Endbericht PG 552 - Drug Hunting

Bernhard Dick, Thorsten Flügel, Henning Garus
Michael Hesse, Philipp Kopp, Philipp Lewe
Dominic Sacré, Till Schäfer, Thomas Schmitz

21. November 2011
Technische Universität Dortmund

Dr. Karsten Klein, Dr. Carsten Gutwenger, Nils Kriege, Prof. Dr. Petra Mutzel
Lehrstuhl XI - Algorithm Engineering
Fakultät für Informatik
Technische Universität Dortmund

Inhaltsverzeichnis

1. Einleitung 5
1.1. Projektgruppen in Dortmund . . . 5
1.2. Aufgabenstellung . . . 5
1.3. Veranstalter und Teilnehmer . . . 6

2. Organisation 7
2.1. Zeitlicher Ablauf . . . 7
2.2. Technische Organisationsmittel . . . 9

3. Seminarphase 10
3.1. Global mapping of pharmacological space . . . 10
3.2. A global view of drug-therapy interactions / How to visually interpret biological data using networks . . . 11
3.3. Scaffoldbasierte Navigation im Chemical Space . . . 11
3.4. Predicting new molecular targets for known drugs / Interaction networks of chemicals and proteins . . . 12
3.4.1. SEA: Predicting new molecular targets for known drugs . . . 12
3.4.2. STITCH: Interaction networks of chemicals and proteins . . . 12
3.5. ChemSpaceShuttle / iPHACE / SARANEA . . . 13
3.5.1. ChemSpaceShuttle . . . 13
3.5.2. iPHACE . . . 13
3.5.3. SARANEA . . . 14
3.6. Chem2Bio2RDF . . . 14
3.7. Target, chemical and bioactivity databases – integration is key / Advanced Biological and Chemical Discovery . . . 16
3.8. Current Trends in Ligand-Based Virtual Screening: Molecular Representations, Data Mining Methods, New Application Areas, and Performance Evaluation . . . 17
3.9. Clustering Methods and Their Uses in Computational Chemistry . . . 17
3.10. Fragment-basierte Methoden zur Klassifikation von Molekülen . . . 18

4. Der Scaffoldbaum und der bisherige Scaffold Hunter 19
4.1. Der Scaffoldbaum . . . 19
4.2. Scaffold Hunter . . . 20
4.2.1. Baumerzeugung . . . 21
4.2.2. Visualisierung des Baumes . . . 21

5. Konzept **23**
5.1. Datenverwaltung . . . 23
5.1.1. Datenimport . . . 23
5.1.2. Datenberechnung . . . 27
5.2. Statistische Analyseverfahren . . . 27
5.3. Allgemeine Designentscheidungen . . . 27

6. Implementierung **30**
6.1. Allgemeine Design- und Strukturentscheidungen . . . 30
6.1.1. Package-Struktur / Modularität / Interfaces . . . 30
6.1.2. Designguide . . . 32
6.2. Lizenz . . . 33
6.3. Verwandte Bibliotheken . . . 33
6.3.1. Hibernate . . . 33
6.3.2. Piccolo2D . . . 34
6.3.3. Batik . . . 36
6.3.4. L2FProd . . . 36
6.3.5. JGoodies . . . 37
6.3.6. JDOM . . . 38
6.3.7. Simple Logging Facade for Java . . . 38
6.3.8. Clustering . . . 39
6.3.9. Guava: Google Core Libraries for Java . . . 40
6.3.10. CDK . . . 41
6.3.11. rangeslider . . . 52
6.3.12. Drag & Drop / Closeable Tabbed Pane . . . 52
6.3.13. JSPF . . . 53
6.3.14. exp4j . . . 54
6.3.15. Open CSV . . . 54
6.3.16. XStream . . . 55
6.4. Interne Datenhaltung . . . 56
6.4.1. Modellentscheidungen . . . 56
6.4.2. Umsetzung in Hibernate . . . 58
6.4.3. Performance Optimierungen . . . 59
6.4.4. Zentrales Handling von Datenbank Exceptions . . . 60
6.5. Vorstellung der Programmoberfläche . . . 62
6.5.1. Hauptfenster . . . 62
6.5.2. Startdialog . . . 63
6.5.3. Optionsdialog . . . 64
6.6. Ansichten . . . 65
6.6.1. Scaffoldbaumansicht . . . 65
6.6.2. Dendrogrammansicht . . . 70
6.6.3. Tabellenansicht . . . 77
6.6.4. Plottansicht . . . 78

6.7. Clustering 78
6.7.1. Hierachisches Clustering 79
6.7.2. NN-Chain Algorithmus 79
6.7.3. Linkage und Distanzmatrix 79
6.8. Lokalisierungskonzept 80
6.9. Optionsframework 80
6.9.1. Konfigurationen aus Sicht der Ansichten 80
6.9.2. Konfigurationen aus Sicht der Editoren 83
6.10. Progress-Worker 86
6.10.1. SwingWorker 87
6.10.2. ProgressWorker 88
6.10.3. ProgressListener 89
6.10.4. WorkerExceptionListener 89
6.10.5. Ausführung mit Fortschrittsdialog 90
6.10.6. Beispiel 90

7. Zukunftsausblick und mögliche Erweiterungen 94

A. Literaturverzeichnis 96

1. Einleitung

1.1. Projektgruppen in Dortmund

Seit 1972 existieren Projektgruppen im Bereich der Informatik an der TU Dortmund. Im Rahmen einer PG soll eine Gruppe aus in der Regel 8 bis 12 Studenten möglichst eigenständig ein Projekt über 2 Semester hinweg durchführen. Diese Projekte gliedern sich meist in aktuelle Forschungsthemen in der Informatik ein und haben zum Ziel, dass die Studenten Erfahrung mit der Durchführung solcher Projekte erlangen. Die Projektgruppen sind, gerade da sie sich von üblichen Lehrveranstaltungen abheben, eine Besonderheit der TU Dortmund.

1.2. Aufgabenstellung

Die Aufgabe der PG552 - Drug Hunting besteht in der Entwicklung bzw. Weiterentwicklung einer Softwarelösung zur Navigation im pharmakologischen Strukturraum. Als Beispiel für eine solche Softwarelösung wurde u.a. das Tool Scaffold Hunter, welche bereits von der Dortmunder PG504 (ChemBioSpace Explorer) entwickelt wurde genannt. Diese ist daraufhin von der PG552 als Entwicklungsbasis aufgegriffen worden.
Die Minimalziele der PG Aufgabenstellung sind:

- Entwurf und Implementierung eines Frameworks zur Integration von zusätzlichen Informationen, das mindestens zwei externe sowie eine lokale Datenquellen umfasst und die beispielhafte Integration in Scaffold Hunter.
- Verknüpfung der Informationen mit den Strukturen im Programm, angemessene Aufbereitung und Darstellung.
- Implementierung weiterer Konzepte der Klassifikation und Navigation im chemischen Strukturraum sowie ihre Visualisierung.
  - Erweiterung des bestehenden Konzepts statischer Strukturgerüstbäume.
  - Anwendung statistischer Methoden zur Klassifikation des Strukturraums.

1.3. Veranstalter und Teilnehmer

Die Projektgruppe wird durch Karsten Klein, Dr. Carsten Gutwenger, Nils Kriege und Prof. Dr. Petra Mutzel des Lehrstuhl XI - Algorithm Engineering der Fakultät für Informatik angeboten und betreut.

Die Teilnehmer der Projektgruppe sind Bernhard Dick, Thorsten Flügel, Henning Garus, Michael Hesse, Philipp Kopp, Philipp Lewe, Dominic Sacré, Till Schäfer, Thomas Schmitz und Ömer Uzun (bis März 2011).

2. Organisation

2.1. Zeitlicher Ablauf

12.10-21.10.2010 Seminarphase und Abstimmung der technischen Organisationsmittel

21.10-28.10.2010 Einarbeitung in die verfügbaren öffentlichen Datenbanken

02.11.2010 Nach dem Vortrag des Chemikers Stefan Wetzel haben wir eine erste grobe Sammlung von Features zusammengestellt, welche wir implementieren wollen.

11.11.2010 Erste Überlegungen zur technischen Umsetzung wurden gemeinsam angestellt. Die dabei aufgetretenen Fragen zur Integration verschiedener Datenbanken, den Darstellungs- und Filtermöglichkeiten und des Clusterings wurden anschließend in vier Kleingruppen geklärt.

18.11.2010 - 22.11.2010 Festlegung unserer genauen Ziele und Entwurf eines Konzeptes. Im Konzept haben wir alle geplanten Funktionen und Grundlagen zusammengefasst, sowie den Umfang des Prototyp I festgelegt.

23.11.2010 Präsentation der Ziele gegenüber den Betreuern und anschließendes Gespräch über noch zu klärende Details und weiteres Vorgehen.

02./06.12.2010 Konzeptpräsentation in Anwesenheit von Chemikern das Max-Planck-Instituts und anschließendes Überarbeiten des Konzepts mit den neu gewonnenen Informationen.

07./08.12.2010 Festlegung des Datenbanklayouts und möglicher Mappings in der Scaffoldbaumansicht.

09.12.2010 Auseinandersetzung mit der bisherigen Codestruktur und Analyse notwendiger Änderungen. Einteilung in Zweiergruppen um Vorbereitungen für die Implementation zu leisten.

16.12.2010 Erarbeitung der Schnittstellen zwischen den einzelnen Gruppen und Beginn der Implementation. Festlegung der weiteren Zeitplanung für das erste Semester.

Januar 2011 Implementation mit zwischenzeitlichen Bestandsaufnahmen und Diskussionen zu den aufgetretenen Problemen.

27.01.2011 Letztes Treffen vor Fertigstellung des Prototyp I.

03.02.2011 Präsentation des Prototyp I in Anwesenheit von Prof. Dr. Petra Mutzel und den Betreuern.

Februar/März 2011 Vorlesungsfreie Zeit / Schreiben des Zwischenberichts.

Anfang April 2011 Planungstreffen für den Ablauf des zweiten Semesters, Zeitpunkt und Umfang des Prototyp II werden festgelegt

April 2011 Arbeiten am Prototyp II: Integration des Clusterings, erste JUnits Tests, DB Nutzung durch die Ansichten und erste Version der Datenintegrations-Dialoge

06.05.2011 Präsentation des Prototyp II in Anwesenheit der Mitarbeiter des MPI. Rückmeldungen der Chemiker decken sich größtenteils mit den Erwartungen.

Mai 2011 Planung und Implementierung einer Beta-Version. Festlegen von Designrichtlinien für die Programmoberfläche. Arbeiten am Import, Export und Calc-Plugin Schnittstellen.

Ende Mai 2011 Erstellen eines Branches für die Betaversion. Diese Version enthält nur (fast) vollständige Features, die nicht funktionierenden Programmpunkte sind auskommentiert oder mit Warnungen versehen.

07.06.2011 Präsentation der Beta am MPI. Direktes Feedback liefert viele nützliche Informationen. Einige Anregungen sind nicht mehr in der PG realisierbar, da Zeit fehlt und / oder die Struktur des Programms zu stark verändert werden müsste, z.B: Verknüpfungen der Moleküle zu ihren Scaffolds in der Tabelle.

Anfang Juni 2011 Planung für die noch verbleibende Zeit. CDK-Bibliothek ist sehr langsam und wird auf mögliche Verbesserungen untersucht.

Juni / Juli 2011 Überarbeitung einiger (CDK-) Algorithmen. Integration diverser Import-, Export-, und Calc-Plugins. Implementation eines Sessionmanagements.

15.07.2011 Abschluss der Implementation. Aufteilung der Aufgaben für den Endbericht und das Handbuch.

31.09.2011 Abgabe des Endberichts

13.10.2011 Präsentation des Programms

2.2. Technische Organisationsmittel

Zur Organisation der Projektgruppe und Implementation des Programmes haben wir uns für mehrere technische Hilfsmittel entschieden. In diesem Abschnitt sollen sie kurz vorgestellt werden.

Mailingliste Zur kurzfristigen Kommunikation und für Ankündigungen haben wir uns nach einer Abstimmung für die Nutzung einer Mailingliste entschieden. Sowohl für die Liste, als auch für die Alternative eines Forums, gab es Vor- und Nachteile. Das entscheidende Argument für die Mailingliste war, dass Foren auf neue Meldungen kontrolliert werden müssen, während eine Mailingliste die Informationen direkt liefert.

Redmine Auf einem vom Lehrstuhl XI betriebenen Server haben wir eine Instanz des Projektmanagement-Tools Redmine eingerichtet. Über diese Webplattform können wir unter anderem Troublemanagement mittels eines Ticketsystems betreiben und ein Wiki pflegen. Das Wiki nutzen wir als Informationsarchiv für Planungsergebnisse, Sitzungsprotokolle und langfristig wichtige Kommunikation, für welche die Mailingliste weniger geeignet ist. Zudem kann über das Redmine der Status des Versionsverwaltungssystem für den Programmcode eingesehen werden und für die Zeitplanung können Meilensteine und Versionen definiert werden. Diesen können dann Troubletickets und Featureanfragen zugeordnet werden.

Eclipse Aus der Aufgabenstellung ging hervor, dass wir in Java programmieren werden. Bei der Programmierumgebung haben wir uns daraufhin auf Eclipse festgelegt, da die Teilnehmer mehr Erfahrung mit Eclipse, als mit der Alternative NetBeans haben. Zudem machte Eclipse einen deutlich professionelleren Eindruck und ist durch eine Vielzahl von Plugins modular erweiterbar. Es lässt sich somit besser den Anforderungen anpassen.

SVN Der Austausch des Programmcodes geschieht mittels Apache Subversion, das zugehörige Repository liegt auf dem selben Server, auf dem Redmine gehostet wird.

3. Seminarphase

Wie bereits in Kapitel 2.1 vorgestellt, fand zu Beginn der Projektgruppe eine Seminarphase statt. Diese Phase sollte einen Einstieg in das interdisziplinäre Feld der Medikamentenforschung zwischen den Fachbereichen Biologie, Chemie und Informatik bieten. Die einzelnen Themen wurden von den Betreuern vorgegeben und sollten von den Projektgruppenteilnehmern aus der Literatur erarbeitet und in Form eines 30-40 minütigen Vortrages vorgestellt werden. Anschließend bestand die Möglichkeit offene Fragen zu den vorgestellten Themen im Rahmen einer Diskussion zu beantworten. Dieser Abschnitt soll eine Übersicht der behandelten Themen geben und den Inhalt der Vorträge in Form einer kurzen Zusammenfassung vorstellen.

3.1. Global mapping of pharmacological space

Der Artikel *„Global mapping of pharmacological space“* [20] von Gaia V. Paolini, Richard H. B. Shapland, Willem P. van Hoorn, Jonathan S. Mason und Andrew L. Hopkins behandelt zwei Methoden zur Untersuchung von chemisch aktiven Struktur Verbindungs Daten (SAR) in der Medizinforschung. Dabei geht es darum, bei bereits existierenden Medikamenten Muster zu finden, welche zu Ansätzen für neue Forschungen führen sollen. Zuerst werden Probleme aufgezeigt, wie etwa die nicht vorhandenen Verbindungen zwischen existierenden Datenbanken, oder die nur kommerzielle Verfügbarkeit der Daten, welche die Forschung auf diesem Gebiet sehr erschweren.

Die erste vorgestellte Methode stammt aus dem Bereich der Polypharmakologie. Hier versucht man Stoffe zu finden, die ähnliche Bindungen eingehen. Proteine werden als ähnlich klassifiziert, wenn sie sich an möglichst viele identische Targets binden können. Dabei ist keine strukturelle Ähnlichkeit notwendig. Diese Methode wurde bereits erfolgreich in der Forschung eingesetzt.

Bei der anderen vorgestellten Methode wird versucht mithilfe von bayesschen Wahrscheinlichkeitsberechnungen die biologische Aktivität eines Moleküls anhand seiner individuellen Daten vorauszusagen. Bei dieser Methode reichen einzelne Eigenschaften nicht aus, Kombinationen aus mehreren Eigenschaften liefern aber nutzbare Ergebnisse. Abschließend wird gesagt, dass die beobachteten Ergebnisse vielversprechend sind, aber in klinischen Studien verifiziert werden müssen.

Nützlich für die Arbeit in der Projektgruppe ist vor allem, dass Clustering/Klassifikation anhand von wenigen Attributen Zusammenhänge zwischen Molekülen aufzeigen kann. Die polypharmakologische Methode ist nicht anwendbar, da hier (fast) vollständige bio-aktivitäts Werte vorhanden sein müssten um brauchbare Ergebnisse zu liefern.

3.2. A global view of drug-therapy interactions / How to visually interpret biological data using networks

Das Paper *„How to visually interpret biological data using networks“*[16] stellt einige grundlegende und häufig verwendete Modelle vor, um biologische Daten mit Hilfe von Netzwerken zu visualisieren und analysieren.
Viele Arten von Daten lassen sich gut in Form von Netzwerken darstellen. Beispielhaft werden in diesem Paper unter anderem Ähnlichkeitsbeziehungen von Proteinen genannt, sowie Interaktionen zwischen Proteinen oder Genen. Am Beispiel von Proteinen der Backhefe (*Saccharomyces cerevisiae*) wird gezeigt, wie deren Wechselbeziehungen als Netzwerk dargestellt werden können, wie sich Daten auf verschiedene graphische Eigenschaften abbilden lassen, und mit welchen Verfahren der daraus entstehende Graph optimiert werden kann. Außerdem werden verschiedene äquivalente Darstellungen als Listen, Netzwerke und Heatmaps verglichen, die sich je nach Art der Daten unterschiedlich gut zu deren Visualisierung eignen.

Die Autoren des Papers *„A global view of drug-therapy interactions“* [18] hatten als Zielsetzung, die Beziehungen zwischen medizinischen Behandlungen und den dabei eingesetzten Medikamenten zu visualisieren. Basierend auf der Datenbank DrugBank wurde ein bipartiter Graph aufgebaut, der jeweils Medikamente mit den dadurch beeinflussten anatomischen Systemen verknüpft. Dieser Graph lässt sich dann in zwei verschiedene Abbildungen zerlegen: Die Medikament-Abbildung enthält nur Medikamente als Knoten, die jeweils verbunden sind, wenn sie in der selben Behandlung eingesetzt werden können. Umgekehrt enthält die Behandlungs-Abbildung nur zu behandelnde Systeme als Knoten, welche verbunden sind, wenn es Medikamente gibt, die in beiden Behandlungen vorkommen.
Anhand dieser beiden Graphen wird im Paper anschließend versucht, Medikamente zu finden, die eine sehr zentrale Rolle spielen. Darauf aufbauend werden Metriken definiert, mit denen sich abschätzen lässt, wie gut sich ein Stoff zur Entwicklung neuer Behandlungsmethoden für komplexe Krankheiten eignet.

3.3. Scaffoldbasierte Navigation im Chemical Space

Es gibt verschiedene Möglichkeiten die Gesamtheit der natürlichen und synthetisierbaren Moleküle zu strukturieren. Neben der Einteilung in Stoffgruppen oder nach medizinischer Wirkung kann auch der atomare Aufbau der Moleküle als Ordnungskriterium herangezogen werden. Letztgenannter Ansatz wird durch Scaffolds verfolgt. Ein Scaffold stellt dabei gewissermaßen das Gerippe eines Moleküls dar. Da sich mehrere Moleküle auf das gleiche Scaffold reduzieren lassen, und mehrere Scaffolds ihrerseits durch weitere Reduktion auf kleinere Scaffolds abgebildet werden können, ergibt sich eine hierarchische Struktur der Molekülmenge.

Zwei Programme die diesen Ansatz verfolgen sind der *Scaffold Hunter* und der *Scaffold Explorer*. Beide erlauben es anhand der Strukturierung mittels Scaffolds durch eine

Molekülmenge zu navigieren und Moleküle mit struktureller Ähnlichkeit aufzuspüren. Weiterhin können beide Programme weiterführende Informationen zu Scaffolds und Molekülen anzeigen. Der Scaffold Explorer ermöglicht es darüberhinaus verschiedene Daten miteinander zu verknüpfen (z.B. die Verknüpfung von Molekülen mit SAR-Tabellen), um die Gewinnung weiterer Erkenntnisse zu unterstützen. Ein weiterer Unterschied zwischen beiden Programmen besteht in der Generierung der baumartigen Anordnung der Scaffolds. Während der Scaffold Hunter vorab eine Erzeugung des Gesamtbaumes verlangt, wird der Baum (oder – bei gleichzeitiger Verwendung unterschiedlicher Regeln für die Baumerzeugung – das Netz) beim Scaffold Explorer sukzessive während des Arbeitens mit dem Programm erstellt.

3.4. Predicting new molecular targets for known drugs / Interaction networks of chemicals and proteins

Bei diesem Vortrag ging es um zwei Verfahren, mit denen Interaktionen zwischen Chemikalien und Proteinen vorhergesagt beziehungsweise erkundet werden können.

3.4.1. SEA: Predicting new molecular targets for known drugs

SEA (Similarity Ensemble Approach) [11] ist ein Verfahren zur Vorhersage von neuen Targets bereits bekannter medizinischer Wirkstoffe. Hierbei wird der Wirkstoff strukturell mit den Liganden der Zielproteine, also Molekülen, die mit diesem Ziel interagieren, verglichen und so eine mögliche Interaktion zwischen ihm und dem neuen Ziel erkannt. Für den Vergleich mit den Liganden werden den Molekülen, abhängig von den darin auftretenden Atombindungen, Fingerprints in Form von Bitvektoren zugewiesen und die Ähnlichkeit anhand der gesetzten Bits beider Vektoren berechnet. Sind nun viele der Liganden ähnlich zu dem Wirkstoff, wird davon ausgegangen, dass auch dieser mit dem Ziel interagiert.

Die Vorhersagequalität dieses Ansatzes wurde von Keiser et al. in mehreren Versuchen getestet [12]. Dabei wurde zunächst in einem retrospektiven Test überprüft, ob sich mit diesem Verfahren bereits bekannte Interaktionen aus einer Datenbasis, die keine Information über diese erneut zu findenden Ziele enthält, erhalten lassen. Anschließend wurden einige der vorhergesagten neuen Ziele, die in keiner Datenbank enthalten waren, in Experimenten auf ihre Richtigkeit überprüft. Unter den durch die Experimente bestätigten neuen Zielen waren sowohl solche, die die eigentliche Wirkung bekannter Wirkstoffe erklären, als auch mögliche Erklärungen für Nebenwirkungen von Medikamenten.

3.4.2. STITCH: Interaction networks of chemicals and proteins

STITCH (Search tool for interactions of chemicals) [14, 13] ist ein Werkzeug zur Erkundung des Raumes der pharmakologisch relevanten Moleküle. Es werden mehrere Datenquellen in einer Datenbank zusammengefasst, wobei den einzelnen Quellen bestimmte Rollen zufallen. So gibt es je nach Typ der beiden interagierenden Moleküle, z.B. Metabolit mit Protein, spezialisierte Datenbanken, die als Quelle benutzt werden. Weiterhin

werden Informationen über die Moleküle und Interaktionen per Text Mining aus medizinischen Veröffentlichungen extrahiert. Für Chemikalien wird zusätzlich die strukturelle Ähnlichkeit zur Vorhersage von Interaktionen verwendet.

An das Werkzeug können über ein Webinterface Suchanfragen gestellt werden. Die Anfrageergebnisse werden in diesem Webinterface als interaktiver Graph dargestellt, in dem unterschiedlich geformte und gefärbte Knoten Chemikalien und Proteine repräsentieren. Interaktionen zwischen diesen werden als Kanten dargestellt, wobei die Kantenfarbe den genauen Typen der Interaktion, z.B. Aktivierung oder Inhibition, und die Kantendicke die aus den Quellen erhaltene Konfidenz dieser Interaktion angeben kann. Der Benutzer kann unter anderem über die Knoten in diesem Graphen navigieren und die Knoten verschieben, um so die Übersichtlichkeit der Darstellung zu erhöhen.

3.5. ChemSpaceShuttle / iPHACE / SARANEA

ChemSpaceShuttle, iPHACE und SARANEA sind drei Applikationen, um Bioaktivitäten und Ähnlichkeiten von Molekülen darzustellen.

3.5.1. ChemSpaceShuttle

ChemSpaceShuttle [9] wurde vom modlab der Goethe Universität Frankfurt am Main entwickelt um Ähnlichkeiten zwischen chemischen Molekülen zu finden. Dazu werden die Moleküle als Punkte im dreidimensionalen Raum dargestellt. Hauptaugenmerk liegt hierbei darauf, möglichst alle Eigenschaften der Moleküle, welche als Tabelle bzw. Matrix vorliegen, gemeinsam so auf den dreidimensionalen Raum zu projizieren, dass ähnliche Moleküle nah bei einander liegen.
Als erste Methode wird der bei der PCA übliche NIPALS Algorithmus verwendet, welcher Koordinaten mehrdimensionaler Matrizen auf welche mit geringerer Dimension projiziert und dabei ursprüngliche Ähnlichkeiten (vergleichbar mit Abständen) zwischen den einzelnen Werten erhält. Eine weitere Methode ist die Projektion mit Hilfe von Encodernetzen. Hierbei werden die Eigenschaften durch mehrere Ebenen einzelner Neuronen geleitet, welche jeder Eigenschaft eine Gewichtung zuweisen, bis am Ende nur noch drei Dimensionen vorhanden sind. Die Gewichtung wird evolutionär nach einer $(1, \lambda)$ Strategie aufgebaut.
Ferner kann ChemSpaceShuttle noch clustern auf der Basis von Self Organizing Maps.

3.5.2. iPHACE

iPHACE [7] ist eine Web-Applikation, welche die Bioaktivität verschiedener Moleküle auf verschiedene Targets mit Hilfe von Heatmaps darstellt. Die Daten im Hintergrund sind fix, da nur eine online verfügbare Version existiert. iPHACE kann auf Basis von Aktivitätsstärken ähnlich stark wirkende Moleküle finden. Ansonsten bietet es hauptsächlich die Möglichkeit, bestimmte Moleküle auf Basis von Aktivitätsgrenzen zu finden. In der Abbildung 3.1 ist iPHACE nach Auswahl von einem Satz von Molekülen zu sehen. Links sind Gruppen von Molekülen auswählbar. Hier zeigt der linke Balken die

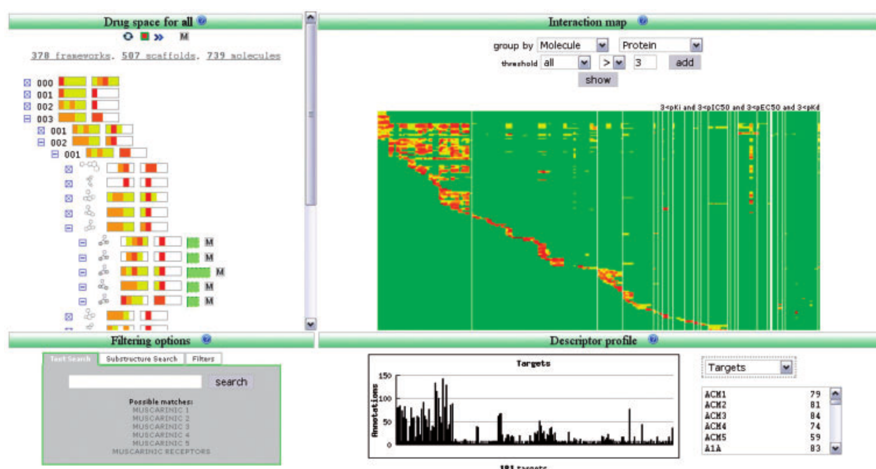


Abbildung 3.1.: Beispielansicht von iPHACE [7]

Bindungsaffinität, der rechte die Effektivität der unter dem Punkt liegenden Molekülen an. Im rechten Teil sieht man das Dendrogramm, die Moleküle in Zeilen und die Zielmoleküle in den Reihen. Im unteren Teil hat man zunächst die Möglichkeit verschiedene Suchfilter zu definieren. Unten rechts kann man in einer Art Histogramm Verteilungen von Eigenschaften über die einzelnen Zielmoleküle anzeigen lassen.

3.5.3. SARANEA

SARANEA [15] ist an der Friedrich-Wilhelms-Universität in Bonn entwickelt worden. Diese Tool soll, ähnlich ChemSpaceShuttle, viele Eigenschaften von Molekülen auf einmal optisch darstellen. Eigenschaften werden hier jedoch nicht auf ein 3D-Universum projiziert, sondern als Graph auf einer 2D Ebene dargestellt. Moleküle werden miteinander verbunden, sobald sie eine gewisse Ähnlichkeit haben. Moleküle die viele Ähnlichkeitsverbindungen untereinander haben, werden als Gruppen beieinander angeordnet.
Weitere Eigenschaften werden auf die Größe der Moleküle sowie ihre Farbe gemappt. Die Beispielansicht in Abbildung 3.2 zeigt links den Graphen, rechts oben ist dieser noch in einer Übersichtskarte zu sehen. Im unteren Teil der Abbildung ist eine Tabelle welche die verschiedenen Eigenschaften der im Graphen ausgewählten Moleküle anzeigt. Rechts kann man Moleküle anhand von Filtern auswählen um einen neuen Graphen zu erstellen.

3.6. Chem2Bio2RDF

Das Projekt Chem2Bio2RDF [4] integriert über 20 chemische, biologische und pharmakologische Datenbanken in einem RDF Repository. Mit Hilfe der Querysprache SPARQL können dann Anfragen an diese Datenbasis formuliert werden.

Beim Datenimport werden die in den verschiedenen Datenbanken verwendeten IDs auf

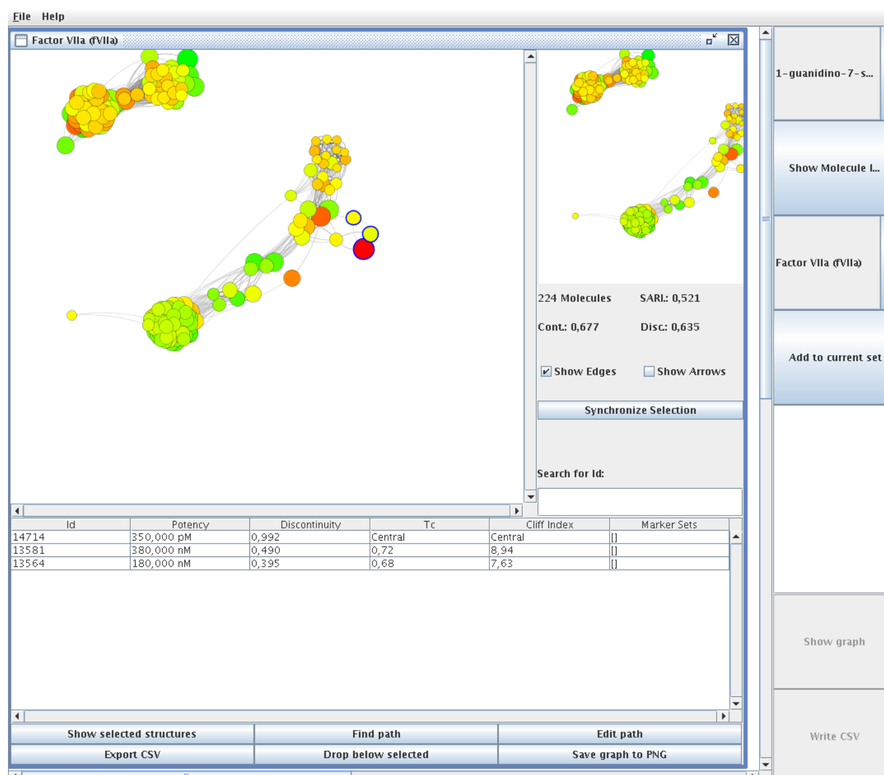


Abbildung 3.2.: Beispielansicht von SARANEA

eine gemeinsame ID für jeden Eintragstyp vereinheitlicht, etwa der *PubChem Compound ID* für chemische Verbindungen. Durch die Verwendung mehrerer Datenbanken ergeben sich auf diese Weise Verknüpfungen zwischen bisher unverknüpften Daten. So lassen sich zum Beispiel Nebenwirkungen von Medikamenten mit biologischen Pfaden in Beziehung setzen.

Neben der Datenintegration wurden im Rahmen des Projektes auch mehrere Tools zur Erleichterung des Umgangs mit dem Repository entwickelt. Darunter befinden sich ein Plugin für die Netzwerkvisualisierungsplattform Cytoscape, welches eine graphische Darstellung der Daten ermöglicht, sowie der *Link Path Generator*, welcher automatisch Verknüpfungen zwischen verschiedenen Datenbanken, sowie die damit assoziierten SPARQL Anfragen generiert.

Durch mehrere detaillierte Fallbeispiele geben die Autoren Einblicke in pharmakologischen Fragestellungen bei denen eine derartig verknüpfte Datenbasis erfolgreich als Hilfsmittel eingesetzt werden kann.

