob der aktuelle Suchjob gerade arbeitet und wie dieser heisst.

Kommunikation von der GUI zum Daemon

Wenn Nachrichten von der GUI zum Daemonen geschickt werden, so geht diesen immer eine Benutzerinteraktion vorraus. Die wohl häufigste Art von `Events` sind `PropertyUpdateEvents`. Immer wenn der Benutzer einen Suchparameter ändert, so wird ein entsprechendes Event erzeugt und über den gewohnten Weg zum `SearchEngine` geschickt. Wenn der Benutzer das Starten eines Suchvorgangs einleitet, so erhält jeder der betroffenen Daemons jeweils ein `JarFileEvent`. Dieses beinhaltet u.a. ein Byte-Array mit dem Inhalt der Jar-Datei des Suchjobs (siehe auch Abschnitt 2.3.1). Das Pendant zu diesem Event ist das sogenannte `AbortSeEvent`. Es wird geschickt, wenn der Benutzer sich dazu entschließt, einen Suchjob zu beenden.

3 Verschiedenes

3.1 Bekannte Probleme

Folgende Probleme wurden im Projektverlauf festgestellt:

- **Interpretation der Java-Zeit**
  Bei der Erstelung eines XY-Liniendiagramms (siehe Abbildung 2.16) oder eines Säulendiagramms (siehe Abbildung 2.15) hat der Benutzer die Möglichkeit auf die X-Achse die Zeit zu legen (z.B. die Dauer der Suche). Dabei wird die Zeit auf der X-Achse in einem vom Benutzer vorgegebenen Format dargestellt. Wird beispielsweise das `TimePattern` „`hh:mm:ss`“ ausgewählt, dann wird die Java-Zeit, welches einen `Long-Wert` darstellt, von der Klasse `java.text.SimpleDateFormat` falsch interpretiert. Die Stundenangabe hat gleich zu Beginn den Wert 1. Die Angaben der Minuten- und Sekunden-Werte stimmen jedoch.

- **Compilerfehler**
  Probleme beim Kompilieren mit dem Sun Java Compiler (vgl Abschnitt 2.1.2).

3.2 Verwendete externe Komponenten

Es durften in das Projekt nur Komponenten integriert werden, die — wie auch das DGPF Projekt selber — unter der LGPL Lizenz[1] stehen.

3.2.1 JFreeChart

Für die Darstellung der Diagramme wurde die Bibliothek `JFreeChart`[4] verwendet. Sie enthält alles, um in Javaprogrammen schöne, ansprechende Diagramme zu erstellen und steht auch unter der LGPL. Es ist aus diesem Grund nicht verwunderlich, dass sich die Bibliothek großer Beliebtheit erfreut, wie die lange Liste auf der `JFreeChart`-Homepage beweist (wird zur Zeit von ca. 40.000 bis 50.000 Entwicklern benutzt). Siehe auch Abschnitt 2.2.9.

[1] GNU Lesser General Public License, siehe auch
`http://www.gnu.org/licenses/lgpl.html`

3.2.2 JDT

Aus den, im Abschnitt 2.1.2 erläuterten Gründen, kam der Compiler *JDT*[2] zum Einsatz. Er ist nun im CVS Repository des DGPF Projekts enthalten und wird beim Kompilieren durch Ant verwendet.
Es gibt aber auch andere Gründe den Eclipse-eigenen Compiler zu verwenden. Besonders bemerkenswert sind die umfangreichen Einstellungen, z.B. die Möglichkeit, die Quelltext-Überpüfung an die persönlichen Bedürfnisse anzupassen. So kann man z.B. entscheiden ob fehlende Javadoc Kommentare bei Public-Deklarationen als Fehler oder Warnnung gemeldet werden, oder gar ignoriert werden sollten.

3.2.3 Tango Icons

Das *Tango Project*[3] versteht sich als Hilfe bei der Erstellung von freier Software mit grafischer Benutzeroberfläche. Die Zielsetzung ist es, Entwicklern von Open Source Projekten mit einheitlichen und freien Icons zu versorgen. Die Icons werden unter der *Creative Commons Attribution Share-Alike Lizenz* veröffentlich. Sie können also ohne Probleme bei DGPF eingesetzt werden. Die verwendeten Icons stammen fast ausschließlich aus dem Tango Projekt. Ausserdem sind noch einige Java Icons enthalten, die auch frei nutzbar sind.