3.7. Target, chemical and bioactivity databases – integration is key / Advanced Biological and Chemical Discovery

In diesem Vortrag über die Paper „Target, chemical and bioactivity databases – integration is key“ [19] und „Advanced Biological and Chemical Discovery (ABCD)“ [3] wurden zwei unterschiedliche Methoden zur Integration mehrerer Datenquellen in eine Datenbank vorgestellt.

Als erstes wurde dabei die sogenannte *Data Federation* behandelt, bei der die Daten in ihren ursprünglichen Datenbanken gespeichert bleiben und nur für direkte Anfragen in ein gemeinsames Format umgewandelt und zusammengeführt werden. Dieses Verfahren hat die Vorteile, dass Anfragen immer mit den aktuellen Daten beantwortet werden können und außerdem keine Kopie der Daten erstellt werden muss und somit Speicherplatz gespart werden kann. Die Nachteile sind allerdings, dass die Daten bei jeder Anfrage abgefragt und umgewandelt werden müssen und so hohe Serverlasten und Antwortzeiten in Kauf genommen werden müssen.

Bei dem zweiten Verfahren, dem sogenannten *Data Warehouse*, werden die Ursprungsdaten in eine zentrale Datenbank kopiert und dabei in das gewünschte Format umgewandelt. Dadurch können Anfragen an die Datenbank direkt und ohne besondere Kenntnis der Datenquellen beantwortet werden. Der Datenbestand selbst wird in zyklischen Abständen aktualisiert. Da bei diesem Verfahren ein normales Datenbanksystem eingesetzt werden kann, können gute Antwortzeiten für Abfragen erreicht werden. Außerdem fallen beim *Data Warehouse* im Gegensatz zur *Data Federation* keine Daten aus, wenn eine der Quelldatenbanken ausfällt oder ihre Struktur verändert wird. Nachteile sind, dass durch die Kopien sämtlicher Ursprungsdaten der Speicherplatzverbrauch sehr groß werden kann und neue oder veränderte Daten einer Ursprungsdatenbank erst nach einer Aktualisierung des Datenbestandes im *Data Warehouse* verfügbar sind.

Eine mögliche Umsetzung des Data Warehouses wurde am Beispiel des *Advanced Biological and Chemical Discovery* gezeigt und erläutert.

3.8. Current Trends in Ligand-Based Virtual Screening: Molecular Representations, Data Mining Methods, New Application Areas, and Performance Evaluation

In dem Vortrag zum Paper „Current Trends in Ligand-Based Virtual Screening: Molecular Representations, Data Mining Methods, New Application Areas, and Performance Evaluation“ [8] von Hanna Geppert, Martin Vogt und Jürgen Bajorath wurden Methoden für das in der Chemoinformatik an Bedeutung gewinnende Thema Data-Mining betrachtet. Das Problem, das ständig ansteigende Datenvolumen von Informationen zu klassifizieren und die Merkmalsunterschiede daraus zu extrahieren, beschäftigte die Forschung enorm. Dabei wurden schon bekannte Verfahren und unter spezifischen Aspekten erweiterte Verfahren betrachtet. Bei der vorhandenen Vielfalt an Methoden haben sich

- Support Vector Machines (SVM)
- Bayessche Methoden
- Entscheidungsbäume

mit einigen Variationen durchgesetzt und werden von der Mehrheit bevorzugt. Auch wurden Molekül-erfassende Bitvektor-Verfahren wie und Structure-Key Fingerprints und Hash-Key Fingerprints, die zur schnelleren Substruktursuche gut geeignet sind, den Teilnehmern vorgestellt.

3.9. Clustering Methods and Their Uses in Computational Chemistry

Der Vortrag zu Kapitel 1 des Buches [6] befasste sich mit dem Thema Clustering in der Chemoinformatik. Dabei lag der Schwerpunkt bei der Vorstellung bewährter Clustering Methoden und einer Analyse dieser Methoden bezogen auf typische Daten, die in der Chemoinformatik anfallen. Zudem wurden die verschiedenen Methoden klassifiziert und eine allgemeine Terminologie (z.B. überwachtes/unüberwachtes Lernen) eingeführt. Vorgestellt wurden die folgenden Methoden als Repräsentanten einer Klasse von Clustering Methoden:

- k-Means (Zentroid basiertes Clustern)
- Jarvis-Patrick (Nearest Neighbour)
- SAHN (Hierarchisches Clustern)
- EM Algorithmus (Mixture Model)

Ziel des Vortrags war es den Teilnehmern einen Überblick über verschiedene Clusteringansätze zu geben. Somit soll den Teilnehmern ermöglicht werden, eine qualifizierte Wahl von Clustermethoden in unserem Projekt vornehmen zu können.

3.10. Fragment-basierte Methoden zur Klassifikation von Molekülen

Dieser Vortrag sollte den Projektgruppenteilnehmern einen Einblick verschaffen, wie sich aus bereits bekannten Molekülen Erkenntnisse über noch unbekannte Moleküle gewinnen lassen. Zum Einstieg wurde der Ablauf der statistischen Datenanalysemethode „Klassifikation“ vorgestellt und die Lernaufgabe „Klassifikation“ mathematisch definiert. Im Anschluss daran wurden verschiedene Klassen von chemischen Merkmalen (Deskriptoren) und ihr jeweiliger Einfluss auf die Güte der Klassifikation präsentiert.
Aufbauend auf diesem Basiswissen, konnten dann zwei recht ähnliche Verfahren (vgl. [5], [22]) zur Berechnung von Deskriptoren vorgestellt werden. Beide Verfahren basieren auf der Idee, die 2D/3D-Struktur aller Moleküle einer gegebenen Datenbasis in kleine Teilstrukturen zu zerschneiden und diese zu sammeln. Für jedes Molekül wird dann ein Merkmalsvektor erzeugt, indem die Anwesenheit/Abwesenheit von „wichtigen“ Teilstrukturen aus der Sammlung innerhalb des Moleküls binär kodiert wird.
Es wurden exemplarisch die Herausforderungen bezüglich Komplexität der Berechnung bzw. Genauigkeit der Deskriptoren und geeignete Lösungsansätze vorgestellt.

4. Der Scaffoldbaum und der bisherige Scaffold Hunter

Die Menge von Molekülen, die für pharmakologische Zwecke in Betracht kommen, ist gigantisch groß – man schätzt sie auf 10^{60} Moleküle. Eine Möglichkeit diese unüberschaubare Menge zu strukturieren besteht darin, sie anhand ihres atomaren Aufbaus anzuordnen. Diese Idee wird durch das Prinzip der Scaffoldbildung verfolgt.

Moleküle werden dabei nach definierten Regeln auf ein Grundgerüst, das Scaffold, reduziert (Abbildung 4.1).

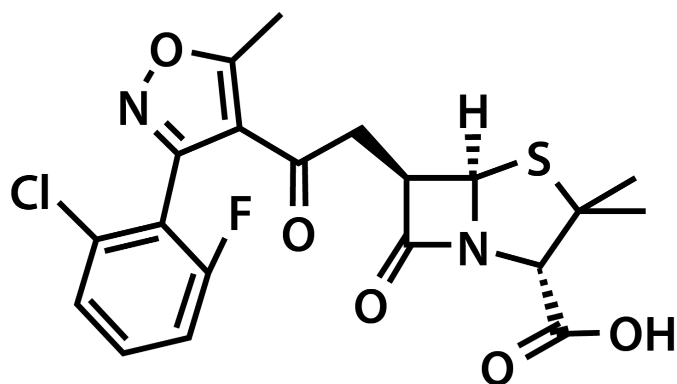
4.1. Der Scaffoldbaum

Die Reduktionsregeln geben vor, welche Teile des Moleküls entfernt werden müssen, um sein zugeordnetes Scaffold zu erhalten. Durch sukzessive Anwendung der Regeln kann dieses Scaffold weiter auf ein kleineres Scaffold reduziert werden[1]. Dabei werden in jedem Schritt Teile des Moleküls bzw. Scaffolds entfernt, sodass das resultierende Scaffold eine immer noch zusammenhängende Verbindung von Ringen darstellt, allerdings mit weniger Ringen als zuvor (Abbildung 4.2).

Der erste Schritt der Scaffoldbildung sieht vor, dass freie Bindungen von Molekülen entfernt werden. Dadurch werden verschiedene Moleküle auf das gleiche Scaffold reduziert. Ein Scaffold kann somit als Repräsentant für diese Menge von Molekülen angesehen werden. Gleichzeitig kann dieses Scaffold aber auch durch Reduktion eines größeren Scaffolds entstehen, welches seinerseits für eine Menge von Molekülen steht. Auf diese Weise ergibt sich eine hierarchische, baumartige Struktur der Molekülgesamtmenge, bei der Scaffolds die Knoten/Blätter darstellen. Abbildung 4.3 zeigt einen Ausschnitt aus einem solchen Scaffoldbaum.

Aus pharmakologischer Sicht hofft man, dass Moleküle, die ähnlich aufgebaut sind, auch ähnliche Eigenschaften aufweisen.

[1] Ein solches Regelwerk mit anpassbaren Regeln wird in [21] vorgestellt



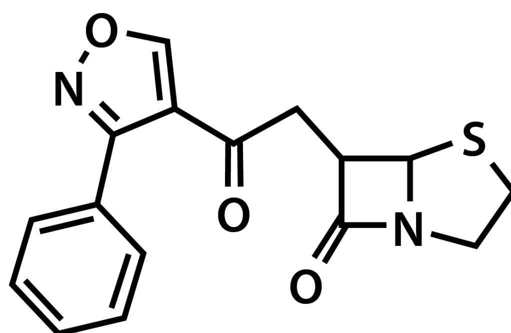


Abbildung 4.1.: Reduktion eines Moleküls auf sein Scaffold durch Entfernen freier Bindungen.

4.2. Scaffold Hunter

Das von der Projektgruppe 504 entwickelte Programm *Scaffold Hunter* entstand zu dem Zweck, diesen Scaffoldbaum ausschnittsweise visualisieren zu können. Zu den weiteren Zielsetzungen gehörten u.a. die Erstellung eines Scaffoldbaums (also die Anordnung einer Molekülmenge in die Baumstruktur), die Navigation in dem erstellten Baum und die Anzeige von weiterführenden Informationen zu einem Molekül.

Der Scaffold Hunter teilt den Workflow des Anwenders in zwei große Teile auf. Als erstes muß der gewünschte Scaffoldbaum erstellt werden. Im zweiten Schritt kann der Anwender bereits erstellte Bäume betrachten. Hier liegt der Schwerpunkt des Scaffold Hunters. Darüberhinaus verwendet der Scaffold Hunter noch eine Benutzerverwaltung, sodass individuelle Einstellungen jedes Nutzer gespeichert werden können.

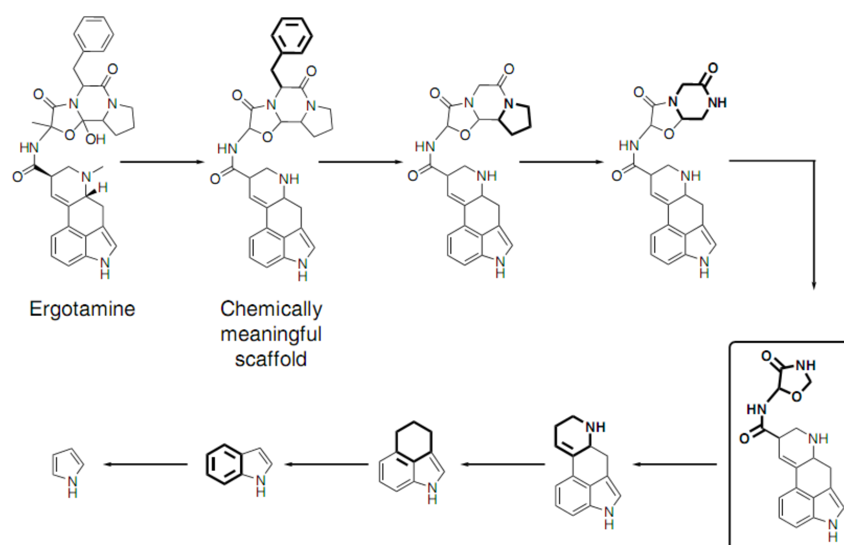


Abbildung 4.2.: Scaffoldreduktion am Beispiel von Ergotamin[21].

4.2.1. Baumerzeugung

Der Baumerzeugungsschritt ist beim Scaffold Hunter in ein eigenes Programm ausgegliedert. Der Anwender kann hier eine Molekülmenge in Form einer *Structure Data Format*-Datei sowie ein Regelwerk für die Scaffoldbildung angeben. Da die anschließende Baumgenerierung viel Zeit in Anspruch nehmen kann, wird der Baum nach seiner einmaligen Erzeugung in einer Datenbank gespeichert.

4.2.2. Visualisierung des Baumes

Beim Start des zweiten Programms, des eigentlichen Scaffold Hunters, muß der Anwender zunächst einen bereits generierten Baum aus einer Datenbank auswählen. Dabei können noch diverse Filter auf die im Baum vorhandenen Moleküle und Scaffolds angewendet werden, um sich im folgenden Arbeitsprozeß auf eine Teilmenge des Baumes beschränken zu können.

Die Hauptfunktionalität des Scaffold Hunters liegt in der Darstellung des Scaffoldbaums. Dazu bietet er verschiedene Funktionen an um in dem Baum zu navigieren, zu zoomen, Unterbäume ein- und ausblenden und Informationen an Knoten und Blättern, sowie in separaten Fenstern anzuzeigen. Weiterhin können Nutzer Kommentare zu Molekülen und Scaffolds verfassen und „Lesezeichen“ setzen, um einzelne Scaffolds schnell wiederzufinden.

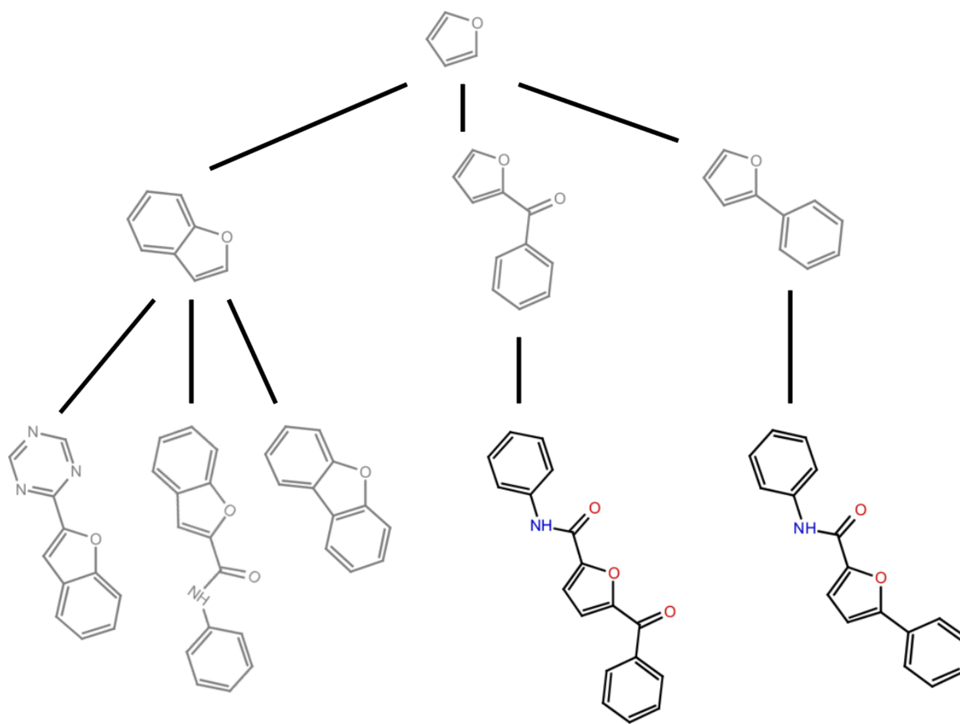


Abbildung 4.3.: Ausschnitt aus einem Scaffoldbaum[23].

5. Konzept

Die Weiterentwicklung des Scaffold Hunters der vorherigen Projektgruppe lässt sich in drei Abschnitte unterteilen. Der erste Abschnitt (Datenintegration) befasst sich mit der vereinfachten Anbindung multipler externer Datenquellen sowie der automatischen Berechnung weiterer Daten zu einem Molekül. Der zweite Abschnitt beinhaltet statistische Analysemethoden auf den integrierten Daten. Der dritte und letzte Abschnitt umfasst allgemeine Designentscheidungen, die visuelle Aufbereitung der gegebenen Informationen sowie den Arbeitsablauf und Bedienkonzepte.

5.1. Datenverwaltung

Für den neuen Scaffold Hunter sollte die Datenverwaltung überarbeitet werden, da die des alten Scaffold Hunters nicht für die Anforderungen des neuen Scaffold Hunters geeignet war. So soll eine zentrale Datenbank für mehrere Benutzer eingerichtet werden können anstatt einzelner Datenbanken auf jedem Rechner der Benutzer. Anders als vorher enthält diese Datenbank dann mehrere Scaffoldbäume und Moleküldatensätze. Auf solch einer zentralen Datenbank können Benutzer dann zusammen auf den gleichen Daten arbeiten und auch Kommentare austauschen. Da diese Änderungen eine große Umstrukturierung der Datenverwaltung zur Folge hatte, wurde die Datenverwaltung fast komplett neu entwickelt.

5.1.1. Datenimport

Von Beginn an war geplant den Datenimport zu einem Teil des Scaffold Hunters zu machen, anstatt ein zweites Programm zu verwenden. Desweiteren sollte die Integration unterschiedlicher Datenquellen ermöglicht werden. Dabei sollten sowohl externe Datenbanken (z.B. *PubChem*) als auch lokale Daten als Datenquelle verwendet werden können. Der Ablauf dieser Datenintegration sollte für den Benutzer nachvollziehbar gestaltet werden, sodass dieser Vertrauen zu den importierten Daten hat.

Datenquellen Das Ziel des Datenimports ist das Einlesen einer oder mehrer *Datenquellen* in die Scaffold Hunter Datenbank. Eine Datenquelle kann dabei verschiedene Formen annehmen, Beispiele sind eine aus einer öffentlichen oder proprietären Datenbank heruntergeladene *sdf-Datei* oder der direkte Zugriff auf eine SQL Datenbank. Generell enthält eine Datenquelle Daten zu chemischen Strukturen. Neben der Beschreibung des chemischen Aufbaus einer Struktur enthält eine Datenquelle in den meisten Fällen weitere Daten. Diese lassen sich in mehrere Kategorieren aufteilen.

Numerische Eigenschaften: In diese Kategorie fallen viele Eigenschaften einer chemischen Struktur diese können entweder gemessen oder aus der Struktur errechnet worden sein. Darunter fallen zum Beispiel die Molekülmasse oder der $\log P$ Wert als Maß für die Wasserlöslichkeit einer Struktur. Interne oder externe Datenbank Kennnummern wie etwa die *Pubchem Compound ID* fallen ebenfalls in diese Kategorie.

Textuelle Eigenschaften: Viele Datenquellen enthalten auch textuelle Daten zu chemischen Strukturen etwa Bezeichner wie *InChI* oder verschiedene *SMILES*[1] Varianten, URLs oder nichtnumerische Datenbank IDs.

Fingerprints: Zur Verwendung beim Fingerprintbasierten Clustering können Fingerprints importiert werden, es wird jedoch vorausgesetzt das diese in einem vom Scaffold Hunter definierten Format vorliegen, sodass diese Option wahrscheinlich nur beim Reimport von aus dem Scaffold Hunter exportierten Daten genutzt werden kann.

Importvorgang Ziel des Importvorgangs ist die Erstellung eines *Datensets* aus einer oder mehreren *Datenquellen*. In diesem Zusammenhang wird das Einlesen einer einzelnen Datenquelle als *Importjob* bezeichnet. Zu Beginn des Importvorgangs erstellt der Benutzer für jede Datenquelle einen Importjob. Zum Einlesen unterschiedlicher Datenquellen stehen hierbei verschiedene *Plugins* zur Verfügung. Anschließend kann der Benutzer die zu importierenden Eigenschaften auswählen und ihren Typ und den fortan verwendeten internen Bezeichner für diese Eigenschaft festlegen. Werden hierbei Eigenschaften aus verschiedenen Datenquellen auf die selbe interne Eigenschaft abgebildet, muss weiterhin eine Strategie zur *Konfliktbehandlung* ausgewählt werden.

Zusammenführen verschiedener Datenquellen Ein Problem, das bei der parallelen Integration mehrerer Datenquellen entsteht, ist die Behandlung konkurrierender Werte aus unterschiedlichen Datenquellen. Es können semantisch identische Attribute in unterschiedlichen Datenbanken vorhanden sein. Wenn diese Attribute auch identische Werte annehmen, kann einfach eine Datenquelle für dieses Attribut ausgewählt werden. Treten jedoch unterschiedliche Werte auf oder fehlen Werte in einer Quelle, so müssen Attribute aus unterschiedlichen zusammengeführt werden. Der Teil des Programms, der das Zusammenführen von Daten übernimmt, wird im folgenden *Merger* genannt. Der *Merger* geht sequentiell vor, d.h. es werden die Daten der einzelnen Datenquellen nacheinander integriert.

Hierbei Betrachtet der *Merger* Strukturen mit dem gleichen SMILES-String als identisch und führt deren Attribute zusammen. Dieses Verhalten kann unter Umständen unerwünscht sein (siehe Seite 26).

Die folgende Aufzählung gibt einen Überblick über angebotenen Strategien zur *Konfliktbehandlung*:

[1] *Simplified Molecular Input Line Entry Specification*

Nicht überschreiben: Es wird derjenige Wert verwendet der als erstes verfügbar ist.

Überschreiben: Bereits vorhandene Werte werden von folgenden Werten überschrieben

Min/Max (bei Numerischen Eigenschaften): Es wird der minimale oder maximale Wert verwendet.

Konkatenation (bei textuellen Eigenschaften): Die konkurrierenden Attribute werden konkateniert.

Zunächst war geplant für numerische Eigenschaften auch das Bilden eines Durchschnitts anzubieten, darauf wurde jedoch während der Entwicklung verzichtet. Der Nutzen dieser Strategie wurde von uns beratenden Chemikern in Zweifel gezogen. Falls verschiedene Werte für die selbe Struktur vorliegen ist es unwahrscheinlich, dass durch Durchschnittbildung ein „besserer“ Wert erlangt wird. In Anbetracht dieser Aussage und aufgrund des zusätzlichen Implementierungsaufwands, den diese Strategie bei der sequentiellen Berechnung verursacht hätte, haben wir von einer Implementierung abgesehen.

Nachvollziehbarkeit Um die Vorgänge während des Datenimports transparent zu machen und so das Vertrauen des Nutzers in die importierten Daten zu gewinnen, wurde entschieden, neben der Fortschrittsanzeige ein ausführliches Protokoll anzuzeigen, welches Ereignisse wie fehlende Daten, auftretende Konflikte und deren Behandlung und unlesbare Strukturinformationen nachhält. Um den Nutzer dabei nicht mit einer Flut von Meldungen zu „erschlagen“ wird dieses Protokoll in einer baumartigen Darstellung angezeigt, welche dem Nutzer zunächst nur einen groben Überblick über auftretende Ereignisse bietet, es aber bei Bedarf ermöglicht Teile des Baums auszuklappen und so eine detaillierte Liste der aufgetretenen Ereignisse anzuzeigen. Abbildung 5.1 zeigt diesen Fortschrittsdialog.

Plugins Um das Programm flexibel zu gestalten und einfaches Nachrüsten weiterer unterstützter Quellformate zu ermöglichen wurde entschieden den Einlesevorgang durch ein Pluginsystem zu realisieren. Zum Erstellen eines *Importjobs* wird zunächst ein Plugin ausgewählt. Dieses stellt zunächst ein pluginspezifisches *Panel* mit Importoptionen zur Verfügung. Diese reichen von einer einfachen Dateiauswahl für das SDF-Plugin zu deutlich umfangreicheren Optionen beim SQL-Plugin. Zum jetzigen Zeitpunkt werden dem Benutzer drei verschiedene Plugins zur Verfügung gestellt:

SDF-Plugin: Ein Plugin zum Import von *Structure Date Files*, einem verbreiteten Format für chemische Strukturen samt Attributen. Viele öffentliche Strukturdatenbanken stellen Downloads in diesem Format zur Verfügung.

CSV-Plugin: Ein Plugin für den Import von *Comma Seperated Value Files*. Dabei handelt es sich um ein einfaches, in verschiedenen Varianten auftretendes Format zur Kodierung von Tabellen, welches von allen gängigen Tabellenkalkulationen gelesen und geschrieben werden kann.

SQL-Plugin: Ein Plugin zum direkten Import von Daten aus einer SQL-Datenbank.

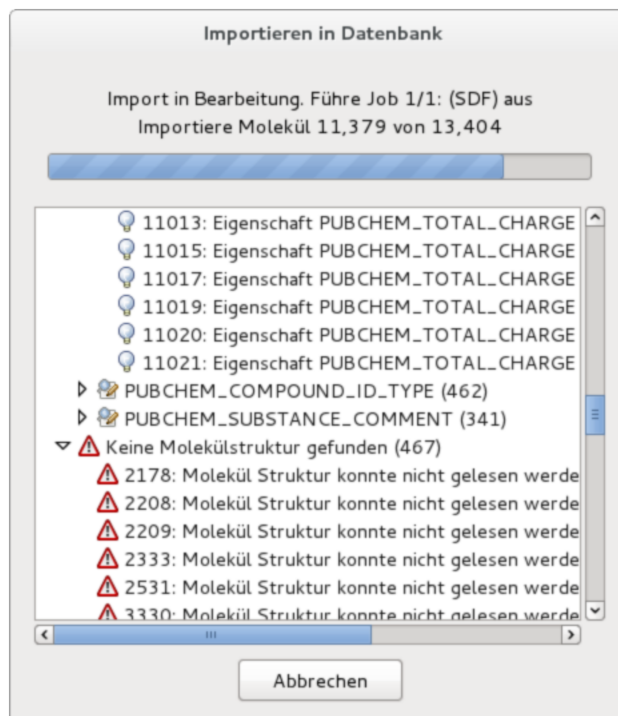


Abbildung 5.1.: Dialog zur Anzeige des Importfortschritts

Nachträgliches Hinzufügen von Daten Nach der Erstellung eines Datensets ist es möglich weitere Attribute zu den Strukturen dieses Datensets hinzuzufügen. Dadurch ist es möglich neue experimentelle oder mittels anderer Programme errechnete Attribute in ein bestehendes Datenset zu integrieren. Die Zuordnung von Attributen zu Strukturen erfolgt dabei entweder über den SMILES-String oder über ein bereits bestehndes, eindeutiges Attribut, wie etwa eine Datenbankkennziffer.

SMILES als eindeutige Schlüssel für Strukturen Während der Entwicklung wurde relativ früh beschlossen den eine Struktur kodierenden SMILES-String als eindeutigen Schlüssel beim zusammenführen von Daten zu verwenden, hierdurch ist es nicht möglich mehrere Einträge mit der gleichen Struktur in die Scaffold Hunter Datenbank zu importieren. Wie sich später herausstellte kann dieses Verhalten unerwünscht sein. So ist es zum Beispiel möglich, das eine chemische Datenbank mehrere Einträge mit der gleichen Struktur und unterschiedlichen Attributen enthält. Dies kann zum Beispiel auftreten wenn die selbe Struktur in unterschiedlichen Lösungen betrachtet wird. Je nach Datenbank kann dabei die verwendete Lösung entweder als Teil der Strukturdaten oder als zusätzliches Attribut angegeben werden. Ist die Lösung Teil der Strukturdaten, so werden unterschiedliche SMILES-Strings für verschiedene Lösungen generiert und die Einträge können getrennt importiert werden. Falls diese Information jedoch durch ein weiteres Attribut zur Verfügung gestellt wird, kann dies momentan beim Import nicht

berücksichtigt werden. Aus diesem Grund werden beim Import alle Einträge aus einer Datenquelle, für welche ein SMILES-String generiert wird, der zu einem bereits aus dieser Datenquelle importierten Eintrag identisch ist, verworfen

5.1.2. Datenberechnung

Für die geplante Dendrogrammansicht (Darstellung eines hierarchischen Clusterings einer Molekülmenge) ist es hilfreich das Berechnen von zusätzlichen Eigenschaften (wie z.B. Fingerprints) für die importierten Moleküle anzubieten. Um den Scaffold Hunter möglichst flexibel und erweiterbar zu gestalten und jedem Benutzer die Wahl von geeigneten Eigenschaften (wie z.B. Fingerprints) passend zum Forschungsthema zu ermöglichen, ist ein Plugin-System zur Berechnung zusätzlicher Eigenschaften geplant.

Berechnungsvorgang Ziel des Berechnungsvorganges ist das Berechnen und Hinzufügen von Eigenschaften zu einem bestehenden *Datenset*. Der Benutzer erstellt einen Berechnungsjob, indem er ein Berechnungsplugin auswählt und nach seinen Bedürfnissen konfiguriert. Wird der Berechnungsjob ausgeführt, erhält das entsprechende Plugin über die Pluginschnittstelle lesenden Zugriff auf die Moleküle und ihre Eigenschaften und kann neue Eigenschaften hinzufügen. Diese Eigenschaften stehen dann, wie alle anderen importierten Eigenschaften auch, im Hauptprogramm zur Verfügung.

Nachvollziehbarkeit Um evtl. auftretende Probleme beim Ausführen eines Plugins für den Benutzer transparent zu gestalten, wird ähnlich wie in 5.1.1 beschrieben neben der Fortschrittsanzeige ein ausführliches Protokoll in baumartiger Darstellung angezeigt.

5.2. Statistische Analyseverfahren

Um zusätzlich zum Scaffoldbaum eine alternative Klassifikationsmöglichkeit der Moleküle zu ermöglichen, wurden statistische Verfahren des unüberwachten Lernens in den Scaffold Hunter integriert. Dabei handelt es sich um SAHN (*Sequential Agglomerative Hierarchic Nonoverlapping*) Clustering Verfahren. Es werden eine Vielzahl von verschiedenen Linkageverfahren und Distanzmaße für numerische Werte und Bit Fingerprints angeboten. Insbesondere der Tanimoto Koeffizient spielt hier aufgrund der vielen binären Merkmalsdeskriptoren in der Biochemie eine wichtige Rolle.

5.3. Allgemeine Designentscheidungen

Generierung des Scaffoldbaums Anders als im bisherigen Scaffold Hunter, der die Erzeugung des Scaffoldbaums in ein separates Programm ausgelagert hatte, ist es nun möglich, sämtliche Arbeitsschritte bei der Benutzung des Scaffold Hunters innerhalb einer graphischen Oberfläche durchzuführen.

Verschiedene Ansichten Die für den Scaffold Hunter namensgebende Scaffoldbaumansicht wurde um weitere Ansichten ergänzt, welche eine alternative Art der Darstellung, Navigation und Filterung von Molekülen ermöglichen. Diese Ansichten stehen prinzipiell gleichberechtigt neben dem Scaffoldbaum zur Verfügung, und besonderes Augenmerk wurde auf deren nahtlose Integration in das Programm, sowie die möglichst gute Unterstützung damit verbundener typischer Arbeitsabläufe gerichtet. Unter anderem stehen zusätzlich zum Scaffoldbaum eine Dendrogrammansicht für hierarchisches Clustering, eine Tabellenansicht sowie ein Streudiagramm für Moleküleigenschaften zur Verfügung.

Subset-Konzept Ein entscheidendes neues Konzept in der Weiterentwicklung des Scaffold Hunters ist das Prinzip der sogenannten *Subsets*, d.h. Teilmengen von Molekülen, die sich jeweils in einer eigenen Ansicht (Scaffoldbaumansicht, Dendrogrammansicht, etc.) darstellen lassen. Ausgangspunkt ist hierbei zunächst das bei der Datenintegration entstandene Datenset, das alle aus der Datenbank geladenen Moleküle enthält. Diese Menge lässt sich nun weiter eingrenzen, indem beliebig viele Moleküle markiert und anschließend in einem neuen Subset abgelegt werden.
Die einzelnen Ansichten bieten hierzu verschiedene Möglichkeiten, zusammengehörige Moleküle auf einfache Art und Weise auszuwählen. So wählt etwa die Markierung eines Scaffolds im Scaffoldbaum alle darunterliegenden Moleküle aus, und in der Dendrogrammansicht lässt sich ein beliebig großes Cluster mit allen darin enthaltenen Molekülen der Auswahl hinzufügen.
Die Auswahl von Molekülen ist hierbei global über alle Ansichten, d.h. wird ein Molekül in einer Ansicht markiert, so wird das selbe Molekül gleichzeitig auch in allen anderen Ansichten ausgewählt. Dadurch ist es relativ unkompliziert möglich, die Lage eines Moleküls in verschiedenen Ansichten zu vergleichen, und bei der Erzeugung eines Subsets lassen sich die Auswahlmethoden verschiedener Ansichten sehr einfach kombinieren.

Navigation Die Integration der neuen Ansichten sowie die Möglichkeit, beliebig viele Teilmengen von Molekülen in jeweils eigenen Ansicht darzustellen, erforderten ein deutlich erweitertes Bedienkonzept. Alle Ansichten werden in Tabs angezeigt, sodass sich bequem zwischen verschiedenen Ansichten und zwischen verschiedenen Subsets umschalten lässt. Auch gibt es die Möglichkeit, mehrere Ansichten gleichzeitig nebeneinander darzustellen. Zu diesem Zweck lässt sich einerseits die Tab-Leiste splitten, so dass zwei Ansichten im selben Fenster angezeigt werden. Andererseits lassen sich aber auch mehrere Hauptfenster mit jeweils eigenen Ansichten erstellen, die bei Bedarf auch auf verschiedene Monitore verteilt werden können.
Ein neues Bedienelement ist die Subsetleiste, die in einer Baumdarstellung den Überblick über das Datenset und alle daraus erzeugten Subsets bietet. Die Subsetleiste ermöglicht es, jedes Subset in einer beliebigen neuen Ansicht zu öffnen, und verfügt über diverse Funktionen zur Verwaltung und Annotation der Subsets.

Annotationen Zur Verbesserung der Übersicht, und um vorausgegangene Arbeitsschritte besser nachvollziehbar zu machen, ist es an verschiedenen Stellen möglich tex-

tuelle Beschreibungen zu Molekülen, Subsets und Tabs hinzuzufügen. Moleküle können darüber hinaus mit Fähnchen versehen werden, um diese beispielsweise als "gesehen" zu markieren.
Alle Annotationen können entweder privat oder öffentlich gemacht werden, um zu bestimmen, ob diese anderen Benutzern der selben Datenbank zur Verfügung gestellt werden sollen.

Session Management Der Zustand einer Scaffold Hunter Sitzung kann jederzeit in der Datenbank gespeichert und bei Bedarf komplett wiederhergestellt werden. Dies umfasst unter anderem die erzeugten Subsets sowie alle aktuell geöffneten Ansichten inklusive Annotationen und Visualisierungs-Parameter.
Alle Sitzungsdaten werden einheitlich in der zentralen Datenbank gespeichert, so dass eine gespeicherte Sitzung ggf. auch an einem anderen Arbeitsplatz wiederaufgenommen werden kann.

6. Implementierung

6.1. Allgemeine Design- und Strukturentscheidungen

Die Entscheidung Java als Programmiersprache für den Scaffold Hunter zu verwenden war von Anfang an klar, da der erste Scaffold Hunter auch in Java geschrieben wurde und wir diesen erweitern sollten. Aus diesem Grund hätte es wenig Sinn gemacht zu einer anderen Programmiersprache zu wechseln. Zusätzlich konnten wir so auch Codefragmente weiterverwenden. Dies haben wir bei einzelnen Funktionen getan, die Grundstruktur aber neu gebaut, da wir durch die Aufgabenstellung größere Umstrukturierungen durchführen mussten und an vielen Stellen andere Ansätze als unsere Vorgänger verfolgt haben. Einzelne Komponenten, wie etwa die Seitenleistenelemente in der Scaffoldbaumansicht haben wir aber größtenteils übernommen, da sie sehr gut in das Konzept passten. Zusätzlich zu dem so gesparten Programmieraufwand wirkt die Umstellung der alten auf die neue Version weniger drastisch, wenn bekannte Elemente wieder vorhanden sind.

Wir haben uns entschlossen einen Logging-Mechanismus zu implementieren, um so detaillierte Debugginginformationen nutzen zu können. Auf diese Weise kann zentral gesteuert werden in wie weit der Benutzer programmiertechnische Informationen zur Laufzeit angezeigt bekommen möchte, ohne das jedes Mal die Ausgabe der Loggingdaten im Code ein- bzw. auskommentiert werden müsste.

6.1.1. Package-Struktur / Modularität / Interfaces

Bei der Entwicklung des Scaffold Hunters wurde versucht, den Code möglichst modular zu halten. Dahinter steckt nicht zuletzt die Hoffnung, daß zukünftige Erweiterungen des Programms, beispielsweise um zusätzliche Ansichten oder Import-Funktionen, ohne allzu tiefgreifende Änderungen des vorhandenen Codes möglich sind.
Für dem Import und die Berechnung von Daten wurden Plugin-Schnittstellen bereitgestellt, die die Entwicklung und Distribution neuer Funktionalität auch außerhalb des Scaffold Hunter Quellcodes erlaubt. Eine Plugin-Schnittstelle für Ansichten war zwischenzeitlich angedacht, wurde aber aufgrund des erforderlichen Mehraufwands im Rahmen dieser PG verworfen. Nichtsdestoweniger gibt es auch hierfür eine Schnittstelle, durch die allgemeine und ansichtsspezifische Programmteile sauber voneinander getrennt sind.

Schnittstelle für Ansichten Das `View`-Interface definiert eine gemeinsame Schnittstelle für alle vom Scaffold Hunter unterstützten Ansichten. Jede Ansicht ist selbst für die Erzeugung ihrer Benutzeroberfläche zuständig, inklusive Menüs, Toolbars, und Seitenleiste. Über das Interface können diese Elemente vom Hauptprogramm abgefragt und an den

entsprechenden Stellen im Hauptfenster plaziert werden. Darüber hinaus besitzen alle Ansichten Eigenschaften wie das angezeigte Subset, Konfigurationseinstellungen sowie den aktuellen Zustand der Ansicht. Diese müssen bei Bedarf abgefragt oder gesetzt werden können, beispielsweise um eine Sitzung zu speichern und wiederherzustellen. Auch hierfür stellt das `View`-Interface entsprechende Methoden bereit.
Zusätzlich gibt es eine abstrakte Klasse namens `GenericView`, die `View` implementiert und als Basisklasse aller konkreten Ansichten dient. Diese vereinfacht die Implementierung von Ansichten und stellt unter anderem Methoden bereit, mit denen Ansichten bei Bedarf auf bestimmte Objekte des Hauptprogramms zugreifen können.

Model-View-Controller-Architekturmuster Wir haben versucht, überall dort das *Model-View-Controller*-Konzept anzuwenden, wo es für die Implementierung des Scaffold Hunters sinnvoll und praktikabel erschien. Während die Trennung zwischen *View* und *Controller* teilweise recht schwammig ist, ist der *Model*-Teil weitgehend sauber vom übrigen Programm getrennt. Positiv bemerkbar macht sich dies insbesondere bei der Kapselung von Zugriffen auf die zugrundeliegende Datenbank. Sowohl das Hauptfenster als auch die Ansichten operieren zu einem großen Teil auf den selben Daten, und stellen im MVC-Konzept verschiedene Views des selben Models dar.
Auch beim Speichern und Wiederherstellen von Sitzungen kommen die Vorteile dieser Trennung zum Tragen, da letztlich alle zu sichernden Daten im Model zu finden sind.

Package-Struktur Auch die Package-Struktur des Scaffold-Hunter-Codes spiegelt die Trennung zwischen Model und View/Controller wieder.
Direkt im Package `edu.udo.scaffoldhunter` befindet sich lediglich die Klasse `ScaffoldHunter` mit der `main`-Funktion. Der Code des Hauptfensters und aller nicht Ansichts-spezifischen Dialoge befindet sich im Package `edu.udo.scaffoldhunter.gui`. Das Package `edu.udo.scaffoldhunter.model` umfaßt den Model-Teil des Programms. In Unter-Packages hiervon sind unter anderem die Datenbank-Schnittstelle (`db`), die Scaffoldbaum-Generierung (`treegen`) sowie die implementierten Clusteringverfahren (`clustering`) zu finden.
Weiterhin gibt es das Package `edu.udo.scaffoldhunter.view` mit dem `View`-Interface und Unter-Packages für die einzelnen Ansichten (`scaffoldtree`, `dendrogram`, `plot` und `table`). Die Plugins für Datenimport und -Berechnung sind in `edu.udo.scaffoldhunter.plugins` zu finden.

6.1.1.1. Plugins

Mit Hilfe des Java Simple Plugin Framework (siehe Abschnitt 6.3.13) haben wir zwei Pluginschnittstellen erstellt. Zum einen eine für den Datenimport, zum anderen eine für die nachträgliche Berechnung von Daten. Dies soll insbesondere eine leichte Anbindung des Scaffold Hunters an neue Datenquellen oder z.B. interne Datenbanken von Forschungseinrichtungen ermöglichen.

Gemeinsamkeiten beider Schnittstellen Jedes Plugin liefert eine Konfigurationsoberfläche, die dem Benutzer bei Auswahl des Plugins im Import- bzw. Berechnungsprozess gezeigt wird. Im Rahmen dieser Konfigurationsfähigkeit kann jedes Plugin Konfigurationsdaten zum Speichern zur Verfügung stellen und erhalten. Zusätzlich stellt jedes Plugin die aktuellen Einstellungen getrennt zur Verfügung und diese werden dem Plugin zu zwei Zeitpunkten mitgeteilt. Zum einen bei der Anzeige der Konfigurationsoberfläche, um nachträgliche Änderungen zu ermöglichen, zum anderen beim Plugindurchlauf, so können mehrere aufeinanderfolgende Durchläufe mit einem Plugin realisiert werden. Ferner können MessageListener bei den Plugins registriert werden, über die Meldungen der Plugins beim endgültigen Durchlauf im Prozessdialog angezeigt werden können.

Schnittstelle der Importplugins Zusätzlich zur gemeinsamen Schnittstelle bieten die Import Plugins noch eine Schnittstelle für eine Vorabprüfung, ob die aktuelle Konfiguration auf jeden Fall zu einem Fehler führt. Weitere Daten erhält das Plugin vom Scaffold Hunter nicht. Als Ergebnis liefert das Plugin die Anzahl der Moleküle, eine Liste mit Eigenschaftsnamen, eine weitere, welche angibt welche numerische Werte enthalten, und alle Moleküle über einen iterativen Zugriff.

Schnittstelle der Berechnungsplugins Die Berechnungsplugins erhalten zum einen eine Liste mit allen aktuell im Datensatz vorhandenen Eigenschaften, auf dieser Basis kann der Benutzer im Plugin wählen, welche Eigenschaften für die Eingabe verwendet werden sollen. Zusätzlich kann das Plugin ein weiteres Konfigurationsfeld einbinden, welches die Auswahlmöglichkeit für Deglykosylierung, Neuberechnung der 2D Struktur und Auswahl nur des größten Fragmentes bietet. Das Plugin kann auch auf Funktionen zugreifen, die die Moleküle entsprechend vorbereiten. Für den Plugindurchlauf erhält das Plugin alle Moleküle des Datasets mit iterativem Zugriff und liefert diese alle iterativ zurück. Dies führt zu einem möglichst geringen Speicheraufwand.

6.1.2. Designguide

Die übersichtliche und einheitliche Gestaltung der Programmoberfläche hat wesentlichen Einfluß auf die Bedienbarkeit eines Programms, insbesondere auch für neue und unerfahrene Nutzer. Aus diesem Grund wurde für den Scaffold Hunter ein Designguide erstellt, der einige grundlegende Eigenschaften der Oberfläche beschreibt. Dieser umfaßt unter anderem Maus- und Tastaturbelegungen, Richtlinien für Icons und Farbwahl, sowie zur Benennung und Anordnung von GUI-Elementen.

Hierbei kommen zwei verschiedene Aspekte der GUI-Gestaltung zum Tragen: Zum einen sollte die Bedienung des Scaffold Hunters in sich möglichst konsistent sein, zum anderen ist aber auch eine gute Anpassung an das jeweilige Betriebssystem wünschenswert.

Bedienung der verschiedenen Ansichten Funktionen wie die Auswahl von Molekülen, das Zoomen und Verschieben des angezeigten Bildausschnitts oder das Setzen von

Fähnchen sind in mehreren oder allen Ansichten verfügbar. Soweit möglich und sinnvoll werden hierzu überall die selben Mausbelegungen und Tastenkürzel verwendet. Abweichungen hiervon sind allerdings dort angebracht, wo gängige UI-Konventionen ein anderes Verhalten erwarten lassen, oder wo häufig genutzte Funktionen sonst unnötig umständlich würden.

Anpassung an das Betriebssystem Die verschiedenen von Java und damit auch vom Scaffold Hunter unterstützten Betriebssysteme unterscheiden sich nicht nur in ihrem Erscheinungsbild, sondern weisen auch eine Vielzahl von mehr oder weniger subtilen Unterschieden in der Bedienung gängiger GUI-Elemente auf. Eine vollständige Umsetzung der UI-Richtlinien aller unterstützten Betriebssysteme ist mit Javas Swing-Bibliothek nicht praktikabel.
Im Scaffold Hunter wurde allerdings dafür Sorge getragen, daß zumindest das Erscheinungsbild (Theme, Schriftart, etc.) weitgehend dem des Betriebssystem und den entsprechenden Einstellungen des Nutzers entspricht. Außerdem wird die Button-Reihenfolge in Dialogen automatisch angepaßt, so daß sich beispielsweise der 'Ok'-Button unter Windows links, und unter anderen Betriebssystemem rechts befindet.

6.2. Lizenz

Der Code des bisherigen Scaffold Hunter bzw. ChemBioSpace Explorer (PG 504) wurde unter der GPLv3 (GNU General Public Licence Version 3 [1]) lizensiert. Da unsere Projektgruppe den bestehenden Scaffold Hunter weiterentwickelt, wird auch unser Code unter der **GPLv3** veröffentlicht. Dies bedeutet insbesondere, dass auch die verwendeten Bibliotheken unter der GPLv3 oder einer kompatiblen Lizenz veröffentlicht sein müssen. Details dazu befinden sich im nächsten Kapitel 6.3.

6.3. Verwandte Bibliotheken

6.3.1. Hibernate

Kurzbeschreibung Das Hibernate Framework ist ein OR-Mapper (Object-Relational-Mapper) zur Anbindung von Relationalen Datenbanksystemen in Java. Darüber hinaus beinhaltet es eine Menge weiterer Funktionen, die über einen einfachen OR-Mapper weit hinaus gehen.

Logo:

Quelle: Hibernate Release 3.6; documentation/manual/en-US/html/images

Herausgeber: JBoss / Red Hat

Homepage: www.hibernate.org

[1] http://www.gnu.org/licenses/gpl.html

Lizenz: LGPL

aktuelle Version (15.03.2011): 3.6.1

verwendete Version: 3.6.0

Verwendungszweck Hibernate wird in unserem Projekt für die interne Datenhaltung verwendet. Dies entspricht den importierten Daten aus externen Datenbanken, sowie Programmeinstellungen und Programmzustandsdaten. Die Projektgruppe hat sich aus den folgenden Gründen für den Einsatz eines OR-Mappers ausgesprochen:

- Die einfache Austauschbarkeit des verwendeten Datenbanksystems erhöht die Flexibilität bezüglich des Einsatzorts und Zwecks.
  - Eine mitgelieferte MySQL Datenbank garantiert den vollen Funktionsumfang der Software auf einem lokalen PC.
  - Im firmenweiten Einsatz kann auf bestehende Datenbankinfrastruktur zurück gegriffen werden (z.B. Oracle, DB2, etc).
- Einfacher Umgang mit den gespeicherten Daten. Zugriff erfolgt direkt über get und set Methoden einer Klasse.
- Performance Optimierungen durch den Einsatz von Caching.
- Speicher Optimierungen durch den Einsatz von Lazy Loading.
- Automatische Anfrageoptimierungen bzgl. des verwendeten Datenbanksystems

Natürlich bietet der Einsatz eines OR-Mappers nicht nur Vorteile. In vielen Fällen sind reine SQL Anfragen schneller, da weniger SQL Anfragen benötigt werden. Ein Beispiel dafür sind hierarchische Objektstrukturen (Ein Objekt besitzt ein Objekt, dass wiederum ein Objekt besitzt, usw.). Für jedes Objekt wird bei dem Einsatz eines reinen OR-Mappers eine einzelne Anfrage benötigt. Hibernate bietet für solche Fälle die Verwendung von HQL (Hibernate Query Language) und nativem SQL an. HQL wird je nach verwendeten Datenbanksystem in nativen SQL Code übersetzt. Performancekritische Programmteile können somit mit geringem Mehraufwand dennoch effizient erledigt werden.

6.3.2. Piccolo2D

Kurzbeschreibung Piccolo2D ist ein Framework zur Gestaltung interaktiver, zoombarer 2D-Grafikoberflächen in Java oder C#. Es basiert auf dem Szenengraphen-Modell und stellt eine Schnittstelle zur Verfügung, die auf eine Graphik-API tieferer Ebene (z.B. Java2D) aufbaut. (vgl. [2])

Herausgeber: ursprünglich Human-Computer Interaction Laboratory, University of Maryland; Weiterentwicklung durch freiwillige Entwicklergemeinde auf der Plattform Google Code

Homepage: www.piccolo2d.org

Lizenz: BSD Lizenz

aktuelle Version für Java (14.11.2011): 1.3.1

verwendete Version: 1.3

Features Piccolo bietet einige Features, die dem Programmierer aufwändige Selbstimplementierungen ersparen (vgl. [2]):

- effizientes Neuzeichnen des Bildschirms
- Boundingbox-Management
- Eventhandling
- Picking[2]
- Animationen
- Layout-Management[3]

Gründe für den Einsatz von Piccolo Für den Einsatz von Piccolo lassen sich folgende Gründe anführen:

- Im Scaffold Hunter müssen viele Moleküle visualisiert werden. Um eine übersichtliche und effiziente Darstellung sicherzustellen, ist der Einsatz von Zoom- und Level-Of-Detail-Techniken unerlässlich. Piccolo ist genau für die Anforderungen zoombarer Benutzeroberflächen entwickelt worden.
- Die Darstellungs- und Interaktions-Komponenten des Scaffold Hunter 1.0 wurden mit dem Piccolo-Framework realisiert. Diese Programmteile wurden größtenteils übernommen und befinden sich nun in der Scaffoldbaumansicht. Piccolo eignete sich ebenfalls für die Dendrogrammansicht, so dass Codeteile auch für diese übernommen werden konnten (siehe Kapitel 6.6.2.2).
- Piccolo macht einen reifen Eindruck und wird von einer öffentlichen Entwicklergemeinde gepflegt. Es existiert eine Diskussionsgruppe[4] in der evtl. auftretende Fragen gestellt werden können.

[2] Ermittlung des Objektes im Szenengraphen, mit dem eine Benutzerinteraktion wie z.B. Mausklick stattfand

[3] Einheitliche Schnittstelle, um die Kindknoten eines Knotens z.B. bei Größenänderungen eines Kindes automatisch neu zu positionieren.

[4] `http://groups.google.com/group/piccolo2d-users`

6.3.3. Batik

Kurzbeschreibung Batik ist eine Java Bibliothek zum Rendern, Transcodieren, Generieren und programmatischen Bearbeiten von *Scalable Vector Graphics (SVGs)*. SVG[5] ist ein gebräuchlicher Standard zur XML-basierten Repräsentation von zweidimensionalen Vektorgrafiken.

Logo:

Quelle: Batik Homepage

Herausgeber: Apache Software Foundation

Homepage: `http://xmlgraphics.apache.org/batik/`

Lizenz: Apache License Version 2.0

aktuelle Version (25.3.2011) 1.7 (Release: 09.01.2008)

verwendete Version: 1.7

Verwendungszweck Im von der PG504 entwickelten Scaffold Hunter werden SVGs zur Darstellung von Molekülen und Scaffolds verwendet. Dabei kommt die Renderingkomponente von Batik zum Einsatz. In geringem Rahmen wird auch von den Bearbeitungsmöglichkeiten Gebrauch gemacht, etwa beim Einfärben ausgewählter Scaffolds. Darüberhinaus werden die Erzeugungs- und Transkodierkomponenten von Batik eingesetzt, um einen Export des Scaffoldbaumes in verschiedene Grafikformate zu ermöglichen.

Da wir beschlossen haben, die Scaffoldbaumansicht weitgehend in unser Projekt zu übernehmen, haben wir uns auch für die Weiterverwendung von Batik entschieden. Auch wenn die Entwicklung von Batik im Augenblick nicht mehr besonders aktiv vorangetrieben zu werden scheint, ist Batik ein relativ reifes Projekt, welches unter Mitarbeit mehrerer Firmen seit Herbst 2000 entwickelt wird. Die in der aktuellen Version befindlichen Features sind für die Zwecke unseres Projekts mehr als ausreichend.

Neben seiner ursprünglichen Verwendung in der Scaffoldbaumansicht wird Batik in unserem Projekt in der Tabellen- und der Dendrogrammansicht zum Rendern von Molekülgrafiken verwendet.

6.3.4. L2FProd

Kurzbeschreibung L2FProd Common Components stellt einige häufig verwendete Komponenten bereit, die in Swing leider nicht vorhanden sind. Dazu gehören ein `PropertySheet`, in dem Attribute von Objekten editiert werden können und eine `JTaskPane`, die zur Anzeige von gruppierten Aktionen in einer Seitenleiste dient, wie dies zum Beispiel bei dem Windows Explorer unter Windows XP der Fall ist. Weiterhin gibt es unter anderem eine `ButtonBar`, in der Buttons nebeneinander platziert werden

[5] `http://www.w3.org/Graphics/SVG/`

können, Dialoge zur Auswahl von Schriftart und Verzeichnis, sowie einen Dialog zur Anzeige eines Tipps des Tages. Ausserdem stellt diese Bibliothek Methoden zur Verfügung, mit denen das Layout der Dialoge des Programms beeinflusst werden kann.

Herausgeber: Frédéric Lavigne

Homepage: http://www.l2fprod.com/common/

Lizenz: Apache Lizenz 2.0

aktuelle Version (23.3.2011): 7.3

verwendete Version: 7.3

Verwendungszweck Von den bereitgestellten Komponenten werden vor allem `PropertySheet` im Optionsdialog und `JTaskPane` in der Seitenleiste des Hauptfensters verwendet. Ersteres hat eine rasche Erstellung des Optionsdialoges ermöglicht und basiert auf einer Tabelle, in der Attribute von Beans mit Hilfe von Editor- und Rendererklassen angezeigt und verändert werden können. Für die Realisierung der Seitenleiste des Hauptfensters wird, wie schon bei der vorigen Version des Scaffold Hunters, die `JTaskPane` verwendet.

6.3.5. JGoodies

Kurzbeschreibung JGoodies Forms stellt Klassen zur leichten Generierung von Dialogen bereit. So kann man z.B. Button-Leisten erzeugen, bei denen die Reihenfolge der OK- und Abbrechen-Buttons automatisch an das Betriebssystem angepasst wird. Weiterhin lassen sich formulorartige Komponenten bequem erstellen.

Herausgeber: JGoodies

Homepage: http://www.jgoodies.com/freeware/forms/index.html

Lizenz: BSD Lizenz

aktuelle Version (23.3.2011): 1.4

verwendete Version: 1.0.7

Verwendungszweck Da der Startdialog auf dem des alten Scaffold Hunters basiert und dort diese Bibliothek eingesetzt wurde, wird sie weiterhin verwendet. Zusätzlich zu den bereits vorhandenen Anwendungsgebieten bei der Erstellung des Start- und des Datenbankprofil-Dialogs wird sie auch für das Layout des Passwortdialogs verwendet.

6.3.6. JDOM

Kurzbeschreibung JDOM bietet eine einfache Schnittstelle zu XML-Dateien.

Herausgeber: JDOM Project

Homepage: http://www.jdom.org/

Lizenz: Eigene im BSD/Apache-Stil

aktuelle Version (23.3.2011): 1.1.1

verwendete Version: 1.0

Verwendungszweck JDOM wird für die Speicherung der lokalen Datenbank-Profile verwendet. Der Grund für die Verwendung dieser XML-Bibliothek liegt darin, dass diese bereits bei dem vorigen Scaffold Hunter für die Speicherung der Benutzerprofile benutzt wurde und so eine rasche Umsetzung möglich war.

6.3.7. Simple Logging Facade for Java

Kurzbeschreibung *Simple Logging Facade for Java* (*slf4j*) ist eine dünne Abstraktionsschicht welche über verschiedenen Java Logging-Frameworks verwendet werden kann. Dies ermöglicht es unter anderem Java Bibliotheken mit Logging Support zu schreiben ohne sich auf ein bestimmtes Logging-Framework festzulegen.

Herausgeber: QOS.ch

Homepage: `http://www.slf4j.org`

Lizenz: MIT License

aktuelle Version (31.3.2011) 1.6.1 (Release: 05.07.2011)

verwendete Version: 1.6.1

Wahl des Logging-Frameworks Nachdem der Beschluss gefasst wurde ein Logging-Framework zur Ausgabe von Debugging Informationen und zum loggen von Fehlern zu verwenden, haben wir verschiedene Logging-Frameworks evaluiert, darunter *log4j*, *slf4j* und die in der Java Klassenbibliothek enthaltene *Java Logging API*. Wir haben uns schließlich für *slf4j* über der *Java Logging API* entschieden, da wir die Einstellungsmöglichkeiten der *Java Logging API* für unsere Zwecke als ausreichend erachtet haben. *log4j* bietet noch eine Vielzahl weiterer Funktionen und Konfigurationsmöglichkeiten die wir jedoch nicht benötigen, etwa Logging per Mail oder über einen *syslog-Daemon*.

Die Verwendung von *slf4j* bietet nicht nur den Vorteil, dass wir das darunter liegende Logging-Framework problemlos austauschen können, falls wir z.B. doch die erweiterten Möglichkeiten von *log4j* nutzen wollen, sondern es erlaubt auch eine einfache Verwendung von *Formatstrings*, eine Funktionalität welche die *Java Logging API* von sich aus nicht anbietet und welche die Konstruktion aussagekräftiger Lognachrichten erleichtert.

6.3.8. Clustering

Um dem Benutzer möglichst viele Clustermethoden zur Verfügung zu stellen und Entwicklungszeit zu sparen, haben wir zu Beginn der Projektgruppe nach bereits bestehenden Clustering-Frameworks recherchiert. Unsere Hauptanforderungen an ein Framework waren Verfügbarkeit unter einer Open-Source-Lizenz, Unterstützung für effizientes hierarchisches Clustering von 10.000 bis 100.000 Molekülen und nach Möglichkeit Unterstützung von in der Chemoinformatik benutzten Distanzmaßen (z.B. Tanimoto-Koeffizient).

Leider gab es für die Sprache Java eine sehr begrenzte Anzahl an Datamining-Frameworks, so dass uns im wesentlichen Weka[6], RapidMiner[7] und Elki[8] zur Auswahl standen.

Weka Weka bietet zwar eine Implementierung eines hierarchischen Clusterings mit diversen Linkage-Verfahren an, bei unserem Performance-Test[9] der mit Weka 3.6 mitgelieferten GUI stellte sich aber heraus, dass der Clustering-Algorithmus den Dienst wegen zu wenig Hauptspeicherplatz versagte.

RapidMiner RapidMiner implementiert einen hierarchischen Clusteringalgorithmus mit kubischer Laufzeit und benötigte für unseren Performance-Test 43 Minuten. Zum Vergleich: Die proprietäre Datenanalysesoftware SPSS[10] benötigte hierfür lediglich 5 Minuten und implementiert unseren Vermutungen zu Folge einen quadratischen Algorithmus.

Elki Elki bietet eine Vielzahl von Clusteringalgorithmen, da es als Benchmarksuite für verschiedene Clusteralgorithmen entwickelt wurde. Unter anderem ist eine Implementierung mit quadratischer Laufzeit[11] für das hierarchische Clustering mit der Single-Linkage-Methode enthalten. Elki ist modular aufgebaut und erlaubt das Erweitern durch beliebige Distanzfunktionen. Weiterhin unterstützt Elki den effizienten Zugriff auf die zu clusternden Datenpunkte durch verschiedene Indexstrukturen. Obwohl der Quelltext von Elki offen im Internet verfügbar ist, steht Elki nicht unter Open-Source-Lizenz. Nach Aussage des von uns kontaktierten Mitarbeiters konnten sich die Entwickler noch nicht auf eine einheitliche Lizenz für das gesamte Framework einigen. Allerdings wurde uns eine Freigabe unter Open-Source-Lizenz durch die einzelnen Autoren in Aussicht gestellt, wenn wir die entsprechenden Teile benennen.

[6] http://www.cs.waikato.ac.nz/ml/weka/
[7] http://rapid-i.com/
[8] http://www.dbs.ifi.lmu.de/research/KDD/ELKI/
[9] Testdaten: 10.000 Beobachtungen mit jeweils 5 Attributen, Testsystem: aktueller Dual Core CPU mit 2,53 GHz, Start der JavaVM mit 4 GB max. Heapgröße
[10] IBM SPSS Statistics: http://www.spss.com/de/software/statistics/
[11] http://www.dbs.ifi.lmu.de/research/KDD/ELKI/release0.3/doc/de/lmu/ifi/dbs/elki/algorithm/clustering/SLINK.html

Fazit Da Weka und RapidMiner auf Grund der für unsere Anforderungen nicht ausreichenden Performanz ausscheiden und die Lizenz des ansonsten recht vielversprechenden Elki ungeklärt ist haben wir eine Eigenimplementation vorgenommen.

6.3.9. Guava: Google Core Libraries for Java

Kurzbeschreibung Die Java Klassenbibliothek ist verglichen mit vielen anderen Sprachen sehr reichhaltig, trotzdem ist sie durchaus noch verbesserungsfähig. So fehlen zum Beispiel die unter anderem aus der STL bekannten Klassen Multiset und Multimap und beim Programmieren schreibt man öfter relativ ähnliche Dreizeiler welche sich gut durch eine Bibliotheksfunktion ersetzen ließen, etwa beim Prüfen von Methodenargumenten oder beim Iterieren über Collections.

An dieser Stelle setzen Bibliotheken wie Guava oder Apache Commons an und stellen weitere Klassen und statische Methoden zur Verfügung, um Lücken in der Standardbibliothek zu schließen.

Herausgeber: Google

Homepage: `http://code.google.com/p/guava-libraries/`

Lizenz: Apache License Version 2.0

aktuelle Version (1.9.2011) r09 (Release: 07.04.2011)

verwendete Version: r09

Verwendungszweck Wie bei einer Erweiterung der Standardbibliothek kaum anders zu erwarten werden verschidene Guava Klassen im Scaffold Hunter an unterschiedlichen Stellen verwendet.

Viele Methoden verwenden `Preconditions` zur Überprüfung von Argumenten. An mehreren Stellen werden weiterhin `Multimaps` und Immutable Collections verwendet. Der von mehren Ansichten verwendete SVG Cache ist im wesentlichen eine mittels `MapMaker` erstellte *Computing Map* mit *soft Values*. Desweiteren wurden mehrfach Hilfsmethoden aus der `Iterables` Klasse eingesetzt anstatt das entsprechende Verhalten in einer Schleife auszuprogrammiern.

Zu guter letzt werden oftmals anstelle der generischen Collection Konstruktoren, die von Guava zur Verfügung gestellten, kompakteren *statischen Factorymethoden* verwendet. Diese ermöglichen es auf die erneute Angabe des Typarguments zu verzichten, falls dieses, wie etwa bei einer direkten Zuweisung inferiert werden kann, was insbesondere bei langen Typargumenten kompakteren Code ermöglicht[12]

[12] Der in Java 7 eingeführte *Diamantoperator* erfüllt die gleiche Aufgabe, jedoch wurde der Scaffold Hunter unter Java 6 entwickelt.

6.3.10. CDK

Kurzbeschreibung Das Chemistry Development Kit (CDK) ist eine sehr umfangreiche, in JAVA geschriebene Bibliothek für die Behandlung chemischer Daten. Sie enthält unter anderem Datenstrukturen zur Repräsentation von Proteinen, Molekülen und deren Bestandteilen, sowie von Reaktionen zwischen Molekülen und bietet die Möglichkeit, auf diesen zum Beispiel verschiedene Ähnlichkeitsmaße zwischen Molekülen oder chemische Eigenschaften von Molekülen zu berechnen.

Herausgeber: Egon Willighagen, Rajarshi Guha, Christoph Steinbeck

Homepage: `http://cdk.sourceforge.net/`

Lizenz: LGPL 2.1

aktuelle Version (27.9.2011) 1.4.4 (Release: 27.09.2011)

verwendete Version: Eigene Version

Verwendungszweck Das CDK wird im Scaffold Hunter für den Export und Import von Molekülen im SDF-Format, für die Generierung von SMILES-Strings zur Identifikation der Moleküle beim Import und bei der Baumgenerierung, sowie für das Erzeugen von Fingerprints verwendet.