3.3 Metriken

3.3.1 Arbeitszeit

Das Team hat sich im Semester regelmäßig zweimal pro Woche getroffen. Meistens Dienstags und Donnerstags, zwischen 10-14 Uhr. Außerdem wurde weiterhin über die Vorlesungszeit hinaus intensiv gearbeitet, mehrmals pro Woche oder auch am Wochenende, jeweils ca. 4-5 Stunden pro Treff.

3.3.2 Lines of Code

Obwohl *Lines of Code*[7] keineswegs eine objektive Softwaremetrik darstellen, können sie trotzdem einen gewissen Anhaltspunkt über den Umfang eines Softwareprojekts liefern. Für das Projekt haben wir mittels einem kleinen Shellscript

a) die Gesamtanzahl der vorhandenen Programmzeilen und

b) die Gesamtanzahl der vorhandenen Programmzeilen abzüglich der Leer- und Kommentarzeilen

gemessen. Grundlage ware hierbei alle `.java`-Dateien in den von uns entwickelten Packages `org.dgpf.gui`, sowie `org.dgpf.search.control`.
Dabei kamen folgende Ergebnisse heraus:

[2] `http://www.eclipse.org/jdt/` - *Eclipse Java Development Tools*

Gesamtprogrammzeilen	Reine Codezeilen
15707	7135

3.3.3 Packages, Klassen und Interfaces

In diesem Abschnitt soll eine Übersicht gegeben werden, welche

(P) Packages,

(C) Klassen und

(I) Interfaces

während dem Projektverlauf entstanden sind.
Anzahl:

Art	Anzahl
Package	13
Klasse	88
Interface	5
Methode	546

Übersicht:

(P) org.dgpf.gui.actions

- (C) AbortSeAction
- (C) CloseAction
- (C) CloseAllAction
- (C) MinimizeAction
- (C) UnminimizeAction

(P) org.dgpf.gui.charts

- (C) Bar3DChart
- (C) ChartRunner
- (I) IChart
- (C) Pie3DChart
- (C) XYLineChart

(P) org.dgpf.gui.charts.descriptions

- (C) DefaultPortDescriptor
- (C) PortDescriptorBase

- (C) StatAveragePortDescriptor
- (C) StatCountPortDescriptor
- (C) StatMaxPortDescriptor
- (C) StatMedianPortDescriptor
- (C) StatMinPortDescriptor
- (C) StatStddevPortDescriptor
- (C) StatSumPortDescriptor
- (C) StatSumSqrPortDescriptor
- (C) StatVariancePortDescriptor

(P) org.dgpf.gui.components

- (C) BorderedPanel
- (C) Button
- (C) CheckBox
- (C) DGPFFileFilter
- (C) DGPFList
- (C) DGPFTitledBorder
- (C) DirChooserPanel
- (C) FileChooser
- (C) FileChooserPanel
- (C) ImagePanel
- (C) MenuBar
- (C) MessageBox
- (C) RadioButton
- (C) Spinner
- (C) SplitPane
- (C) TextArea
- (C) TextField
- (C) TextPane
- (C) Tree

(P) org.dgpf.gui.controls

- (C) BoundedDoublePropertyControl
- (C) BoundedIntPropertyControl

- (C) BoundedLongPropertyControl
- (C) ControlMap
- (C) DGPFMouseListener
- (C) FactoryMap
- (I) IControlFactory
- (I) IControl
- (C) PropertyControl
- (C) TimePropertyControl

(P) org.dgpf.gui.dialogs

- (C) AboutDialog
- (C) ConnectToHostDialog
- (C) DialogBase
- (C) HostlistEditor
- (C) HostlistTableModel
- (C) SettingsDialog
- (C) StartJarDialog

(P) org.dgpf.gui.tabs

- (C) ChartTab
- (I) ITab
- (C) LogTab
- (C) PropertyTab

(P) org.dgpf.gui.utils

- (C) BrowserLaunch
- (C) Host
- (C) HostListUtils
- (C) LanguageSwitcher
- (C) Layout
- (C) PrefUtils
- (C) TimePattern

(P) org.dgpf.gui.wizards

- (C) WizardBase

(P) org.dgpf.gui.wizards.chartchooser

- (C) CCWzrdFirstCard
- (C) CCWzrd
- (C) CCWzrdSecCardBC
- (C) CCWzrdSecCard
- (C) CCWzrdSecCardPC
- (C) CCWzrdSecCardXYLC

(P) org.dgpf.search.control

- (C) Daemon

(P) org.dgpf.search.control.protocol

- (C) AbortSeEvent
- (C) Commands
- (I) ICommunicationPeer
- (C) JarFileEvent
- (C) NetworkCommunicationPeer
- (C) PropertyUpdateEvent
- (C) ProtCircularBuffer
- (C) ReadingThread
- (C) ReceivingThread
- (C) RunningStatusEvent