Vorgehensweise Im Laufe der Projektgruppe hat sich leider herausgestellt, dass die Geschwindigkeit des CDK nicht für die anvisierte Datensatzgröße von einigen Zehntausend Molekülen reicht. So hat der Import größerer Datensätze mit der ursprünglich verwandten Version 1.2.8 einige Stunden gedauert.

Nach einer Untersuchung des Quellcodes der Bibliothek stand fest, dass diese nie auf Geschwindigkeit optimiert wurde und sich leicht beschleunigen lässt. Es wurde daher ungefähr ein Monat von einem Projektgruppenmitglied darauf verwandt, Patches für diese Bibliothek zu entwickeln.

Zur Entwicklung der Patches wurde ein Profiling-Tool verwendet um die für die hohe Import-Dauer verantwortlichen Teile des CDK zu ermitteln. Da jeweils bei der Entwicklung eines Patches nur der Einfluss auf einen einzelnen Datensatz untersucht wurde, erfolgte nach Fertigstellung der Patches ein umfangreicher Test der Auswirkungen der einzelnen Patches auf verschiedene Algorithmen des CDK. Diese Untersuchung ist insbesondere für die Integration der entwickelten Patches in das offizielle CDK von Bedeutung.

Um sicherzustellen, dass durch die entwickelten Patches keine Fehler entstanden sind, wurde am Ende ein Vergleich der durch das veränderte CDK erzeugten SMILES-Strings eines Datensatzes mit den originalen SMILES durchgeführt.

Da im Laufe der Projektgruppe die Version 1.4.0 des CDK veröffentlicht wurde und die Patch-Entwicklung auf Basis einer aktuellen Revision aus dem offiziellen git-Repository des CDK erfolgte, wurde der Scaffold Hunter auf die Version 1.4.x umgestellt. Dadurch

ist es auch möglich, jederzeit auf die neueste Version umzusteigen, indem diese ausgecheckt wird und die entwickelten Patches, soweit noch nicht im offiziellen CDK enthalten, angewendet werden.

Änderungen Es wurden folgende Patches entwickelt:

0001-made-the-creation-of-morgan-numbers-N-times-faster Bei der Erstellung des SMILES-Strings eines Moleküls werden im CDK Morgan-Zahlen zur Beschreibung des Graphen, der sich aus den Atomen und den Bindungen eines Moleküls ergibt, verwendet. Die Morgan-Zahlen entstehen durch sukzessives Kumulieren von Knotenmarkierungen der Nachbarn eines Knoten. Dabei werden die Knotengrade als initiale Markierungen verwendet und anschließend n Mal die neuen Markierungen durch Aufsummieren der Nachbarmarkierungen gebildet, wobei n die Anzahl der Atome des Moleküls ist. Dadurch hat jeder Knotengrad Einfluss auf die Morgan-Zahl eines jeden Knoten, so dass die Morgan-Zahlen die Topologie des Graphen kodieren. Die dabei enstandenen Zahlen dienen der Festlegung einer eindeutigen Reihenfolge der Atome bei der Generierung von kanonischen SMILES-Strings.

Leider wurde bei dem originalen CDK in jedem Iterationsschritt für jeden Knoten die Nachbarschaft erneut mit der Methode `getConnectedAtomsList()` aus `AtomContainer` bestimmt, wobei diese wiederum alle Knoten auf die Nachbarschaft zu dem aktuellen Knoten hin untersucht hat.

In der gepatchten Version werden die Nachbarschaften einmal ermittelt und während den Iterationsschritten nur noch auf die gespeicherten Werte zugegriffen.

Da ein Atom in einem Molekül typischerweise an sehr wenigen Bindungen beteiligt ist, hat die Methode `getMorganNumbers()` jetzt eine Laufzeit von $O(n^2)$ statt der vorigen $O(n^3)$.

Dieser Patch ist bereits Teil des offiziellen CDK.

0002-reduced-the-number-of-morgannumber-calculations Die Morgan-Zahlen wurden an verschiedenen Stellen in den Klassen `SmilesGenerator` und `BondTools` berechnet, die bei der SMILES-Generierung jeweils für jedes Atom des Moleküls aufgerufen wurden. Dies war unnötig, da das Molekül während der SMILES-Berechnung nicht verändert wird. Deshalb wurden die entsprechenden `getMorganNumbers()`-Aufrufe in die `createSMILES*()`-Methoden verschoben und die Resultate an die Methoden, in denen diese verwendet werden, weitergeleitet. Dadurch werden die Morgan-Zahlen nur noch konstant oft, statt wie bisher n Mal, berechnet.

Desweiteren enthält dieser Patch einige kleinere Verbesserungen, die ebenfalls mehrfache Berechnungen entfernen.

0003-improved-speed-of-AtomContainer-queries Die Ermittlung der Nachbarschaft eines Atoms wird nicht nur bei der Berechnung der Morgan-Zahlen, sondern wie die anderen Methoden aus `AtomContainer` auch an vielen anderen Stellen eingesetzt.

Da der `AtomContainer` die Atome und Bindungen nur in einem Array speichert, haben diese Methoden jeweils eine lineare Laufzeit.

Dieser Patch verringert die Laufzeit der Anfragen an den `AtomContainer`, indem dieser intern die Bindungen der Atome sowie die Indizes der Atome und Bindungen in `HashMap`s speichert.

0004-made-Path.getIntersectionSize-much-faster Bei der Suche nach allen Ringen eines Moleküls werden iterativ Pfade ermittelt und Pfadpaare durch Ermittlung ihrer Schnittmenge als Ring-Kandidaten bestimmt.

Dieser Patch beschleunigt die dabei verwendete Schnittgrößenbestimmung durch den Einsatz eines `HashSet`s. Dadurch muss in `getIntersectionSize()` nicht mehr der gesamte andere Pfad durchlaufen werden und die Laufzeit der Methode sinkt von $O(m * n)$ auf $O(min(n, m))$.

0005-AtomContainer-no-longer-allows-direct-manipulation Dieser Patch ändert lediglich die Sichtbarkeit der Arrays in `AtomContainer` und passt die Klasse `Ring` dementsprechend an.

0006-moved-RingSet.contains-to-AtomContainerSet Hier werden die `AtomContainer`-Array-Indizes in einer `HashMap` im `AtomContainerSet` gespeichert. Dadurch kann die in `RingPartitioner` und damit indirekt auch in SmilesGenerator verwendete Methode `RingSet.contains()` durch eine einfache Anfrage an diese `HashMap` in `AtomContainerSet` ersetzt werden. Weiterhin lassen sich auch die `get-` und `setMultiplier()` Methoden dort beschleunigen, da der Index des `AtomContainers` bereits feststeht und nicht über alle `AtomContainer` iteriert werden muss.

Allerdings können durch diesen Patch nicht länger mehrere Kopien einer Instanz in einem `AtomContainerSet` vorhanden sein, was von `AtomContainerSetTest` vorausgesetzt wird.

0007-added-a-version-of-getIntersectionSize Dieser Patch fügt eine Variante von `getIntersectionSize()` in `Path` hinzu, die stoppt, sobald die Schnittgröße einen bestimmten Wert erreicht. Dadurch lassen sich Anfragen der Art `intersectionSize >= x` effizienter realisieren.

0008-made-the-search-for-all-rings-much-faster Dieser Patch verbessert die an sehr vielen Stellen des CDK, insbesondere bei der SMILES-Generierung, verwendete Suche nach allen Ringen. Bei der Suche nach alle Ringen wird über alle Atome des Moleküls iteriert, wobei die an dem jeweiligen Atom beginnenden Pfade kombiniert und wie bei dem vierten Patch beschrieben auf entstandene Kreise untersucht werden.

Ursprünglich wurde hier in jedem Schritt der Knoten mit dem kleinsten Grad bestimmt und alle Pfade darauf untersucht, ob sie an diesem beginnen und ob sie einen Kreis bilden. Dabei wurde, sobald ein Pfad gefunden wurde, der an dem Knoten beginnt, alle nachfolgenden Pfade in der Pfadliste jedesmal erneut darauf

untersucht, ob sie auch an dem Knoten beginnen. War dies der Fall, wurde die exakte Schnittgröße beider Pfade bestimmt und die Vereinigung der beiden Pfade je nach Schnittgröße in die Liste der neuen Pfade oder die der potentiellen Kreise eingefügt. Hierbei ist lediglich ein Schnitt der Größe eins oder zwei interessant, da die beiden Pfade sonst keinen Kreis bilden können. Nach der Untersuchung aller Pfade wurden die nicht mehr benötigten Pfade aus der Pfadliste gelöscht, die neuen Pfade eingefügt und der Knoten nach der Untersuchung der potentiellen aus dem Graphen entfernt.

Durch den Patch wird zunächst die langsame Bestimmung des Knoten mit dem kleinsten Grad während der Iteration durch eine Sortierung der Knoten nach dem Knotengrad vor der Iteration ersetzt, so dass die Gesamtlaufzeit für diesen Teil von $O(n^2)$, mit dem originalen `AtomContainer` aufgrund `getConnectedBondsCount()` sogar $O(n^3)$, auf $O(n * \log(n))$ sinkt. Dadurch entspricht die Reihenfolge genau der, die auch in dem Artikel [10] verwendet wird, auf den in den Kommentaren der Klasse `AllRingsFinder` verwiesen wird.

Weiterhin werden pro Knoten zunächst die Pfade der Liste bestimmt, die an diesem beginnen. Dadurch werden Pfade, die dies nicht tun, nur noch einmal statt für jeden dort beginnenden Pfad betrachtet.

Für die Berechnung der Schnittgröße wird die im vorigen Patch eingeführte Methode verwandt, da Pfade mit mehr als zwei gemeinsamen Knoten keinen Kreis bilden können und so die genaue Schnittgröße nicht benötigt wird.

Nach der Untersuchung aller Pfade werden diese in ein `LinkedHashSet` eingefügt, da dort schneller gelöscht werden kann als in der ursprünglich verwendeten `ArrayList` von Pfaden. `LinkedHashSet` wurde `HashSet` vorgezogen, da dieses die Reihenfolge erhält und sonst bei mehrfacher SMILES-Generierung durch die möglicherweise unterschiedliche Pfadreihenfolge verschiedene SMILES entstehen können. Die unterschiedliche Knotenreihenfolge hat auch einen Einfluss auf die SMILES-Generierung, allerdings ist diese Reihenfolge deterministisch, so dass auch immer die gleichen SMILES erzeugt werden.

Darüberhinaus wird bei der Untersuchung der potentiellen Ringe in `detectRings()` nicht mehr das erste Element gelöscht, da dies in dem verwendeten `Vector` sehr langsam ist. Auch die Atome werden nicht mehr aus dem `AtomContainer` gelöscht, da dies ebenso langsam ist und aufgrund der neuen Bestimmung der Reihenfolge nicht mehr nötig ist.

0009-only-calculate-morgan-numbers-when-they-are-needed Die Berechnung der Morgan-Zahlen in `SmilesGenerator` wurde verschoben, da diese nur nötig sind, wenn chirale SMILES generiert werden sollen.

0010-only-search-for-all-rings-if-we-re-generating-chiral-smiles Auch die Ringe werden nur benötigt, wenn chirale SMILES generiert werden.

0011-multipliers-are-now-stored-as-doubles In dem `AtomContainerSet` befinden sich `multiplier`, die als `Double` gespeichert und durch die vom Scaffold Hunter aufge-

rufenen Methoden nicht verwendet werden. Dieser Patch spart dadurch Speicherplatz, dass der Typ der `multiplier` von `Double` auf `double` geändert wird. Dies ist möglich, da die `multiplier` im CDK nur Zahlenwerte annehmen und keine `null`-Einträge auftauchen.

0012-changed-Double-to-double-everywhere Dieser Patch korrigiert weitere unnötige Vorkommen von `Double` zu `double`.

0013-now-that-the-multipliers-have-the-same-type Da jetzt alle `multiplier`-Arrays wieder den gleichen Typ verwenden, kann das effizientere `System.arraycopy()` statt der for-Schleifen verwendet werden.

Die restlichen Patches ergänzen lediglich die Lizenzangaben um den Autor und fügen einige fehlende JavaDoc-Beschreibungen von Parametern hinzu.

Korrektheit Neben den möglichen Programmierfehlern führt der achte Patch zu einer Änderung im Verhalten des CDK, da dieser durch die geänderte Strategie der Auswahl der Atomreihenfolge in `AllRingsFinder` auch die gefundenen Kreise in anderer Reihenfolge zurückgibt, die auch die erzeugten SMILES-Strings beeinflussen kann. Um sicherzustellen, dass das CDK auch weiterhin gültige SMILES-Strings erzeugt, wurde ein Korrektheitstest durchgeführt.

Dieser erfolgte durch den Vergleich des Resultates eines Datenimports gefolgt von einer Baumgenerierung mit der offiziellen Version 1.4.2 sowie derselben Version, auf die alle entwickelten Patches angewendet wurden. Dazu wurden zwei Patches für den Scaffold Hunter entwickelt, die bei der SMILES-Generierung in der Klasse `CanonicalSmilesGenerator` sowie bei der Baumgenerierung in `ScaffoldTreeGenerator` die SMILES-Strings der erzeugten Moleküle und die Datenbank-ID des Moleküls, das gerade in den Scaffoldbaum eingefügt wird, ausgeben. Für die Tests wurde Eclipse so konfiguriert, dass die Konsolenausgabe des Scaffold Hunters in einer Log-Datei gespeichert wird. Anschließend wurde der Datensatz pubchem_substance_data_600.sdf mit den beiden zu vergleichenden CDK-Versionen importiert, so dass für jede CDK-Version ein Datensatz in der Datenbank vorhanden war. Für jeden dieser Datensätze wurde dann jeweils nach einem Neustart des Scaffold Hunters der Baum in den Standardeinstellungen generiert, so dass für beide Baumgenerierungen je eine Log-Datei vorhanden war. Um diese beiden Dateien vergleichen zu können, wurde ein kleines Python-Skript geschrieben. Dieses kopiert alle Dateien mit der Endung *.log des aktuellen Verzeichnisses und sortiert dabei die Dateiinhalte nach den Datenbank-IDs, so dass die kopierten Log-Dateien dann die Moleküle und die dazu generierten Scaffolds aufgrund der fortlaufenden ID-Vergabe in der Importreihenfolge enthalten. Diese Sortierung ist nötig, da die Moleküle bei der Baumerzeugung in zufälliger Reihenfolge durchlaufen werden. Die beiden sortierten Log-Dateien wurden dann mit einem Vergleichswerkzeug untersucht. Dabei sind lediglich die erwarteten Unterschiede aufgefallen. Zum einen treten vertauschte SMILES-Paare auf. So wird zum Beispiel mit dem gepatchten CDK zunächst ein Scaffold mit dem SMILES c1ccc(cc1)c2cc[nH]n2 erzeugt, gefolgt von c1ccc(cc1)c2ccn[nH]2, während diese mit dem originalen CDK in der

umgekehrten Reihenfolge auftreten. Der Grund hierfür liegt darin, dass bei der Baumerzeugung anhand einiger Regeln ein Parent-Scaffold des aktuell betrachteten Moleküls oder Scaffolds aus allen möglichen Parents ausgewählt wird. Falls die Regeln keine eindeutige Auswahl zulassen, wird das lexikographisch kleinste Scaffold gewählt, wofür wiederum die SMILES erzeugt werden müssen.

Weiterhin werden durch die zufällige Reihenfolge der Moleküle einige Parent-Scaffolds bei der einen Baumerzeugung erst dann untersucht, wenn sie bereits Parent eines anderen Moleküls oder Scaffolds sind, während dieser Scaffold bei der anderen Baumerzeugung ausgehend vom selben Molekül zum ersten Mal betrachtet wird, und so auch dessen Vorgänger-Scaffolds erzeugt werden. Dies äussert sich in dem Vergleich durch scheinbar zusätzlich generierte SMILES in beiden Log-Dateien.

Da sich diese zusätzlich erzeugten SMILES unterscheiden könnten und innerhalb der Log-Dateien nicht leicht vergleichen lassen, wurden mit dem Befehl `grep 'smiles' logfile | sort > smilesfile` die Scaffold-SMILES aus beiden Dateien extrahiert und lexikographisch sortiert in Dateien gespeichert. Die beiden resultierenden Dateien wurden dann wieder miteinander verglichen. Hierbei traten wieder einige wenige zusätzliche SMILES in beiden Dateien auf. Durch die Sortierung ließ sich hier aber erkennen, dass es sich bei den dazugekommenen SMILES fast nur um mehrfach auftretende SMILES handelt, die bei der einen Baumerzeugung einmal mehr oder weniger als in der anderen Baumerzeugung generiert wurden. Die einzige Ausnahme macht hier C1=COCCO1, das bei der Baumerzeugung mit dem originalen CDK auftritt, bei der gepatchten Version jedoch nicht. Dort tritt aber der SMILES C1CO=CO1 auf, der wiederum bei dem originalen CDK nicht vorkommt. Diese beiden SMILES wurden mit Hilfe von Daylight Depict [13] verglichen, mit dem Ergebnis, dass sie dieselbe chemische Struktur darstellen.

Da alle hier beschriebenen Unterschiede in derselben Art auch dann auftreten, wenn man zwei Baumerzeugungsvorgänge miteinander vergleicht, die mit dem originalen CDK auf demselben bereits importierten Datensatz durchgeführt wurden, kann man davon ausgehen, dass das gepatchte CDK keine Fehler enthält.

Geschwindigkeit Auf den Beispieldatensatz drugbank_approved.sdf angewendet ergab sich alleine durch den ersten Patch eine Beschleunigung des Imports von 3,5 Stunden auf etwa 6 Minuten. Nach Anwendung aller Patches fiel die Importdauer sogar auf etwa eine Minute.

Um die Auswirkung der einzelnen Patches auf die Geschwindigkeit zu untersuchen wurden die restlichen Patches in einige Gruppen thematisch ähnlicher Änderungen zusammengefasst. Diese Patchgruppen wurden jeweils auf das Referenz-CDK 1.4.2 angewendet, genauso wie ein Patch der CDK-Entwickler, der das Package `org.openscience.cdk.silent` einführt, welches eine schnellere Variante des `org.openscience.cdk.nonotify`-Moduls beinhaltet. Der erste Patch ist ab Version 1.4.2 im CDK enthalten, so dass dieser nicht mehr separat getestet wurde. Der Silent-Patch wurde hinzugenommen, da er insbesondere einen eigenen `AtomContainer` enthält, der so in Kombination mit den gesamten entwickelten Änderungen getestet werden kann. Die resultierenden CDK-Varianten sind

[13] http://www.daylight.com/daycgi/depict

Variante	enthaltene Patches
cdk-1.5.0.git-silent	Silent-Patch
cdk-1.5.0.git-custom	Patches 0002-0010 und Silent-Patch
cdk-1.5.0.git-custom-double-multipliers	Patches 0002-0013
cdk-1.5.0.git-double-multipliers	Patches 0006 und 0011-0013
cdk-1.5.0.git-fast-atomcontainer	Patches 0003 und 0005
cdk-1.5.0.git-fast-path	Patch 0004
cdk-1.5.0.git-fast-ringsearch	Patches 0004, 0007, 0008 und 0010
cdk-1.5.0.git-fast-ringset-contains	Patch 0006
cdk-1.5.0.git-fewer-morgannumbers	Patch 0002 und 0009

Tabelle 6.1.: CDK-Varianten

in Tabelle 6.1 abgebildet.

Die Geschwindigkeitsmessung erfolgte dann mit dem Japex-Framework [14], das auch von den CDK-Entwicklern verwendet wird. Es werden Treiber für die Ringsuche und die SMILES- und Fingerprint-Generierung verwendet, und jeweils für alle erzeugten CDK-Varianten die Zeit gemessen, die für die Verarbeitung einiger Testdatensätze benötigt wird. Die verwendeten Datensätze sind drugbank_approved.sdf, das bei der Patch-Entwicklung als Referenz verwendet wurde, pubchem_416_benzenes.sdf, das von den CDK-Entwicklern für Tests verwendet wird, sowie ein Datensatz aus ZINC [15] zur Kontrolle. Weiterhin wird ein Datensatz verwendet, der nur das Molekül Amatain enthält, da dieses mit dem ursprünglichen CDK nicht importiert werden konnte, weil die Ringsuche nach fünf Sekunden abbricht. In den Treibern wird dementsprechend die Zeitbegrenzung in `AllRingsFinder` deaktiviert.

Gemessen wurden die Zeiten für die Ringsuche sowie für die SMILES- und Fingerprint-Erzeugung. Die Ringsuche und die SMILES-Generierung werden an vielen Stellen des CDK eingesetzt, so dass Beschleunigungen dort sich an vielen Stellen auswirken sollten. Die beiden Fingerprinter wurden als indirekte Anwender dieser ausgesucht. Weiterhin werden `EStateFingerPrinter` und die SMILES-Erzeugung im Scaffold Hunter eingesetzt, so dass die Auswirkungen dort von besonderem Interesse sind.

Es wurde fast durchgehend eine sehr geringe Iterationsanzahl von 1 bis 15 verwendet, da einige der Tests so bereits sehr lange brauchen. Die fünfmalige SMILES-Generierung mit allen CDK-Varianten benötigt für den Drugbank-Datensatz zum Beispiel insgesamt 5 Stunden. Aufgrund der wenigen Iterationen sind geringfügige Unterschiede zwischen den CDK-Varianten nicht als repräsentativ anzusehen. Um hier genauere Ergebnisse zu erhalten, könnten die Tests mit einer größeren Anzahl an Durchläufen, die in der Testkonfigurationsdatei über den Parameter `runsPerDriver` eingestellt werden kann, wiederholt werden. Bei mehr als einem Durchlauf erhält man durch das Framework auch Angaben über die Standardabweichung.

[14] `http://japex.java.net/`
[15] `http://zinc.docking.org/`

Variante	Drugbank [ms]	Pubchem [ms]	Zinc [ms]	Amatain [ms]
nonotify	248689.213	4132.731	15806.181	114701.211
silent	254409.299	4259.789	15425.576	115247.147
custom	8493.619	1270.095	12233.542	430.305
custom-silent	39610.981	2121.249	13007.195	1208.499
custom-double-multipliers	8439.288	1273.906	12361.046	425.532
double-multipliers	251180.394	3984.659	15666.859	115054.906
fast-atomcontainer	150926.63	2696.033	14143.439	115258.420
fast-path	241119.849	3954.577	15756.877	22010.202
fast-ringsearch	250824.802	4086.774	15758.483	4639.626
fast-ringset-contains	248355.728	4170.812	15921.737	115476.322
fewer-morgannumbers	53334.577	2122.710	13291.674	113141.304
Iterationen	5	50	5	1

Tabelle 6.2.: Zeitmessungen der SMILES-Erzeugung

Die Ergebnisse der Zeitmessungen auf einem Core 2 Duo mit 2,4 GHz sind in den Tabellen 6.2, 6.3, 6.4 und 6.5 angegeben. Dabei entsprechen alle Varianten bis auf nonotify und custom-silent den in Tabelle 6.1 angegebenen. nonotify entspricht der silent-Variante, wobei die Tests aber mit dem `NoNotificationChemObjectBuilder` durchgeführt wurden, während silent selbst den `SilentChemObjectBuilder` verwendet. Dieser wird auch durch custom-silent verwendet, so dass dieses der custom-Variante mit allen Patches bis auf den veränderten `AtomContainer` entspricht. Für alle anderen Tests wurde der `NoNotificationChemObjectBuilder` eingesetzt.

Die SMILES-Erzeugung zeigt eine deutliche Beschleunigung von etwa 20% bei dem Zinc-Datensatz bis zu einer mehr als 250-fachen Geschwindigkeitssteigerung bei Amatain. Bei diesem Molekül wird dies durch die veränderte Ringsuche erreicht, während die anderen drei Datensätze von der geringeren Anzahl an Morgan-Zahl-Berechnungen und den schnelleren `AtomContainer`-Anfragen profitieren.

Bei der Ringsuche ergab sich für Amatain eine noch größere Beschleunigung als bei der SMILES-Erzeugung, wobei hier wieder dieselben Patches der Grund für diese Verbesserung sind. Diese führen auch bei den anderen Datensätzen zu deutlich kürzeren Laufzeiten. Überraschenderweise hat die Änderung am `AtomContainer` bei Pubchem und Zinc aber eine negative Auswirkung auf die Geschwindigkeit, so dass custom-silent dort schneller als custom ist und dieses bei dem Zinc-Datensatz sogar von dem originalen CDK geschlagen wird. Dies dürfte daran liegen, dass in diesen Datensätze weniger komplexe Moleküle vorhanden sind, bei denen die Ringsuche nicht von der Vorverarbeitung profitiert. Angesichts der sehr schnellen Ringsuche in diesen Molekülen ist dies aber zu verschmerzen.

Die Erzeugung eines 1024 Bit langen Fingerprints profitiert dagegen wie in Tabelle 6.4 zu sehen bei allen Datensätzen ausschließlich von dem verbesserten `AtomContainer`. Es ergibt sich dabei eine Beschleunigung von 20 bis 45%.

Variante	Drugbank [ms]	Pubchem [ms]	Zinc [ms]	Amatain [ms]
nonotify	12356.448	173.600	173.810	110336.433
silent	12363.697	176.614	173.968	111335.705
custom	615.969	147.771	212.194	202.210
custom-silent	512.600	107.957	150.071	157.291
custom-double-multipliers	574.616	148.303	215.199	205.715
double-multipliers	12403.973	178.646	180.481	111224.291
fast-atomcontainer	12754.71	229.490	253.511	111765.064
fast-path	2731.164	131.476	159.554	18750.770
fast-ringsearch	508.547	108.229	149.706	157.325
fast-ringset-contains	12426.925	179.976	180.898	111679.970
fewer-morgannumbers	12424.474	179.237	178.985	111754.115
Iterationen	15	25	25	1

Tabelle 6.3.: Zeitmessungen der Ringsuche

Variante	Drugbank [ms]	Pubchem [ms]	Zinc [ms]	Amatain [ms]
nonotify	21923.718	4908.307	7734.636	60.882
silent	21763.980	4908.109	7692.112	60.392
custom	11849.900	3346.248	5648.107	38.090
custom-silent	20652.513	5232.004	7648.147	62.745
custom-double-multipliers	11782.537	3363.983	5603.641	39.128
double-multipliers	20277.019	4911.452	7591.441	61.084
fast-atomcontainer	11804.432	3219.994	5592.924	38.445
fast-path	22211.985	4967.809	7642.871	61.206
fast-ringsearch	22326.322	4977.615	7943.196	62.653
fast-ringset-contains	21199.410	4984.421	7682.875	60.236
fewer-morgannumbers	21362.108	4960.121	7527.420	62.228
Iterationen	1	1	1	50

Tabelle 6.4.: Zeitmessungen der Fingerprint-Erzeugung

Variante	Drugbank [ms]	Pubchem [ms]	Zinc [ms]	Amatain [ms]
nonotify	31023.858	3761.550	7551.655	110661.689
silent	31197.324	3748.285	7566.127	111542.663
custom	18757.997	3802.485	7630.874	451.109
custom-silent	19523.052	3686.624	7487.447	472.977
custom-double-multipliers	19093.191	3842.622	7744.327	538.010
double-multipliers	31781.637	3860.931	7576.712	112929.231
fast-atomcontainer	31103.628	3869.128	7720.755	112833.161
fast-path	22514.561	3743.263	7526.324	19045.141
fast-ringsearch	19776.440	3750.328	7522.737	423.424
fast-ringset-contains	31876.223	3837.107	7653.357	112846.706
fewer-morgannumbers	31900.949	3791.115	7537.133	112869.561
Iterationen	1	1	1	1

Tabelle 6.5.: Zeitmessungen der EState-Fingerprint-Erzeugung

Verwendet man allerdings den `EStateFingerprinter` für die Erzeugung der Fingerprints, erhält man bei den Zinc und Pubchem-Datensätzen gar keine Beschleunigung. Amatain zeigt eine ähnliche Beschleunigung wie bei der Ringsuche, während der Drugbank-Datensatz zwar von den gleichen Patches profitiert, aber nur eine Beschleunigung um 40% gegenüber der 20-fachen Beschleunigung bei der Ringsuche zeigt.

Insgesamt stellen die entwickelten Patches eine deutliche Verbesserung dar, so dass der Import der Datensätze im Scaffold Hunter deutlich weniger Zeit benötigt und durch die schnellere Ringsuche auch Moleküle importiert werden können, für die dies mit dem unveränderten CDK nicht möglich ist. Auch die Fingerprint-Erzeugung profitiert von den Verbesserungen, wenn auch nicht bei allen Datensätzen und auch nicht in dem gleichen Maße wie die SMILES-Erzeugung und die Ringsuche.

Speicherverbrauch Während der Geschwindigkeitsmessungen ist bei Verwendung des `DefaultChemObjectBuilders` für die Erzeugung der Molekül-Instanzen ein sehr hoher Speicherverbrauch aufgefallen, der wie in Tabelle 6.6 dargestellt wird sogar von der Anzahl der Iterationen abhängt. Der Grund hierfür dürfte in den vielen Listener-Objekten liegen, die die erzeugten Moleküle, Atome usw. referenzieren und so ein Freigeben des Speichers bis zur Messung der nächsten Variante verhindern.

Bei der Verwendung von `NoNotification`- und `SilentChemObjectBuilder` dagegen trat dieser Effekt nicht auf. Der Speicherverbrauch blieb sowohl bei der Ringsuche als auch der SMILES-Generierung fast immer unter 50 MB, selbst bei 500 Iterationen der Ringsuche im Pubchem-Datensatz. Einzige Ausnahme war der Drugbank-Datensatz, der während der Ringsuche 150 MB und während der SMILES-Generierung 80 MB benötigte.

In der Tabelle ist auch ersichtlich, dass der beschleunigte `AtomContainer` wie erwartet durch die vielen `HashMaps` deutlich mehr Speicher benötigt. Überraschenderweise fällt

Variante	Pubchem [MB]	Zinc [MB]	Zinc [MB]
nonotify	367	94	320
silent	389	96	320
custom	3692	702	2954
custom-silent	3680	686	2956
custom-double-multipliers	3689	692	2944
double-multipliers	443	109	378
fast-atomcontainer	3532	657	2828
fast-path	394	99	350
fast-ringsearch	393	121	363
fast-ringset-contains	455	134	386
fewer-morgannumbers	404	133	360
Iterationen	50	5	25

Tabelle 6.6.: Speicherverbrauch mit `DefaultChemObjectBuilder` während der Ringsuche

dieser Effekt bei den Varianten fast-path, fast-ringsearch und fast-ringset-contains nur sehr gering aus, wobei diese Erhöhung auch auf eine Messungenauigkeit des verwendeten Werkzeugs VisualVM zurückzuführen sein könnte. Darauf deuten auch die unterschiedlichen Werte für nonotify und silent bei dem Pubchem-Datensatz hin.

Die beiden double-multipliers Varianten verwenden zwar wie erwartet etwas weniger Speicher als die entsprechenden Varianten ohne die letzten drei Patches, allerdings liegt diese Differenz innerhalb der beobachteten Messungenauigkeit. Bei deutlich komplexeren Molekülen mit mehr Ringen sollte sich der für die Ringsuche benötigte Speicherplatz mit diesem Patches jedoch deutlich reduzieren lassen. Ein Beispiel hierfür ist Dodecahydrocloso-dodecaborat [16], das für die Ringsuche weit mehr als 4 GB benötigt und aktuell vom Scaffold Hunter nicht importiert werden kann. Bei einer Ringsuche mit der custom-Variante belegten die in den `AtomContainerSets` gespeicherten `Double`-Werte 17% des insgesamt beanspruchten Speichers von 1 GB. Diese 174 MB ließen sich mit der custom-double-multipliers Variante auf 58 MB senken, was ungefähr 11% weniger Speicherverbrauch insgesamt bedeutet.

Mögliche weitere Verbesserungen Die Ringsuche ist ein vielversprechender Kandidat für weitere Verbesserungen. Es lassen sich aufgrund des Speicherverbrauchs und der teilweise extrem langen Dauer sehr viele Moleküle noch nicht mit dem Scaffold Hunter importieren. Ein Ansatz könnte die Suche nach den Ringen auf kleineren Teilen der Moleküle sein, so dass nur die gefundenen Ringe und die Pfade, die den Rand des untersuchten Gebietes berühren, im Speicher gehalten werden müssen. Die Pfade der benachbarten Gebiete müssten dann miteinander kombiniert und auf Kreise untersucht werden. Die Suche auf den Gebieten ließe sich wiederum parallelisieren. Auch wäre die Verwendung

[16] `http://www.ebi.ac.uk/chebi/advancedSearchFT.do?searchString=33594`

von Externspeicher für die Ringe und Pfade denkbar, so dass auch Moleküle importiert werden können, für die der vorhandene Hauptspeicher nicht ausreicht.

6.3.11. rangeslider

Kurzbeschreibung RANGESLIDER ist keine Bibliothek sondern lediglich eine Erweiterung der JSlider Klasse. Während ein JSlider über nur einen Regler - zur Eingabe eines einzigen Wertes - verfügt besitzt ein Rangeslider zwei Regler, um dadurch einen Wertebereich einstellen zu können.

Herausgeber: Ernest Yu

Homepage: http://ernienotes.wordpress.com/2010/12/27/creating-a-java-swing-range-slider/

Lizenz: MIT Lizenz

aktuelle Version (20.8.2011): initiales Release vom 21. Dezember 2010

verwendete Version: initiales Release vom 21. Dezember 2010

Verwendungszweck Beim Scatterplot werden Rangeslider verwendet, um den auf einer Achse dargestellten Wertebereich eingrenzen zu können.

6.3.12. Drag & Drop / Closeable Tabbed Pane

Kurzbeschreibung Der in Javas Swing-Bibliothek vorhandenen Tableiste fehlen viele Features, die Nutzer heutzutage aus diversen anderen Applikationen gewöhnt sind. Dazu gehört insbesondere die Möglichkeit, einzelne Tabs über einen 'X'-Button zu schließen, sowie die Reihenfolge von Tabs per Drag & Drop zu ändern.
Um diese Features im Scaffold Hunter nutzen zu können, wurden zwei unabhängige Erweiterungen der Standard-JTabbedTane kombiniert, und um einige zusätzliche Funktionen erweitert. Ausgangspunkte für die neue Klasse *DnDCloseableTabbedPane* waren die Klassen *CloseableTabbedPane* aus der XNap Commons Bibliothek, sowie *DnDTabbedPane* von Terai Atsuhiro.

Herausgeber: Felix Berger, Steffen Pingel (XNap); Terai Atsuhiro (DndTabbedPane)

Homepage: http://xnap-commons.sourceforge.net/
http://terai.xrea.jp/Swing/DnDTabbedPane.html

Lizenz: LGPL 2.1+ (XNap); Public Domain (DndTabbedPane)

Verwendungszweck Die erweiterte Tableiste wird für die Ansicht-Tabs im Hauptfenster verwendet. Dadurch lassen sich Tabs relativ bequem verschieben und schließen, lediglich das Verschieben zwischen verschiedenen Tableisten bzw. Fenstern ist nach wie vor nur per Menü möglich.

6.3.13. JSPF

Kurzbeschreibung Java Simple Framework (JSPF) ist ein auf Einfachheit getrimmtes Framework für Java, um zur Laufzeit nachladbare Plugins schnell realisieren zu können. Eine Pluginschnittstelle wird hier durch ein Java Interface definiert. Einzelne Plugins müssen dieses Interface bereitstellen und werden über eine Annotation als Plugin etabliert. Das JSPF gibt einem dann die Möglichkeit, Instanzen dieser Plugins als Objekt im Programm zu laden. (vgl. [1])

Herausgeber: R. Biedert und Nicolas Delsaux (sehr kleines OS Projekt)

Homepage: `http://code.google.com/p/jspf/`

Lizenz: BSD2 Lizenz

aktuelle Version: 1.0.2

verwendete Version: 1.0.2

Features

- Laden von Plugins aus JAR, Verzeichnissen, komplettem Classpath, per HTTP
- Threadsafe
- Typesafe
- Dependency Injection
- Sehr einfache Konfiguration
- Fast vollständig Annotation basierend
- Unterstützt Caching
- Plugins können isoliert werden
- Pluginexport über ERMI, LipeRMI, JSON, XMLRPC
- Autodiscovery von remote Plugins
- Appletkompatibel

(vgl. [1])

Gründe für den Einsatz von JSPF Bei der Recherche nach einem Sinnvollen Plugin Framework für den Scaffold Hunter wurde neben dem JSPF auch das Java Plugin Framework (JPF) und der Pluginstandard OSGi, sowie die Möglichkeit einer Eigenimplementation evaluiert. JPF hatte einen etwas größeren Funktionsumfang als das JSPF, wird aber anscheinend nicht mehr weiterentwickelt, hier gab es das letzte Release in 2007. OSGi bietet zwar einen sehr großen Funktionsumfang und ist als Standard etabliert, ist jedoch für den Einsatz im Scaffold Hunter deutlich überdimensioniert. Bei einer Eigenimplementation wäre zwar genau bekannt, wie das Pluginframework arbeitet und somit eventuelle spätere Erweiterungen daran vermutlich leichter möglich, der Implementationsaufwand für eine stabile Lösung wäre jedoch zu hoch. Die Wahl fiel auf JSPF, insbesondere aufgrund der sehr einfachen Struktur, der Tatsache, dass die von uns benötigten Features stabil implementiert und dieses Projekt zum Entwicklungszeitpunkt als lebendig angesehen werden konnte.

Verwendungszweck Das JSPF wird verwendet, um leichte Anpassungen an weitere Daten- und Berechnungsquellen zu ermöglichen, es stellt die Basis für sowohl die Import- als auch für die Berechnungsplugins (Calcplugins) sowie deren Verwaltung dar.

6.3.14. exp4j

Kurzbeschreibung Exp4j ist ein einfacher Parser zur Auswertung mathematischer Ausdrücke.

Herausgeber: Frank Asseg

Homepage: `http://projects.congrace.de/exp4j/`

Lizenz: Apache License Version 2.0

aktuelle Version (1.9.2011) 0.2.8 (Release: 18.08.2011)

verwendete Version: 0.1.38

Wahl von exp4j Exp4j ist eine vergleichsweise kleine und einfache Bibliothek und für den Einsatz beim Datenimport voll und ganz ausreichend. Weiterhin ist der Code relativ überschaubar und gut verständlich.

Verwendungszweck Beim Datenimport ist es möglich einen mathematischen Ausdruck anzugeben um etwa die importierten Werte zu logarithmieren *exp4j* wird eingesetzt um diese Ausdrücke zu parsen und anzuwenden.

6.3.15. Open CSV

Kurzbeschreibung OpenCSV ist eine kleine OpenSource Bibliothek mit der man CSV (Comma-Separated Values) Dateien lesen und schreiben kann. Hierbei unterstützt es beliebige Trenn- und Quotezeichen.

Herausgeber: Glen Smith, Sean Sullivan, Scott Conway (sehr kleines OS Projekt)

Homepage: `http://opencsv.sourceforge.net/`

Lizenz: Apache 2.0

aktuelle Version: 2.3

verwendete Version: 2.3

Features

- Lesen von CSV Dateien
- Wechselnde Anzahl von Datenfeldern pro Zeile
- Trenn- und Quotationzeichen können konfiguriert werden
- Komplettes Dokument vollständig einlesen oder in Iteratorweise
- Erstellen von CSV Dateien

Gründe für den Einsatz von OpenCSV OpenCSV machte einen stabilen Eindruck und die Einarbeitungszeit war sehr kurz. Ferner waren die Tatsache der leichten Konfigurierbarkeit sowie die anscheinend aktive Entwicklung zentrale Entscheidungsgründe für den Einsatz von OpenCSV.

Verwendungszweck OpenCSV wird im Scaffold Hunter für den Datenexport in CSV Dateien (z.B. für eine Weiterverarbeitung in Tabellenkalkulationen) sowie im CSV Import Plugin für das Lesen von CSV Dateien verwendet.

6.3.16. XStream

Kurzbeschreibung XStream ist eine Bibliothek, die die Serializierung und Deserialisierung von Java-Objekten in Form von XML (oder JSON) ermöglicht.
Während das in Java eingebaute *Serializable*-Interface nur die Serialisierung als Binärdaten unterstützt, erzeugt XStream ohne nennenswerten Mehraufwand relativ saubere und lesbare XML-Daten aus fast beliebigen Java-Objekten. Dazu sind im Allgemeinen keinerlei Änderungen an den zu serialisierenden Klassen erforderlich. Bei Bedarf können eigene Converter-Klassen geschrieben werden, die eine größere Kontrolle über die XML-Baumstruktur der serialisierten Daten erlauben.

Herausgeber: XStream Project

Homepage: `http://xstream.codehaus.org/`

Lizenz: BSD Lizenz

aktuelle Version (08.09.2011): 1.4.1 (Release: 11.08.2011)

verwendete Version: 1.4.1

Verwendungszweck Die XStream-Bibliothek wird beim Speichern von Sitzungen und Einstellungen in der Datenbank an den Stellen verwendet, wo eine Modellierung der entsprechenden Datenstrukturen direkt im Datenbankschema nicht sinnvoll erschien. Dies umfasst im Wesentlichen die Konfigurationseinstellungen, den Zustand der Hauptfenster, sowie den Zustand der einzelnen Ansichten. Im Fall der Dendrogrammansicht umfasst dies auch das komplette Ergebnis des Clusterings.

6.4. Interne Datenhaltung

6.4.1. Modellentscheidungen

Da das alte Datenbankmodell aufgrund zahlreicher Änderungen nicht übernommen werden konnte, wurde ein vollständig neues Modell entwickelt und nur einige Grundideen aus dem alten Modell beibehalten. Es ist nun möglich, mehrere Bäume auf unterschiedlichen Mengen von Molekülen zu berechnen. In der bisherigen Version des Scaffold Hunters konnte pro Datenbankschema nur ein Datensatz von Molekülen importiert werden. Für andere Baumgenerierungsregeln musste ebenfalls ein neues Datenbankschema angelegt werden, wenn das alte nicht verloren gehen sollte. Zudem werden die Profile der einzelnen Benutzer mit ihren Presets, Programmeinstellungen und Programmzustandsinformationen zum Speichern und Wiederherstellen von Arbeitssitzungen(Sessions) in der Datenbank gespeichert. Kommentare und Schnellannotationen zu Molekülen und Scaffolds werden ebenfalls in der Datenbank gespeichert.

Im Zentrum des neuen Modells steht das `Dataset`. Solch ein `Dataset` steht für eine importierte Menge von Molekülen. Es enthält hauptsächlich eine Liste von Molekülen (in der Datenbank realisiert als 1:n-Beziehung vom Molekül zum `Dataset`) und eine Liste von dafür generierten Scaffoldbäumen. Neben anderen Feldern, die selbstredend sind und hier nicht erläutert werden, enthält ein `Dataset` außerdem noch die Eigenschaftsdefinitionen, die beschreiben, welche Eigenschaften mit welcher Bedeutung für die Moleküle und Scaffolds in diesem `Dataset` zur Verfügung stehen. Aufgeführte Eigenschaften sind aber nicht zwingend für alle Moleküle oder Scaffolds verfügbar. Es darf fehlende Werte geben. Die Eigenschaftswerte selbst werden in vier Tabellen abgelegt, aufgeteilt nach Scaffolds und Molekülen bzw. nach textuellen (String) und numerischen (Double) Werten. Dort wird neben dem Wert das Scaffold bzw. Molekül gespeichert, zu dem dieser Wert gehört. Zu den Scaffoldbäumen werden hauptsächlich die zugehörigen Scaffolds gespeichert. Diese wiederum enthalten ihre hierarchische Beziehung zueinander. Eine zusätzliche n:n-Beziehung zu den enthaltenen Molekülen ist notwendig, da ein Scaffold pro Baum mehrere Moleküle enthält und es mehrere Bäume pro `Dataset` geben kann. Dadurch, dass der gleiche Scaffold in unterschiedlichen Bäumen durch Filterung auf Molekülebene oder andere Baumgenerierungsregeln andere Moleküle besitzen kann, ist es außerdem notwendig, alle Scaffolds für jeden Baum neu zu speichern. Sowohl Moleküle als auch Scaffolds verweisen mit ihrer `Structure_ID` auf die Tabelle `Structure_Data`, in der Daten gespeichert werden, die sowohl für Scaffolds als auch für Moleküle existieren. Diese Verallgemeinerung hat den Vorteil, dass bei Fähnchen(`Banners`) und Kommentaren(`Comments`) nicht zwischen Molekülen und Scaffolds unterschieden werden muss.

Abbildung 6.1.: internes Datenbanklayout

Zusätzlich zu den auf ein `Dataset` bezogenen Daten werden auch die Benutzerprofile und ihre persönlichen Einstellungen in der Datenbank gespeichert. Die Benutzerprofile werden in der Tabelle `Profile_Data` gespeichert. Zu einem Benutzer werden die von ihm erstellten Kommentare abgelegt. Diese beinhalten neben dem Text und dem zugehörigen Molekül bzw. Scaffold auch die Information, ob sie öffentlich sind, also auch von anderen Nutzern gesehen werden können. Zusätzlich hat ein Benutzer eine Liste von Sessions, in denen der aktuelle Programmzustand samt aller erstellten Subsets gespeichert werden kann.

6.4.2. Umsetzung in Hibernate

Hibernate wurde in unserem Programm eingesetzt, um die Zuordnung zwischen Java Objekten und relationalen Datenbanksystemen zu ermöglichen. Auf Programmebene kann dann mit POJOs (*Plain Old Java Object*) gearbeitet werden.

Konfigurationsdateien Hibernate bietet zwei verschiedene Möglichkeiten, die Beziehungen zwischen den Klassen und der Datenbank zu modellieren. Zum einen ist dies in den Quellcodedateien über Annotationen und zum anderen über gesonderte XML Dateien möglich. Die Datenbankgruppe hat sich für die zweite Variante entschieden, da somit der Code etwas übersichtlicher bleibt und die Trennung von Programmlogik und Datenbankanbindung unserer Meinung nach sauberer ist.

Es gibt ein XML Wurzeldokument, in dem alle globalen Eigenschaften der Datenbankverbindung definiert werden. Dieses enthält unter anderem das Updateverhalten bei Veränderungen des Datenbankschemas oder Einstellungen für das Caching. Desweiteren werden in diesem Dokument alle weiteren XML Dokumente eingebunden, die die einzelnen Klassen- und Tabellenbeziehungen modellieren. Hibernate greift über Reflections auf die Attribute von POJO Klassen zu und verknüpft deren Inhalt mit einer Spalte in einer Tabelle eines Datenbankmodells. In jedem dieser Verknüpfungsdokumente steht also, welche Klasse welcher Tabelle im Datenbankmodell entspricht und welche Attribute auf welche Spalten abgebildet werden. Ein Datensatz aus dieser Tabelle entspricht dann genau einer Instanz dieser Klasse.

Neben reinen Attributen kann Hibernate auch Verknüpfungen von Klassen zueinander modellieren. Wenn Objekte auf andere Objekte oder Mengen von Objekten verweisen, können 1:1, 1:n und n:n Beziehungen verwendet werden. Die entsprechenden Zwischentabellen für die n:n Beziehungen legt Hibernate selbstständig an. Neben den Verknüpfungen werden in diesen Dateien auch Constraints für die Datenbank und das Ladeverhalten (z.B. Lazy Loading) definiert.

Austauschbarkeit des verwendeten Datenbanksystems Hibernate verfügt über eine automatische Erkennung des Datenbankdialektes. So kann jedes von Hibernate unterstützte Datenbanksystem ohne weiteren Konfigurationsaufwand verwendet werden. Zudem werden die Datenbankanfragen entsprechend dem verwendeten Datenbanksystem auf Leistung optimiert. Die Liste der unterstützten Datenbanksysteme umfasst alle bekannteren Datenbanksysteme. Nach einiger Überlegung hat sich die Datenbankgruppe

allerdings dennoch für eine manuelle Konfiguration des Datenbankdialekts entschieden. Dies lag daran, dass manche Datenbankfeatures wie Kaskadierung beim Löschen von Einträgen mit der automatischen Dialekterkennung nicht funktioniert haben.

6.4.3. Performance Optimierungen

Ein zentrales Ziel unserer Projektgruppe war es die Performance des Scaffold Hunter dahingehend zu optimieren, dass auch für sehr große Datensätze mit mehr als 100.000 Molekülen ein Arbeiten möglich ist. Wenn jedes dieser Moleküle mehr als 20 Eigenschaften (*Properties*) besitzen kann, multipliziert sich die Gesamtanzahl der Eigenschaftswerte auf über 2.000.000. Ein zentrales Problem war daher vor allem der Speicherverbrauch der Anwendung. Hinzu kam, dass im Gegensatz zum bisherigen Scaffold Hunter mehrere Ansichten auf ein und die selben Daten zugreifen müssen. Ein direktes Abfragen der Daten aus jeder Ansicht heraus würde also zu unnnötiger Datenbankkommunikation und Speicherplatzverbrauch führen. Da die verwendete Datenbank (vor allem im professionellen Umfeld) wahrscheinlich über ein Netzwerk zur Verfügung gestellt wird (und nicht lokal auf der Workstation installiert ist), ist zudem die Anzahl der SQL Anfragen ein entscheidendes Performancekriterium. Um die Balance zwischen dem Speicherplatzverbrauch und der Abfragehäufigkeit der Daten zu erhalten, haben wir daher das folgende Konzept entworfen.

Beim Programmstart wird als aller erstes das Profil eines Benutzers geladen. Das Profil des Benutzers mit all seinen Presets, Kommentaren und Fähnchen wird über die komplette Zeit nach dem Einloggen im Speicher gehalten. Hat der Benutzer danach eine Session ausgewählt, wird die Session mit dem dazugehörigen Dataset, den Eigenschaftsfdefinitionen, dem Baum, allen Subsets und den darin enthaltenen Molekülen und den Scaffolds zu den Molekülen und dem Baum geladen. Dabei werden Objektreferenzen zwischen den Subsets und den Molekülen so aufgelöst, dass die verschiedenen Subsets immer das gleiche Java Objekt referenzieren, wenn es sich um ein und dasselbe Molekül handelt. Dies ist nicht selbstverständlich, da die Daten aus der Datenbank gelesen werden müssen und somit überprüft werden muss ob das entsprechende Objekt schon im Programmspeicher existiert. Die vielseitige Referenzierung von Objekten war daher eines der Hauptprobleme in unserem Datenbankmodell und existierte für eine Vielzahl von Klassen. Ein Designziel des Datenbanklayouts war deshalb eine möglichst lose Kopplung der Objekte. Im Gegensatz zu den Molekülen, war dieses Vorgehen bei den Scaffolds nicht möglich. Die Eltern-Kind Beziehung der Scaffolds ändert sich mit jedem unterschiedlichen Subset, da nur ein Teil der Kinder in dem Subset sein muss. Für jede Scaffoldbaumansicht müssen daher alle Scaffolds dieses Baumes erneut in den Arbeitsspeicher geladen werden. Es befinden sich nun alle zum Betrieb des Scaffold Hunter erforderlichen Daten, außer den Eigenschaftswerten, im Arbeitsspeicher. Nicht benötigte Datasets, Bäume, Profile, Moleküle und Scaffolds werden nicht geladen. Um die Menge der zu ladenen Daten weiter reduzieren zu können, besteht die Möglichkeit vor dem Erstellen einer neuen Session die Moleküle und Scaffolds eines Datasets zu filtern. Das Wurzelsubset entspricht in diesem Falle nicht allen Molekülen des Datasets, sondern nur einer Teilmenge davon.

Da bei großen Datasets nicht alle Eigenschaften in den Arbeitsspeicher passen, werden

die Eigenschaften nur bei Bedarf geladen. Die Tabelle benötigt z.B. nur die Daten des aktuell sichtbaren Bereichs, der Clusteringalgorithmus nur die Eigenschaften über die geclustert werden soll. Damit auch hier nicht jede Ansicht die Daten einzeln laden muss, wurde ein locking Mechanismus implementiert. Jede Ansicht teilt der Datenbankverwaltung mit, welche Eigenschaften sie für welche Strukturen benötigt. Nach Gebrauch, gibt sie diese wieder frei. Das Besondere dabei ist, dass die Daten nur ein mal geladen werden müssen, wenn zwei oder mehr Ansichten gleichzeitig auf die Daten zugreifen. Dazu wird für jedes Molekül und jede Eigenschaft mitgezählt wie viele Locks es gibt.

Um die Anzahl der SQL Anfragen weiter zu reduzieren, wurden an möglichst vielen Stellen Batch Inserts, Updates und Selects verwendet. Die Datenbankschicht bietet also viele Funktionen, die für einen Menge von Molekülen, Scaffolds oder Eigenschaften funktionieren und diese dann mit einer festen Anzahl von SQL Anfragen (unabhängig von der Größe der Menge) abarbeiten.

Baumstrukturen sind ein weiteres Problem für die Anzahl der SQL Abfragen. In unserem Programm kamen diese unter anderem bei den Scaffolds vor. Hibernate würde standardmäßig damit Anfangen einen Knoten mit einer Anfrage zu laden. In einer nächsten Abfrage würden dann die Kinder dieses Knotens geladen. In der nächsten Abfrage die Kinder des ersten Kindes, ... Daher steigt die Anzhal der Abfragen linear zur Anzahl der Knoten. In diesen Fällen hat es sich als performanter herausgestellt, alle Knoten manuell in einer Abfrage zu laden und später manuell die Referenzen zwischen diesen zu setzten.

An vielen Stellen des Scaffold Hunter mussten Werte aus mehreren anderen Werten berechnet werden. Es ist z.B. möglich im Scaffoldbaum kumulierte Eigenschaften (Minimum, Maximum, Durchschnitt, etc) aller Moleküle zu einem Scaffold oder über den Unterbaum eines Scaffolds zu berechnen. Damit für diese Berechnung nicht erst alle Eigenschaften geladen und anschließend auf Programmebene berechnet werden müssen, bietet die Datenbankverwaltung an, diese Berechnungen direkt auf der Datenbank durchzuführen. Somit wird der Speicherbedarf verringert und die Performance der Berechnung erhöht.

Das folgende Beispiel dient als anschauliche Erklärung für eine Klasse weiterer Performanceoptimierungen in Bezug auf das Laden von Objekten. Um zu wissen welche Session geladen werden muss, braucht der Benutzer erst die Möglichkeit diese aus einer Liste von Sessions auszuwählen. Für diese Liste sind jedoch nur die Titel der Session und nicht das komplette Sessionobjekt plus referenzierter Objekte notwendig. Um das Laden jeder einzelnen Session für den Titel zu verhindern, wurde der Datenbankverwaltung eine Methode hinzugefügt, mit der sich eine Liste aller Sessiontitel für das aktuelle Profil abfragen lässt. Als zweiter Schritt kann dann mit Hilfe des Titels einer Session das Sessionobjekt geladen werden.

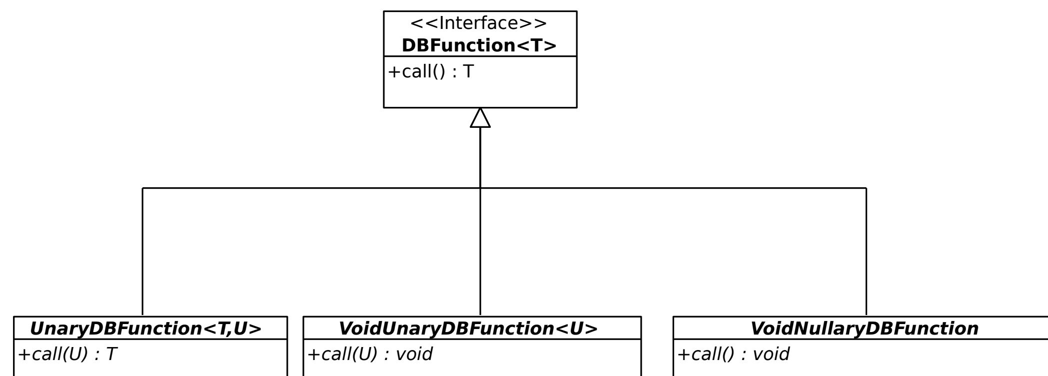
6.4.4. Zentrales Handling von Datenbank Exceptions

Der Zugriff auf die Datenbank erfolgt generell über Methoden der Klasse `DBManager`. Im Fehlerfall werfen diese Methoden `DatabaseExceptions`. Oftmals besteht im Fehlerfall der einzig sinnvolle Weg eine solche *Exception* zu behandeln darin, dem Benutzer

anzubieten den fehlgeschlagenen Datenbankzugriff zu wiederholen oder das Programm zu beenden. In manchen Fällen kann es auch sinnvoll sein dem Benutzer zu erlauben, den fehlgeschlagenen Datenbankzugriff zu ignorieren.

Von Beginn an stand fest, dass dieses Handling von Exceptions wenn möglich zentralisiert werden sollte, anstatt den Code für die Anzeige und Logik eines entsprechenden Dialogs an mehreren Stellen im Programm zu wiederholen. Hierzu wurde schließlich die Klasse `DBExceptionHandler` geschrieben.

Die Klasse `DBExceptionHandler` Die Klasse `DBExceptionHandler` stellt statische Methoden zur Verfügung, welche eine übergebene Methode aufrufen und eine auftretende `DatabaseException` durch Anzeige eines Dialogs mit den Optionen „Wiederholen“, „Programm Beenden“ und falls gewünscht „Ignorieren“ behandeln. Ist der Aufruf erfolgreich gibt diese Methode den von der übergebenen Methode zurückgegebenen Wert zurück. Da es in Java nicht möglich ist Methoden direkt als Argumente zu übergeben[17] wird stattdessen eine Klasse verwendet, welche das Interface `DBFunction<T>` implementiert.



Das Interface `DBFunction<T>` Analog etwa zum Interface `Runnable` aus der Standardbibliothek dient dieses Interface einzig zur Übergabe einer Methode. Im Gegensatz zur `run` Methode verfügt diese jedoch über einen Rückgabewert. Wie bei `Runnable` bietet sich bei der Implementierung die Verwendung *anonymer innerer Klassen* an. Zur einfacheren Implementierung stehen desweiteren mehrere abstrakte Klassen zur Verfügung welche verschiedene Argument und Rückgabewert Kombinationen als *anonyme innere Klasse* ermöglichen:

Klasse	Rückgabewert	Argumente
`VoidNullaryDBFunction`	nein	0
`VoidUnaryDBFunction`	nein	1
`UnaryDBFunction`	ja	1

[17] Theoretisch bieten Javas Reflection Features eine Möglichkeit zur Übergabe von Methoden, diese ist jedoch relativ umständlich und im Gegensatz zu der hier verwendeten Variante nicht typsicher.

6.5. Vorstellung der Programmoberfläche

Der Aufbau des Graphischen Oberfläche orientiert sich am bisherigen Scaffold Hunter, so dass es weiterhin einen Startdialog zur Auswahl eines Datenbank-Profils gibt, bevor das eigentliche Hauptfenster geöffnet wird. Für die Konfiguration des Programms bzw. der einzelnen Ansichten gibt es einen neu gestalteten und vereinheitlichten Optionsdialog.

6.5.1. Hauptfenster

Das Hauptfenster wurde um einige neue Elemente ergänzt, um den neu hinzugekommenen Fähigkeiten des Programms Rechnung zu tragen. Der bestehende Code des alten Scaffold Hunters erwies sich hierbei als zu unflexibel, so dass hier trotz oberflächlicher Ähnlichkeit der alten und neuen Version kein Code übernommen werden konnte. Stattdessen wurde dieser Teil des Programms komplett neu implementiert in der Hoffnung, hierbei von vornherein alle geplanten Features in der Architektur des Codes berücksichtigen zu können.

Prinzipiell beibehalten wurden die Menüleiste und Werkzeugleiste am oberen Rand des Fensters, sowie die Seitenleiste am linken Rand. Hier stellte sich das Problem, dass der neue Scaffold Hunter über sehr verschiedene Ansichten verfügt, die jeweils auch unterschiedliche Arten von Informationen darstellen müssen und unterschiedliche Benutzereingaben erwarten. Aus diesem Grund sind die gesamte Seitenleiste sowie Teile der Menü- und Werkzeugleiste ansichtsspezifisch und werden abhängig von der gerade aktiven Ansicht ein- oder ausgeblendet. Die Kontrolle über den Inhalt dieser GUI-Elemente liegt hierbei weitgehend bei den jeweiligen Ansichten selbst.

Menüleiste Die Menüleiste verfügt über die Untermenüs „Sitzung“, „Auswahl“, „Subset“, „Fenster“ und „Hilfe“, sowie ein ansichtsspezifisches Untermenü, dessen Name jeweils der gerade aktiven Ansicht entspricht.
Das Sitzungs-Menü enthält grundlegende Funktionen zum Erstellen und Speichern einer Sitzung, zum Ändern von Programmeinstellungen, sowie zum Beenden des Programms. Die Auswahls- und Subset-Menüs bieten diverse Operationen auf Molekülauswahl und auf dem gerade aktiven Subset an. Dazu gehören unter anderem diverse Funktionen zur Modifizierung der Auswahl, zum Erstellen von Subsets aus der Auswahl, oder zum Setzen von Fähnchen an den ausgewählten Molekülen. Das Fenster-Menü erlaubt unter anderem das Óffnen und Schließen von Fenstern und Ansichten. Außerdem finden sich hier Funktionen zum Anpassen des Fensterlayouts, beispielsweise durch Aufteilung des Fensters in zwei Hälften, oder durch Ein- oder Ausblenden der Seitenleisten.
Ein großer Teil der Menüeinträge ist zusätzlich auch in den Kontextmenüs von Subsetleiste, Tabs und den einzelnen Ansichten verfügbar.

Werkzeugleiste Die Werkzeugleiste besteht aus zwei Teilen: Die linke Hälfte enthält allgemeine Funktionen, die unabhängig von der aktiven Ansicht zur Verfügung stehen, während die rechte Hälfte abhängig von der gerade aktiven Ansicht wechselt.

Ansicht-Tabs Dies ist der zentrale Teil des Hauptfensters, in dem beliebig viele Ansichten in Tabs untergebracht werden können. Bei Bedarf ist auch eine horizontale oder vertikale Teilung der Tab-Leiste möglich, so dass zwei Ansichten gleichzeitig nebeneinander sichtbar gemacht werden können.

Seitenleiste Wie auch im ursprünglichen Scaffold Hunter ist die Seitenleiste mit Hilfe der Klasse JTaskPane aus der L2FProd-Bibliothek realisiert. Dies gibt dem Benutzer die Möglichkeit, nur die gerade benötigten Informationsfenster anzeigen zu lassen, und andere Elemente einfach auszublenden. Da der Aufbau der Seitenleiste von der zugehörigen Ansicht abhängt, wird der Zustand der jeweiligen Informationsfenster für jede Ansicht einzeln gespeichert, und beim Wechsel der Ansicht entsprechend wiederhergestellt. Bei Bedarf kann außerdem die komplette Seitenleiste temporär ausgeblendet werden, um mehr Platz für die Ansichten selbst zu gewinnen.

Subsetleiste Ein neu hinzugekommenes Element der Benutzeroberfläche ist die Subsetleiste am rechten Rand des Hauptfensters. Hier werden das Datenset und die daraus generierten Subsets in Form eines Baumes dargestellt. Die Subsetleiste ist der einzige Teil der Bedienoberfläche, dessen Erscheinungsbild und Funktionen völlig unabhängig von der gerade aktiven Ansicht sind. Ebenso wie die Seitenleiste kann auch die Subsetleiste bei Bedarf ausgeblendet werden.
Das Kontextmenü der Subsetleiste enthält Funktionen zur Navigation zwischen verschiedenen Subsets und Ansichten sowie zur Erstellung, Löschung und Annotation von Subsets.

Auswahlpanel Am unteren Rand der Subsetleiste wird die Größe der aktuellen Molekülauswahl angezeigt. Da unter Umständen ein Teil der ausgewählten Moleküle nicht in der aktiven Ansicht enthalten ist, gibt es hierzu zwei Angaben: zunächst die Gesamtgröße der Auswahl, und darunter die Größe der Schnittmenge von Auswahl und aktivem Subset. Ergänzt wird dies durch Buttons, um aus der jeweiligen Auswahl neue Subsets zu erzeugen, und einen Button, der die gesamte Auswahl auf Null zurücksetzt.

6.5.2. Startdialog

Der Startdialog dient zum Start des Scaffold Hunters mit einem gewählten Datenbank-Profil und zur Verwaltung der lokal gespeicherten Profile. Ein Datenbankprofil enthält neben dem Datenbanktyp, alle zur Verbindung mit einer bestimmten Datenbank benötigten Daten, die Speicherung des Datenbankpassworts ist dabei optional.

Dazu enthält der Dialog eine Combobox, in der alle auf der Festplatte gespeicherten Profile eingetragen sind, sowie Buttons, über die weitere Dialoge zum Erstellen und Bearbeiten von Datenbankprofilen und Benutzern geöffnet werden können. Zusätzlich bietet der Dialog eine Combobox zur Sprachauswahl.

Zum Speichern der Profile auf der Festplatte wird auf die *Java Preferences API* zurückgegriffen, diese verwendet eine an das ausführende Betriebssystem angepasste Methode um die Einstellungen zu persistieren: Unter Windows werden die Einstellungen in

der *Registry* abgelegt, während sie unter Unix, Linux und Mac OS in XML-Dateien im *home Verzeichnis* des aktuellen Benutzers gespeichert werden.