3.4 Coding Styles

Damit sich auch spätere Entwickler in dem Projekt zurecht finden, wurde auf die Einhaltung der *Coding Styles* geachtet. Im DGPF Projekt gibt es einheitliche Konventionen, wie der Quelltext strukturiert wird. Neben dem konsequenten Einsatz von *Javadoc* und Vorgaben bezüglich Einrückung und Klammersetzung, gibt es außerdem folgende Richtlinien für Bezeichner:

Art	**Präfix**
Membervariablen	`m_`
Parameter	`p_`
Lokale Variablen	`l_`

3.5 Ausblick

Im Rahmen der Projektarbeit wurden weitere Aufgaben erkannt. Nachfolgend werden ein paar Punkte aufgeführt, wie die Benutzeroberfläche sinnvoll erweitert werden könnte:

- **Automatisierte Erstellung von JAR-Dateien**
  Die Suchjobs in Form von JAR-Dateien könnten durch die GUI automatisch generiert werden. So könnten Grundfunktionalitäten der Jobs bequem über die Oberfläche zusammengestellt werden, ohne dass dafür direkt Java Programmtext geschrieben werden müsste.

- **Mehr Diagrammtypen**
  Relativ leicht ließen sich weitere Diagrammtypen verwirklichen.

- **Sprachunterstützung**

  - Dynamisches Umschalten der Sprache, also ohne ein Neustart des Programms.
  - Es könnte ein Mechanismus implementiert werden, der überprüft, welche `ResourceBundles` verfügbar sind und dann dynamisch eine Liste der vorhandenen Sprachen erzeugt.
  - Übersetzungen in weitere Sprachen.

- **Erstellen von Diagrammen aus Log-Dateien**
  Es wäre denkbar, aus zuvor abgespeicherten Log-Dateien, die Statistiken zu extrahieren und aus ihnen Diagramme zu erstellen.

4 Projektzusammenfassung

Die im Sommersemester 2006 entstandene Projektarbeit erweitert das *Distributed Genetic Programming Framework* um eine grafische Benutzeroberfläche sowie einen Hintergrunddienst. Die Software wurde in der Programmiersprache Java implementiert. Als Entwickleroberfläche kam hauptsächlich *Eclipse* zum Einsatz. Außerdem wurde die Projektverwaltung durch *Ant* automatisiert. Mittels der Benutzeroberfläche können mehrere Hintergrunddienste über das Netzwerk ferngesteuert werden. Der Hintergrunddienst selbst dient als Container für Suchjobs, die `SearchEngines`. Diese können verteilt arbeiten und werden über die Oberfläche gestartet und bei Bedarf beendet. Außerdem können die Jobs bequem überwacht werden. Die Statusinformationen werden übersichtlich auf dem Bildschirm ausgegeben und als Diagramm in Echtzeit visualiert. Die Textausgabe kann in zwei Formaten in Dateien mitgeloggt werden und die Diagramme lassen sich als Grafiken exportieren. Bisher gab es beim DGPF Projekt eine Textausgabe am Bildschirm. Darüber hinaus lassen sich die Suchparameter während der Laufzeit anpassen. Der Programmaufbau ist möglichst modular gehalten, um spätere Erweiterungen zu vereinfachen. So auch bei der Kommunikation zwischen GUI und Hintergrunddienst: Die Netzwerkkomponente ist komplett austauschbar. Auch die Art des `SearchEngines` ist nicht durch das System begrenzt. So können neue `SearchEngines` implementiert werden, ohne dass die GUI oder der Hintergrunddienst angepasst werden muss, obwohl sich die Eigenschaften und Parameter des `SearchEngines` ändern. Die Benutzeroberfläche verfügt über einen Editor, der es ermöglicht, bequem Rechnerlisten zusammenzustellen, zu speichern und zu laden. Das Programm ist bisher auf Deutsch und Englisch übersetzt. Andere Sprachen können hinzugefügt werden, indem *ResourceBundles* erstellt werden. Die Oberfläche besitzt eine interne Fensterverwaltung, bei der jedes Fenster einen verbundenen Hintergrunddienst darstellt. Das Projekt wurde mit Hilfe der Bibliothek *JFreeChart* und dem *Tango Icon Project* umgesetzt.