Zusätzlich zu den Profilen wird auch der Name des zuletzt gewählten Profils gespeichert, damit dieses beim nächsten Start automatisch ausgewählt werden kann.

Verändert der Benutzer ein Profil, so wird diese Änderung unmittelbar gespeichert. Dabei werden allerdings nicht alle möglichen Eingaben akzeptiert. So darf es keine Profile mit gleichem oder leerem Namen geben, dies ermöglicht es gespeicherte Profile eindeutig anhand ihres Namens zu identifizieren. Auch dürfen die Profildaten nicht leer sein. Bei allen Eingaben im Profildialog werden etwaige Leerzeichen am Anfang und Ende entfernt. Die Ausnahme bildet hier das optionale Passwort. Falls das Häkchen zum Speichern aktiviert ist, wird die Eingabe in diesem Feld unverändert als Passwort übernommen. Bei URLs für netzwerkbasierte Datenbanken wird bei Bedarf das zum ausgewählten Datenbanktyp passend Präfix ergänzt.

Weiterhin enthält der Dialog einen Login-Button, mit dem die Verbindung zur Datenbank hergestellt und der Dialog zur Sessionauswahl angezeigt wird, dabei werden die Daten des aktuell gewählten Profils benutzt, um sich mit dem `DBManager` in die Datenbank einzuloggen. Ist in dem Profil kein Datenbankpasswort gespeichert, wird ein Dialog angezeigt, in dem der Benutzer dieses eingeben kann.

6.5.3. Optionsdialog

Die Ansichten können unterschiedliche Konfigurationen mit jeweils verschiedenen Gültigkeitsbereichen haben. Diese werden im Scaffold Hunter in vier Kategorien eingeteilt: Global, Ansichtstyp, Ansicht und Status.

Die globale Konfiguration gilt für alle Ansichtstypen und kann zum Beispiel die Sprache der Benutzeroberfläche beinhalten. Ansichtstyp-Konfigurationen gelten für alle Ansichten eines bestimmten Typs und eignen sich so zum Beispiel für das Speichern der Schriftart einer Überschrift. In Konfigurationen der Ansichtskategorie lassen sich Einstellungen eintragen, die nur für die zugehörige Instanz des Ansichtstypen gültig sind. Ein Beispiel hierfür ist das Raster eines Zeichenprogramms, das in jeder Ansicht unterschiedlich sein kann. Die Status-Kategorie enthält interne Konfigurationen, die nicht explizit über einen Konfigurationsdialog durch den Benutzer veränderbar sein müssen. Der aktuell sichtbare Bildausschnitt eines Zeichenprogramms ist ein Beispiel hierfür.

Der Grund für diese feine Unterteilung liegt darin, dass die Ansichten beim Starten wiederhergestellt werden sollen, und so eine einheitliche Schnittstelle für das Speichern des Zustands in der Datenbank wünschenswert ist. Weiterhin erleichtert die Unterteilung die Verwaltung der Ansichten und ihrer Zustände, da diese so nur einmal implementiert werden musste.

In dem Optionsdialog können nur Konfigurationen der Kategorien Ansicht und Ansichtstyp editiert werden. Beide sind über einen Menüeintrag des Hauptfensters unter „Datei“ erreichbar, wobei „globale Optionen“ für die Kategorie Ansichtstyp steht und „Optionen“ die Ansichtsinstanz-Konfigurationen anzeigt. Alle Konfigurationen der gewählten Kategorie werden beim Öffnen des Dialoges in einer Registerkartenansicht angezeigt, wobei die zu der aktuell in dem Hauptfenster aktiven Ansicht gehörende Konfi-

guration automatisch angewählt wird.

Die zu einer Konfiguration gehörenden Einstellungen werden tabellarisch angezeigt, wobei unter der Tabelle eine Beschreibung zu der aktuell gewählten Eigenschaft angezeigt wird und Eigenschaften gruppiert angezeigt werden. Diese Gruppen können nach Bedarf ausgeblendet werden.

Klickt der Benutzer auf die Anzeige des Wertes in der rechten Spalte, wird der vom Typ der Eigenschaft abhängige Editiermodus aktiviert. Dies kann eine Eingabezeile sein, in der der neue Wert eingegeben werden kann, oder es wird der Wert, zum Beispiel im Falle einer Checkbox, direkt geändert. Weiterhin kann ein Auswahlfenster mit den möglichen Werten der Eigenschaft angezeigt werden. Bei komplexeren Objekten werden am rechten Rand der Spalte Buttons eingeblendet. Mit diesen Buttons lassen sich der eingetragene Wert löschen und Dialogfenster öffnen, so dass beliebige Objekte für die Konfigurationen verwendet werden können.

Unter dem Beschreibungsfeld befinden sich die üblichen „Ok“, „Anwenden” und „Abbrechen“-Buttons, mit denen die in dem Dialog vorgenommenen Änderungen verworfen oder in die entsprechenden Ansichten übernommen werden können.

Da sich beliebig viele Ansichten erstellen lassen, gibt es eine Standardkonfiguration pro Ansichtstyp, die für neu erzeugte Ansichten verwendet wird. Diese lässt sich bei dem Editieren einer Ansichtskonfiguration über den „Als Standard setzen“ Button unten links setzen.

6.6. Ansichten

6.6.1. Scaffoldbaumansicht

Die Scaffoldbaumansicht wurde im Wesentlichen aus dem Ursprungsprojekt Scaffold Hunter übernommen. Im Rahmen der Projektgruppe wurden die Teile, die die Ansicht bildeten, aus dem ursprünglichen Scaffold Hunter extrahiert und an die neue Umgebung angepasst. Insbesondere an die neue gemeinsame View-Struktur und an das neue Datenbankmodell. Dabei wurden die im Ursprungsprojekt vorhandenen Features weitestgehend beibehalten, neben den neuen ansichtsübergreifenden Elementen wurden auch weitere Visualisierungsmöglichkeiten zur Ansicht hinzugefügt.

Der Code für die Scaffoldbaumansicht wurde aus dem urspünglichen Scaffold Hunter kopiert, da das Datenbankmodell im Rahmen dieser Projektgruppe vollständig neu geschrieben wurde, wurden dabei bereits erste Änderungen an der Datenbankanbindung gemacht, auch wurden einige Teile, welche von noch nicht implementierten Komponenten abhingen zunächst auskommentiert. Diese wurden im weiteren Projektverlauf mit Fertigstellung der entsprechenden Komponenten angepasst oder neu geschrieben.

Layouts Die drei Scaffoldbaumlayouts (*Radial Layout*, *Balloon Layout*, *Linear Layout*) wurden praktisch vollständig aus dem Originalprojekt übernommen. Es wurden lediglich einige Bugfixes vorgenommen und die Methode zum Ablegen von Layouteinstellungen wurde in Teilen an das Ansichtsübergreifende Optionsframework angepasst.

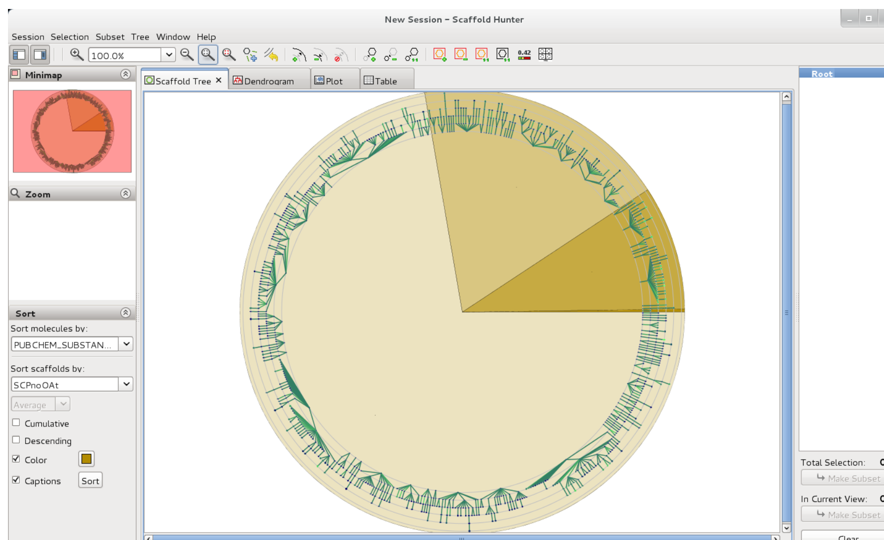


Abbildung 6.2.: Scaffoldbaumansicht

Anzeige von Teilbäumen Der ursprüngliche Scaffold Hunter bot die Möglichkeit weitere Tabs zu öffnen, welche Teile des gesamten Scaffoldbaum enthalten. Da die neue Scaffoldbaumansicht mit Ausnahme von virtuellen Scaffolds welche nötig sind um den Zusammenhang des Baumes zu gewährleisten nur diejenigen Scaffolds anzeigt, welchen im aktuellen *Subset* mindestens ein Molekül zugeordnet wird dieses Feature durch das Subsetprinzip bereits abgedeckt um eine direkte Möglichkeit zur Anzeige von Teilbäumen zu bieten wurde im Kontextmenü die Option hinzugefügt ein Subset zu erstellen, welches genau die den Scaffolds eines Unterbaumes zugeordneten Moleküle enthält.

Interaktion Sowohl die Maus-, als auch die Tastaturbedienung wurden ebenfalls weitestgehend aus dem Ursprunsprojekt übernommen, es wurden lediglich einige Tastaturkürzel verändert um diese an programmweite Standards anzupassen. Tiefgreifendere Änderungen ergaben sich bei der *Selektion*. Im Ursprungsprojekt basierte Selektion auf Scaffolds in der neuen Version können jedoch Moleküle selektiert werden. Es wurde entschieden Selektion für Scaffolds von der Selektion der einem Scaffold zugeordneten Moleküle abhängig zu machen und auf dieser Basis Scaffolds in drei Gruppen einzuteilen:

nicht selektiert kein dem Scaffold zugeordnetes Molekül ist ausgewählt

teilselektiert mindestens eins, aber nicht alle dem Scaffold zugeordneten Moleküle sind ausgewählt

(vollständig) selektiert alle dem Scaffold zugeordneten Moleküle sind ausgewählt

Dabei werden teilselektierte und selektierte Scaffolds durch unterschiedliche Farben gekennzeichnet.

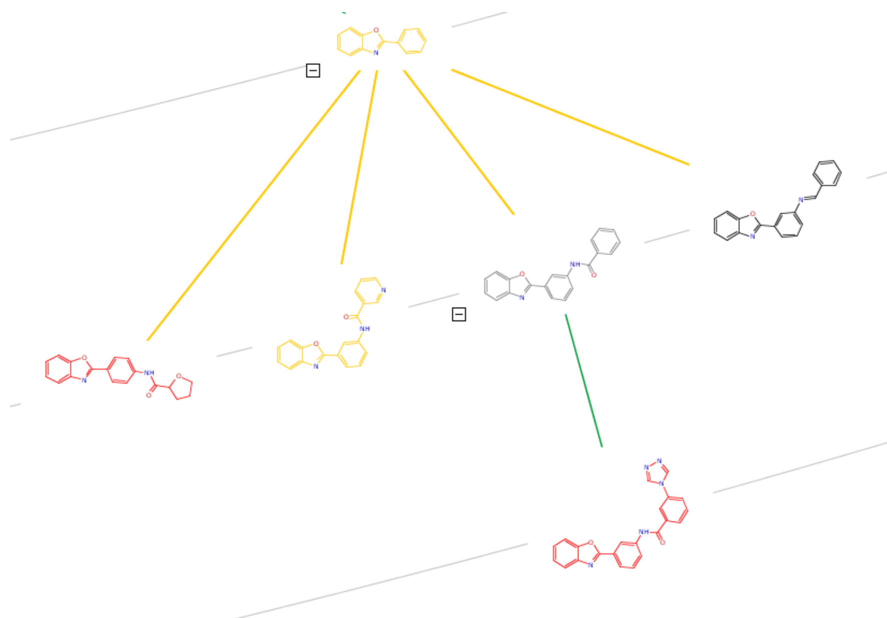


Abbildung 6.3.: nicht selektierte (schwarz), teilselektierte (gelb) und vollständig selektierte (rot) Scaffolds

graphischer Export Die Funktionen welche es erlauben den gesamten Baum oder die aktuelle Ansicht als Vektor- oder Rastergrafik zu exportieren wurden ebenfalls übernommen. Bedingt durch einige interne Änderungen mussten dabei jedoch einige Teile des Exportcodes neu geschrieben werden:

In der Originalversion wurden viele visuelle Optionen, wie etwa die Kantendicke oder die Schwellwerte für den semantische Zoom der Knoten des Scaffoldbaumes durch statische Methoden in einer zentralen Klasse gesteuert. Beim graphischen Export wurden die aktuellen Einstellungen zwischengespeichert und durch für den Export angepasste Einstellungen ersetzt. So wurde zum Beispiel durch entsprechende Anpassung der Schwellwerte die detailreichste Ebene des semantischen Zooms aktiviert.

Bei der Überarbeitung ist diese zentrale Klasse weggefallen. Einige Einstellungen wurden in das Optionsframework aufgenommen. Andere, welche eher Konstanten entsprachen wurden passenden Klassen zugeordnet. So sind jetzt etwa die Schwellwerte für den semantischen Zoom Teil eines *Enums* welches die Zoomlevel repräsentiert.

Um weiterhin eine an den Export angepasste Darstellung verschiedener Elemente des Scaffoldbaumes zu ermöglichen wurden die Klasse `ExportCamera` und das Interface `ExportPaintNode` entwickelt. Eine `ExportCamera` ist eine `PCamera` welche eine zusätzliche Methode zur Verfügung stellt um den Szenengraph in einer für den Export angepassten Version zu zeichnen. Beim Aufruf dieser Methode erfolgt ein Durchlauf durch den Szenengraph dabei wird für Knoten welche das Interface `ExportPaintNode` implementieren eine gesonderte `exportPaint` Methode aufgerufen, auf allen übrigen Knoten wird die normalerweise von Piccolo zum zeichnen des Szenengraphen verwendete `fullPaint`

Methode aufgerufen, da das zeichnen mittels dieser Methode normalerweise rekursiv erfolgt werden zunächst alle Kinder eines Knoten entfernt und nach dem Aufruf wieder hinzugefügt.

Visualisierungsmöglichkeiten Im Ursprungsprojekt gab es bereits mehrere Möglichkeiten um einem Scaffold zugeordnete Daten im Baum darzustellen: Der Hintergrund eines Scaffolds konnte entsprechend eines Wertes eingefärbt werden, entweder durch Zuordnung von Farben zu Intervallen oder durch Zuordnung einer Farbe zu einer Eigenschaft und Anpassung der Farbintensität gemäß des zugeordneten Wertes. Weiterhin war es möglich einem Scaffold zugeordnete Verteilungen mittels kleiner Diagramme Unterhalb der Scaffolds zu visualisieren.

Diese Visualisierungsmöglichkeiten wurden bei der Überarbeitung übernommen. Da sich durch die Einführung von Subsets die Menge der einem Scaffold zugeordneten Moleküle jederzeit ändern kann mussten dabei die aus Molekülen eines Scaffolds kumulierte Werte (Maximum,Minimum und Durchschnitt über die Molekülwerte) bei Bedarf berechnet werden, anstatt diese beim Import vorzuberechnen und in der Datenbank abzulegen.

Zusätzlich wurden weitere Optionen zur Visualisierung hinzugefügt: An jedem Scaffold kann ein Wert als Label angezeigt werden. Der Wertunterschied zwischen zwei über eine Kante verbundene Scaffolds kann auf die Breite dieser Kante abgebildet werden. Desweiteren ist es möglich die zwei benachbarten Scaffolds zugeordneten Werte als Farbverlauf auf der Kante darzustellen.

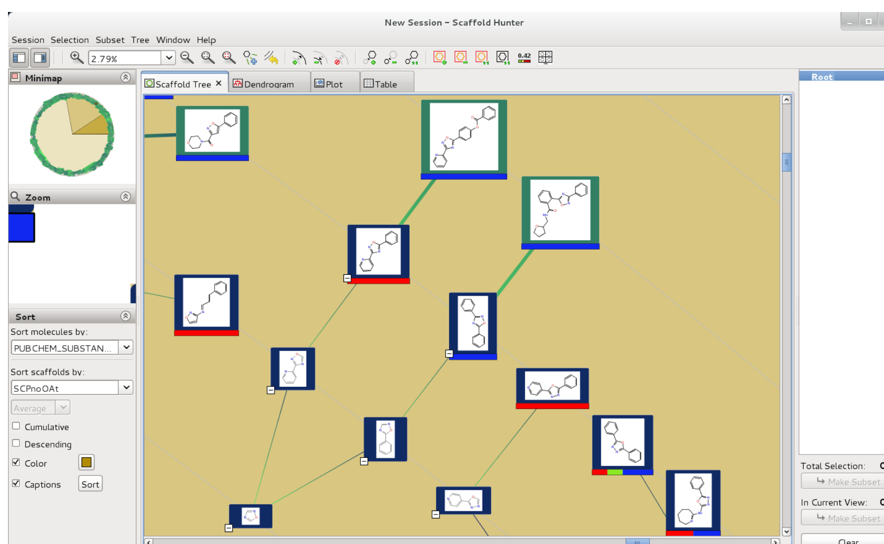


Abbildung 6.4.: Zoom in den Scaffoldbaum mit eingefärbtem Hintergrund, Diagrammen, Wertunterschieden auf der Kantenbreite und Farbverlauf auf den Kanten.

Es war ursprünglich geplant zusätzlich den Vordergrund entsprechend eines Wertes einzufärben sowie die Knotengröße entsprechend eines Wertes anzupassen. Wir haben

uns im Verlauf des Projektes gegen eine zusätzliche Einfärbung für den Vordergrund entschieden, da dieser bereits eingefärbt wird um die Auswahl zu visualisieren. Die Änderung der Knotengröße bereitete bei der Implementierung unerwartet viele Probleme, sodass wir auch von dieser Option wieder abgesehen haben.

Molekülansicht Im Ursprungsprojekt gab es eine Ansicht, welche die Moleküle eines ausgewählten Scaffolds in einem eigenen Tab anzeigt. Diese war jedoch vollständig vom Scaffoldbaum getrennt und bot weder weitere Daten zu den Molekülen noch Interaktionsmöglichkeiten.

Mit dem Gesamtkonzept, welches den Scaffoldbaum nur als eine Ansicht auf eine Menge von Molekülen betrachtet, schien es uns wichtig, die Verbindung von Scaffolds und Molekülen hervorzuheben. Zu diesem Zweck eine Funktion implementiert die es ermöglicht im Scaffoldbaum auf eine *Molekülansicht* umzuschalten. Diese zeigt die einem Scaffold zugeordneten Moleküle im Vordergrund und das Scaffold im Hintergrund (siehe Abbildung 6.5). Sind dem Scaffold mehr als neun Moleküle aus dem aktuellen Subset zugeordnet kann der Benutzer mittels der Pfeile im Unteren Teil des Knotens durch die Moleküle blättern.

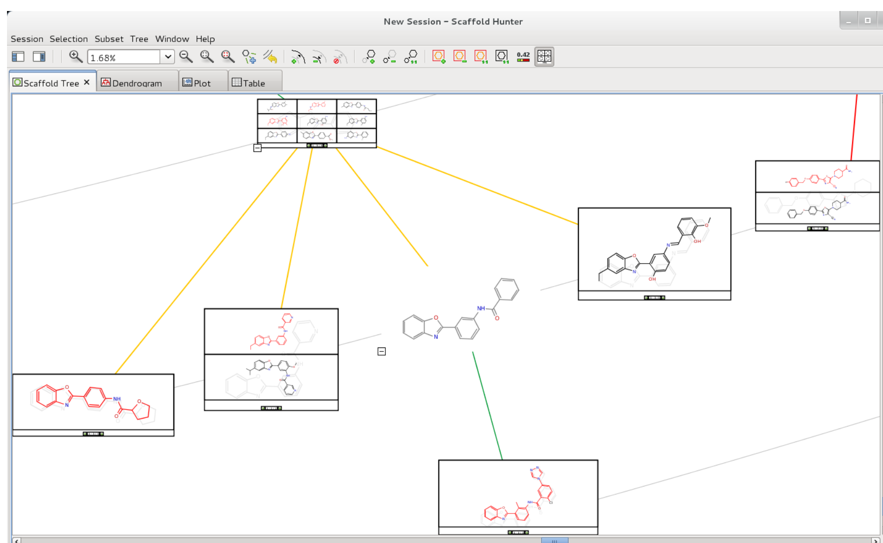


Abbildung 6.5.: Scaffoldbaum mit aktivierter Molekülansicht.

Reicht ein Knoten mit aktiver Molekülansicht über den Rand des aktuellen *Viewports* hinaus, wird er so skaliert und verschoben dass er auf dem zur Verfügung stehenden Platz vollständig zu sehen ist.

Moleküle in der aktuellen Auswahl werden farblich hervorgehoben. In der Seitenleiste gibt es darüber hinaus eine Option um die Moleküle jedes Scaffolds entsprechend eines ausgewählten Wertes zu sortieren. Um weitere Daten zu einem Molekül anzuzeigen kommt der auch an anderen Stellen im Programm verwendete Tooltip zum Einsatz (siehe Abbildung 6.6).

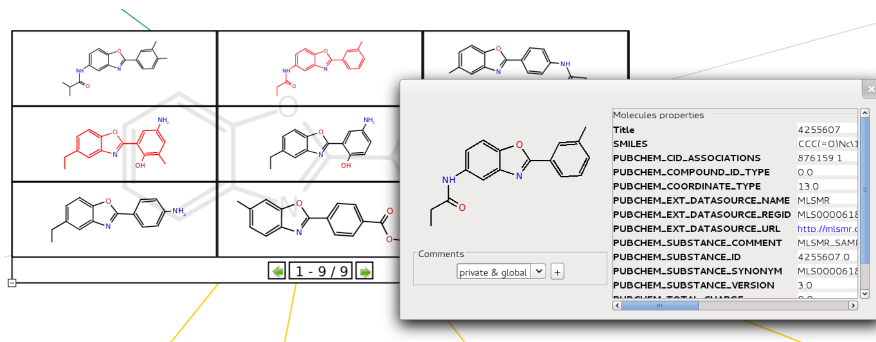


Abbildung 6.6.: Molekülansicht mit Tooltip

6.6.2. Dendrogrammansicht

Eine der neu implementierten Ansichten ist die Dendrogrammansicht. In ihr kann der Nutzer ein *hierarchisches Clusterverfahren* auf ein Subset von Molekülen anwenden und sich das Ergebnis anzeigen lassen. Ein hierarchisches Clusterverfahren dient in diesem Fall dazu, dass Moleküle abhängig von ihrer Ähnlichkeit in einer zweidimensionalen Darstellung angeordnet werde. Hierbei werden jeweils die zwei ähnlichsten Moleküle bzw. Cluster von Molekülen in einer Baumstruktur zusammengefasst und dieser Schritt solange wiederholt, bis nur noch ein Knoten vorhanden ist, der die Wurzel des Dendrogramm(-Baum)s bildet. Der so entstandene *binäre Baum* wird in der Ansicht mithilfe des 2D-Grafik-Frameworks *Piccolo* dargestellt (siehe Abbildung 6.7). Dabei werden an den Blättern des Baumes zusätzlich die Moleküle als SVGs angezeigt.

Die Ähnlichkeit zwischen zwei zusammengefassten Teilclustern/Molekülen wird durch den Abstand der Moleküle zur jeweiligen Wurzel visualisiert. Auf diese Weise kann der Benutzer auf einen Blick erkennen, wie ähnlich sich diese Cluster/Moleküle sind.

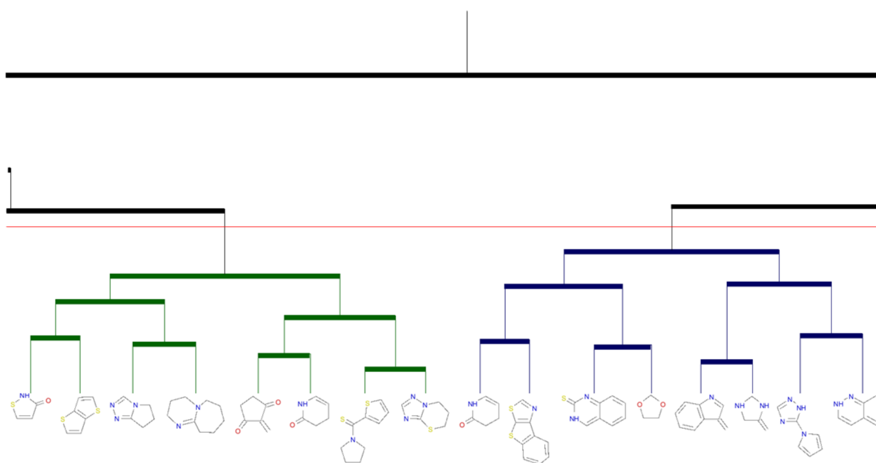


Abbildung 6.7.: Ausschnitt aus einem Dendrogramm Baum

6.6.2.1. Ablauf

Die Dendrogrammansicht wird erst einen Baum anzeigen, wenn dies explizit vom Benutzer angestoßen wird. Dies hängt damit zusammen, dass ein Clustering nicht im Voraus berechnet werden kann (wie etwa der Scaffoldbaum), sondern erst wenn die entsprechenden Parameter eingestellt wurden und die zu clusternde Auswahl feststeht. Da die Berechnung im Zweifelsfall sehr lange dauern kann, wird die Berechnung nur manuell gestartet. Das Starten des Clusterns erfolgt dabei über den entsprechenden Button in der Seitenleiste oder im Menü.

6.6.2.2. Designentscheidungen und Implementierungsdetails

Einsatz von Piccolo Wir haben uns bei der Umsetzung der Dendrogrammansicht aus mehreren Gründen für den Einsatz des Piccolo Frameworks entschieden. Der primäre Grund für die Wahl des Frameworks ist die Nutzung des *Szenegraphenkonzeptes* von Piccolo, da dies sehr geeignet ist, um Bäume zu zeichnen bzw. sie zu verändern. Es lassen sich beispielsweise komplette Teilbäume verschieben, in dem lediglich der Wurzelknoten neu positioniert wird. Ein weiterer Grund liegt darin, dass Piccolo bereits im Scaffold Hunter eingesetzt wurde. Eine gemeinsame Nutzung verhindert neue Abhängigkeiten durch eventuell benötigte weitere Bibliotheken. Zusätzlich können Codeteile aus der Scaffoldbaumansicht (z.B. Seitenleistenelemente) weiter benutzt werden, was zum einen die Neuentwicklung einspart und es zum anderen erleichtert, die verschiedenen Ansichten optisch einheitlicher wirken zu lassen.

Umsetzung des MVC-Konzeptes Das Ergebnis des Clustering ist im wesentlichen eine Baumstruktur, bei der in jedem Knoten nur Referenzen auf die jeweiligen Kinder (Knoten oder Moleküle) sowie die (Un-)Ähnlichkeit dieser gespeichert ist. Blattknoten speichern zusätzlich noch eine Referenz auf das zugehörige Molekül. Die Dendrogrammansicht bindet dieses Modell ein, ohne es zu verändern und ergänzt es durch eine grafischen Repräsentation. Wir haben uns für diese Trennung von Modell und grafischer Repräsentation entschieden, da es einerseits das *Model-View-Controller* Konzept unterstützt und zusätzlich weitere Views auf dem gleichen Modell arbeiten können.

Codestruktur Die Klasse `DendrogramView` ist die Oberklasse dieser Ansicht. Sie erzeugt sämtliche Elemente, die vom Hauptfenster abgefragt werden können (Ansicht, Seitenleistenelemente und Toolbaricons). Außerdem erzeugt sie das `DendrogramCanvas` mit dem darzustellenden Modell als Parameter. Das Canvas erzeugt ein Dictionary, in dem von jedem `ModelNode` ein Verweis zum dazugehörigen `ViewNode` hinterlegt ist. Dies ist nötig, da im Modell keine von der Ansicht abhängigen Informationen gespeichert werden. Zusätzlich stößt der Konstruktor des Canvas die Generierung des Szenengraphen an. Zu jedem `ModelNode` wird ein `ViewNode` erzeugt, der von der Piccolo-Klasse `PNode` erbt und die View-spezifischen Daten enthält. Dazu gehören die Koordinaten, bzw. Verschiebung der Knoten, die Kanten (`DendrogramEdge`) zwischen den Baumknoten und Darstellungsinformationen wie etwa die Farbe. Bei der Generierung des Szenengraphen erzeugt jeder Knoten zuerst seine Kinder (falls vorhanden) und berechnet dann aus deren Daten seine eigene Position. Die Clustering Klassen sind von den View Klassen getrennt und werden in 6.7 beschrieben. Die Verknüpfung bildet das ClusteringListener Interface welches von der DendrogramView Klasse implementiert wird.

Generierung des Szenengraphen Der Aufbau des Szenengraphen geschieht rekursiv durch eine „*post order*“-Traversierung des Modells. Dabei werden zunächst die absoluten Positionen aller Knoten berechnet, beginnend mit der Position (0,0) beim am weitesten links stehenden Nachfahren der Wurzel (Rekursionsboden). Dies ist notwendig, damit zur Berechnung der Position eines Knotens bereits die Positionen der Kinder bekannt sind.

Auf diese Weise kann die x-Koordinate des rechten Kindes eines Elternknotens v passend zur x-Koordinate des linken Kindes berechnet und die x-Koordinate von v mittig zu der Ausdehnung seiner Kinder bestimmt werden. Die y-Koordinate eines inneren Knotens entspricht dabei jeweils dem Maximum der y-Koordinaten seiner Kinder zuzüglich den Wert ihrer Unähnlichkeit.

Das Szenengraphenkonzept basiert auf relativen Koordinaten. Wenn die Berechnung der absoluten Positionen eines Knotens abgeschlossen ist, wurden die absoluten Positionen seiner Kinder bereits berechnet. An dieser Stelle ist es nun möglich, die relative Verschiebung der Kinder im Szenengraphen zu berechnen.

Visualisierung des Szenengraphen Als Repräsentanten der Blätter werden die SVGs der entsprechenden Moleküle angezeigt. Um die einzelnen SVGs der Moleküle, die teilweise aus mehreren Zusammenhangskomponenten bestehen können besser zu trennen, wird um jedes SVG ein Rahmen gezeichnet. Ab einem gewissen Zoomlevel sind die Details der SVG nicht mehr sichtbar, deshalb werden ab einem Schwellwert die SVGs durch schwarze Rechtecke ersetzt. Vom Benutzer hinzugefügte Banner werden unter den SVGs / Rechtecken als Fähnchen / farbliche Flecke angezeigt Die inneren Knoten werden durch rechteckige flache Balken visualisiert, die so breit sind, dass sie von der Mitte des linken zur Mitte des rechten Kinds reichen. Zusätzlich gibt es noch Kanten von den Eckpunkten der Knoten zu ihren Kindern. Wir haben uns für diese Darstellung der inneren Knoten entschieden, da auf diese Weise durch Klicken auf diese der Unterbaum an/abgewählt werden kann.

6.6.2.3. Funktionen der Dendrogrammansicht

Selektion von Clustern Im Gespräch mit Chemikern haben wir erfahren, dass die Dendrogramm-Visualisierungen die in der Industrie angewandt werden, eine verschiebbare Leiste, die *Selectionbar* besitzen. Mit dieser Leiste kann der Clusterbaum in mehrere Unterbäume aufgeteilt werden, welche jeweils die Moleküle eines Cluster enthalten. Diese Leiste wird durch *Drag-and-Drop* verschoben. Um die erzeugten Cluster deutlicher voneinander abzugrenzen, werden sie unterschiedlich eingefärbt. Diese Farbveränderung wurde in unseren ersten Versionen während des Drag-Vorgangs unmittelbar aktualisiert; wir haben uns aber aus Performancegründen dazu entschieden, das Neuzeichnen erst beim Loslassen der Leiste auszuführen.

Markieren von Clustern In allen Ansichten soll es möglich sein, Moleküle und Scaffolds zu markieren, damit diese dann in den anderen Ansichten hervorgehoben werden. Diese Funktion haben wir um die Möglichkeit erweitert, auf einen inneren Baumknoten zu klicken, um so auch ganze Cluster an- und abwählen zu können.

Einbindung der Tabellenansicht In einem Programm, welches die von uns befragten Chemiker bei Bayer für die Auswertung von Clusterings benutzen, wird ein Dendrogramm gemeinsam mit einer Tabelle mit Moleküldaten angezeigt und beide Darstellungselemente können direkt miteinander interagieren. Es erschien uns sinnvoll, den

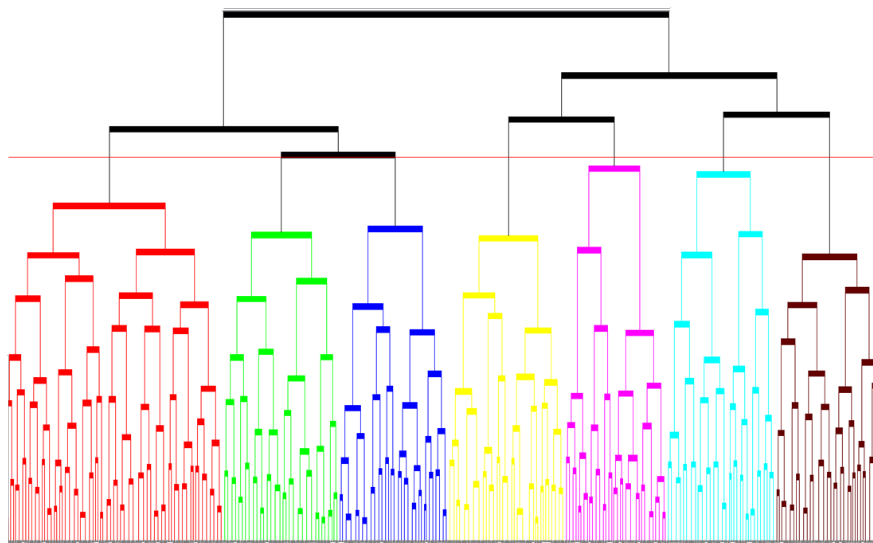


Abbildung 6.8.: Beispiel für die farbliche Unterteilung durch die Selektionsleiste

Chemikern auch innerhalb des Scaffold Hunters ein gewohntes Interface zur Verfügung zu stellen. Aus diesem Grund haben wir die Möglichkeit implementiert eine Tabelle einzublenden, die noch stärker mit dem Dendrogramm interagiert als andere parallele Ansichten. Zu diesem Zweck kann der Benutzer bei Bedarf unterhalb des Dendrogramms eine Tabelle einblenden, wobei das Platzverhältnis zwischen Dendrogramm und Tabelle beliebig einstellbar ist. Die angezeigte Tabelle entspricht dabei einer speziellen Instanz der Tabellenansicht, die aus der Dendrogrammansicht heraus erzeugt wird. Sie wird nicht als gleichberechtigte Ansicht angesprochen (besitzt also keine direkte Verbindung zum Mainframe), sondern enthält die gleichen Moleküle wie das Dendrogramm und ist standardmäßig auch genauso sortiert. Zusätzlich zu den normalen Spalten verfügt die Tabelle in diesem eingebetteten Modus über eine Clusterspalte. In dieser werden die von der Selektionsleiste ausgewählten Cluster von links nach rechts durchnummeriert dargestellt. Um die Cluster auch optisch besser identifizieren zu können, sind die Zeilen der Clusterspalte mit den gleichen Farben hervorgehoben wie die korrespondierenden Cluster im Baum. Die Detail Ansicht der Tabelle wird in der Dendrogramm Seitenleiste eingeblendet, wenn die Tabelle sichtbar ist. Ausserdem werden die Toolbarbuttons der Tabelle direkt über ihr angezeigt.

Zoom Unsere ursprüngliche Implementation des Zooms beruhte auf der Transformation der Hauptkamera, die die Ansicht auf den Szenengraphen steuert. Diese Methode entspricht der Realisierung des Zooms in der Scaffoldbaumansicht und ließ sich sehr einfach implementieren, da lediglich eine in x- und y-Richtung gleiche Skalierung der Kamera mit den entsprechenden Piccolo-Funktionen durchgeführt werden musste.

Das Problem bei Dendrogrammen ist aber, dass diese im Regelfall sehr breit und

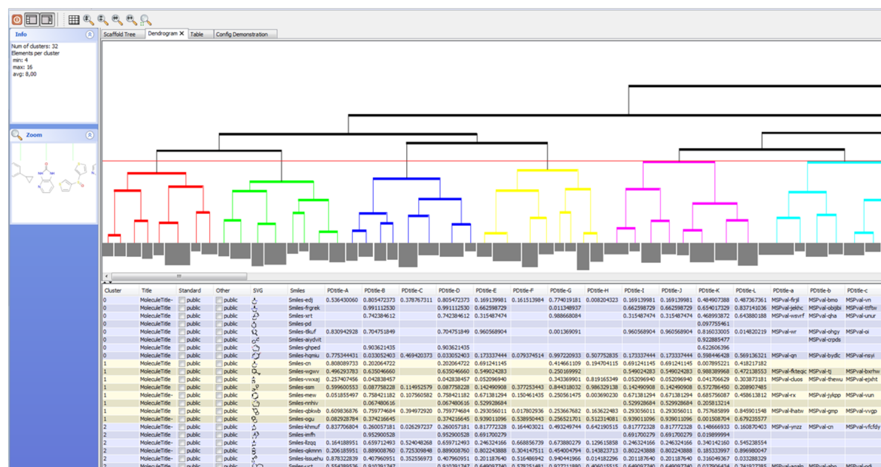


Abbildung 6.9.: Dendrogrammansicht inklusive Tabelle und Zoomfenster

gleichzeitig sehr flach sind. Dies führt dazu, dass bei komplett herausgezoomter Ansicht mit der oben beschriebenen Methode nur ein schmaler horizontaler Strich in der vertikalen Mitte des Ansichtbereiches zu sehen ist. Um diese schlechte Ausnutzung des Ansichtbereiches zu umgehen und eine vollständige Einpassung des Dendrogrammes zu erreichen, war es erforderlich den *Skalierungswert* für die x- und y-Koordinaten der Kamera getrennt einstellen zu können. Diese Methode hat jedoch, wenn sie wie oben beschrieben über eine Transformation der Hauptkamera realisiert wird, ein entscheidendes Problem: Da sich die Skalierungswerte rekursiv auf den gesamten Szenengraphen auswirken, werden die durch SVGs dargestellten Moleküle in den Blättern des Dendrograms verzerrt dargestellt, falls eine Achse stärker als die andere skaliert wird.

Aus diesem Grund haben wir die Zoom-Funktion nicht über eine Skalierung der Kamera realisiert, sondern den Szenengraphen modifiziert und eigene, angepasste Zoomfunktionen (x-Zoom, y-Zoom) implementiert. Der Szenengraph wurde von uns in einen „Baumknoten“ (enthält das Dendrogramm ohne Moleküle) und einen „Blattknoten“ (enthält nur die Moleküle) getrennt. Beim vertikalen Zoom wird nur die y-Koordinate des Baumknoten skaliert. Der horizontale Zoom hingegen skaliert die x- und y-Koordinate des Blattknoten und die entstehenden (unerwünschten) Höhenveränderungen werden durch gegenläufige Skalierung der y-Koordinate des Baumknoten ausgeglichen.

Zur Steigerung der Darstellungsgeschwindigkeit und aus Gründen der Übersichtlichkeit haben wir semantische Zoomstufen implementiert, die die Molekül-SVGs durch graue Rechtecke ersetzen, wenn die Struktur des Moleküls auf Grund zu kleiner Zeichenfläche nicht mehr erkennbar ist. Zwischenzeitlich hatten wir die Idee vollständige Teilbäume des Dendrogrammes innerhalb von bestimmten semantischen Zoomlevel auszublenden und durch gröbere Darstellungskomponenten zu ersetzen. Dies würde die

Übersichtlichkeit für den Benutzer erhöhen, da die Teilbäume des Dendrogramms „aus weiter Ferne“ betrachtet nicht zu einem schwarzen Bereich kollabieren würden und Teilstrukturen weiterhin erkennbar blieben. Da die bisherige Realisierung des Dendrogramms aber bereits jetzt zu Speicherproblemen führt, und weitere Darstellungskomponenten zusätzlichen Speicher benötigen würden, haben wir diese Idee nicht umgesetzt.

6.6.2.4. Zusätzliche Informationsfenster

In der Seitenleiste haben wir vier Elemente (*StartClustering Button, Vergößerungsglas, Statistische-Information, Cluster-Information*) integriert. Ursprünglich hatten wir geplant auch die *Minimap*, in der angezeigt wird, auf welchen Teil des Baum gerade gezoomt ist, aus der Scaffoldbaumansicht zu übernehmen. Diese hat zwar zunächst fehlerfrei funktioniert, wir mussten sie aber aus Performancegründen entfernen, da sie unkontrolliert ständiges Neuzeichnen der gesamten Ansicht ausgelöst hatte. Dieses Problem hängt mit Piccolo zusammen und wir haben bis jetzt keine anwendbare Lösung finden können. Als fünftes Element wird während die Tabelle eingeblendet ist die Zugehörige Detailansicht angezeigt.

StartClustering Button Durch diesen Button wird ein Fenster geöffnet, in dem die Clustering Parameter eingestellt werden können. Zusätzlich wird hier angezeigt, durch Einfärben der Schrift und einen Informationstext, ob das Subset der Ansicht geändert wurde, ob also ein neues Clustering gestartet werden sollte.

Vergrößerungsglas Neben der Übersichtsanzeige gibt es in der Scaffoldbaumansicht auch ein Zoomfenster, welches den Bereich um die Maus vergrößert darstellt. Dieses Seitenleistenelement haben wir übernommen und es nur dahingehend abgeändert, dass der vergrößerte Bereich der Maus nur noch horizontal folgt und dabei immer auf das Molekül zentriert bleibt, welches an der entsprechenden x-Koordinate ist.

Statistische-Information Zusätzlich haben wir noch ein Element integriert, in dem Informationen über die durch die Clusterselektionsleiste gewählten Cluster zu sehen sind: Die Anzahl der ausgewählten Cluster, sowie deren maximale, minimale und durchschnittliche Größe. Diese Informationen sollen es dem Nutzer erlauben besser abschätzen zu können, ob die Leiste wie gewünscht positioniert ist oder nicht. Aus diesem Grund werden die Informationen in dem Infobereich auch unmittelbar beim Verschieben der Leiste aktualisiert.

Cluster-Information Dieses Element zeigt Informationen über die genutzten Clustering Parameter an, also welche *Linkage, Distanz*, Eigenschaftsspalten ausgewählt wurden.

6.6.2.5. Probleme und ausstehende Features

Bestehende Probleme Im Konzeptentwurf hatten wir festgelegt, dass wir minimal 100.000 Moleküle in jeder Ansicht darstellen können wollen. Dies ist aus mehreren Gründen nicht möglich. Bei unseren Tests[18] war die Darstellung von 2^{16} (65.536) Molekülen mit verschmerzbaren Performanceeinbußen möglich, bei 2^{17} (131.072) Molekülen brach aber die *JVM* aufgrund von Speicherproblemen zusammen. Ein Vergrößern des zugewiesenen Arbeitsspeichers ist nicht praktikabel, da nicht erwartet werden kann, dass die Anwender über mehr als 4GB Arbeitsspeicher verfügen. Ein Grund für den enormen Speicherverbrauch ist der Piccolo-Szenengraph, der neben den Molekülen ihre Größeninformationen sowie die inneren Knoten des Clustermodells und die zu visualisierenden Kanten enthält. Da es sich um einen binären Baum handelt, sind also zusätzlich zu den Blättern noch einmal genauso viele innere Knoten und pro innerem Knoten jeweils zwei Kanten zu speichern. Wir haben schon versucht Speicherplatz auf Kosten von Rechenzeit zu sparen, waren dabei aber noch nicht erfolgreich genug. Neben der Ansicht hat auch das Clustering selber eine Grenzen. Da für die Datenhaltung der Distanzmatrix ein zweidimensionales Array verwendet wird und die Array Größe in Java auf 2^{32} begrenz ist, sind maximal 65536 Elemente verarbeitbar.

Ausstehende Features Ein mögliches Feature ist eine Unterteilung der Selectionbar, um die Clustergrößen individuell einstellen zu können. Wir haben uns aber gegen die Implementierung innerhalb der PG entschlossen, da wir den Arbeitsaufwand für ein Nice-to-have Feature als zu hoch geschätzt haben.

6.6.3. Tabellenansicht

Die Tabelle listet sämtliche Moleküle der gerade aktiven Molekülmenge samt ihren Eigenschaften und gesetzten Markierungen ("Fähnchen") auf. Sie dient hauptsächlich dazu, sich einen Überblick über den Datenbestand des aktuellen Subsets zu verschaffen und alle Informationen zu einem Molekül einsehen zu können, falls diese in einer anderen Ansicht nicht verfügbar sind.

Die Tabelle unterstützt das sog. *Lazy Loading*, bei dem der Tabelleninhalt erst dann aus der Datenbank geladen wird wenn er angezeigt werden soll. Dadurch lässt sich der Speicherbedarf für große Tabellen erheblich reduzieren. Um eventuelle Verzögerungen bei der Reaktion auf Benutzereingaben zu reduzieren, lädt die Tabelle dabei nicht nur die aktuell sichtbaren Daten sondern zusätzlich auch noch die Daten im näheren Umfeld des sichtbaren Ausschnitts (*preloading*).

Da die Tabelle oftmals mehr Spalten enthält als – aufgrund der Fensterbreite – gleichzeitig angezeigt werden können, lassen sich, vom linken Tabellenrand ausgehend, beliebig viele Spalten als *fest* markieren. Fest bedeutet hierbei, dass diese Spalten nicht horizontal mitscrollen sondern ständig sichtbar bleiben. Da die Tabellenspalten beliebig umgeordnet werden können lässt sich auf diese Weise jede Spalte am linken Tabellenrand fixieren. Dies soll helfen, die Übersicht in der Tabelle zu steigern. Realisiert wurde diese Funktion durch die Verwendung zweier nahtlos nebeneinanderstehender Einzeltabellen, von

[18] Testsystem: aktueller Dual Core CPU mit 2,53 GHz, Start der JavaVM mit 4 GB max. Heapgröße

denen sich nur eine horizontal scrollen lässt. Da beide Tabellen die selben unterliegenden Strukturen verwenden (Datenmodel, Spaltenmodel, Zeilensortierung etc.), arbeiten sie synchron und erwecken den Eindruck eine einzige Tabelle zu sein.

6.6.4. Plottansicht

Der Scatterplot dient dazu numerische Daten von Molekülen in Form eines zwei- oder dreidimensionalen Punktdiagramms anzuzeigen. Die Zuordnung von Moleküldaten zu den zwei bzw. drei Achsen kann dabei vom Anwender frei gewählt werden. Zusätzlich können bis zu zwei weitere Molekülwerte durch die Farbe und die Größe eines Punktes repräsentiert werden.

Eine der Zielsetzungen hierbei war es bis zu 100.000 Punkte in akzeptabler Geschwindigkeit in das Diagramm zeichnen zu können. Um dies zu erreichen wurde für das Zeichnen der Punkte auf die normalen Zeichenmethoden von Java (wie etwa Graphics.drawOval) verzichtet. Stattdessen wird ein Array von Integerwerten als Datenquelle eines BufferedImages verwendet. Gezeichnet wird, indem ARGB-Werte direkt in das Array hineingeschrieben werden. Nachdem das Diagramm auf diese Weise in das BufferedImage gezeichnet wurde wird das Image mittels Graphics.drawImage angezeigt.

Das eigentliche Zeichnen eines Punktes zerfällt in zwei Teilaufgaben: Erstens muss die korrekte Position des Punktes bestimmt werden, und zweitens müssen die zu diesem Punkt zugehörigen Pixel in der zugehörigen Farbe unter Berücksichtigung des Tiefenwertes (bei drei Dimensionen) gesetzt werden. Zur schnellen Berechnung der Punktpositionen werden zunächst die Einheitsvektoren des angezeigten Koordinatensystem errechnet. Manipulationen der Ansicht- wie rotieren oder zoomen - wirken sich nur auf diese Einheitsvektoren aus. Die Positionen der einzelnen Punkte werden dann als Linearkombination der Einheitsvektoren errechnet. Da die Punkte als unterschiedlich große Kreise gezeichnet werden sollten, die Berechnung von Kreisformen aber recht viel Zeit beansprucht, wurden Kreise von 1-20 Pixeln Durchmesser vorberechnet und entsprechend optimierte Zeichenmethoden generiert.

6.7. Clustering

Ein Ziel dieser Projektgruppe war es, statistische Methoden zur Klassifikation des Strukturraums anzubieten. Zu diesem Zweck haben wir ein SAHN (*Sequential Agglomerative Hierarchic Nonoverlapping*) Clustering implementiert. Dieses Clustering erzeugt einen Baum in dem Moleküle sukzessiv zu Clustern verschmolzen werden. Das Ergebnis des Clusterings wird in der Dendrogrammansicht angezeigt. Als Parameter müssen ein Linkageverfahren, eine Distanzfunktion und eine Auswahl von Eigenschaften, über die die Distanzen berechnet werden, gewählt werden. Die gewählte Distanzfunktion schränkt die wählbaren Eigenschaften ein, da abhängig von der Funktion nur numerische oder spezielle Fingerprints geclustert werden können.

6.7.1. Hierachisches Clustering

Beim Hierachischen Clustering, werden zunächst alle Elemente einem eigenen Cluster zugeordnet. Danach werden aus der Menge der Cluster die beiden mit der geringsten Cluster-Distanz (abhängig vom Linkage Verfahren und Distanzmaß) ausgewählt und dann verschmolzen. Dies wird solange wiederholt, bis nur noch ein Cluster übrig ist. Der so erzeugte Baum kann beispielsweise in einem Dendrogramm angezeigt werden. Als zusätzlicher Informationsgehalt kann in jedem Knoten die Distanz zwischen den Kindern gespeichert werden.

6.7.2. NN-Chain Algorithmus

Aufgrund der schlechten Verfügbarkeit von Clusteringframeworks mit akzeptablen Laufzeiten in Java wurde eine Eigenimplementierung des NN-Chain Algorithmus vorgenommen. Dieser führt das Clustering mit einer Laufzeit von $O(n^2)$ durch. Der verwendete Hauptspeicher variiert je nach Linkageverfahren und Distanzmaß. Bei Euklidschen Distanzmaß mit einem Zentroid basiertem Linkageverfahren wird linear viel Speicher gebraucht und bei allen anderen Kombinationen quadratisch viel Speicher. Der NN-Chain Algorithmus wurde dem Buch [17] entnommen. Dieser verwendet Nearest Neighbour Ketten um die Laufzeit des klassischen Verfahrens zu minimieren. Bei der Implementierung wurde Wert auf starke Modularität gelegt. Es ist ohne weiteres Möglich weitere Linkage Verfahren oder Distanzmaße zu implementieren ohne den Code des NN-Chain Algorithmus selber verändern zu müssen.

6.7.3. Linkage und Distanzmatrix

Wir bieten eine Reihe an bekannten Linkageverfahren an, wie etwa Single Linkage, Ward Linkage und Complete Linkage. Diese Verfahren unterscheiden sich im einzelnen dadurch, wie die Inter-Cluster-Distanzen aus den Distanzen der einzelnen Cluster-Elementen berechnet werden Die Meisten dieser Verfahren benötigen eine sogenannte Distanzmatrix, in welcher zu allen Clusterpaaren die jeweilige Distanz gespeichert ist. Da es zu Beginn ein Cluster pro Element gibt, ist die Größe der Matrix quadratisch in der Anzahl der zu verarbeitenden Moleküle. Da in unserer Implementierung ein zweidimensionales Array zum Speichern der Distanzen verwendet wird (welches von JAVA intern eindimensional verwaltet wird) und JAVA Arrays nur mit Integer Werten adressieren kann, gibt es eine maximal Anzahl an Elementen die geclustert werden kann. Diese liegt bei etwa 65000 Elementen. Mit etwas Programmieraufwand kann diese Limitierung umgangen werden. Da das Clustern von 65000 Elementen jedoch eh viel zu lange dauern würde und der Speicherplatzverbrauch mehr als 4 GiB betragen würde, ist dies eher eine theoretische Grenze und wir haben uns den Aufwand der Implementierung gespart. Aus Sicherheitsgründen wird der Anwender deshalb darüber Informiert, falls eine Matrix mit mehr Einträgen benötigt würde. Es gibt einige Kombinationen aus Linkageverfahren und Distanzfunktion, die keine Matrix benötigen. Die Distanzfunktion muss hierbei Euklidisch sein und die Linkage-Verfahren müssen sich auf einen Zentroiden reduzieren lassen (Centroid, Median, Ward). Die Berechnung der Inter-Cluster-Distanz hängt dann nur

von diesem „Punkt“ ab und nicht von den Einzeldistanzen der enthaltenden Elemente. Diese Verfahren sind linear im Speichergebrauch, da es einen Zentroiden pro Cluster zu speichern gibt und maximal so viele Cluster wie Elemente existieren können.

6.8. Lokalisierungskonzept

Um die grafische Oberfläche an die Sprache des Nutzers anpassen zu können, wird im Scaffold Hunter ein auf `ResourceBundles` basierendes Lokalisierungskonzept eingesetzt. Diese bestehen aus Textdateien, in denen sich Schlüssel und die Übersetzungen der dazugehörigen Texte befinden. Der Zugriff auf diese Übersetzungen ist über die Klasse `edu.udo.scaffoldhunter.util.I18n` möglich. Dazu greifen die dort definierten Methoden auf das im Package `edu.udo.scaffoldhunter.resources` liegende `ResourceBundle` der aktuell eingestellten Sprache zu und geben die zu dem Schlüssel gehörende Übersetzung aus. Den Zugriffsmethoden lassen sich zu dem Schlüssel auch weitere Parameter übergeben, die bei entsprechender Formatierung des Textes dann in die Übersetzung eingefügt werden.

6.9. Optionsframework

Da es möglich sein sollte, prinzipiell beliebige Ansichten anzuzeigen, ansichtsabhängige Einstellungen zu ändern und die Ansichten bei einem Neustart des Scaffold Hunters in ihrem alten Zustand wiederherzustellen, wurde ein umfangreicher Konfigurationsmechanismus benötigt.

6.9.1. Konfigurationen aus Sicht der Ansichten

Für die Ansichten und ihre Konfigurationen bedeutet dieser Konfigurationsmechanismus einige Anforderungen hinsichtlich ihrer Klassenstruktur.

Ansichten

Jede Ansichtsklasse erbt von `GenericView`, welches vier Konfigurationsobjekte bereitstellt:

GlobalConfig Dies ist die von allen Ansichten gemeinsam benutzte Konfiguration. Eine mögliche Einstellung, die hier gespeichert werden kann, ist zum Beispiel die Sprache.

ViewClassConfig Dieses Konfigurationsobjekt wird von allen Instanzen des gleichen Ansichttyps geteilt. Dieses könnte zum Beispiel eine Schriftart oder Farbe sein, so dass alle Ansichten gleichen Typs auch gleich aussehen. Es enthält auch eine StandardViewInstanceConfig, die beim Erstellen neuer Ansichten verwendet wird.

ViewInstanceConfig Diese Konfiguration gilt nur für genau diese eine Ansichtsinstanz. Typisch hierfür sind zum Beispiel Rastereinstellungen in einem Zeichenprogramm. Die Konfiguration lässt sich als Standard für diesen Ansichtstyp speichern.

ViewState Der interne Zustand der Ansicht, der für die Wiederherstellung der Ansichten beim Laden einer Sitzung verwendet wird. Dies könnte zum Beispiel der sichtbare Bildausschnitt sein.

Bei der Deklaration der Ansichtsklasse werden die drei ansichtsspezifischen Konfigurationstypen explizit angegeben. Falls die Ansicht eine dieser Konfigurationen nicht benötigt, muss der entsprechende Basistyp angegeben werden, Eine mögliche Deklaration einer Ansichtsklasse ist also zum Beispiel `MyView extends GenericView<ViewInstanceConfig, MyClassConfig, MyState>`.

Die Ansichten haben über `GenericView` Zugriff auf die globale Konfiguration und die ansichtsspezifischen Konfigurationsobjekte. Diese werden der Ansicht bei der Erstellung übergeben, wobei sie an den `GenericView`-Konstruktor weitergegeben werden müssen. Weitere Änderungen an den Konfigurationen werden über den durch `GenericView` bereitgestellten PropertyChange-Mechanismus propagiert, so dass die Ansicht dort Listener für die entsprechenden in `View` definierten Eigenschaften registrieren sollte. Falls eine Ansicht später eine Konfiguration ändert, muss dies über `firePropertyChange()` propagiert werden, damit auch die anderen Ansichten darüber benachrichtigt werden.

Um die Ansichten mit ihren Konfigurationen im Scaffold Hunter nutzen zu können, müssen diese in `ViewClassRegistry` per `registerClass()` registriert werden.

Konfigurationen

Damit die einzelnen Bestandteile der Konfigurationen angezeigt und verändert werden können, müssen die selbsterstellten Konfigurationsklassen `get()`-/`is()`- und `set()`-Methoden für diese bereitstellen. Jede `get()`- bzw. `is()`-Methode gehört zu einem Attribut, das den Suffix nach `get`/`is` als Namen hat, allerdings mit einem Kleinbuchstaben beginnt. Ist eine entsprechende `set()`-Methode für das Attribut vorhanden, lässt sich dieses Attribut nicht nur lesen, sondern auch verändern. Die Attribute müssen zusätzlich mit `@ConfigProperty` annotiert werden, damit diese im Optionsdialog angezeigt werden. Auf diese Weise lassen sich Variablen vor der Veränderung durch den Nutzer schützen und können dennoch mit der Sitzung gespeichert und wiederhergestellt werden.

In den `set()`-Methoden der Konfigurationsklassen gibt es die Möglichkeit, übergebene neue Werte zurückzuweisen, indem eine `PropertyVetoException` ausgelöst wird. Hierbei ist zu beachten, dass im Falle eines zurückgewiesenen Wertes unmittelbar der alte über die `set()`-Methode gesetzt wird. Ausserdem wird eine Fehlermeldung angezeigt, an die die zur Ausnahme gehörende Nachricht angehängt wird, so dass diese den Grund für das Veto beschreiben sollte. Das Veto kann auf folgende Weise erzeugt werden:

```
public void setConstrained(String constrained)
    throws PropertyVetoException
{
```

```
    if ("bla".equals(constrained)) {
      throw new PropertyVetoException("'bla' is not a valid value",
        new PropertyChangeEvent(this, "constrained",
                  this.constrained, constrained));
    }
    this.constrained = constrained;
}
```

Änderungen der Konfigurationen werden erst an die Ansichten weitergegeben, wenn im Optionsdialog auf Ok oder Apply geklickt wurde. Benachrichtigt werden die Ansichten dann über den PropertyChange-Mechanismus. Die Listener werden durch `GenericView` in den `set()`-Methoden über das Setzen einer neuen Konfiguration benachrichtigt und erhalten als Parameter einen der Property-Strings aus `View`, die angeben, welche Art Konfiguration sich verändert hat, und sowohl die alte als auch die neue Konfiguration. Dadurch kann die View auf die Änderungen mit einer Anpassung der Anzeige reagieren und herausfinden, was sich genau geändert hat, um nur die relevanten Teile anpassen zu müssen.

Attributinformationen

Zu jeder Konfigurationsklasse kann eine `BeanInfo`-Klasse erstellt werden, die in dem gleichen Paket bzw. der gleichen umgebenden Klasse liegt und den gleichen Namen mit einem angehangenen `BeanInfo` hat. Sie beschreibt, welche Attribute der Konfiguration in dem `PropertySheetPanel` angezeigt werden sollen und wie diese angezeigt und editiert werden können. Von der für den Optionsdialog verwendeten Bibliothek L2FPROD ist `BaseBeanInfo` als Basisklasse vorgesehen. Um die Implementierung der `BeanInfo`-Klassen zu erleichtern, gibt es in `gui.util` unter anderem folgende Hilfsklassen:

`ExtendedBeanInfo` erweitert `BaseBeanInfo` um das Hinzufügen von Attributen und Manipulieren von deren gesetzten Editoren über den `PropertyDescriptor`. Dadurch kann in Unterklassen für einzelne Attribute gezielt die Editorklasse gesetzt werden.

`PropertiesBeanInfo` erweitert `ExtendedBeanInfo` um das automatische Registrieren der Attribute. Übergibt man dem Konstruktor nur die Konfigurationsklasse, werden alle Attribute, die über eine öffentliche und mit `@CongfigProperty` annotierte `get()`-Methode oder `is()`-Methode verfügen, per Reflection automatisch registriert. Alternativ kann man zusätzlich ein Array übergeben, in dem die zu registrierenden Attributnamen als `Strings` oder in `PropertyAssociatedClasses`-Instanzen stehen. Dann werden nur die in dem Array enthaltenen Attribute registriert. In den `PropertyAssociatedClasses`-Instanzen stehen neben den Attributnamen auch die dem Attribut zugeordneten Editor- und Rendererklassen. Diese können auch mit der Methode `setPropertyAssociated-Classes()` für einzelne Attribute gesetzt werden.

Die zu den Konfigurationsklassen gehörigen BeanInfo-Klassen werden bei der Erstellung des Optionsdialoges durch BeanInfoResolver gesucht und instanziiert. Dabei wird die Klasse PropertiesBeanInfo verwendet, falls für eine Konfigurationsklasse keine entsprechende BeanInfo-Klasse gefunden wird. Dadurch müssen die BeanInfo-Klassen nur

erstellt werden, falls für ein Attribut eine spezielle Editorklasse benötigt wird, was leider bei Enums der Fall ist.

Lokalisierung

Zu den Konfigurationsklassen können `ResourceBundle`s angelegt werden. Diese sind Unterklassen von `I18nResourceBundleWrapper`, die im gleichen Paket oder der gleichen umgebenden Klasse liegen wie die Konfigurationsklasse, und den gleichen Namen mit RB als Anhang tragen. In der zu implementierenden Methode `getKeyArray()` werden Schlüssel als Strings in einem Array zurückgegeben, die für die Suche nach den Übersetzungen verwendet werden. Dabei wird über die Klasse `I18n` in der eingestellten Lokalisierungsdatei nach den Schlüsseln, denen jeweils `"Config."` und der `RessourceBundle`-Klassenname gefolgt von einem Punkt vorangestellt werden, gesucht. Dadurch können sämtliche zu übersetzenden Begriffe in einer Datei eingetragen werden und sind über die Präfixe leicht zu finden.

Für jedes Attribut der Konfigurationsklasse können jeweils Schlüssel für Name, Beschreibung und Kategorie angegeben werden. Die Attribute werden in dem Optionsdialog nach den Kategorien gruppiert angezeigt. Bei dem Attribut wird der Name als Schlüssel verwendet, die Beschreibung wird unter dem Schlüssel Name.shortDescription gesucht, die Kategorie unter Name.category. Falls es zu einem Attribut keinen Beschreibungsschlüssel gibt, wird in dem Beschreibungsfeld des Dialogs die unter dem Attributnamen gefundene Übersetzung angezeigt.

Möchte man also Beschreibung und Name eines Attributs `value` der Konfigurationsklasse `SomeConfig` übersetzen, muss das zurückgegebene Array die Schlüssel `"value"` und `"value.shortDescription"` enthalten. In der deutschen Lokalisierungsdatei `Messages_German.properties` müsste dann die Zeile `"Config.SomeConfigRB.value.shortDescription = Beispiel-Wert"` enthalten sein, sowie eine entsprechende Zeile für `value`.

6.9.2. Konfigurationen aus Sicht der Editoren

Zu jedem Typ, der in einer Konfigurationsklasse verwendbar sein soll, müssen ein Editor und ein Renderer existieren. Der Renderer wird zum Anzeigen des Wertes benutzt, während der Editor verwendet wird, sobald das zu editierende Element angewählt wird. Dieses Konzept stammt aus der verwendeten Klasse `JTable`. Die Editor- und Renderer-Klassen können für den zu editierenden Typen registriert werden, oder explizit in der `BeanInfo`-Klasse angegeben werden, falls `PropertiesBeanInfo` als Basisklasse gewählt wird.

Editoren

Die Bibliothek L2FPROD stellt in `com.l2fprod.common.beans.editor` bereits einige Editorklassen zur Verfügung.

AbstractPropertyEditor ist die Basis für alle Editorklassen. Diese können eine der in der PropertyEditor-Dokumentation [19] angegebenen Möglichkeiten, z.B. Editieren als String oder per eigener Component, verwenden.

Attribute vom Typ Boolean können per BooleanAsCheckBoxPropertyEditor über eine JCheckBox geändert werden. Für Strings gibt es StringPropertyEditor, das genauso wie StringConverterPropertyEditor ein JTextField verwendet.

StringConverterPropertyEditor erlaubt das Editieren allgemeiner Klassen als String in diesem Textfeld. Dazu werden die Daten mittels in ConverterRegistry eingetragenen Konvertern von und nach String konvertiert. Es existieren Konverter für verschiedene Klassen aus java.awt, Boolean und die Standard-Java-Klassen, die von Number abgeleitet sind.