Aufteilung

Nun soll dargelegt werden, wer welche Aufgaben in der Projektarbeit übernommen hat. Es muss angemerkt werden, dass die folgende Aufstellung bloß grobe Zuständigkeiten beschreibt, denn es war immer so, dass die Aufgaben und Probleme im Team gemeinsam bewältigt wurden, so dass eine klare Abgrenzung oft garnicht möglich ist.

Podlich	**Kumsiashvili**	**Dietrich**
Dialoge	Benutzereinstellungen	Artwork (Icons, Buttons)
Darstellung der statistischen Daten mit Schaltflächen	Dialoge	Dialoge
Grafische Darstellung der statistischen Daten mit Diagrammen	Editor für Rechnerliste (XML Dateiformat)	Editor für Rechnerliste
Interne Fenster	Hintergrunddienst	Hintergrunddienst
Protokollierung	Infopanel	Interner Desktop
Sprachübersetzung	Interner Desktop	Interne Fenster
Wizards (Diagrammwizard)	Mnemonic	Netzwerk
	Netzwerk	Sprachunterstützung
	Sprachübersetzung	Sprachübersetzung

Abbildungsverzeichnis

2.1 Die grafische Benutzeroberfläche. 11
2.2 Dialog für Benutzereinstellungen 13
2.3 Editieren von Rechnerliste. 14
2.4 Auswahl des Diagramm-Typ mit dem Diagramm-Wizard. 15
2.5 Diagrammeinstellungen für ein XY-Liniendiagramm. 16
2.6 Gewinnung der Daten aus der SearchEngine. 22
2.7 Setzen und Abfragen der statistischen Daten. 22
2.8 Der SearchEngine-Zustand. 23
2.9 Zusammenarbeit zwischen Feldern und Properties 24
2.10 `Property`-Gruppen in Form von Tabs. 25
2.11 Schaltfläche für eine `TimeProperty`. 26
2.12 Schaltfläche für eine `Boolean-Property`. 26
2.13 Schaltfläche für eine `BoundedIntProperty`. 27
2.14 `JFreeChart`-Kreisdiagramm. 31
2.15 `JFreeChart`-Säulendiagramm. 32
2.16 `JFreeChart`-XY-Liniendiagramm. 33
2.17 Verbindung zwischen GUIs und Daemons 36
2.18 Die Netzwerkkommunikation über Events 37

Index

Ant, 8, 40

Browser, 17

Compiler
    JDT, 9, 40
    Sun Java Compiler, 9
Control, 25
CVS, 9

Diagramm, 30
    -wizard, 14

Eclipse, 10
Event, 36, 38
    AbortSeEvent, 38
    IEventListener, 21, 36
    IEventSource, 36
    JarFileEvent, 38
    PropertyUpdateEvent, 35, 38
    RunningStatusEvent, 38
    SearchUpdateEvent, 20, 21, 28, 37, 38

Grafische Benutzeroberfläche, 5
GridBagLayout, 18

InternalFrame, 12, 21

JAR-Datei, 34, 38
Java Reflection, 24, 34
Javadoc, 9, 44
JFreeChart, 29, 39

LayoutManager, 12
Lizenz
    Creative Commons, 40
    LGPL, 39
Locale, 20

Mac OS X, 17
Microsoft Windows, 17

NetworkCommunicationPeer, 36, 38

POSIX, 17, 18
Properties, 23
Protokollierung, 32

Rechnerliste, 13, 14, 28
ResourceBundle, 19

Schaltflächen, 25
SearchEngine, 24, 38
SearchState, 20, 21, 38
SourceForge, 29
Suchparameter, 38

Tango Project, 40
TCP/IP, 35
    Netzwerkport, 34

Unix, 17

XML, 14

Literaturverzeichnis

[1] *DGPF - An Adaptable Framework for Distributed Multi-Objective Search Algorithms Applied to the Genetic Programming of Sensor Networks*, 2006.

[2] *Genetic Programming Techniques for Sensor Networks*, number 5, JUL 2006.

[3] freedesktop.org. Tango desktop project. `http://tango.freedesktop.org/`.

[4] Object Refinery Limited. Jfreechart. `http://www.jfree.org/jfreechart/`, 2006. [Online; Stand 29 August 2006 11:01:55].

[5] The Jakarta Project. Ant. `http://ant.apache.org/`.

[6] Thomas Weise. Distributed genetic programming framework. `http://sourceforge.net/projects/dgpf` and `http://dgpf.sourceforge.net/`.

[7] Englische Wikipedia. Source lines of code. `http://en.wikipedia.org/wiki/Source_lines_of_code`.