Diese Konverter werden auch in dem NumberPropertyEditor verwendet, welcher Unterklassen von Number akzeptiert, so dass auch für beliebige Zahlentypen leicht Editorklassen erstellt werden können. Für Double, Integer usw. sind diese bereits vorhanden.

Für Klassen mit einigen wenigen Werten, von denen jeweils ein Wert auswählbar sein soll, wie zum Beispiel Enums, eignet sich der ComboBoxPropertyEditor, bei dem mit setAvailableValues()/-Icons() die Werte und zugehörige Bilder gesetzt werden.

Weiterhin gibt es Editorklassen, die die Auswahl von Farben, Schriftarten oder Verzeichnissen erlauben.

Renderer

Renderer müssen für die Verwendung in dem auf PropertySheetTable aufbauenden Optionsdialog das Interface TableCellRenderer implementieren.

Die Bibliothek stellt in com.l2fprod.common.swing.renderer einige Renderer zur Verfügung. Der Standard-Renderer ist DefaultCellRenderer. Er erlaubt die Anzeige als String mit einem Icon, wobei für die Anzeige als String das ConverterRegistry-Singleton verwendet wird. Falls dort kein Konverter registriert ist, wird toString() verwendet. Weiterhin gibt es DateRenderer, das ein Datum unter Beachtung der eingestellten Sprache als Text ausgibt, und ColorCellRenderer, das eine Farbe als einfarbiges Icon und RGB Werte anzeigt. BooleanCellRenderer stellt den boolschen Wert per JCheckBox dar.

Konverter

Konverter konvertieren Objekte von einem Typen zu einem anderen. Dazu implementieren diese das Interface l2fprod.common.util.Converter. In dessen convert()-Methode ist für die übergebene Zielklasse wie auch für das übergebene Objekt zu prüfen, ob die Typen unterstützt werden. Dadurch kann ein Konverter für verschiedene Typenpaare verwendet werden. Verwendet werden sie vor allem durch DefaultCellRenderer für die Konvertierung von und nach String.

[19] http://download.oracle.com/javase/6/docs/api/java/beans/PropertyEditor.html

von	nach	Konverter
Boolean	String	BooleanConverter
String	Boolean	BooleanConverter
Dimension	String	AWTConverters
String	Dimension	AWTConverters
DimensionUIResource	String	AWTConverters
Insets	String	AWTConverters
String	Insets	AWTConverters
InsetsUIResource	String	AWTConverters
Point	String	AWTConverters
String	Point	AWTConverters
Rectangle	String	AWTConverters
String	Rectangle	AWTConverters
Font	String	AWTConverters
FontUIResource	String	AWTConverters
Number	N	NumberConverters
N	N	NumberConverters
N	String	NumberConverters
String	N	NumberConverters

Tabelle 6.7.: Registrierte Konverter

Registrierung

Die Konvertierungsklassen können in dem Singleton `ConverterRegistry` registriert werden. Für Editoren und Renderer werden eigene Registryklassen verwendet, deren Instanzen in `PropertySheetTable` erstellt werden. Diese sind über die Methoden `getRendererRegistry()` bzw. `getEditorRegistry()` der Klasse `PropertySheetPanel` zugänglich, so dass eigene Klassen registriert werden können.

Alle drei Registryklassen registrieren Standardklassen in ihren Konstruktoren.

Die standardmäßig registrierten Konverter sind in Tabelle 6.7 aufgelistet, wobei N ein beliebiges Element aus {`Double`, `Float`, `Integer`, `Long`, `Short`} ist. Diese Zuordnungen gelten auch für die entsprechenden primitiven Datentypen wie `boolean` und `int`.

Für die Editoren und Renderer gelten die in Tabelle 6.8 definierten Standardzuordnungen. Auch hier gelten die Zuordnungen wieder für die jeweiligen primitiven Datentypen.

X ist entweder `JCalendarDatePropertyEditor`, falls `com.toedter.calendar.JDateChooser` verfügbar ist, oder `NachoCalendarDatePropertyEditor`, falls `net.sf.nachocalendar.components.DateField` verfügbar ist.

`Object` hat selbstverständlich keinen Editor, da bei unbekanntem Typ nicht bekannt ist, wie dieser editiert werden kann. Es ist aber der `DefaultCellRenderer` für `Object` eingetragen, der `toString()` nutzt, falls er keinen Konverter findet. Dadurch kann zumindest jeder Typ angezeigt werden, was insbesondere bei schreibgeschützten Attributen, die nur eine `get()`-Methode haben, sinnvoll ist. Als Folge davon sind zum Beispiel

Typ	Renderer	Editor
Object	DefaultCellRenderer	
String		StringPropertyEditor
Boolean	BooleanCellRenderer	BooleanAsCheckBoxPropertyEditor
Byte	DefaultCellRenderer	
Character	DefaultCellRenderer	
Double	DefaultCellRenderer	DoublePropertyEditor
Float	DefaultCellRenderer	FloatPropertyEditor
Integer	DefaultCellRenderer	IntegerPropertyEditor
Long	DefaultCellRenderer	LongPropertyEditor
Short	DefaultCellRenderer	ShortPropertyEditor
Color	ColorCellRenderer	ColorPropertyEditor
Date	DateRenderer	X
File		FilePropertyEditor
Dimension		DimensionPropertyEditor
Insets		InsetsPropertyEditor
Rectangle		RectanglePropertyEditor

Tabelle 6.8.: Registrierte Editoren und Renderer

`byte` und `Byte` aktuell nicht editierbar, werden aber korrekt angezeigt.

Bei Renderern und Editoren wird entlang der Vererbungshierarchie nach einer registrierten Klasse gesucht, während bei den Konvertern nur nach genau dem einen Typen gesucht wird. Hier wäre natürlich genauso eine Suche nach einer passenden Klasse wünschenswert. Dazu müsste aber die benutzte Bibliothek angepasst werden.

6.10. Progress-Worker

Bei der GUI-Programmierung mit Swing in Java werden die durch Benutzeraktionen ausgelösten Ereignisse in einem speziellen Thread, dem Event Dispatcher Thread (EDT), verarbeitet. Auch das Zeichnen der GUI-Komponenten wird durch solche Ereignisse in dem EDT durchgeführt. Da jeweils nur ein Ereignis gleichzeitig behandelt und diese Behandlung nicht unterbrochen werden kann, dürfen die Methodenaufrufe, in denen die Ereignisse verarbeitet werden, nicht zu lange dauern. Anderfalls würde die GUI nicht mehr oder verzögert auf folgende Benutzerbefehle reagieren.

Um dennoch länger dauernde Berechnungen aus der GUI heraus starten zu können, gibt es das Konzept der Worker-Threads, die mit Hilfe der Klasse `SwingWorker` erstellt werden können. Diese Worker-Threads erlauben neben der parallelen Ausführung auch eine einfache Aktualisierung der GUI zur Anzeige des Fortschritts sowie von Zwischen- und Endergebnissen. Dafür bietet die Klasse Methoden an, die durch den Worker aufgerufen werden können und Benachrichtigungen auslösen, die automatisch auf dem Event Dispatcher Thread ausgeführt werden. Dadurch können in ihnen auch

GUI-Aktualisierungen durchgeführt werden.

Im Rahmen der Projektgruppe ist eine an die speziellen Bedürfnisse der aus der GUI heraus angestoßenen nebenläufigen Operationen angepasste Version des Java-SwingWorkers entstanden. Zum einen war die Fortschrittsbenachrichtigung bei der ursprünglichen Version nicht ausreichend, da diese nur Fortschrittswerte von 0 bis 100 zulässt, was nicht für eine genaue Anzeige der zu importierenden Moleküle ausreicht, da dies mehr als 50000 sein können. Weiterhin wurde ein Mechanismus zur Fehlerbehandlung benötigt, da die nebenläufigen Prozesse zum Beispiel Operationen auf der Datenbank ausführen, die fehlschlagen können, so dass der Benutzer die Möglichkeit benötigt, diese zu stoppen, erneut zu starten oder weiterlaufen zu lassen.

Das Worker-System besteht aus den beiden Worker-Klassen `SwingWorker` und `ProgressWorker` sowie einigen Hilfsklassen für die Benachrichtigungsmechanismen für den Fortschritt und die Fehlerbehandlung.

6.10.1. SwingWorker

Der `SwingWorker` ist ein modifizierter Standard-`SwingWorker` aus Java, ohne den Fortschrittsmechanismus, dafür mit Methoden zum erneuten Start des Workers per `restart()`, und zum Aufruf von `Runnable`-Objekten auf dem Event Dispatcher Thread (EDT) per `submit()`. Weiterhin kann in Subklassen die weitere Ausführung des Workers unterbunden werden, indem dort eine `StopException` geworfen wird.

`restart()` startet den Worker erneut, wobei intern `FutureTask.cancel()` aufgerufen wird. Diese Methode stoppt den Worker leider nicht immer augenblicklich, so dass diese Methode nur benutzt werden sollte, wenn sichergestellt werden kann, dass der Worker bereits gestoppt ist, oder unmittelbar nach dem Aufruf stoppt. Dies ist zum Beispiel per `StopException` aus dem Worker heraus möglich. Löst man eine `StopException` aus, wird der Worker unmittelbar beendet, es wird also nichtmal mehr `done()` aufgerufen.

In dem Worker kann per `isRestartingWorker()` geprüft werden, ob er gerade erneut gestartet wird, was z.B. in `done()` benutzt werden kann, um eine Benachrichtigung über das Ende des Workers zu unterbinden. Dies wird zum Beispiel im `ProgressWorker` für die Fehlerbehandlung eingesetzt. Weiterhin kann mit `Thread.interrupted()` geprüft werden, ob der Worker per `worker.cancel(true)` von aussen gestoppt wurde. Da dieses Stoppen des Workers nicht immer unmittelbar umgesetzt wird, sollte die Überprüfung in der `doInBackground()`-Methode des Workers vor möglicherweise langwierigen Operationen geschehen, und in diesem Falle die `doInBackground()`-Methode verlassen werden um die weitere Ausführung des Workers zu unterbinden.

Eigene `Runnable`-Instanzen können per `submit()` auf den EDT gelegt werden, wobei der gleiche Mechanismus wie bei den PropertyChange-Nachrichten und bei `publish()` verwendet wird, so dass die Reihenfolge der Benachrichtigungen der Listener die gleiche wie die der Aufrufe im Worker ist. Der Mechanismus verwendet wiederum intern einen `Timer` um die `Runnable`s auf dem EDT aufzurufen, so dass eine kleine Verzögerung zwischen Aufruf von `submit()` im Worker und Start des `Runnable` liegen kann.

Für die anderen Funktionen des `SwingWorker`s gelten alle Beschreibungen weiter, wie

zum Beispiel die im “Swing Concurrency Tutorial”[20] Insbesondere lässt sich der Worker von aussen per `cancel()` stoppen, und der `publish()`-Mechanismus funktioniert weiterhin.

Allerdings sollte `get()` nicht unmittelbar nach `execute()` aufgerufen werden, obwohl der Thread dann wartet, bis der Worker beendet wird, da dann durch den intern verwendeten `FutureTask` der Aufruf der `done()`-Methode verhindert werden kann. In solchen Fällen müssen der Aufrufer-Thread und der Worker manuell synchronisiert werden, zum Beispiel per `java.util.concurrent.Exchanger` wie in der Klasse `ProgressWorkerTest`.

6.10.2. ProgressWorker

Der `ProgressWorker` erbt von `SwingWorker` und ergänzt ihn um `ProgressListener` und `WorkerExceptionListener`.

6.10.2.1. Progress

Es werden Methoden bereitgestellt, mit denen die `ProgressListener` registriert und aus den Subklassen heraus aufgerufen werden können, wobei die exakt gleichen Methodennamen wie in diesen Listenern verwendet werden. Die Beschreibung dieser Befehle befindet sich im nachfolgenden Abschnitt. Der `ProgressWorker` ruft automatisch zu Anfang unmittelbar vor der `doInBackground()`-Methode `setProgressIndeterminate(true)` und nach dem Beenden des Workers `finished()` mit den entsprechenden Werten auf. Letzeres geschieht in der `done()`-Methode, die also nicht durch Subklassen überschrieben werden sollte. Wird ein Worker direkt per `StopException` beendet, wird diese Methode nicht aufgerufen, so dass die `ProgressListener` keine Benachrichtigung über die Beendigung per `finished()` erhalten. Der Worker muss in solchen Fällen diese Methode selber aufrufen (so z.B. in `handleException` durchgeführt). Wird der `ProgressWorker` von ausserhalb über `cancel(true)` beendet, resultiert dies in einer Benachrichtigung der `ProgressListener` über einen Aufruf von `finished(null, true)` durch `done()`.

6.10.2.2. Exceptions

Für die Fehlerbehandlung gibt es die Methode `handleException()`, die an entsprechender Stelle im `catch`-Block aus der `doInBackground()`-Methode aufgerufen werden kann. Dabei werden sämtliche registrierte `WorkerExceptionListener` nacheinander aufgerufen, wie dies im Abschnitt 6.10.4 erläutert wird. Natürlich kann auch eine gewöhnliche Ausnahmebehandlung im `catch`-Block durchgeführt werden, allerdings ist dann ein Neustart oder ähnliches des Workers etwas aufwändiger und auch fehleranfälliger, da die in der Methode `handleException()` vorgenommenen Aufrufe dann selbstständig ausgeführt werden müssen.

[20] `http://download.oracle.com/javase/tutorial/uiswing/concurrency/index.html`

6.10.3. ProgressListener

Die `ProgressListener` werden durch den `ProgressWorker` immer auf dem Event Dispatcher Thread ausgeführt, so dass hier beliebige GUI-Operationen möglich sind. Ein `ProgressListener` kann vier verschiedene Nachrichtenarten vom Worker empfangen:

setProgressIndeterminate `true` gibt an, dass der Worker nicht abschätzen kann, wieviel Arbeit zu erledigen ist. Vor dem Aufruf von `doInBackground` setzt der `ProgressWorker` den `indeterminate`-Modus auf `true`. Ein Worker muss dementsprechend diesen Modus auf `false` setzen, sofern er weiss, wieviel er zu tun hat. Der Wert kann beliebig oft geändert werden, falls sich bei der Verarbeitung die Situation ändert.

setProgressBounds Gibt die erwarteten Grenzen des Fortschrittes an. Der Listener sollte seine Anzeige auf diese Grenzen hin anpassen. Es wird kein Standardwert durch den `ProgressWorker` gesetzt. Auch dieser Wert kann beliebig verändert werden.

setProgressValue Gibt den aktuellen Fortschritt an. Auch hier wird kein Standardwert gesetzt, und der Wert kann offensichtlich beliebig oft verändert werden.

finished Dies signalisiert dem Listener das Ende des Workers. Diese Methode sollte nicht direkt im Worker aufgerufen werden, da dies bereits durch den `ProgressWorker` mit dem korrekten Ergebnis geschieht.

Diese Befehle können auch direkt auf einer `ProgressWorker`-Instanz aufgerufen werden, wobei die Klasse `ProgressWorker` die Benachrichtung der registrierten Listener vornimmt. Dabei ist durch die Verwendung von `SwingWorker.submit()` die Reihenfolge der Nachrichten untereinander und in Bezug auf die per `firePropertyChange()` versandten Nachrichten garantiert.

6.10.4. WorkerExceptionListener

`WorkerExceptionListener` werden per `addExceptionListener()` bei dem `ProgressWorker` registriert und dienen der Behandlung der im Worker auftretenden Exceptions von aussen. Dazu wird die Methode `exceptionThrown()` in dem Worker Thread aufgerufen und die zu behandelnde Exception übergeben. Zur Behandlung des Fehlers kann z.B. ein Dialog angezeigt werden, in dem der Benutzer gefragt wird, was zu tun ist. Dabei muss der eigene Thread blockiert werden, was z.B. die Methoden `JOptionPane.show*()` und `Dialog.setVisible()` bereits erledigen. Das Ergebnis dieser Behandlung wird dann durch die Methode zurückzugeben. Hierbei gibt es die folgenden Rückgabewerte:

NOT_HANDLED Die Exception wurde nicht behandelt, da z.B. der Typ nicht durch diesen Listener behandelt werden kann.

CONTINUE Der Listener hat die Exception behandelt und ist der Meinung, dass der Worker fortgesetzt werden kann. Der `ProgressWorker` reagiert darauf, indem er

den Worker weiterlaufen lässt, falls dies möglich ist. Je nachdem, an welcher Stelle diese Behandlung auftritt, kann dies verschiedene Konsequenzen haben:

- War im Worker an der Stelle, an der die Exception aufgetreten ist, kein `try-catch`-Block mit `handleException()` Aufruf, wird der Rest des Workers natürlich nicht weiter ausgeführt, da die Exception intern durch den `SwingWorker` (genauer dessen `FutureTask`) gefangen wurde und erst durch `ProgressWorker.done()` behandelt wird. Dabei wird dann `finished(null, true)` aufgerufen und der Worker normal beendet.
- Wurde die Exception hingegen im Worker selber behandelt, wird der restliche Code nach dem `catch`-Block natürlich normal ausgeführt. Insbesondere können, wenn man die Exception-Behandlung in einer Schleife per `handleException()` durchführt, die restlichen Schleifendurchläufe durchgeführt werden, wie bei einer normalen Exception-Behandlung.

RESTART Der Listener hat die Exception behandelt und ist der Meinung, dass der Worker erneut gestartet werden soll. Der `ProgressWorker` reagiert darauf, indem der Worker beendet und neu gestartet wird. Dabei wird wieder der anfängliche indeterminate Modus gesetzt, die Listener erhalten aber kein `finished()` für den ersten (nicht komplett beendeten) Durchlauf des Workers.

STOP Der Listener hat die Exception behandelt und ist der Meinung, dass der Worker beendet werden muss. Der `ProgressWorker` reagiert darauf, indem auf den `ProgressListenern` `finished(null, true)` aufgerufen, und anschließend der Worker sofort beendet wird.

Die Behandlung durch die Listener wird sofort gestoppt, sobald ein Listener einen Wert ungleich `NOT_HANDLED` zurückgibt. Geben alle `WorkerExceptionListener` `NOT_HANDLED` zurück, wird die Exception von `ProgressWorker` selbst durch Ausgabe des Stacktrace behandelt und der Worker fortgesetzt, falls möglich. Dieses Standardverhalten war der Hauptgrund für die Einführung des Exceptionhandlers, da ansonsten Exceptions verloren gehen konnten, wenn man das Ergebnis des Workers nicht per `get()` abgeholt hat.

6.10.5. Ausführung mit Fortschrittsdialog

Zur Ausführung eines Workers mit einem Fortschrittsdialog steht die statische Methode `executeWithProgressDialog()` in der Klasse `ProgressWorkerUtil` zur Verfügung, in der ein Dialog erstellt wird, der ein `ProgressPanel` enthält. Diese Methode kann natürlich Vorlage für einen eigenen Fortschrittsdialog sein, der z.B. zusätzlich den Fortschritt als Zahl anzeigt.

6.10.6. Beispiel

Die Code-Listings 6.1, 6.2 und 6.3 zeigen exemplarisch den Aufbau von einem Worker, einer Klasse zur Fortschrittsbenachrichtigung sowie einer fehlerbehandelnden Klasse.

Listing 6.1: Beispiel-Worker

```
class SomeWorker extends ProgressWorker<Integer, Void>
{
  @Override
  protected Integer doInBackground() {
    setProgressIndeterminate(false);
    setProgressBounds(0,n-1);
    Integer k = 0;
    for (int i=0; i<n; ++i) {
      try {
        k = doSomething(k);
        setProgressValue(i);
      } catch (Exception e) {
        handleException(e);
      }
    }
    return k;
  }
}
```

Die in der Klasse `SomeWorker` benutzte Methode `doSomething()` beinhaltet dabei die Worker-Berechnung, deren Ergebnis in `k` gespeichert und anschließend als Endergebnis der Methode zurückgegeben wird. Bei der Berechnung kann eine `SomeException` genannte Exception auftreten. Es werden allerdings sämtliche Exceptions abgefangen, so dass auch z.B. bei einer `NullPointerException` immerhin der Stack durch die Standardfehlerbehandlung des `ProgressWorkers` ausgegeben wird.

In der Klasse `SomeProgressListener` wird eine Instanz einer fiktiven Fensterklasse `SomeWindow` zur Darstellung des Fortschritts in deren Statuszeile verwendet.

Die Fehlerbehandlung mittels der Klasse `SomeExceptionListener` zeigt bei dem Auftreten einer Exception des Typs `SomeException` einen einfachen Fehlerdialog, in dem der Benutzer auf Ignorieren oder Abbrechen klicken kann. Bei einem Klick auf Abbrechen wird der Worker gestoppt, andernfalls wird er fortgesetzt.

Mit den so definierten Klassen lässt sich dann auf die in Listing 6.4 dargestellte Weise ein Worker mit einem Fortschrittsdialog erstellen. `SomeProgressDialog` enthalte dabei ein `ProgressPanel`, das durch den Konstruktor beim `worker` registriert wird, und einen Abbrechen-Button. Ein Klick auf diesen löse `worker.cancel(true)` aus, was den `worker` beendet, sobald dies möglich ist.

Listing 6.2: Beispiel für Fortschrittsanzeige

```
class SomeProgressListener implements ProgressListener<Integer>
{
  private int N;
  private SomeWindow someWindow;

  SomeProgressListener(SomeWindow someWindow) {
    this.someWindow = someWindow;
  }

  @Override
  public void setProgressValue(int progress) {
    someWindow.setStatusMessage("done: " + progress + "/" + N);
  }

  @Override
  public void setProgressIndeterminate(boolean indeterminate) {
  }

  @Override
  public void setProgressBounds(int min, int max) {
    this.N = max-min+1;
  }

  @Override
  public void finished(Integer result, boolean cancelled) {
    someWindow.setStatusMessage("done");
  }
}
```

Listing 6.3: Beispiel für Fehlerbehandlung

```
class SomeExceptionListener implements WorkerExceptionListener
{
  @Override
  public ExceptionHandlerResult exceptionThrown(Throwable e) {
    if (e instanceof SomeException) {
      int result = showSomeErrorDialog(e);
      if (result == SomeErrorDialog.ABORT) {
        return ExceptionHandlerResult.STOP;
      } else {
        return ExceptionHandlerResult.CONTINUE;
      }
    } else {
      return ExceptionHandlerResult.NOT_HANDLED;
    }
  }
}
```

Listing 6.4: Beispiel-Anwendung

```
SomeWindow window = new SomeWindow();
SomeWorker worker = new SomeWorker();
SomeProgressDialog progressDialog =
  new SomeProgressDialog(worker);
worker.addProgressListener(new SomeProgressListener(window));
worker.addExceptionListener(new SomeExceptionListener());
worker.execute();
progressDialog.setVisible(true);
```

7. Zukunftsausblick und mögliche Erweiterungen

Da der Scaffold Hunter sehr modular aufgebaut ist, eignet er sich gut, um an verschiedenen Stellen erweitert zu werden. Wir haben bei der Planung und Implementierung sehr darauf geachtet, klare Schnittstellen zu definieren, die ein einfaches Erweitern möglich machen. Dazu gehören unter anderem die Ansichten, die Import-, Calc- und Export-Plugins, die Linkage- und Distanzklassen im Clustering, sowie Filterfunktionen. Da die Datenbank über das Framework Hibernate angesprochen wird, sind Portierungen auf weitere Datenbanken (theoretisch) ohne großen Aufwand möglich.

Einige Funktionen über die wir uns Gedanken gemacht haben, welche es aber aus verschieden Gründen nicht mehr in die aktuelle Version geschafft haben sind:

Heatmap

Eine *Heatmap* zeigt eine Farbverteilung über eine Eigenschaft für eine Reihe von Strukturen. Heatmaps werden vor allem für die visuelle Repräsentation von *chemischer Aktivität* der Moleküle gegen ein bestimmtes *Target* genutzt. Bei der Einbindung in den Scaffold Hunter könnte dies als neue Ansicht geschehen, die Molekülen anhand ausgewählter Eigenschaften Farben zuweist, beispielsweise in Form einer Matrix. Dabei sind theoretisch alle numerischen Eigenschaften als Grundlage denkbar, oder eine neue äktivitäts Eigenschaft wird in das Programm integriert.

Nichthierarchisches Clustering

Nichthierarchisches Clustering, zum Beispiel *K-Means* könnte in einer neuen Ansicht implementiert, oder in eine Bestehende integriert werden. Anbieten würde sich die Plottansicht, bei der die Cluster farblich hervorgehoben werden könnten.

Externe Links

In die Detailansicht bzw. Tooltips der Moleküle / Scaffolds könnten Links eingebettet werden, die zu externen Programmen / Webseiten verweisen. Eine Möglichkeit ist ein Link zu z.B. *PubChem* der direkt die Seite des entsprechenden Moleküls öffnet. Die Hyperlinks könnten dabei entweder als URL-Eigenschaft gespeichert werden, oder dynamisch anhand der Compound ID erzeugt werden. Dabei müsste für den Nutzer eine Möglichkeit geschaffen werden, ein URL Schema zu erstellen, nach dem der externe Link aus einer Basis-URL und der Compound ID (oder einem beliebigen anderen Attribut aus dem Datenset) erzeugt wird. Alternativ ließe sich auch ein Calc-Plugin schreiben,

welches eine Eigenschaft mit einem URL-String erzeugt. Der Benutzer könnte dann in der entsprechenden Plugin-Konfiguration ein URL-Schema und eine einzusetzende Eigenschaft wie Compound-ID angeben, nach der das Plugin den URL-String erzeugt. Zur Zeit wandelt zumindest das Tooltip-Fenster Moleküleigenschaften, die "wie URL-Strings aussehen" (realisiert über reguläre Ausdrücke) in einen klickbaren Hyperlink um. Hier wäre es sinnvoll die Hyperlinks z.B. auch über das Kontextmenü eines Scaffolds/Moleküls anzubieten.

Download Plugins

Eine weitere Erweiterungsmöglichkeit stellen Download Plugins dar, die es ermöglichen das Ergebnis einer Internetsuche direkt zu importieren. Der Nutzer kann also aus dem Scaffold Hunter heraus eine Suche in einer Onlinequelle, z.B. PubChem starten, sich die gefundenen Ergebnisse anzeigen lassen und sie dann auf Knopfdruck in die Datenbank importieren.

Calc Plugins

Im *Chemical Development Toolkit* (CDK) befinden sich noch eine Reihe weiterer Deskriptoren und Fingerprints, für die Berechnungsplugins erstellt werden könnten. Weiterhin wäre ein Normalisierungsplugin denkbar, welches Eigenschaften entsprechend einer Vorschrift normiert.

Substruktur-Suche

In der ursprünglichen Version des Scaffold Hunters war eine *Substruktur-Suche* integriert. Diese wieder einzubauen wäre sicher eine sinnvolle Maßnahme. Die benötigte Berechnung der Substruktur-Fingerprints ließe sich leicht in Form eines Calc-Plugins integrieren.

Text-Suche

Eine Suchfunktion zum Selektieren von Molekülen / Scaffolds, die in den Kommentaren oder String-Eigenschaften einem Suchkriterium entsprechen wäre eine zusätzliche Erweiterungsmöglichkeit.

A. Literaturverzeichnis

[1] http://code.google.com/p/jspf/.

[2] http://www.piccolo2d.org/index.html.

[3] Dimitris K. Agrafiotis, Simson Alex, Heng Dai, An Derkinderen, Michael Farnum, Peter Gates, Sergei Izrailev, Edward P. Jaeger, Paul Konstant, Albert Leung, Victor S. Lobanov, Patrick Marichal, Douglas Martin, Dmitrii N. Rassokhin, Maxim Shemanarev, Andrew Skalkin, John Stong, Tom Tabruyn, Marleen Vermeiren, Jackson Wan, Xiang Y. Xu, and Xiang Yao. Advanced Biological and Chemical Discovery (ABCD): Centralizing Discovery Knowledge in an Inherently Decentralized World. *Journal of Chemical Information and Modeling*, 47(6):1999–2014, November 2007.

[4] Bin Chen, Xiao Dong, Dazhi Jiao, Huijun Wang, Qian Zhu, Ying Ding, and David Wild. Chem2bio2rdf: A semantic framework for linking and data mining chemogenomic and systems chemical biology data. *BMC Bioinformatics*, 11(1):255, 2010.

[5] Mukund Deshpande, Michihiro Kuramochi, Nikil Wale, and George Karypis. Frequent substructure-based approaches for classifying chemical compounds. *IEEE Trans. on Knowl. and Data Eng.*, 17:1036–1050, August 2005.

[6] Geoff M. Downs and John M. Barnard. Clustering Methods and Their Uses in Computational Chemistry. *Reviews in Computational Chemistry*, 18:1–40, 2002.

[7] Ricard Garcia-Serna, Oleg Ursu, Tudor I. Oprea, and Jordi Mestres. iphace. *Bioinformatics*, 26:985–986, April 2010.

[8] Hanna Geppert, Martin Vogt, and Jürgen Bajorath. Current Trends in Ligand-Based Virtual Screening: Molecular Representations, Data Mining Methods, New Application Areas, and Performance Evaluation. *Journal of Chemical Information and Modeling*, 50(2):205–216, February 2010.

[9] Alireza Givehchi, Axel Dietrich, Paul Wrede, and Gisbert Schneider. ChemSpaceShuttle: A tool for data mining in drug discovery by classification, projection, and 3D visualization. *QSAR and Combinatorial Science*, 22(5):549–559, 2003.

[10] Th. Hanser, Ph. Jauffret, and G. Kaufmann. A new algorithm for exhaustive ring perception in a molecular graph. *Journal of Chemical Information and Modeling*, 36:1146–1152, 1996.

[11] Michael J. Keiser, Bryan L. Roth, Blaine N. Armbruster, Paul Ernsberger, John J. Irwin, and Brian K. Shoichet. Relating protein pharmacology by ligand chemistry. *Nature Biotechnology*, 25(2):197–206, feb 2007.

[12] Michael J. Keiser, Vincent Setola, John J. Irwin, Christian Laggner, Atheir I. Abbas, Sandra J. Hufeisen, Niels H. Jensen, Michael B. Kuijer, Roberto C. Matos, Thuy B. Tran, Ryan Whaley, Richard A. Glennon, Jérôme Hert, Kelan L. H. Thomas, Douglas D. Edwards, Brian K. Shoichet, and Bryan L. Roth. Predicting new molecular targets for known drugs. *Nature*, 462:175–181, November 2009.

[13] Michael Kuhn, Damian Szklarczyk, Andrea Franceschini, Monica Campillos, Christian von Mering, Lars Juhl Jensen, Andreas Beyer, and Peer Bork. Stitch 2: An interaction network database for small molecules and proteins. *Nucleic Acids Research*, 38(suppl 1), 2010.

[14] Michael Kuhn, Christian von Mering, Monica Campillos, Lars Juhl Jensen, and Peer Bork. Stitch: Interaction networks of chemicals and proteins. *Nucleic Acids Research*, 36(suppl 1), 2008.

[15] Eugen Lounkine, Mathias Wawer, Anne Mai Wassermann, and Jürgen Bajorath. Saranea: A freely available program to mine structure-activity and structure-selectivity relationship information in compound data sets. *Journal of Chemical Information and Modeling*, 50(1):68–78, 2010.

[16] Daniele Merico, David Gfeller, and Gary D Bader. How to visually interpret biological data using networks. *Nature Biotechnology*, 27(10):921–924, October 2009.

[17] F. Murtagh. Chapter 3. In *Multidimensional Clustering Algorithms*, pages 59–88. Physica Verlag, 57th edition edition, 1985.

[18] Jose C Nacher and Jean-Marc Schwartz. A global view of drug-therapy interactions. *BMC Pharmacology*, 8(1):5, 2008.

[19] T. Oprea and A. Tropsha. Target, chemical and bioactivity databases – integration is key. *Drug Discovery Today: Technologies*, 3(4):357–365, FebApr 2006.

[20] Gaia V. Paolini, Richard H. Shapland, Willem P. van Hoorn, Jonathan S. Mason, and Andrew L. Hopkins. Global mapping of pharmacological space. *Nature Biotechnology*, 24(7):805–815, July 2006.

[21] Ansgar Schuffenhauer, Peter Ertl, Silvio Roggo, Stefan Wetzel, Marcus A. Koch, and Herbert Waldmann. The scaffold tree − visualization of the scaffold universe by hierarchical scaffold classification. *Journal of Chemical Information and Modeling*, 47(1):47–58, 2007. PMID: 17238248.

[22] Junmei Wang and Tingjun Hou. Drug and Drug Candidate Building Block Analysis. *Journal of Chemical Information and Modeling*, 50(1):55–67, January 2010.

[23] Stefan Wetzel, Karsten Klein, Steffen Renner, Daniel Rauh, Tudor I. Oprea, Petra Mutzel, and Herbert Waldmann. Interactive exploration of chemical space with Scaffold Hunter. *Nature Chemical Biology*, 5(8):581–583, June 2009.